Computer Networks

Slides courtesy Kurose & Ross

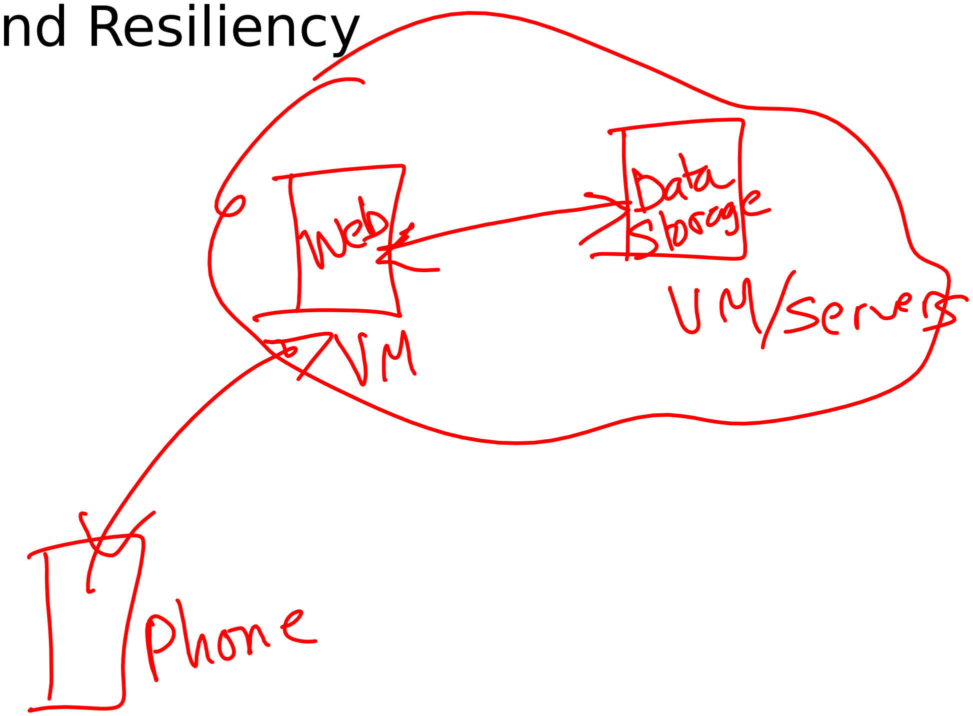
Agenda

- Computer networks, primarily from an application perspective
- Protocol layering
- Client-server architecture
- End-to-end principle
- TCP
- Socket programming

(1) (2) (3) Remote Procedure Calls (RPC)

Why Networking?

- All communication takes place over computer networks
- Networking affects how we design distributed systems:
  - Architecture
  - Performance
  - Reliability and Resiliency



Networking Goals

- Reliable delivery of data (packets)
- Low latency delivery of data [~ ms]
- Utilize physical networking bandwidth
- Share network bandwidth among multiple agents

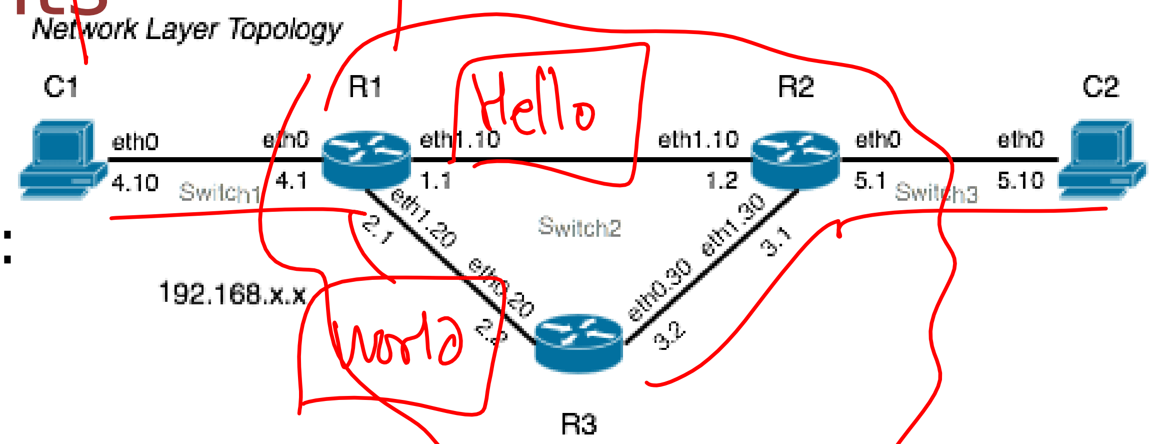
end devices

cloud servers

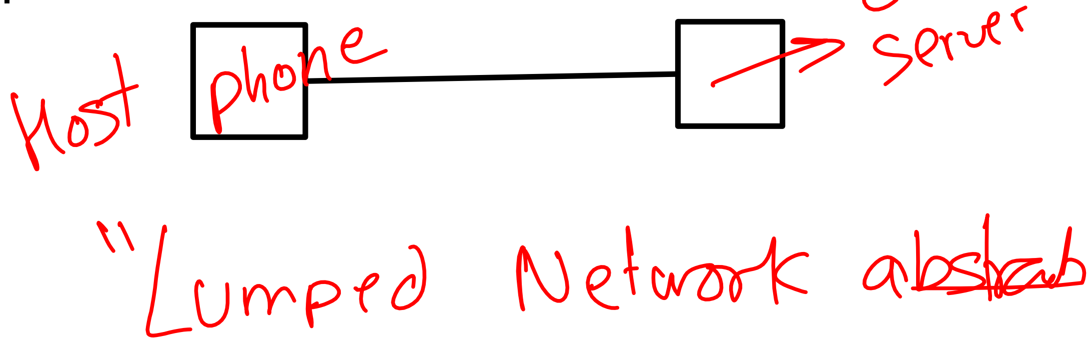
:

Network Elements

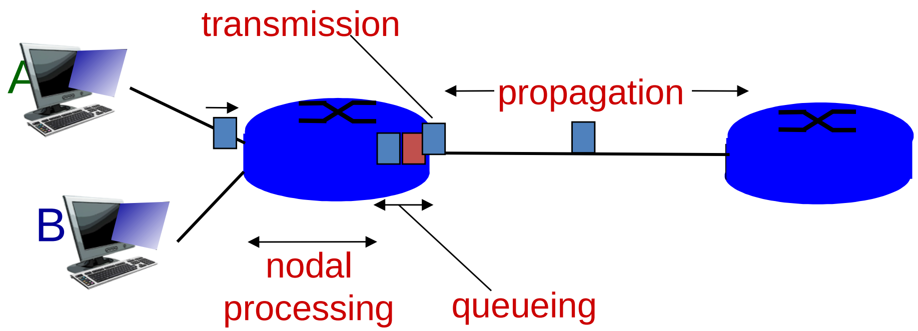
- Links:
  - Wired or wireless
- Hosts or end-points:
  - Servers/clients
- Packets:
  - Units of data transmission
- Switches, Routers, Middleboxes:
  - Receive, process, forward packets



Abstraction



Four sources of packet delay



$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

d_{proc}: nodal processing

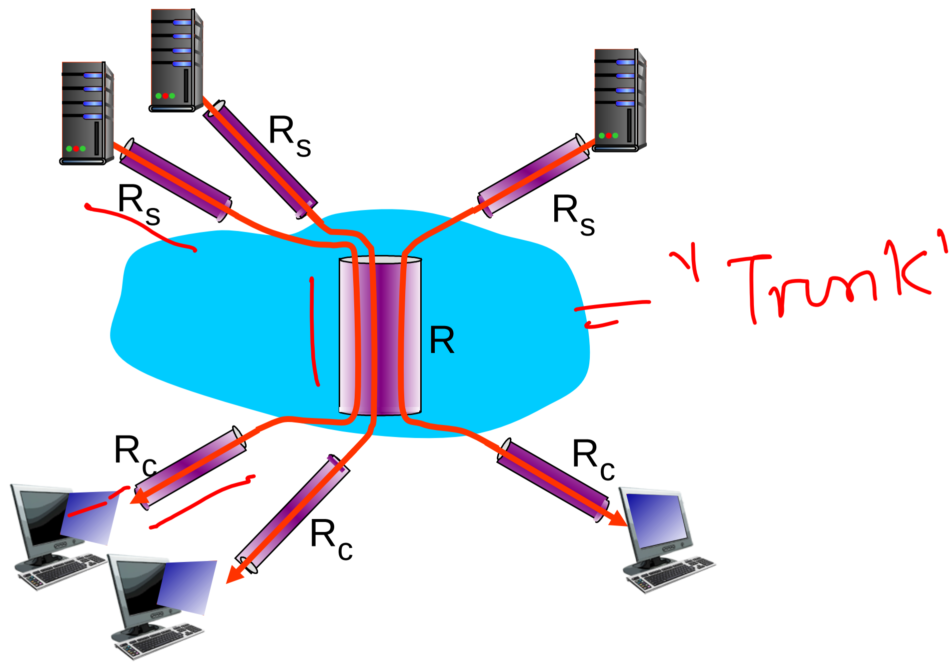
- check bit errors
- determine output link
- typically < msec

d_{queue}: queueing delay

- time waiting at output link for transmission
- depends on congestion level of router

Throughput: Internet scenario

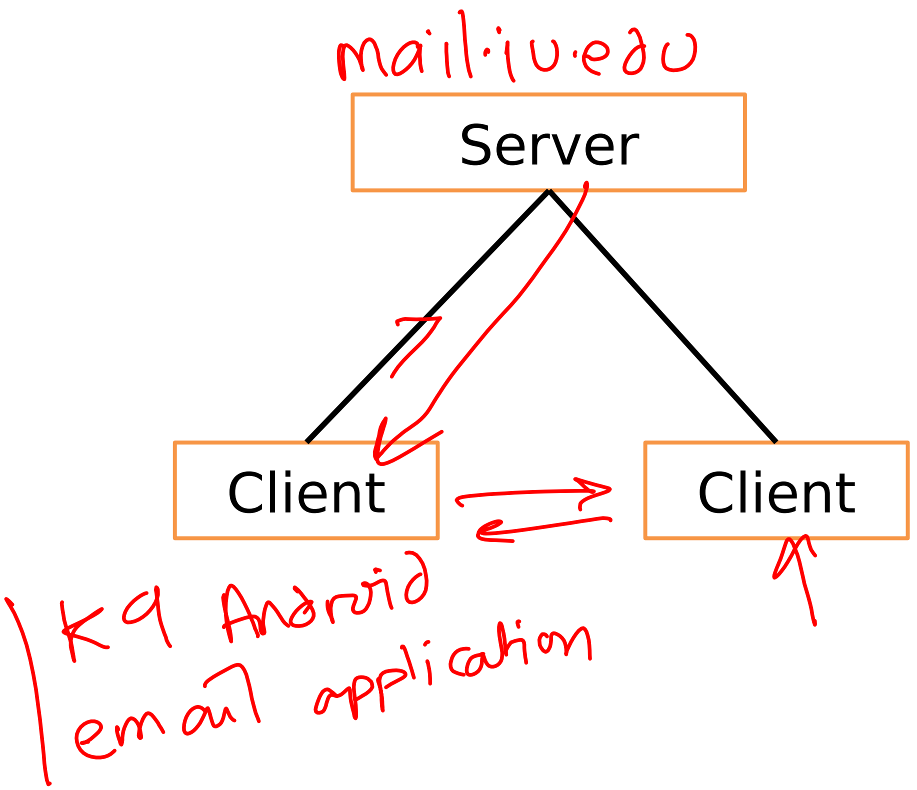
- per-connection end-end throughput: $\min(R_c, R_s, R/10)$
- in practice: R_c or R_s is often bottleneck



10 connections (fairly) share backbone bottleneck link R bits/sec

"Network flow":
(src, dest, ..., ...)

Client-server architecture



Server:

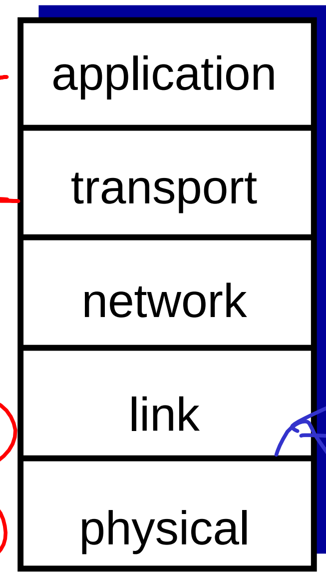
- always-on host
- permanent IP address
- data centers for scaling

Clients:

- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

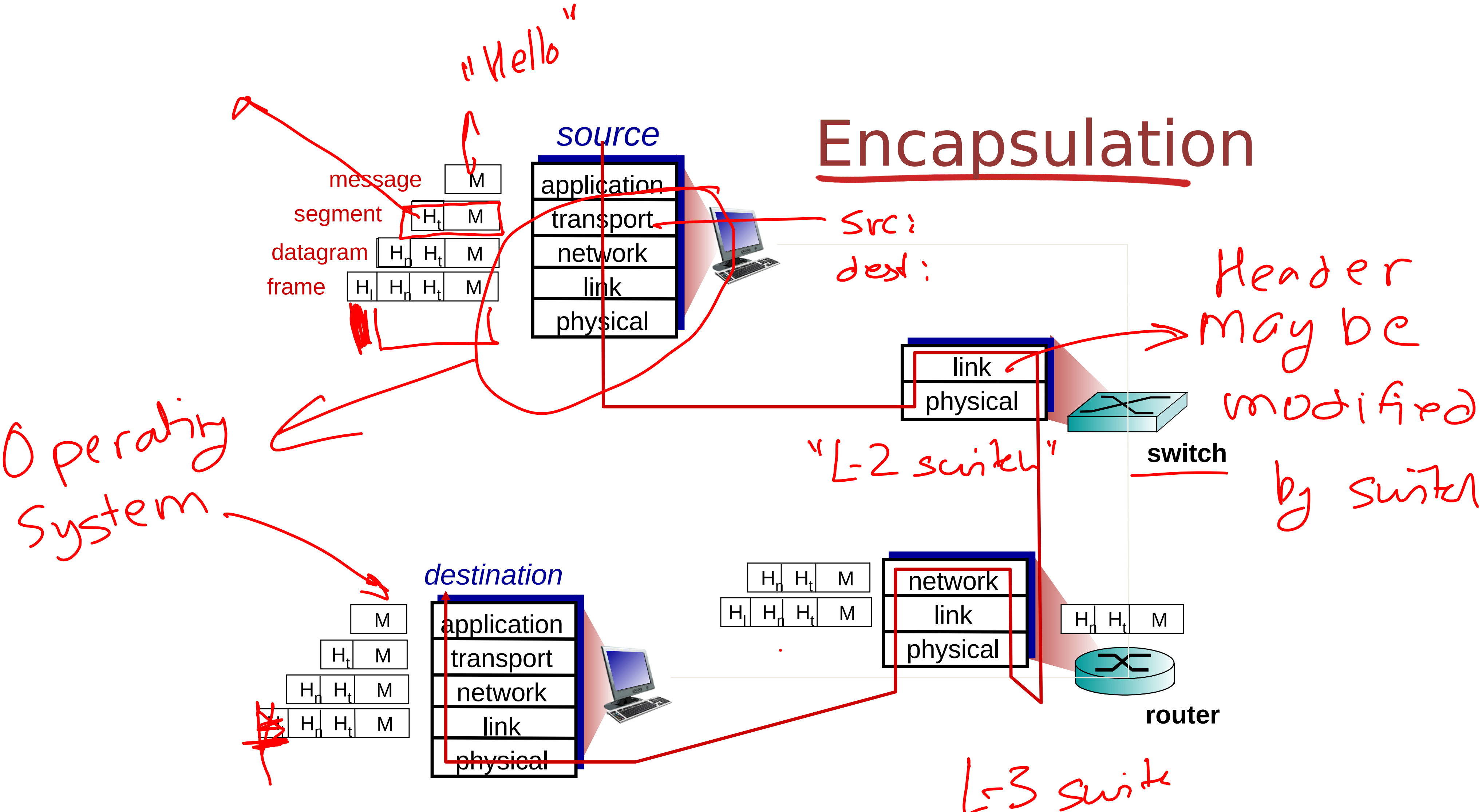
Internet protocol stack

- *application:* supporting network applications
  - FTP, SMTP, HTTP
- *transport:* process-process data transfer
  - TCP, UDP
- *network:* routing of datagrams from source to destination
  - IP, routing protocols
- *link:* data transfer between neighboring network elements
  - Ethernet, 802.111 (WiFi), PPP
- *physical:* bits “on the wire”



meta data, state, infor for the protocol itself to work.

Encapsulation



App-layer protocol defines

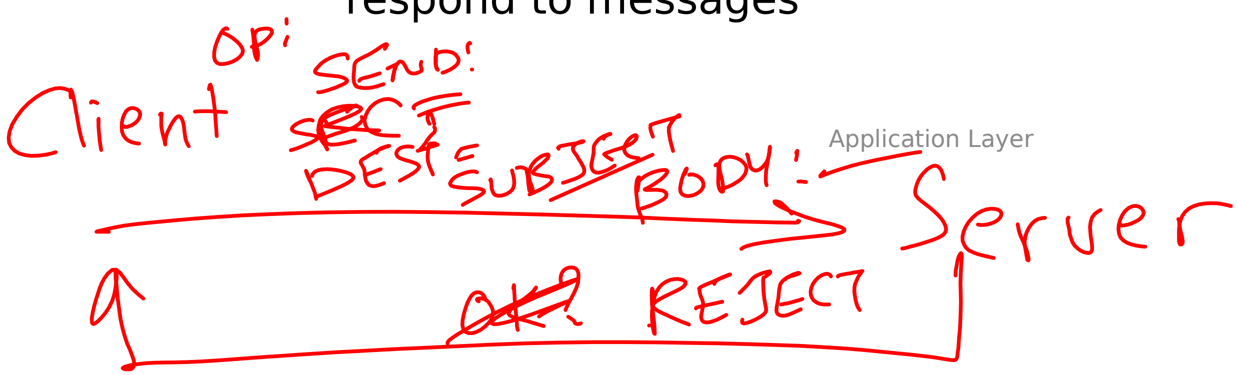
- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

open protocols:

- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

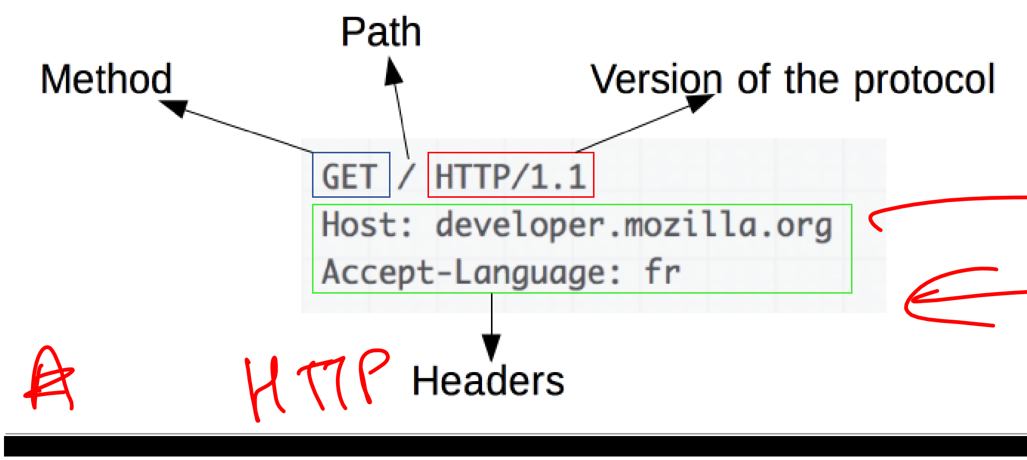
- e.g., Skype



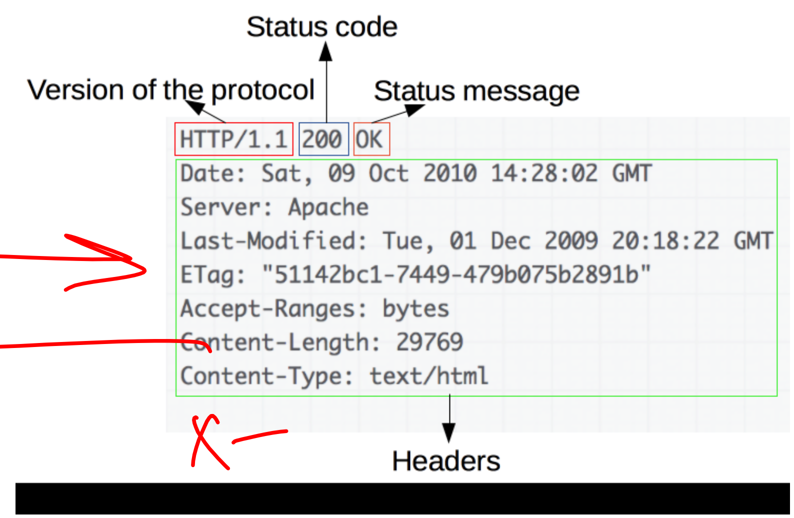
— Web Requests

— API

HTTP Header Example



Request



Response

HTTP overview

uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"

- server maintains no information about past client requests

aside

protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

What transport service does an app need?

data integrity

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- other apps ("elastic apps") make use of whatever throughput they get

security

- encryption, data integrity, …

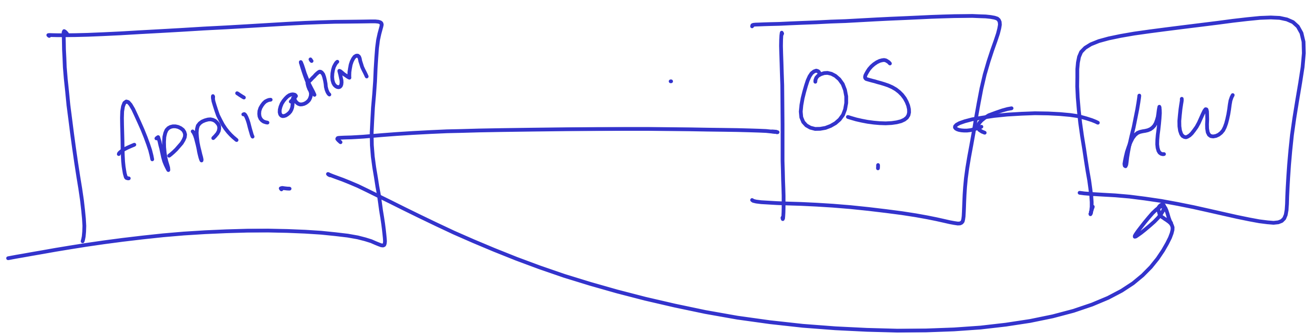
Principle Of End-To-End System Design

"END-TO-END ARGUMENTS IN SYSTEM DESIGN" J.H. Saltzer, D.P. Reed and D.D. Clark

- *Where to implement functionality in a distributed system?*
  - Especially relevant in networking
- Example: Copy a file across the network reliably
  - Option 1 : Copy file, and then verify contents using checksums
  - Option 2 : Build a perfectly reliable network, routers, etc.
- Even with a perfectly reliable network, things can go wrong
  - Need application level verification anyway

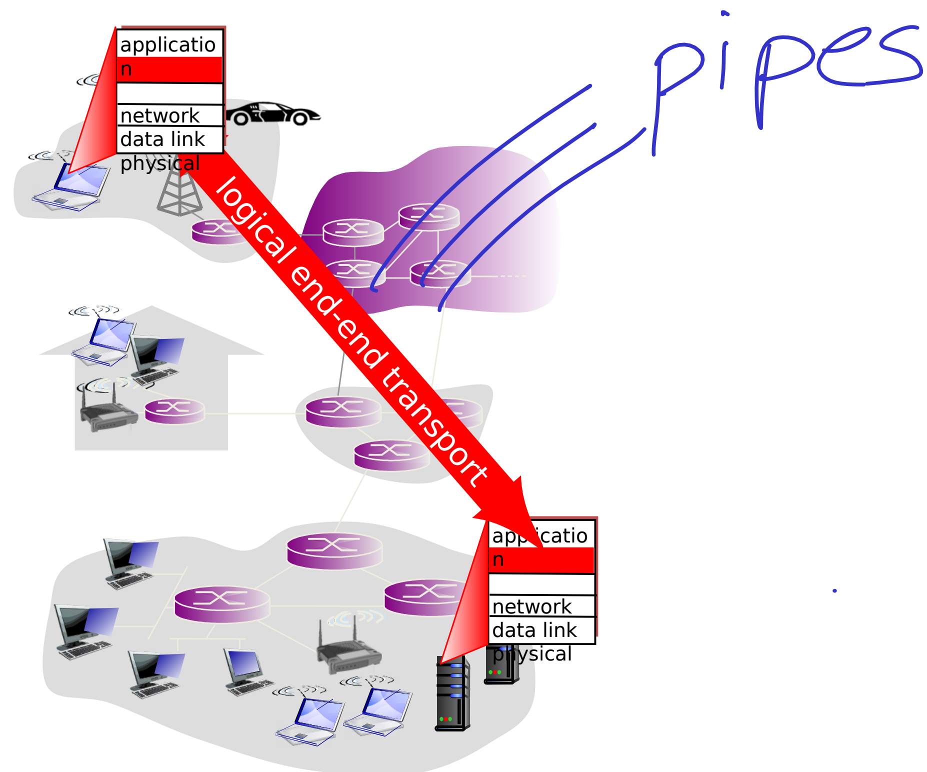
Principle Of End-To-End System Design (2/2)

- It is better to implement functionality at the "ends" of the network (aka the hosts) ← clients/servers
  - Enables effective layering
  - Better to implement functionality at higher layers of abstraction
- Also useful in non-network settings like operating systems
  - Implementing system calls in hardware is not a great idea



Transport services and protocols

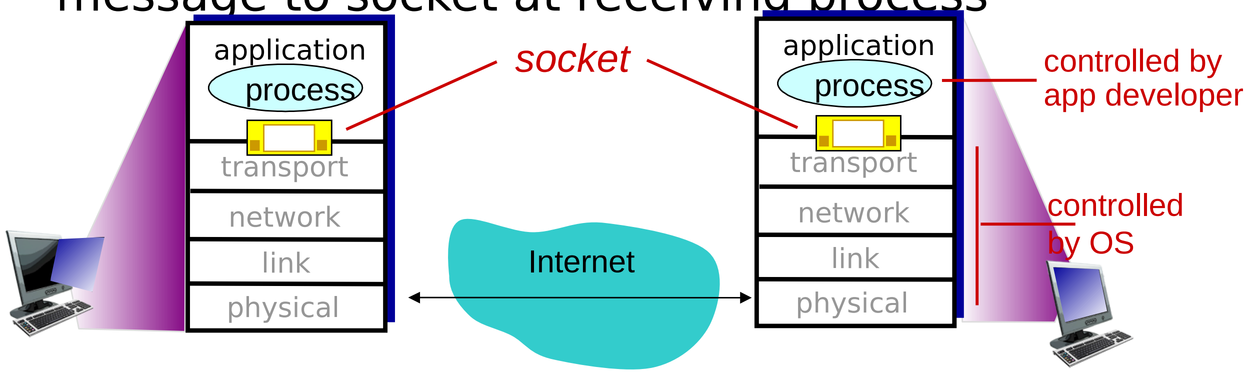
- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol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



Sockets

≡ pipe endpoints

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Transport vs. network layer

IP address

port numbers

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Socket : IPaddress : port number

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

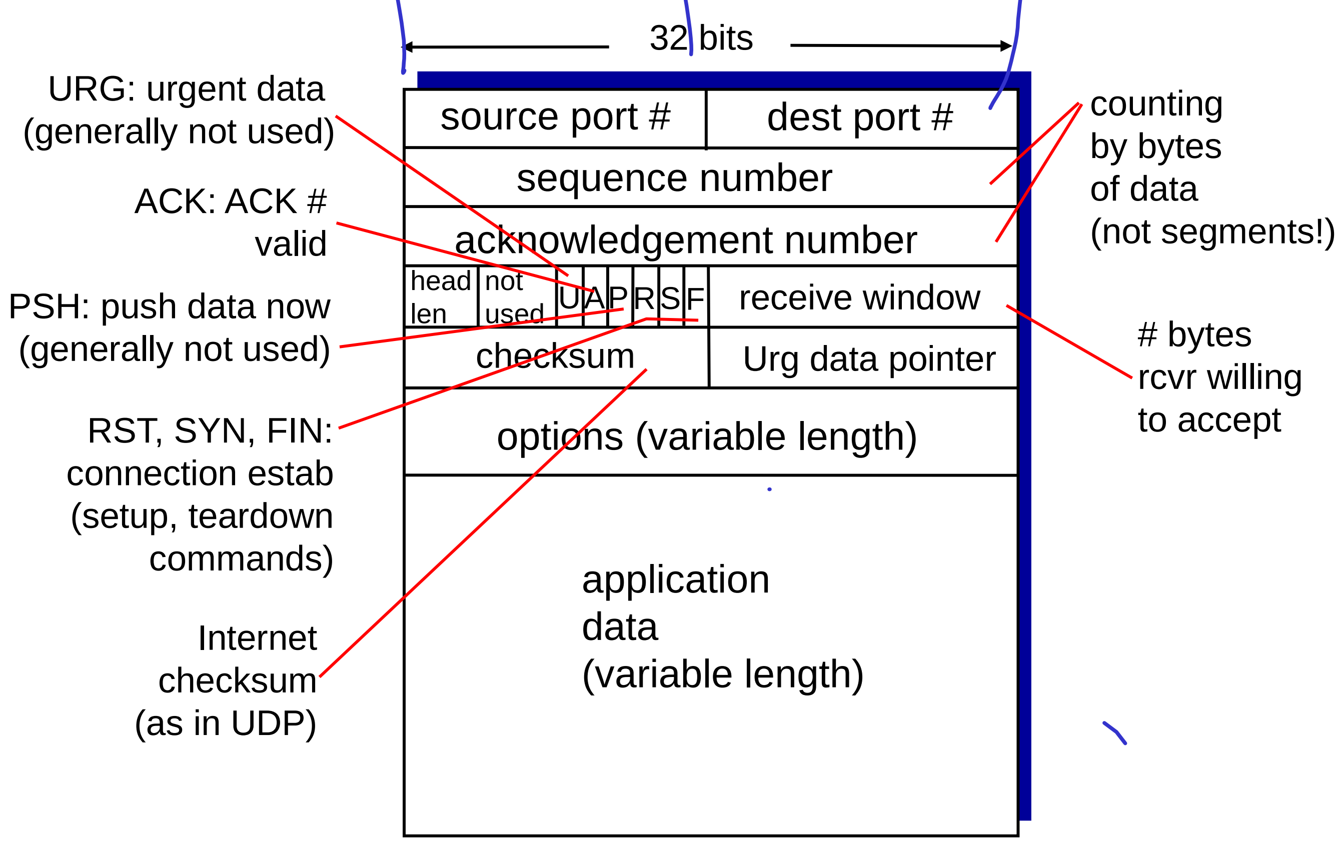
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

TCP: Overview RFCs: 793,1122,1323, 2018, 2581

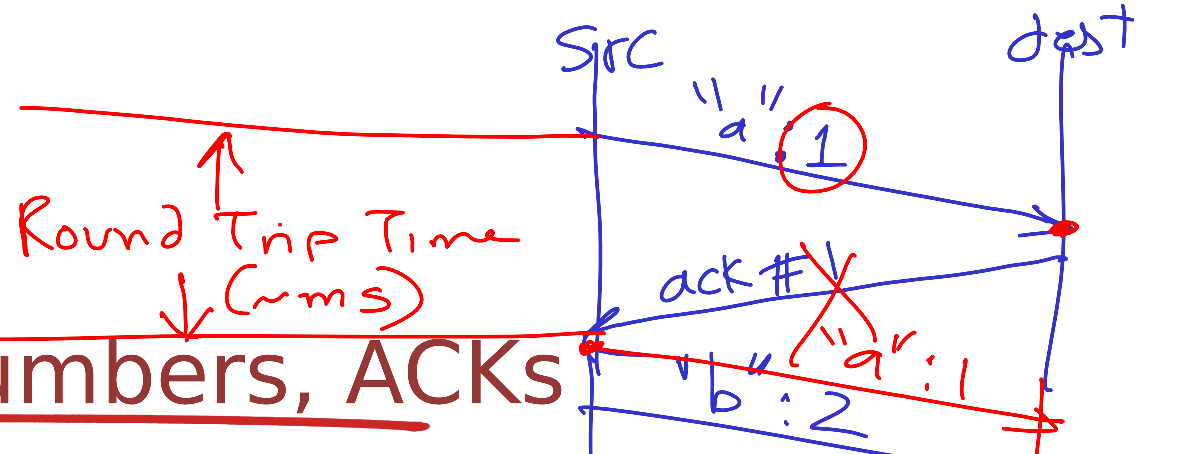
- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no “message boundaries”
- pipelined:
  - TCP congestion and flow control set window size

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

TCP segment structure



TCP seq. numbers, ACKs



sequence numbers:

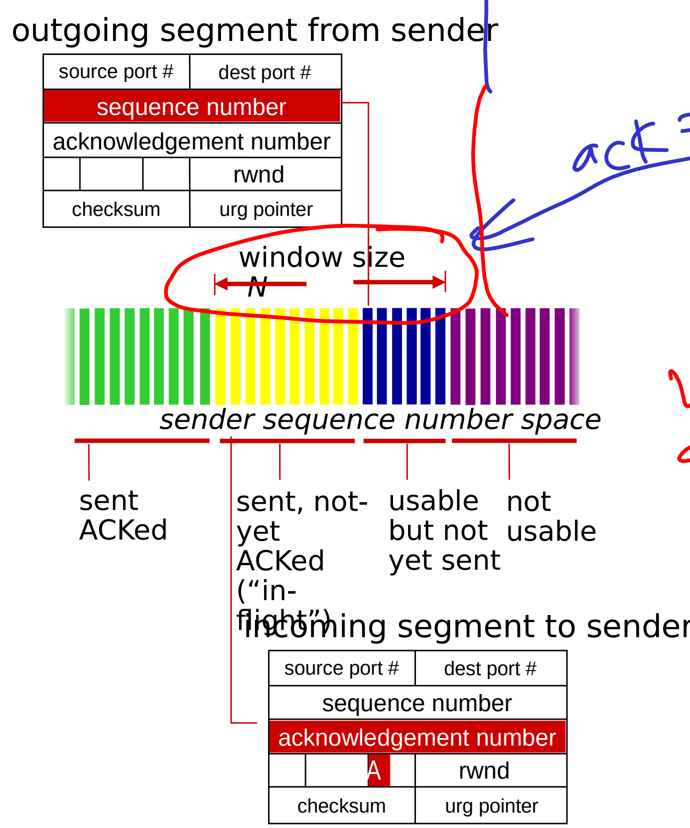
- byte stream "number" of first byte in segment's data

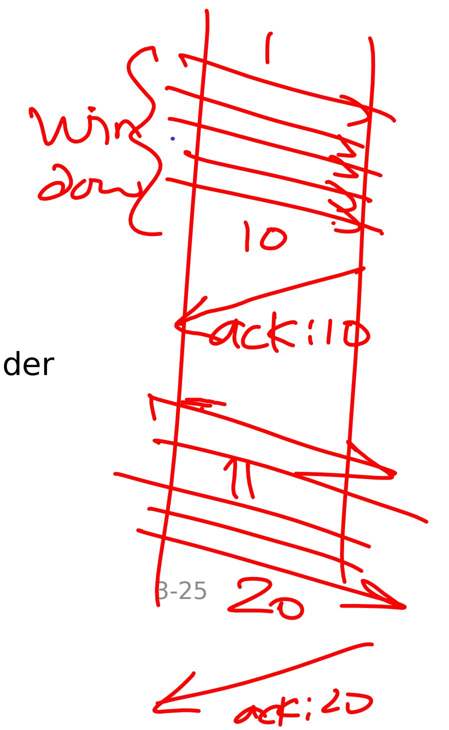
acknowledgements:

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

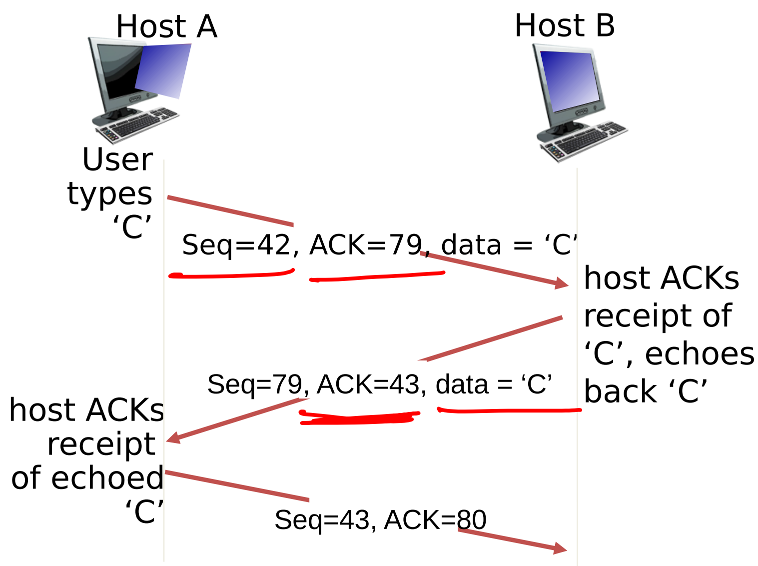
Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor





TCP seq. numbers, ACKs



simple telnet scenario

TCP sender events:

data rcvd from app:

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

timeout:

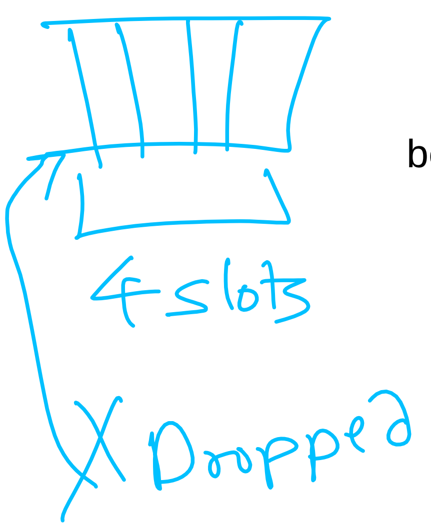
- retransmit segment that caused timeout
- restart timer

ack rcvd:

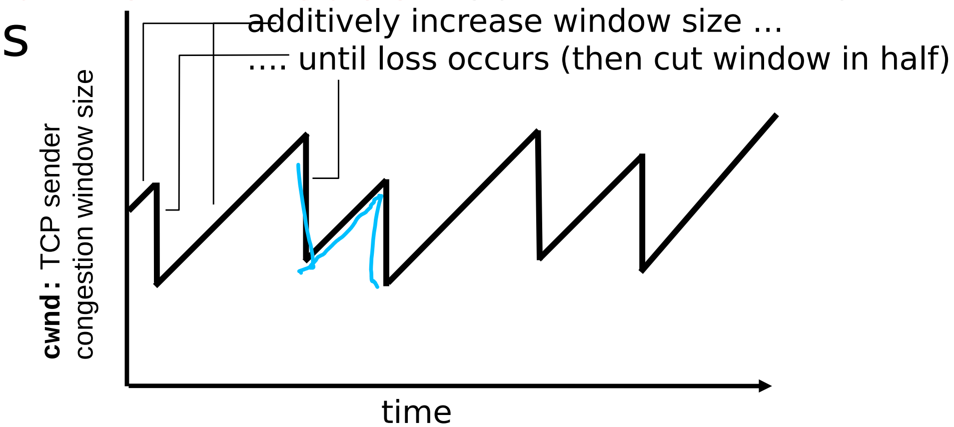
- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

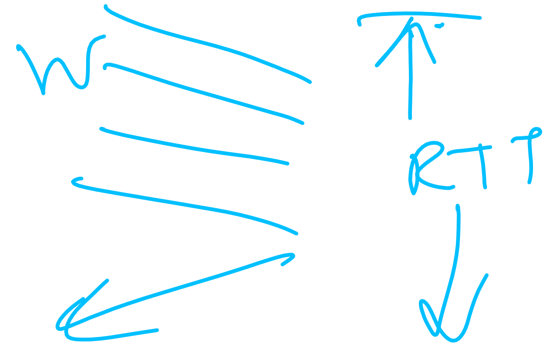
TCP congestion control: additive increase multiplicative decrease

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **cwnd** by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut **cwnd** in half after loss



AIMD saw tooth behavior: probing for bandwidth





TCP Performance

- Ideal: Window-size/Round-Trip-Time
- Throughput = Window-size/RTT*(sqrt(2/3)*packet-loss-probability)
- Performance also depends on receive-buffer sizes

Socket programming *with UDP*

UDP: no “connection” between client & server

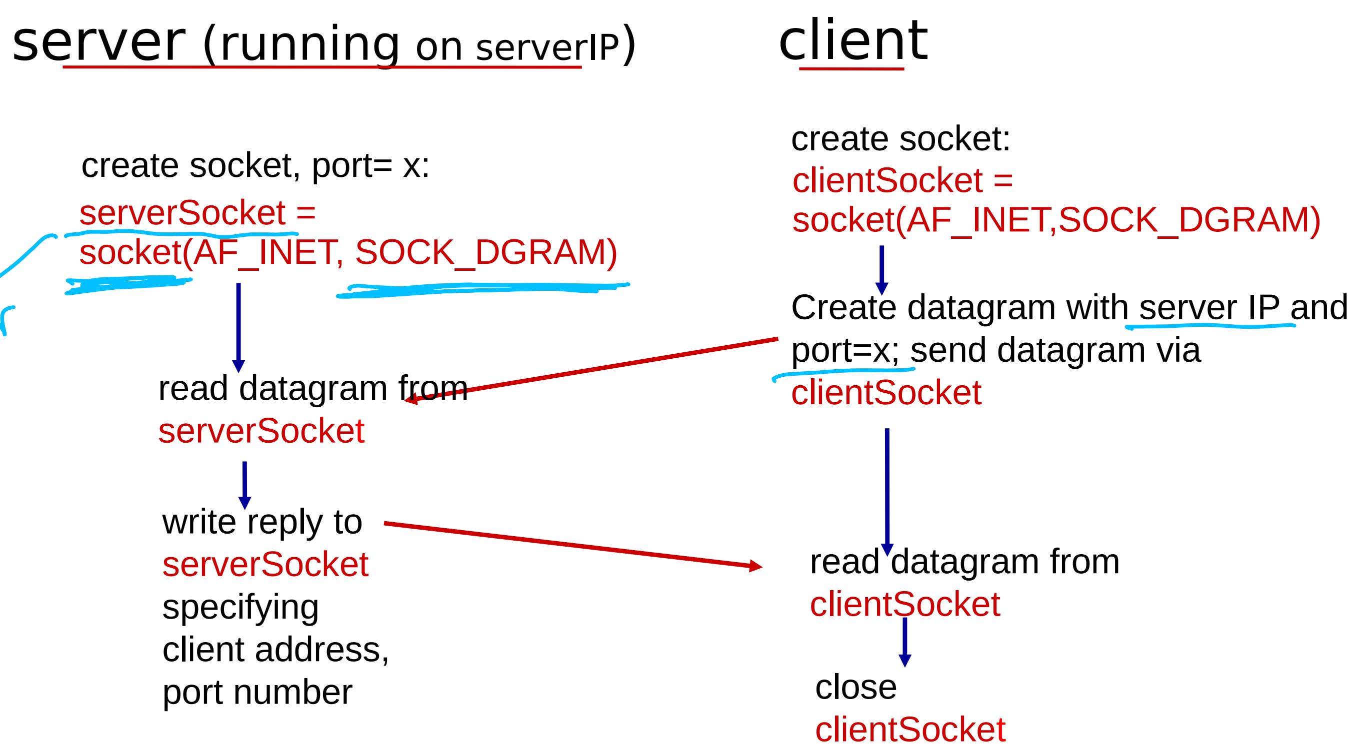
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- rcvr extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes (“datagrams”) between client and server

Client/server socket interaction: UDP



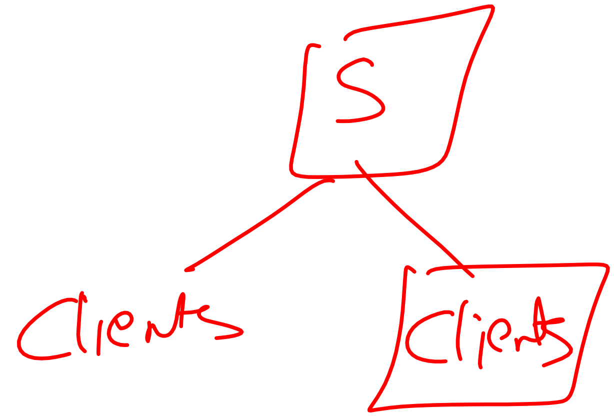
Example app: UDP client

Python UDPClient

include Python's socket library →

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(socket.AF_INET,
                      socket.SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message,(serverName, serverPort))
modifiedMessage, serverAddress =
                      clientSocket.recvfrom(2048)
print modifiedMessage
clientSocket.close()
```

- create UDP socket for server → `clientSocket = socket(...)`
- get user keyboard input → `message = raw_input(...)`
- Attach server name, port to message; send into socket → `clientSocket.sendto(...)`
- read reply characters from socket into string → `modifiedMessage, serverAddress = clientSocket.recvfrom(2048)`
- print out received string and close socket → `print modifiedMessage`, `clientSocket.close()`



Example app: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print "The server is ready to receive"
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, clientAddress)
```

- create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)
- bind socket to local port number 12000 → serverSocket.bind(('', serverPort))
- loop forever → while 1:
- Read from UDP socket into message, getting client's address (client IP and port) → message, clientAddress = serverSocket.recvfrom(2048)
- send upper case string back to this client → serverSocket.sendto(modifiedMessage, clientAddress)

Blocks until a msg is rcvd

Socket programming *with TCP*

client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

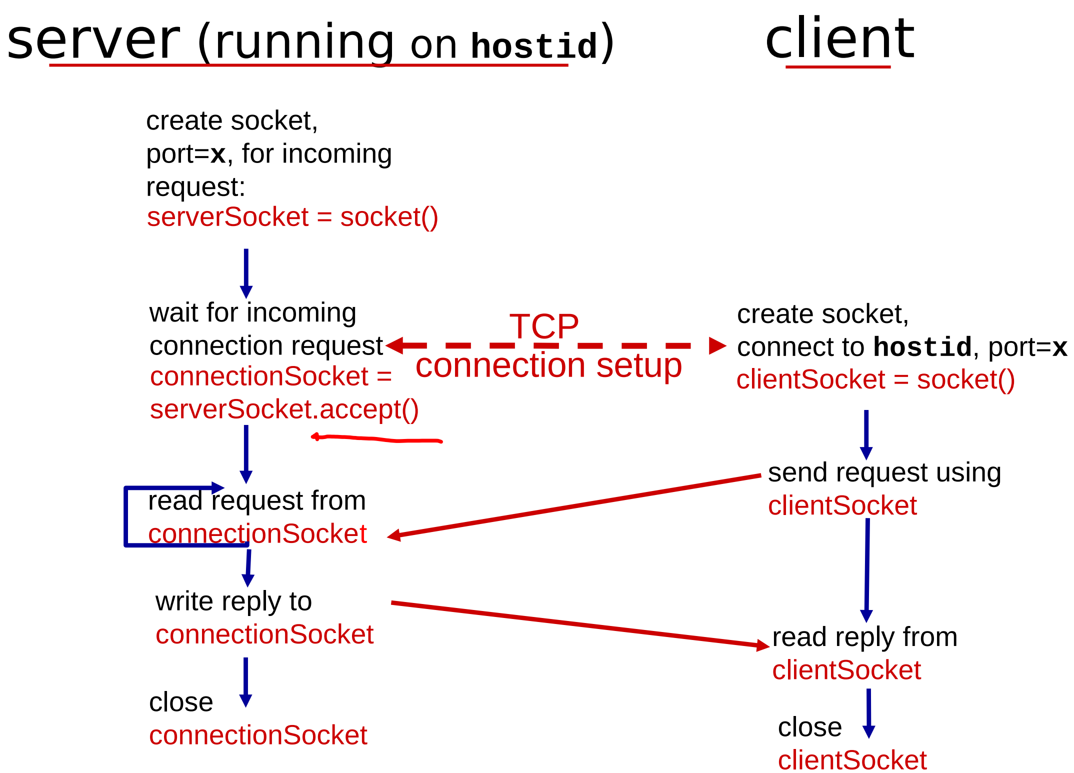
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients

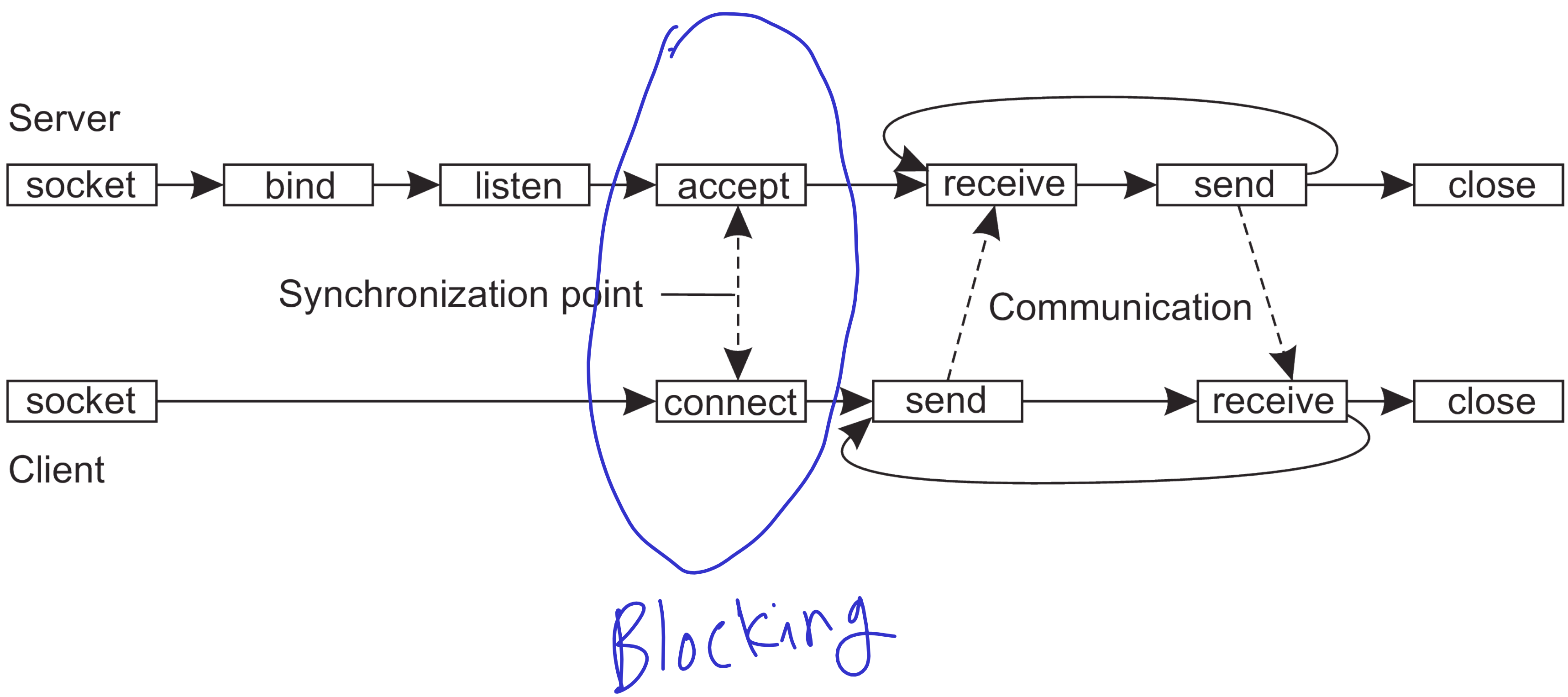
application viewpoint:

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Client/server socket interaction: TCP



Socket Programming With TCP



Socket Example

```
# An example script to connect to Google using socket
# programming in Python
import socket # for socket
import sys

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print "Socket successfully created"
except socket.error as err:
    print "socket creation failed with error %s" %(err)

# default port for socket
port = 80

try:
    host_ip = socket.gethostbyname('www.google.com')
except socket.gaierror:

    # this means could not resolve the host
    print "there was an error resolving the host"
    sys.exit()

# connecting to the server
s.connect((host_ip, port))

print "the socket has successfully connected to google \
on port == %s" %(host_ip)
```

Send HTTP Request

Example app: TCP client

Python TCPClient

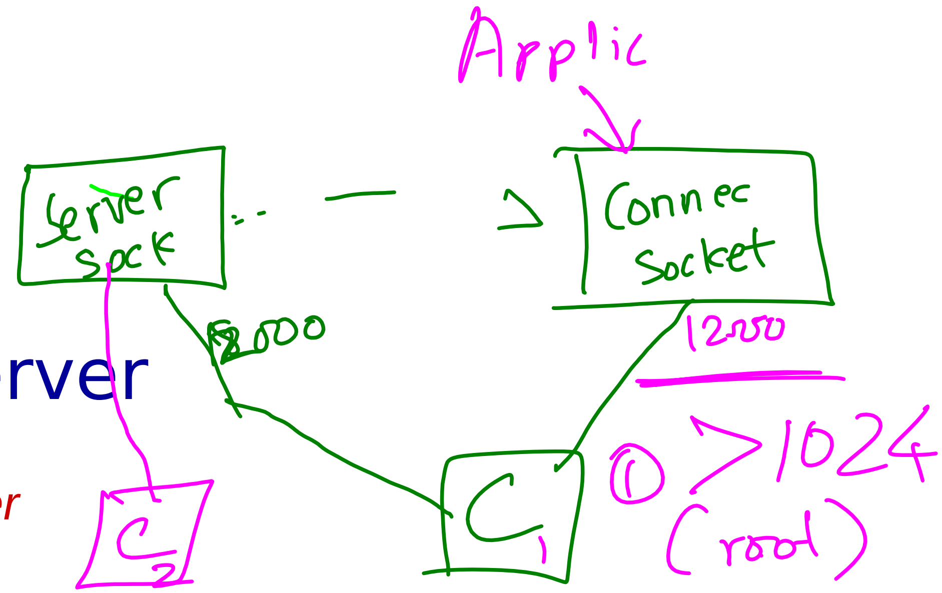
create TCP socket for server, remote port 12000

No need to attach server name, port

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

Blocking

Example app: TCP server



Python TCPServer

create TCP welcoming socket →

server begins listening for incoming TCP requests →

loop forever →

server waits on accept() for incoming requests, new socket created on return →

read bytes from socket (but not address as in UDP) →

close connection to this client (but *not* welcoming socket) →

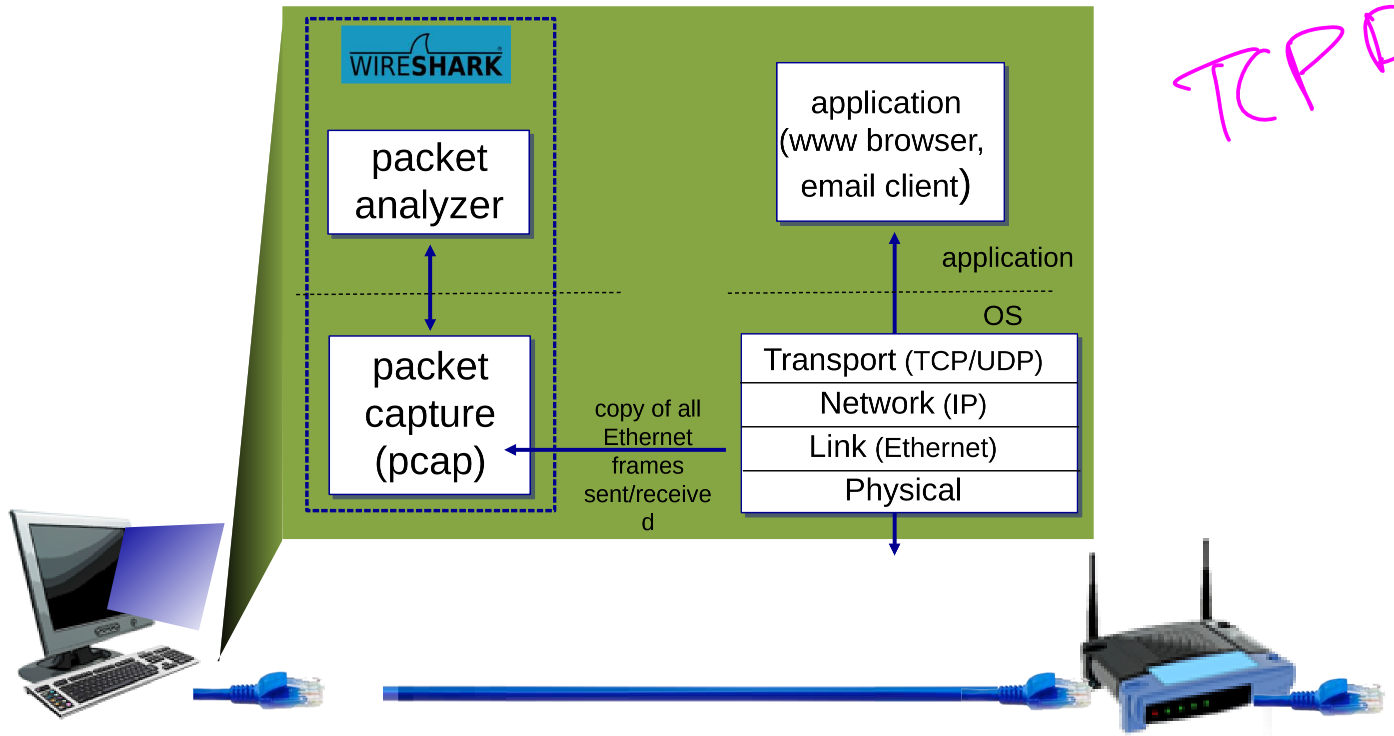
```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
print ‘The server is ready to receive’
while 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

Block

Higher Level Networking

- Client/server code abstracted out (python's twisted framework)
- Message queues: Kafka, ZeroMQ, etc
- Durability of messages (can persist on disk)
- Message lifetimes (time to live)
- Filtering, queueing policies
- Batching policies
- Delivery policies (at most once, at least once, etc)

Debugging Networks: Packet Capture



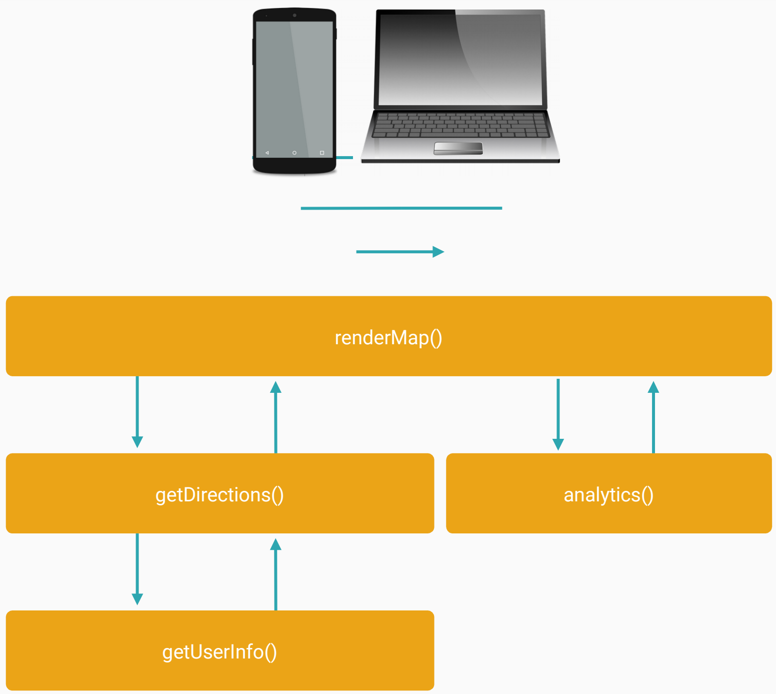
TCPDUMP

Separation of Concerns

- Break problem into separate parts
- Solve each problem independently
- Encapsulate data across layers
- Protocol: Rules for communication within same layer
- Service: Abstraction provided to layer above
- API: Concrete way of using that service
- Layering+Encapsulation Example

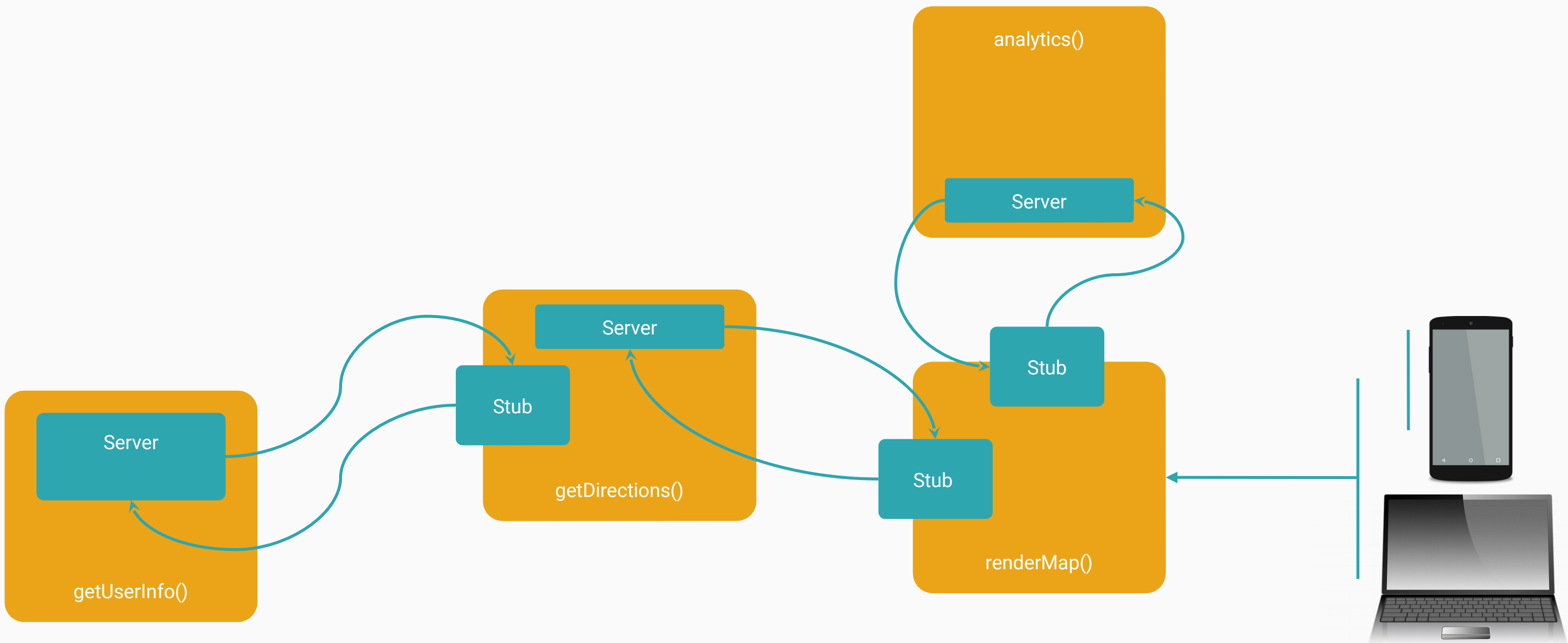
Remote Procedure Calls

Moving to Microservices



https://pixabay.com/en/laptop-computer-open-screen-blank-310098/

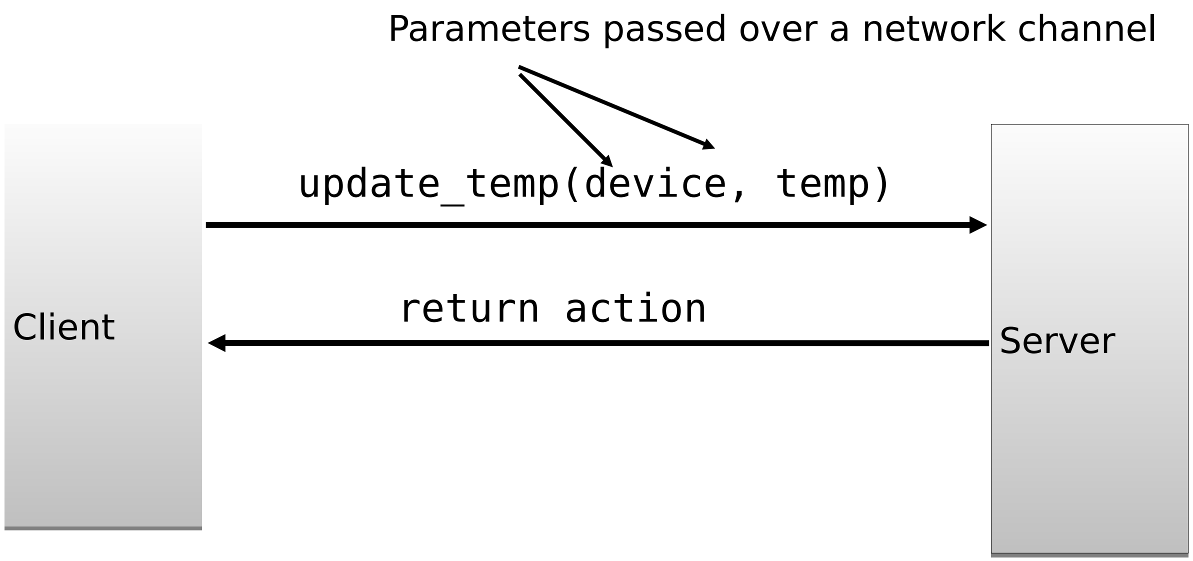
Moving to Microservices



Remote Procedure Calls

- Procedure (function) calls a well known and understood mechanism for transfer of data and control within a program/process
- Remote Procedure Calls : extend conventional local calls to work *across* processes.
  - Processes may be running on different machines
  - Allows communication of data via function parameters and return values
  - RPC invocations also serve as notifications (transfer of control)

RPC Example


Parameters passed over a network channel
update_temp(device, temp)
return action
Client
Server

RPC Advantages

- Clean and simple to understand semantics similar to local procedure calls
- Generality: all languages have local procedure calls
  - RPC libraries augment the procedure call interface to make RPCs appear similar to local calls
- Abstraction for a common client/server communication pattern

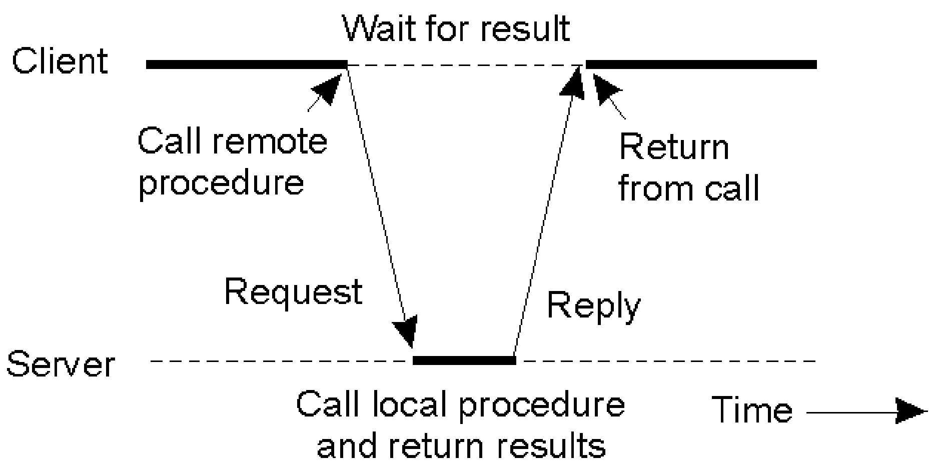
```
push_temp(name) {
    t = get_current_temp();
      return update_temp (name, t);  //RPC
      }
```

Challenges

- RPCs impose new challenges not faced in local calls
- How to pass parameters?
  - Passing data over a network raises issues like endian-ness
  - Pointers: machines may not share an address space
- How to deal with machine failures?
  - Local procedures are assumed to always run
  - A remote machine running an RPC may face crashes, network issues
  - Need to consider failure semantics in RPC implementations
- How to integrate RPCs with existing language runtimes?
  - Seamless local and remote calls
  - Integrate RPCs with language caller/callee interface

RPC Semantics

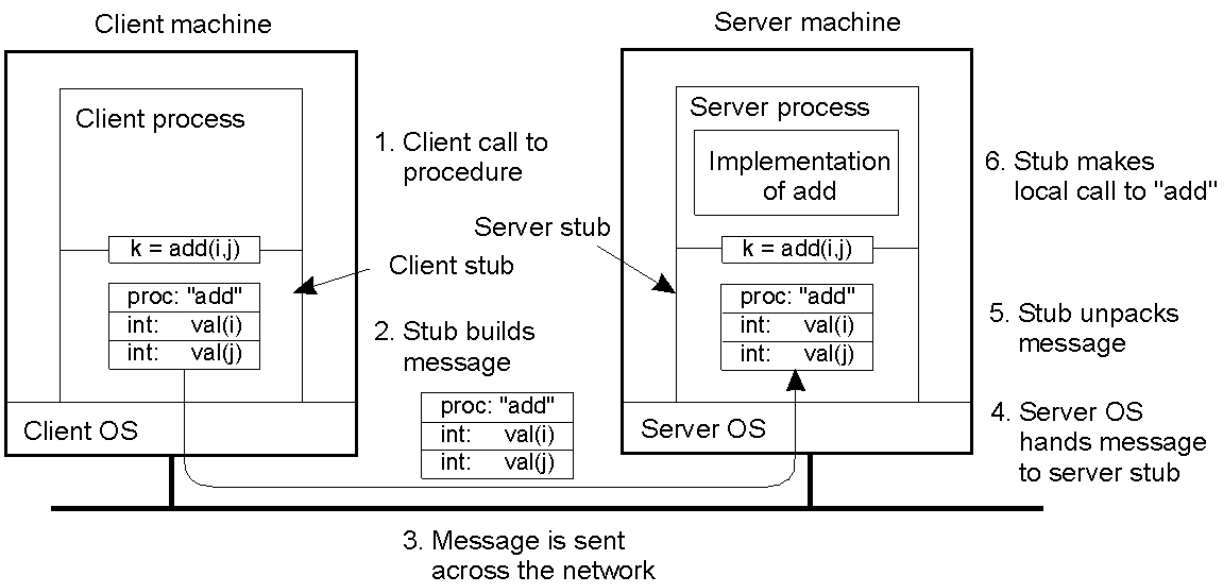
- Usually, RPCs are blocking
  - Thus, also useful for synchronization



How RPCs Work

- Each process has 2 additional components:
  - Code stubs
  - RPC runtime
- Code stubs “translate” local calls remote calls
  - Pack/unpack parameters
- RPC runtime transmits these translated calls over the network
  - Wait for result

How RPCs Work



Parameter Passing

- Local procedure parameter passing
  - Call-by-value
  - Call-by-reference: arrays, complex data structures
- Remote procedure calls simulate this through:
  - Stubs – proxies
  - Flattening – marshalling
  - Serializing local, in-memory representation
- Related issue: global variables are not allowed in RPCs

Client And Server Stubs

- Client makes procedure call (just like a local procedure call) to the client stub
- Server is written as a standard procedure
- Stubs take care of packaging arguments and sending messages
- Packaging parameters is called *marshalling*
- Stub compiler generates stub automatically from specs in an Interface Definition Language (IDL)
  - Simplifies programmer task

Steps of RPC

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Marshalling

- Problem: different machines have different data formats
  - Intel: little endian, SPARC: big endian
- Solution: use a cross-platform, general, standard representation
  - Convert in-memory object representation to a standardized “wire” format
  - Example: external data representation (XDR)
- Problem: how do we pass pointers?
  - If it points to a well-defined data structure, pass a copy and the server stub passes a pointer to the local copy
- What about data structures containing pointers?
  - Prohibit
  - Dereference and send (used by most RPC implementations)
  - Chase pointers over network
- Marshalling: transform parameters/results into a byte stream (serialization of parameters)

Binding

- Problem: how does a client locate a server?
  - How does caller code locate and call the callee
  - Use bindings (similar to how symbols are bound to variables during run-time in local programs)
- Server
  - Export server interface during initialization
  - Send name, version no, unique identifier, handle (address) to binder
- Client
  - First RPC: send message to binder to import server interface
  - Binder: check to see if server has exported interface
    - Return handle and unique identifier to client

Binding Comments

- Binding can be at **run-time**
  - Better handling of partial failures (clients can try other advertised end-points, protocols, etc.)
  - Increased dynamism
- Exporting and importing incurs overheads
- Binder can be a bottleneck
  - Use multiple binders
- Binder can do load balancing

Failure Semantics

- *Client unable to locate server*: return error
- *Lost request messages*: simple timeout mechanisms
- *Lost replies*: timeout mechanisms
  - Make operation idempotent
  - Use sequence numbers, mark retransmissions
- *Server failures:* did failure occur before or after operation?
  - At least once semantics / Idempotent (SUNRPC)
  - At most once
  - No guarantee
  - Exactly once: desirable but difficult to achieve

More Failure Semantics

- *Client failure:* what happens to the server computation?
  - Referred to as an *orphan*
  - *Extermination*: log at client stub and explicitly kill orphans
    - Overhead of maintaining disk logs
  - *Reincarnation*: Divide time into epochs between failures and delete computations from old epochs
  - *Gentle reincarnation*: upon a new epoch broadcast, try to locate owner first (delete only if no owner)
  - *Expiration*: give each RPC a fixed quantum *T*; explicitly request extensions
    - Periodic checks with client during long computations

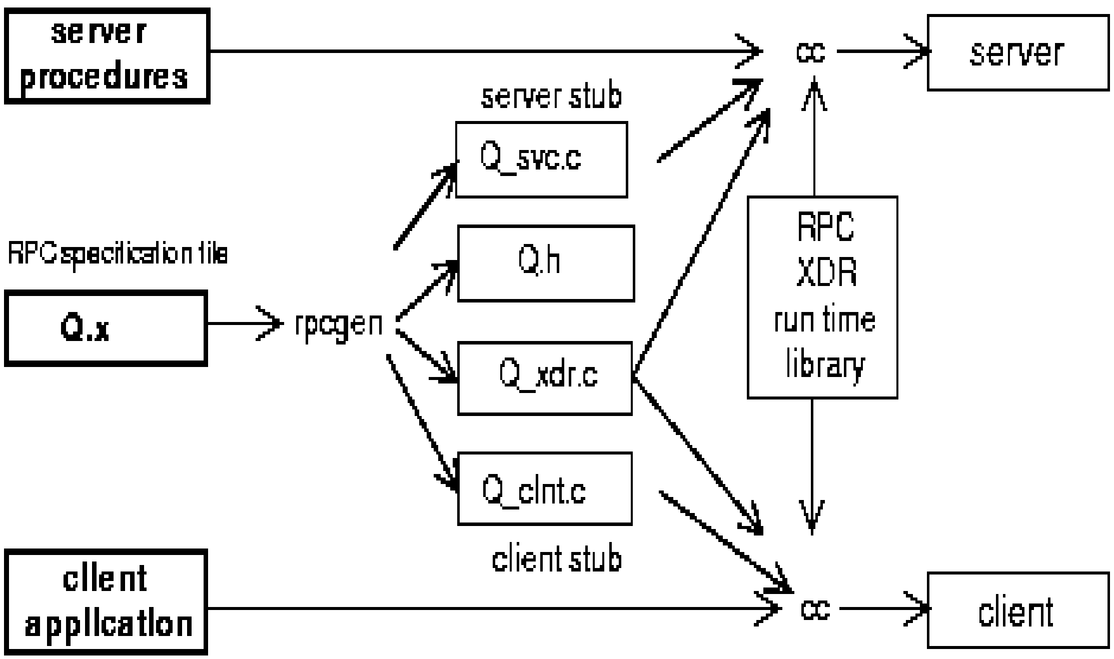
Implementation Issues

- Choice of protocol [affects communication costs]
  - Use existing protocol (UDP) or design from scratch
  - Packet size restrictions
  - Reliability in case of multiple packet messages
  - Flow control
- Copying costs are dominant overheads
  - Need at least 2 copies per message
    - From client to NIC and from server NIC to server
  - As many as 7 copies
    - Stack in stub – message buffer in stub – kernel – NIC – medium – NIC – kernel – stub – server

Sun RPC

- One of the most widely used RPC systems
- Developed for use with NFS (Network File System)
- Built on top of UDP or TCP
  - TCP: stream is divided into records
  - UDP: max packet size < 8912 bytes
  - UDP: timeout plus limited number of retransmissions
  - TCP: return error if connection is terminated by server
- Multiple arguments marshaled into a single structure
- At-least-once semantics if reply received, at-least-zero semantics if no reply. With UDP tries at-most-once
- Use SUN’s eXternal Data Representation (XDR)
  - Big endian order for 32 bit integers, handle arbitrarily large data structures

Sun RPC program structure



Protocol Buffers

IDL (Interface definition language)
Describe once and generate interfaces for any language.

Data Model
Structure of the request and response.

Wire Format
Binary format for network transmission.

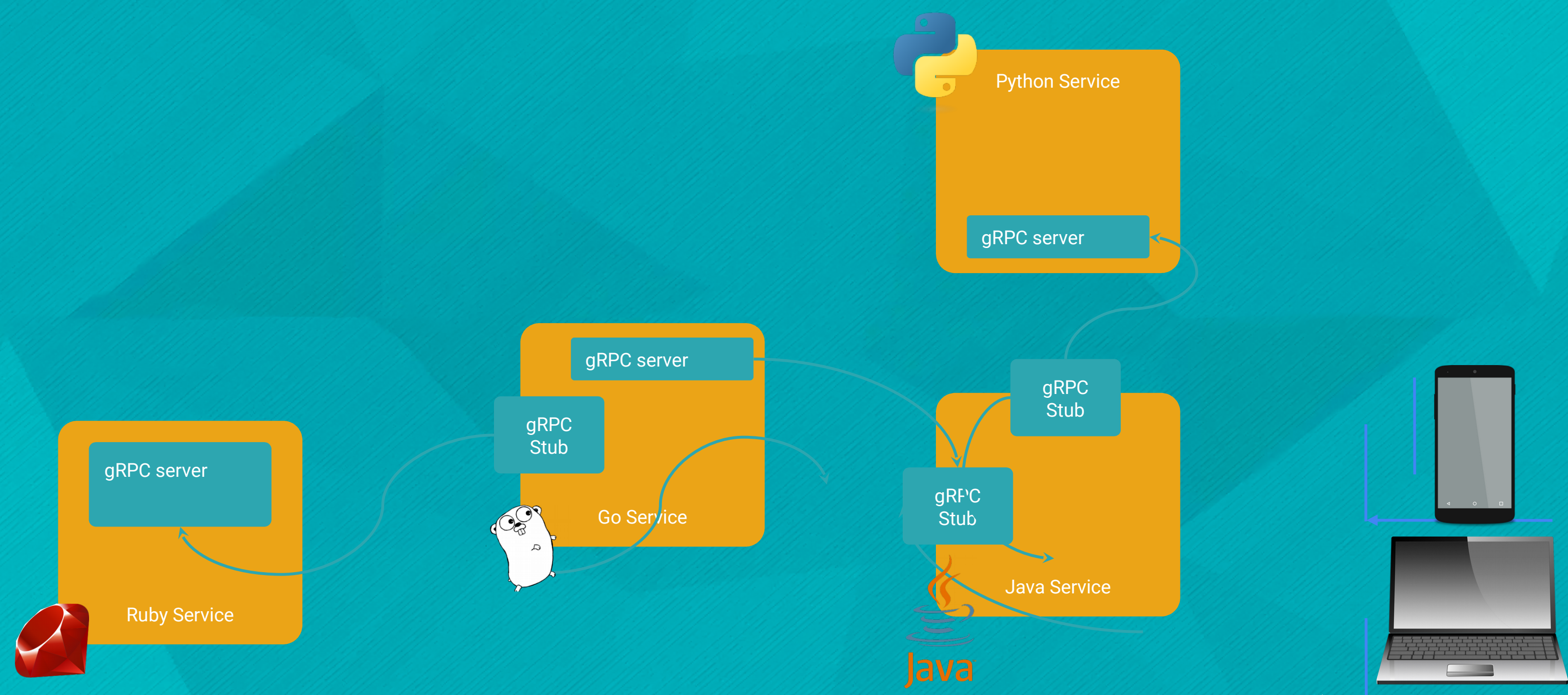
```
syntax = “proto3”;

message Person {

    string name = 1;
    int32 id = 2;
    string email = 3;

    enum PhoneType {
      MOBILE = 0;
      HOME = 1;
      WORK = 2;
    }

    message PhoneNumber {
      string number = 1;
      PhoneType type = 2;
    }

    repeated PhoneNumber phone = 4;
}
```

Service Definitions

```
service RouteGuide {
  rpc GetFeature(Point) returns (Feature);
  rpc RouteChat(stream RouteNote) returns (stream RouteNote);
}
```

```
message Point {
  int32 Latitude = 1;
¨  int32 Longitude = 2;
}

message Feature {
 string name = 1;
 ¨Point location = 2;
}

message RouteNote {
  Point location = 1;
  string message = 2;
}
```

Multiple Language Support



Modern RPCs and Protocol Buffers

- Many distributed systems use RPCs today (like Mesos)
- Common paradigm: serialize function calls in some serialization format (XML, JSON,…) and send over HTTP (xmlrpclib, etc.)
- HTTP servers unpacks and executes the remote call
  - POST http://foo.com/api/function-name {arg1:x, arg2:y}

Protocol Buffers

- Relatively new (2008) serialization format from Google
- Binary format. Faster than JSON/XML
  - Con: Not self documenting

```
message Point {
  int32 x = 1 ; //Field “tags”, since names are not included in the message
  int32 y = 2 ;
  String name = 3 ;
}
Repeated Points point = 4 ; //List/array
```

- Getters and setter methods created for each message during compilation (protoc)
- Access via msg.fieldname() (for example, point.x())
- Multiple languages supported

gRPC: A Modern RPC Framework

- “Service” : Function declaration
  - Unary: Single response for a request
  - Streaming: Multiple streaming requests result in single response
- Uses HTTP/2 as transport
  - Messages are just POST requests. Request name is URI, params is content
  - Can multiplex multiple requests onto single TCP connection
- At-most-once failure semantics, but other schemes using retries possible
- Can use load balancers
- GRPC Python: https://www.semantics3.com/blog/a-simplified-guide-to-grpc-in-python-6c4e25f0c506/