CAN 201 Computer Network

Intro to network

Basic Concepts

- Hosts: terminal systems
- Packet switches: forward packages, chunks of data
  - routers, switches
- Communication links:
  - fiber, radio, satellite
  - transmission rate: bandwidth
- Networks: collections of devices, routers, links
  - managed by organization

Nut and bolts view

- Hosts: running network apps; server and client
- communication links
- packet switches: routers and link-layer switches
- Internet: network of networks
  - Interconnected ISPs (Internet service provider)
- Protocols: control sending receiving of messages: format
  - format, order of messages sent and received among network entities
  - actions taken on message transmission, receipt
- Standards:
  - RFC: Request for comments
  - IETF: Internet Engineering Task Force

Hosts

1. Takes application message
2. breaks into smaller chunks, packets, with length L bits
3. Transmits packet into access network at transmission rate R
   - link transmission rate, aka., link capacity, aka., link bandwidth.

$\text{packet transmission delay} = \text{time needed to transmit L-bit packet into link} = \frac{L}{R}$

Physical media

Wired, wireless communication links.

- Guided media: signals propagate in solid media: copper, fiber, coaxial cable, glass
- Unguided media: signals propagate freely, e.g., radio

Physical:

Twisted pair: TP, two insulated copper wires: 双绞线

- category 5:100 Mbps, 1Gbps Ethernet
- Category 6:10Gbps

Coaxial cable: 同轴线

- two concentric copper conductors
- bidirectional

- broadband: multiple channels on cable; HFC

Fiber optic cable: glass fiber carrying light pulses, 1 pulse = 1 bit. 光纤
- high-speed operation: p2p transmission: 10-100Gbps
- Low error rate: repeaters spaced far apart; not affected by electromagnetic noise

Radio:

- Signal carried in electromagnetic spectrum
- No physical wire
- Bidirectional 双工
- Reflection; Obstruction by objects; interference 反射, 被物件阻碍, 互相干涉

Types:
- WLAN: Wireless LAN: 54 Mbps — 9.6Gbps
- Wide-area: Cellular: 4G: ~100Mbps; 5G: ~1Gbps
- Satellite: Kpbs ~ 45Mpbs; starlink: ~ 1440Mbps; At least 270ms e2e delay

PSTN: Dial-up Internet access
拨号网络: use public switched telephone network (PSTN)
- dialing a telephone number over conventional telephone line.
- Modems: Modulator and demodulator:
  - Decode audio signals into data
  - encode digital signals back into audio
- 56Kpbs ~ 10min for a music

DSL: Digital subscriber line
Telephone line base:
- Have a splitter
- central office DSL Access Multiplexer (DSLAM)
- Data over DSL phone line goes to Internet
- Voice over DSL phone line goes to telephone net

```
Phone -------------------.                         .⟶ Telephone network
                          \                       /
Computer ⟶ DSL modem ⟶ splitter ⟶ DSLAM ⟶ ` ⟶ ISP
```

- Upstream ~ < 3Mbps
- Downstream ~ < 50Mbps
- ADSL: Asymmetric Digital Subscriber Line: faster

FTTH: fiber to the home
Pptical network terminator → Optical splitter → Optical line terminator → Internet

Passive Optical Networks (PONs) distribution architecture.

Wireless access networks
Base stations used to provide the signal.

- Wireless LANs:
  - 802.11b/g/n: 11,54, 450 Mbps
  - 802.11ax: 9.6 Gbps (MIMO)
- Wide-area wireless access
  - provided by ISP

- >= 1Gbps

Enterprise networks
Companies, Universities, … Mix of wired, wireless link, connecting a mix of switches and routers

Data center networks
high bandwidth links connected servers

Core
Mesh of interconnected routers

Packet-switching: hosts break application layer msg into packets. Network forwards packet from on router to next, across links on pat from source to destination.

- Forwarding: switching: local action: move arriving packets from router’s input link to appropriate router output link ← local forwarding table

24
- Routing: global action: determine source destination path taken by packet

Packet switching: store-and-forward:

```
+--------+
| source | ⟶ R bps ⟶ Router → R bps → Destination
+--------+                ^
   ^                  +-------+
 +-++++               | Store |
 |L|||| Packets       +-------+
 +-++++ (L bits/ pkg)
```

- $\frac{L}{R}$ s to transmit (push out) L-bit packaet into link at R bps.
- Store and forward: entire packet must arrive at router before it can be transmitted on next link
- E2E delay = 2L/R (assuming no propagation delay) 无传播延时

Switching: queueing delay, loss
Circuit switching:
- dedicated resources: no sharing: circuit like (guaranteed) performance
- Circuit segment idle if not used by call
- commonly used in traditional telephone networks

Frequency Division Multiplexing: FDM 频分复用(FDM，Frequency Division Multiplexing)就是将用于传输信道的总带宽划分成若干个子频带（或称子信道），每一个子信道传输 1 路信号。频分复用要求总频率宽度大于各个子信道频率之和，同时为了保证各子信道中所传输的信号互不干扰，应在各子信道之间设立隔离带，这样就保证了各路信号互不干扰（条件之一）。频分复用技术的特点是所有子信道传输的信号以并行的方式工作，每一路信号传输时可不考虑传输时延，因而频分复用技术取得了非常广泛的应用。频分复用技术除传统意义上的频分复用(FDM)外，还有一种是正交频分复用(OFDM)。

Time Division Multiplexing: 时分复用技术（time-division multiplexing, TDM）是将不同的信号相互交织在不同的时间段内，沿着同一个信道传输；在接收端再用某种方法，将各个时间段内的信号提取出来还原成原始信号的通信技术。这种技术可以在同一个信道上传输多路信号 [1]。

Packet Switching: Queuing and loss: 组交换采用存储转发技术。实质上是采用了在数据通信的过程中断续（或动态）分配传输带宽的策略（断续分配传输带宽）。

将报文分成更小的等长的数据段，每一个数据段加上一些控制信息（诸如目的地址和原地址等）后组成首部（header），构成一个分组（packet）。分组又称为“包”，分组的首部也可称为“包头”。

```
  +----+
1 |    | → R = 100 Mb/s
  +----+    \      +-----------+
             -   → | switcher | → R=1.5Mb/s → next link
  +----+    /      +-----------+
2 |    | →            +-++++
  +----+              | |||| Queuing
                      +-++++
```

If arrival rate (in bits) to link exceeds transmission rate of link for a period of time:
- packets will queue, wait to be transmitted to link
- packets dropped if memory fills up

PS advantages:
- resource sharing
- simpler, no call setup

PS drawbacks:
- excessive congestion possible: delay and loss
- protocols needed for reliable data transfer, congestion control

How to provide circuit-like behavior PS?
- Bandwidth guarantees

Internet structure

Hosts connect to Internet via access Internet Service Providers (ISPs)
- Access ISPs in turn must be interconnected
- So that any two hosts (anywhere!) can send packets to each other
- Resulting network of networks is very complex
- Evolution driven by economics, national policies

网络由若干结点(node)和连接这些结点的链路(link)组成。 互联网是“网络的网络”。 习惯上，与网络相连的计算机常称为主机。 网络把许多计算机接连在一起。互联网则把许多网络通过路由器连接在一起。

因特网结构：因特网结构用一句话概括就是网络的网络：端系统通过接入 ISP（access ISPs ）连接到 Internet，接入 ISP 必须进一步互连，保证任意两个主机才可以互相发送分组。

ISP 互联: 对等链路, IXP 目前的互联网是一种多层次 ISP 结构，ISP 根据覆盖面积的大小分为第一层 ISP、区域 ISP 和接入 ISP。互联网交换点 IXP 允许两个 ISP 直接相连而不用经过第三个 ISP。

```
            .---------------------------.
           /                             \
 +------------+   +------------+   +------------------+
 | Tier 1 ISP |---| Tier 1 ISP |---| Content provider |
 +------------+   +------------+   +------------------+
  | /  \            /          \      /
  |/   +-----+     /            +-----+
  /\   | IPX |    /            /| IPX |
 /  \  +-----+   /            / +-----+
/    -/     \   /            /    |
    /     /|     +-------------+ /     |
   /     /  \    | Reginal ISP |/      |
  /     /    \   +-------------+       |
 /     /      \    |                   |
/     /        \   |                   |
+------------+  +------------+ +------------+
| Access ISP |  | Access ISP | | Access ISP |
+------------+  +------------+ +------------+
```

At “center”: small # of well-connected large networks

- “tier-1” commercial ISPs (e.g., Level 3, Sprint, AT&T, NTT), national & international coverage
- content provider networks (e.g., Google, Facebook): private network that connects its data centers to Internet, often bypassing tier-1, regional ISPs

Performance

速率：数据传输速率或称比特率。往往指额定速率或标称速率 带宽

- “带宽”原本指信号具有的频带宽度，即最高频率与最低频率之差。
- 网络中的“带宽”通常是数字信道所能传送的“最高数据率”

时延

1. 分组交换为什么会产生丢包和时延呢？

   Packages queue in router buffers
   - packet arrival rate to link (temporarily) exceeds output link

   capacity
   - then, packets queue, wait for turn

2. 四种分组延迟
   - $d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$,
     - 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
     - d_{trans}: transmission delay:
       - L: packet length (bits)
       - R: link bandwidth (bps)
       - $d_{\text{trans}} = \frac{L}{R}$
     - d_{prop}: propagation delay: 传播
       - d: length of physical link
       - s: propagation speed (~2.9x108 m/sec)
       - $d_{\text{prop}} = \frac{d}{s}$
   - 发送时延(传输延迟) d=L/R。L 为分组长度，R 为链路带宽
   - 传播时延(电磁波在信道中传输一定距离所需划分的时间)=信道长度（m）/信号的传播速度（m/s）
   - 处理时延：差错检测，确定输出链路，通常小于 msec
   - 排队时延：等待输出链路可用，取决于路由器拥塞程度
   - 注意：对于高速网络链路，提高的是发送速率而不是传播速率。

3. 排队时延
   - 流量强度=La/R
     - R:链路带宽
     - L:分组长度
     - a:平均分组到达速率
   - La/R 0:平均排队延迟很小
   - La/R 趋近于 1:平均排队延迟很大
   - La/R>1:超出服务能力，平均排队延迟无限大!

Packet loss:
- Queue (aka buffer) preceding link in buffer has finite capacity
- Packet arriving to full queue dropped (aka lost)
- Lost packet may be retransmitted by previous node, by source end system, or not at all

Throughput: 吞吐量
- 即时吞吐量：给定时刻的速率；
- 平均吞吐量: 一段时间的平均速率。
- bottleneck link: link on end-end path that constrains e2e Throughput.

Per-connection end-end throughput: min{R_{client} , R_{server} , R_{total}/connected devices count}
- In practice: R_{client} or R_{server} is often bottleneck

Internet protocol stack
- 体系结构
  - OSI 7 层体系结构：物理层、数据链路层、网络层、运输层、会话层、表示层、应用层 ISO/OSI = ISO/Open System Interconnection • presentation: allow applications to interpret meaning of data, e.g., encryption, compression, machine-specific conventions • session: synchronization, checkpointing, recovery of data exchange • Internet stack “missing” these layers! • these services, if needed, must be implemented in application
    - 物理层：传输 数据比特流
    - 数据链路层：传输 数据帧: Ethernet(以太网)，Wifi,PPP(电话线上网)
    - 网络层：传输 数据包: IP 协议，路由协议
    - 运输层：传输 数据报文: TCP(面向连接的),UDP(无连接的)
    - 会话层：传输 数据报文
    - 表示层：传输 数据报文
    - 应用层：传输 数据报文: 文件传输（FTP），电子邮件（SMTP），Web（HTTP）
  - TCP/IP 5 层体系结构：物理层、数据链路层、网络层、运输层、应用层
    - application: supporting network applications: FTP, SMTP, HTTP, DNS
    - 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, 802.11 (WiFi), PPP
    - physical: bits “on the wire”

```
Host     +-------------+ Source
         | application |   |  message           | M |
         +-------------+   |
         |  transport  |   |  segment        |H_t| M |
         +-------------+   |
         |   network   |   |  datagram   |H_n|H_t| M |
         +-------------+   |
         |    link     |   |  frame  |H_I|H_n|H_t| M |
         +-------------+   |
         |  physical   |   |
         +-------------+   v

Switch   +-------------+   |
         |    link     |   |
         +-------------+   |
         |  physical   |   |
         +-------------+   v

            link                         |H_n|H_t| M |

Router   +-------------+   |
         |   network   |   |             |H_n|H_t| M |
         +-------------+   |
         |    link     |   |         |H_I|H_n|H_t| M |
         +-------------+   |
         |  physical   |   |
         +-------------+   v

Host     +-------------+ Destination
         |  physical   |   |
         +-------------+   |
         |    link     |   |  frame  |H_I|H_n|H_t| M |
         +-------------+   |
         |   network   |   |  datagram   |H_n|H_t| M |
         +-------------+   |
         |  transport  |   |  segment        |H_t| M |
         +-------------+   |
         | application |   |  message            | M |
         +-------------+   v
```

- Divide complex systems to simple components
- Easy for maintenance
- Flexible for updating

Application Layer

Creating a network app Write programs that:
- run on (different) end systems
- communicate over network
- e.g., web server software communicates with browser software

No need to write software for network-core devices
- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation

Host(Program), Sockets, Network

Processes communicating: Process: program running within a host
- within same host, two processes communicate using inter-process communication (IPC, defined by OS)
- processes in different hosts communicate by exchanging messages

Sockets
- Process sends/receives messages to/from its socket

- Have 2 socket for 2-process RPC, each for pulling and pushing, having their own buffer compare to pipe, share one buffer and half duplex, sockets does full duplex.

To receive messages, process must have identifier
- Host device has unique 32-bit IPv4 and/or 128-bit IPv6
- Process network identifier:
- IPv4:port 192.168.1.100:80
- [IPv6]:port [240e:3a1:4cb1:69d0:f40c:4269:74a2:7ea3]:80

Client-Server:
- server:
  - Always-on host
  - Permanent IP address
  - Often in data centers, for scaling
- Clients
  - Contact, communicate with server
  - May be intermittently connect to the internet
  - May have dynamic IP address
  - Do not communicate directly with each other

Peer-to-Peer:
- No always-on server is needed
- End systems directly exchange data
- Client process / server process on the same host
- Peers request service from other peers, provide service in return to other peers
  - Self scalability – new peers bring new service capacity, as well as new service demands
- Peers are intermittently connected
- Maybe dynamic IP addresses -> complex managemen

Application Layer protocol:
- Paradigm : CS or/and P2P
- 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
- Message timing: when and how
- Open protocols: Defined in RFCs; e.g., HTTP, SMTP, FTP
- Private protocols: e.g., Skype, Games, you own protocols…

Transport service requirements: common apps

Applications	Data loss	Throughput	Time sensitive
File transfer	No loss	Elastic	No
E-mail	No loss	Elastic	No
Web documents	No loss	Elastic	No
Real-time video/audio	Loss-tolerant	Based on quality*	Yes 100 ms
Streaming video/audio	Loss-tolerant	Based on quality*	Yes few second
Interactive games	Loss-tolerant	Few kpbs	Yes 100 ms
Text messaging	No loss	Elastic	Yes or No

*Audio: 5k to 1Mpbs, Video: 10kpbs - 10 Mbps or more

Internet transport protocols services

TCP service:

- Reliable transport between sending and receiving process
- Flow control: sender won't overwhelm receiver
- Congestion control: throttle sender when network overloaded
- Does not offer: timing, minimum throughput guarantee, security
- Connection-oriented: setup required between client and server processes

UDP service:

- Unreliable data transfer between sending and receiving process
- Does not offer: reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup,
- Simple and fast

Application	App layer protocol	Underlying transport Protocol
E-mail	SMTP [RFC 2821], POP3, IMAP	TCP
Remote terminal Access	Telnet [RFC 854], SSH	TCP
Web	HTTP [RFC 2616] HTTPS	TCP
File Transfer	FTP [RFC 959], SFTP	TCP
Multimedia	HTTP / RTP [RFC 1889]	TCP or UDP
VoIP	SIP, RTP or proprietary	TCP or UDP

Securing TCP - Secure Sockets Layer - SSL TCP & UDP

- No encryption
- Cleartext psws -> Internet

SSL

- Provides encrypted TCP connection
- Data integrity
- End-point authentication

SSL is at app layer

- Apps use SSL libraries, that "talk" to TCP

SSL socket API

- Cleartext psw -> encrypted psw -> Internet

For TCP:

```
 0                              15 16                              31
|        source port (16)        |      destination port (16)      | \
|                       Sequence number (32)                       |  |
|                     Acknowledge number (32)                      |  |
| head |   reserve   |U|A|P|R|S|F|                                 |  20 Bytes
|      |             |R|C|S|S|Y|I|        Window size (16)         |  |
| (4)  |     (6)     |G|K|H|T|N|N|                                 |  |
|         check sum (16)         |           pointer (16)          | /
~                                                                  ~
|                                                                  |
|                              options                             |
```

TCP 端口号: TCP 的连接是需要四个要素确定唯一一个连接:（源 IP，源端口号）+ （目地 IP，目的端口号） 所以 TCP 首部预留了两个 16 位作为端口号的存储，而 IP 地址由上一层 IP 协议负责传递 源端口号和目地端口各占 16 位两个字节，也就是端口的范围是 2^16=65535 另外 1024 以下是系统保留的，从 1024-65535 是用户使用的端口范围

TCP 的序号和确认号： 32 位序号 seq：Sequence number 缩写 seq ，TCP 通信过程中某一个传输方向上的字节流的每个字节的序号，通过这个来确认发送的数据有序，比如现在序列号为 1000，发送了 1000，下一个序列号就是 2000。 32 位确认号 ack：Acknowledge number 缩写 ack，TCP 对上一次 seq 序号做出的确认号，用来响应 TCP 报文段，给收到的 TCP 报文段的序号 seq 加 1。

TCP 的标志位 每个 TCP 段都有一个目的，这是借助于 TCP 标志位选项来确定的，允许发送方或接收方指定哪些标志应该被使用，以便段被另一端正确处理。 用的最广泛的标志是 SYN，ACK 和 FIN，用于建立连接，确认成功的段传输，最后终止连接。

- SYN：简写为 S，同步标志位，用于建立会话连接，同步序列号；
- ACK： 简写为.，确认标志位，对已接收的数据包进行确认；
- FIN： 简写为 F，完成标志位，表示我已经没有数据要发送了，即将关闭连接；
- PSH：简写为 P，推送标志位，表示该数据包被对方接收后应立即交给上层应用，而不在缓冲区排队；
- RST：简写为 R，重置标志位，用于连接复位、拒绝错误和非法的数据包；
- URG：简写为 U，紧急标志位，表示数据包的紧急指针域有效，用来保证连接不被阻断，并督促中间设备尽快处理；

```
              Client                               Server
                |         SYN=1, seq=J               |
    SYN_SENT    | Connect Request --------------->   |
                |                                    |
                |  SYN=1, ACK=1 ack=J+1, seq=K       |
                | <--------------------- Response    | SYN_RCVD
                |                                    |
                |         ACK=1, ack=K+1             |
 ESTABLISHED    | Confirm ----------------------->   | ESTABLISHED
                |                                    |
```

第一次握手： 客户端将 TCP 报文标志位 SYN 置为 1，随机产生一个序号值 seq=J，保存在 TCP 首部的序列号(Sequence Number)字段里，指明客户端打算连接的服务器的端口，并将该数据包发送给服务器端，发送完毕后，客户端进入 SYN_SENT 状态，等待服务器端确认。

第二次握手： 服务器端收到数据包后由标志位 SYN=1 知道客户端请求建立连接，服务器端将 TCP 报文标志位 SYN 和 ACK 都置为 1，ack=J+1，随机产生一个序号值 seq=K，并将该数据包发送给客户端以确认连接请求，服务器端进入 SYN_RCVD 状态。

第三次握手： 客户端收到确认后，检查 ack 是否为 J+1，ACK 是否为 1，如果正确则将标志位 ACK 置为 1，ack=K+1，并将该数据包发送给服务器端，服务器端检查 ack 是否为 K+1，ACK 是否为 1，如果正确则连接建立成功，客户端和服务器端进入 ESTABLISHED 状态，完成三次握手，随后客户端与服务器端之间可以开始传输数据了。 注意:我们上面写的 ack 和 ACK，不是同一个概念:

小写的 ack 代表的是头部的确认号 Acknowledge number， 缩写 ack，是对上一个包的序号进行确认的号，ack=seq+1。 大写的 ACK，则是我们上面说的 TCP 首部的标志位，用于标志的 TCP 包是否对上一个包进行了确认操作，如果确认了，则把 ACK 标志位设置成 1。

```
              Client                               Server
                |             FIN=M                  |
     FIN_WAIT_1 | DisConnect Request -----------> |
                |                                 |
                |         ack=M+1                 |
     FIN_WAIT_2 | <--------------------- Response | CLOSE_WAIT
                |                                 |
                |            FIN=N                |
      TIME_WAIT | <----------- DisConnect Request | LAST_ACK
                |         ACK=1, ack=N+1          |
                | Confirm ----------------------> | CLOSED
                |                                 |
```

挥手请求可以是 Client 端，也可以是 Server 端发起的，我们假设是 Client 端发起：

- 第一次挥手： Client 端发起挥手请求，向 Server 端发送标志位是 FIN 报文段，设置序列号 seq，此时，Client 端进入 FIN_WAIT_1 状态，这表示 Client 端没有数据要发送给 Server 端了。
- 第二次分手：Server 端收到了 Client 端发送的 FIN 报文段，向 Client 端返回一个标志位是 ACK 的报文段，ack 设为 seq 加 1，Client 端进入 FIN_WAIT_2 状态，Server 端告诉 Client 端，我确认并同意你的关闭请求。
- 第三次分手： Server 端向 Client 端发送标志位是 FIN 的报文段，请求关闭连接，同时 Client 端进入 LAST_ACK 状态。
- 第四次分手： Client 端收到 Server 端发送的 FIN 报文段，向 Server 端发送标志位是 ACK 的报文段，然后 Client 端进入 TIME_WAIT 状态。Server 端收到 Client 端的 ACK 报文段以后，就关闭连接。此时，Client 端等待 2MSL 的时间后依然没有收到回复，则证明 Server 端已正常关闭，那好，Client 端也可以关闭连接了。

HTTP

HTTP: hypertext transfer protocol (超文本传输协议)

- Application layer protocol
- Client/server model
  - Client: browser that requests, receives, (using HTTP protocol) and show Web objects （Render）
  - Server: Web server sends (using HTTP protocol) objects in response to requests
- Uses TCP:
  1. Client initiates TCP connection (creates socket) to server, port 80(443 for https)
  2. Server accepts TCP connection from client
  3. HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
  4. TCP connection closed
- “stateless”: server maintains no information about past client requests
  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

Request Msg: format

```
| method | sp | URI | sp | version | cr | lf |      -- request line
| header field name: | | value | cr | lf |          \
~                                      ~           - request header
| header field name: | | value | cr | lf |          /
| cr | lf |                                         -- empty line
| body ~                                   |      -- body -- POST
```

HTTP v1.1 request method: CRUD: resource/data/information: GET, POST, PUT, DELETE, PATCH, HEAD, TRACE, OPTIONS, CONNECT

Key-value pairs, metadata about the request and the client:
- Host name
- Authentication
- Content types
- User-agent information
- Caching / Cookies
- Types of connections

Non-persistent HTTP
- At most one object sent over TCP connection
- connection then closed
- Downloading multiple objects required multiple connections

Persistent HTTP: Multiple objects can be sent over single TCP connection between client, server

RTT (Round Trip Time): time for a small packet to travel from client to server and back round trip time HTTP response time:
- One RTT to initiate TCP connection
- One RTT for HTTP request and first few bytes of HTTP response to return
- File transmission time
- Non-persistent HTTP response time = 2RTT+ file transmission time

```
Client                          Server
  |                               |
  | Connect Request --------------> | \
  |                               |  RTT 1
  | <--------------------- Response | /
  |                               |
  | File Request -----------------> | \
  |                               |  RTT 2
  | <------------ File transmission | /
  | file received
```

Persistent HTTP
Non-persistent HTTP issues:
- Requires 2 RTTs per object
- OS overhead for each TCP connection
- Browsers often open parallel TCP connections to fetch referenced objects

Persistent HTTP issues:
- Server leaves connection open after sending response
- Subsequent HTTP messages between same client/server sent over open connection
- Client sends requests as soon as it encounters a referenced object
- As little as one RTT for all the referenced objects

Status Code
The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. These codes are fully defined in section 10. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user. The client is not required to examine or display the Reason- Phrase.

The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- 1xx: Informational - Request received, continuing process
- 2xx: Success - The action was successfully received, understood, and accepted
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfill an apparently valid request

The individual values of the numeric status codes defined for HTTP/1.1, and an example set of corresponding Reason-Phrase's, are presented below. The reason phrases listed here are only recommendations – they MAY be replaced by local equivalents without affecting the protocol.

```
Status-Code    =
      "100"  ; Section 10.1.1: Continue
    | "101"  ; Section 10.1.2: Switching Protocols
    | "200"  ; Section 10.2.1: OK
    | "201"  ; Section 10.2.2: Created
    | "202"  ; Section 10.2.3: Accepted
    | "203"  ; Section 10.2.4: Non-Authoritative Information
    | "204"  ; Section 10.2.5: No Content
    | "205"  ; Section 10.2.6: Reset Content
    | "206"  ; Section 10.2.7: Partial Content
    | "300"  ; Section 10.3.1: Multiple Choices
    | "301"  ; Section 10.3.2: Moved Permanently
    | "302"  ; Section 10.3.3: Found
    | "303"  ; Section 10.3.4: See Other
    | "304"  ; Section 10.3.5: Not Modified
    | "305"  ; Section 10.3.6: Use Proxy
    | "307"  ; Section 10.3.8: Temporary Redirect
    | "400"  ; Section 10.4.1: Bad Request
    | "401"  ; Section 10.4.2: Unauthorized
    | "402"  ; Section 10.4.3: Payment Required
    | "403"  ; Section 10.4.4: Forbidden
    | "404"  ; Section 10.4.5: Not Found
    | "405"  ; Section 10.4.6: Method Not Allowed
    | "406"  ; Section 10.4.7: Not Acceptable

    | "407"  ; Section 10.4.8: Proxy Authentication Required
    | "408"  ; Section 10.4.9: Request Time-out
    | "409"  ; Section 10.4.10: Conflict
    | "410"  ; Section 10.4.11: Gone
    | "411"  ; Section 10.4.12: Length Required
    | "412"  ; Section 10.4.13: Precondition Failed
    | "413"  ; Section 10.4.14: Request Entity Too Large
    | "414"  ; Section 10.4.15: Request-URI Too Large
    | "415"  ; Section 10.4.16: Unsupported Media Type
    | "416"  ; Section 10.4.17: Requested range not satisfiable
    | "417"  ; Section 10.4.18: Expectation Failed
    | "500"  ; Section 10.5.1: Internal Server Error
    | "501"  ; Section 10.5.2: Not Implemented
    | "502"  ; Section 10.5.3: Bad Gateway
    | "503"  ; Section 10.5.4: Service Unavailable
    | "504"  ; Section 10.5.5: Gateway Time-out
    | "505"  ; Section 10.5.6: HTTP Version not supported
    | extension-code
```

```
extension-code = 3DIGIT
Reason-Phrase  = *<TEXT, excluding CR, LF>
```

Cookies

Four components:

1. cookie header line of HTTP response message
2. cookie header line in next HTTP request message
3. cookie file kept on user's host, managed by user's browser
4. back-end database at Web site

Web cache

Cache acts as both client and server

- server for original requesting client
- client to origin server
- Typically cache is installed by ISP (university, company, residential ISP)

Why Web caching?

- reduce response time for client request
- reduce traffic on an institution's access link
- Internet dense with caches: enables "poor" content providers to effectively deliver content (so too does P2P file sharing)

Conditional GET

- Goal: don't send object if cache has up-to-date cached version
- no object transmission delay
- lower link utilization
- cache: specify date of cached copy in HTTP request If-modified-since: `<date>`
- server: response contains no object if cached copy is up-to-date: HTTP/1.0 304 Not Modified

HTTP/2, HTTP/3

Key goal: decreased delay in multi-object HTTP requests

HTTP1.1: introduced multiple, pipelined GETs over single TCP connection

- server responds in-order (FCFS: first-come-first-served scheduling) to GET requests
- with FCFS, small object may have to wait for transmission (head-of- line (HOL) blocking) behind large object(s)
- loss recovery (retransmitting lost TCP segments) stalls object transmission

HTTP/2: [RFC 7540, 2015] increased flexibility at server in sending objects to client:

- methods, status codes, most header fields unchanged from HTTP 1.1
- transmission order of requested objects based on client-specified object priority (not necessarily FCFS)
- push unrequested objects to client
- divide objects into frames, schedule frames to mitigate HOL blocking

HTTP/2: objects divided into frames, frame transmission interleaved

HTTP/2 over single TCP connection means:

- recovery from packet loss still stalls all object transmissions
- as in HTTP 1.1, browsers have incentive to open multiple parallel TCP connections to reduce stalling, increase overall throughput
- no security over raw TCP connection

HTTP/3: adds security, per object error- and congestion- control (more pipelining) over UDP

QUIC: Quick UDP Internet Connections
application-layer protocol, on top of UDP
- increase performance of HTTP
- deployed on many Google servers, apps (Chrome, mobile YouTube app)

Adopts approaches we'll study in chapter for connection establishment, error control, congestion control
- error and congestion control: "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- connection establishment: reliability, congestion control, authentication, encryption, state established in one RTT

TCP vs. QUIC:
- TCP: TCP (reliability, congestion control state) + TLS (authentication, crypto state): 2 serial handshakes
- QUIC: reliability, congestion control, authentication, crypto state: 1 handshake

RESTful API
Representational State Transfer Application Programming Interface

A set of recommended styles/rules for API design
- Use of standard HTTP methods (GET, POST, PUT, DELETE, etc.) to operate on resources.
- Resources are identified via URIs, where each URI represents a resource.
- Statelessness, meaning the server does not store any client session information between requests.
- Use of standard HTTP status codes to represent the outcome of requests.
- Over 93% of developers using RESTful API in their projects

DNS: domain name system
Application-layer protocol:
- C/S architecture
- UDP (port 53)
- hosts, name servers communicate to resolve names (name / address translation)

Distributed database implemented in hierarchy of many name servers

DNS services
- Hostname to IP address translation(A)
- Host aliasing (cname): canonical, alias names
- Mail server aliasing(mx)
- Load distribution: Replicated Web servers: many IP addresses correspond to one name

Why not centralize DNS?
- Single point of failure
- Traffic volume
- Distant centralized database
- Maintenance

DNS
- Humongous distributed database:
  - ~ billion records, each simple

- handles many trillions of queries/day:
  - many more reads than writes
  - performance matters: almost every Internet transaction interacts with DNS - msecs count!
- organizationally, physically decentralized:
  - millions of different organizations responsible for their records
- “bulletproof”: reliability, security

A distributed, hierarchical database:

```
                                  +----------------------------+
Root                              |      Root DNS Servers      |
                                  +----------------------------+
                                  /               |            \
                                 /                |             \
Top Level Domain      com DNS servers     org DNS servers     edu DNS servers    ...

Authoritative     ....com
```

Root name servers:
- Contacted by local name server that can not resolve name
- Root name server:
- Contacts authoritative name server if name mapping not known
- Gets mapping
- Returns mapping to local name server

13 logical root name “servers” worldwide: Each “server” replicated many times

TLD, authoritative servers:
- Top-level domain (TLD) servers: Responsible for com, org, net, edu, aero, jobs, museums, and all Top-level country domains, e.g.: cn, uk, fr, ca, jp
  Eg.: Network Solutions maintains servers for .com TLD; Educause for .edu TLD (https://net.educause.edu/)
- Authoritative DNS servers:
  - Organization’s own DNS server(s), providing authoritative hostname to IP mappings for organization’s named hosts
  - Can be maintained by organization or service provider

Local DNS name server
- Does not strictly belong to hierarchy
- Each ISP (residential ISP, company, university) has one: Also called “default name server”
- to find yours: MacOS: `% scutil --dns`; Windows: `>ipconfig /all`
- When host makes DNS query, query is sent to its local DNS server
  - Has local cache of recent name-to-address translation pairs (but may be out of date!)
  - Acts as proxy, forwards query into hierarchy

caching, updating records
- Once (any) name server learns mapping, it caches mapping: Cache entries timeout (disappear) after some time (TTL)
- TLD servers typically cached in local name servers: thus root name servers not often visited
- Cached entries may be out-of-date: If name host changes IP address, may not be known Internet-wide until all TTLs expire
- Update/notify mechanisms proposed IETF standard: RFC 2136

Records

DNS: distributed database storing resource records (RR)

RR format: (name, value, type, ttl)

type=A
- name is hostname
- value is IP address

type=NS
- name is domain (e.g., foo.com)
- value is hostname of authoritative name server for this domain

type=CNAME
- name is alias name for some “canonical” (the real) name
- “www.taobao.com”“ is really “www.taobao.com.danuoyi.tbcache.com”
- value is canonical name

type=MX
- value is name of mailserver associated with name

Query and reply messages, both with same message format.

```
|   identification   |        flag         |
|    # questions     |    # answer RRs     |
|   # authority RRs  |  # additional RRs   |
|    questions (vairable # of questions)   |
|     answers (vairable # of questions)    |
|    authority (vairable # of questions)   |
| additonal info (vairable # of questions) |
```

Message header:
- identification: 16 bit # for query, reply to query uses same #
- flags:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative
- Name, type fields for a query
- RRs in response to query
- Records for authoritative servers
- Additional “helpful” info that may be used

New record

Register name feimax.com at DNS registrar (e.g., net.cn)
- Normally, you don’t need to set up the NS record
- Insert A record for the IP address of your host
- Insert MX record for email
- …

P2P

No always-on server
- Arbitrary end systems directly communicate
- Peers change IP addresses

Examples:
- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)

File distribution time: client-server
- server transmission: must sequentially send (upload) N file copies:
  - time to send one copy: $\frac{F}{u_s}$,
  - time to send N copies: $N\frac{F}{u_s}$.
- client: each client must download file copy
  - d_{min} = min client download rate
  - max client download time: $\frac{F}{d_{\text{min}}}$

time to distribute F to N clients using client-server approach: $D_{c-s} > \max\left\{N\frac{F}{u_s},, \frac{F}{d_{\text{min}}}\right\}$ NF ← increase linearly in N

File distribution time: P2P
- server transmission: must sequentially send (upload) at least one file copies: time to send one copy: F/us
- client: each client must download file copy: min client download time: F/dmin
- clients: as total must download NF bits: max upload rate (limiting max download rate) is $u_s + \Sigma u_i$

time to distribute F to N clients using P2P approach: $D_{\text{P2P}} > \max\left\{\frac{F}{u_s},, \frac{F}{d_{\text{min}}}, N\frac{F}{u_s+\Sigma u_i}\right\}$

BitTorrent
- Efficient content distribution system using file swarming.
- The throughput increases with the number of downloaders via the efficient use of network bandwidth

Peers in torrent send/receive file pieces(chunks)
- Tracker: a central server keeping a list of all peers tracks peers participating in torrent
- Torrent/Swarm: group of peers exchanging chunks of a file

To share a file or group of files, the initiator first creates a .torrent file, a small file that contains:
- Metadata about the files to be shared
- Information about the tracker, the computer that coordinates the file distribution

Downloaders first obtain a .torrent file, and then connect to the specified tracker, which tells them from which other peers to download the pieces of the file.

Metadata of .torrent
- SHA-1 hashes of all pieces
- A mapping of the pieces to files
- Piece size
- Length of the file
- A tracker reference

BitTorrent
- Peer joining torrent:
  - has no pieces, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers (“neighbors”)
- While downloading, peer uploads pieces to other peers
- Peer may change peers with whom it exchanges pieces

- Peers may come and go
- Once peer has entire file, it may (selfishly) leave or remain in torrent

Seeder = a peer that provides the complete file. Initial seeder = a peer that provides the initial copy.
Leecher -> Seeder
- As soon as a leecher has a complete piece, it can potentially share it with other downloaders.
- Eventually each leecher becomes a seeder by obtaining all the pieces, and assembles the file. Verifies the "checksum" of the file

Piece selection policy
- The order in which pieces are selected by different peers is critical for good performance.
- If an inefficient policy is used, then peers may end up in a situation where each has all identical set of easily available pieces, and none of the missing ones.
- If the original seed is prematurely taken down, then the file cannot be completely downloaded!

Piece selection - Micro view
- Rarest First: General rule
  - Determine the pieces that are most rare among your peers, and download those first.
  - This ensures that the most commonly available pieces are left till the end to download.
- Random First Piece: Special case, at the beginning
  - Initially, a peer has nothing to trade
  - Important to get a complete piece ASAP
  - Select a random piece of the file and download it
- Endgame Mode: Special case
  - Near the end, missing pieces are requested from every peer containing them.
  - This ensures that a download is not prevented from completion due to a single peer with a slow transfer rate.
  - Some bandwidth is wasted, but in practice, this is not too much.

Internal Mechanism
- Built-in incentive mechanism (where all the magic happens):
- Choking Algorithm
- Optimistic Unchoking

Choking:
- Choking is a temporary refusal to upload. It is one of BT's most powerful idea to deal with free riders (those who only download but never upload).
- For avoiding free riders and avoiding network congestion
- Tit-for-tat strategy is based on game-theoretic concepts.

Optimistic unchoking
- A peer sends pieces to those four peers currently sending her chunks at highest rate
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- Every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 4
- Reasons:
  - To discover currently unused connections that are better than the ones being used
  - To provide minimal service to new peers

Upload-Only mode:
- Once download is complete, a peer can only upload. The question is, which nodes to upload to?
- Policy: Upload to those with the best upload rate. This ensures that pieces get replicated faster, and new seeders are created fast

Socket

Socket: door between application process and end-end-transport protocol

Two socket types for two transport services:
- UDP: unreliable datagram
- TCP: reliable, byte stream-oriented

Application Example:
1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

UDP:
- UDP: no "connection" between client & server
  - No handshaking before sending data
  - Sender explicitly attaches IP destination address and port # to each packet
  - Receiver 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
create socket              |          | create socket, port = x
clientSocket =             |          | serverSocket =
socke(AF_INET, SOCK_DGRAM) |          | socket(AF_INET, SOCK_DGRAM)
                           |          |
Create datagram with server |         |
IP:x; send datagram        | -------->| read datagram: serverSocket.recv_from
via `clientSocket`         |          |
                           |          |
read datagram form         | <--------| write reply to serverSocket
clientSocket               |          | specifying client address, port
                           |          ~
                           v
                   close clientSocket
```

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 (more in Chap 3)

Application viewpoint: TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server

```
                          Client       Server
                             |           | create socket, port=x
                             |           | serverSocket = socket(AF_INET, SOCK_STREAM)
create socket                |           |
connect to host:x            | ← TCP → | wait for incoming connection
clientSocket = socket()      |           | cn = serverSocket.accept()
                             |           |
sending request              | --------> | read request: cn.recv()
                             |           |
read reply                   | <-------- | write reply: cn.send()
                             |           |
                             v           v
                           close       close
```

Transport Layer

Provide logical communication between app processes running on different hosts

- Transport protocols run in end systems
  - Send side: breaks app msg into segments, passes to network layer
  - Rcv side: reassembles segments into messages, passes to app layer
- Transport-layer protocols for Internet: TCP and UDP

Transport layer: logical communication between processes: Relies on, enhances, network layer services

Network layer: logical communication between hosts

10 kids in Ann’s house sending letters to 10 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

Sender:
- pass an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP

Receiver:
- extracts application-layer message
- checks header values
- receives segment from IP
- demultiplexes message up to application via socket

Two Principal Internet transport protocols Unreliable, unordered delivery: UDP
- No-frills extension of “best-effort” network layer (internet protocol)

Reliable, in-order delivery: TCP
- Congestion control
- Flow control
- Connection setup

Services not available:
- Delay guarantees
- Bandwidth guarantees

Multiplexing/demultiplexing

Multiplexing as sender: handle data from multiple sockets, add transport header (later used for demultiplexing)

Demultiplexing as receiver: use header info to deliver received segments to correct socket

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 suitable socket

UDP: Created socket has host-local port number:

```
Socket = socket(AF_INET, SOCK_DGRAM)
Socket.bind(('', 12345))
```

When creating datagram to send into UDP socket, must specify Destination IP address; Destination port number

```
clientSocket.sendto(msg,(server_name, server_port))
```

host receives UDP segment:
- Checks destination port # in segment
- Directs UDP segment to socket with that port #

IP datagrams with same dest. port #, but different source IP addresses and/or source port numbers will be directed to same socket at dest

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: non-persistent HTTP will have different socket for each request

UDP: User Datagram Protocol [RFC 768]

Feature:
- Simple and straightforward
- Best effort
- Lost
- Connectionless
- No handshaking
- Each UDP segment handled independently of others: Out-of-order to APP

UDP use:
- Streaming multimedia apps
- DNS
- HTTP/3

Reliable transfer over UDP:
- Add reliability at application layer
- Application-specific error recovery!

UDP sender actions:
- pass an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP

UDP receiver actions:
- extracts application-layer message
- checks UDP checksum header value
- receives segment from IP
- demultiplexes message up to application via socket

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

```
|     source port #     |      dest port #      |
|        length         |       checksum        |
~                                               ~
|        application data (payload)             |
```

Goal: detect “errors” in transmitted segment Sender:
- Treat segment contents, including header fields, as sequence of 16-bit integers
- Checksum: addition (one’s complement sum) of pseudo header, UDP header and UDP data
- 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 nonetheless?

Checksum:

```
unsigned short checksum;
unsigned short *b;
while ((b = input()) ≠ NULL) {
  if ((((int)checksum + *b) & (1 << 16)) > 0) {
    // if addition has overflow
    // add carried 1 to back of sum
    // which is, add 1 directly
    checksum += *b;
    checksum += 1;
  } else {
    checksum += *b;
  }
}
checksum = ~checksum;
```

Principles of reliable data transfer

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)

Sender, receiver do not know the "state" of each other, e.g., was a message received. Unless communicated via a message.

Reliable data transfer protocol (rdt): interfaces;
- rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
- sender-side implementation of rdt reliable data transfer protocol
- udt_send(): called by rdt to transfer packet over unreliable channel to receiver
- Bi-directional communication over unreliable channel
- rdt_rcv(): called when packet arrives on receiver side of channel
- receiver-side implementation of rdt reliable data transfer protocol
- deliver_data(): called by rdt to deliver data to upper layer

```
             +---------+                      +-----------+
             | sending |                      | receiving |
             | process |                      |  process  |
             +---------+                      +-----------+
                   | data ||||  <----data---->       | data ||||
 rdt_send ----------------------        ---------------------------
                   |                                 ^
                   v                                 |
 sender-side implementation of rdt    receiver-side implementation of rdt
                   ^                                 ^
                   |              packet             |
                udt_send     |Header| data ||||   rdt_rcv
                   |                                 |
                   `---->    unreliable channel <----`
```

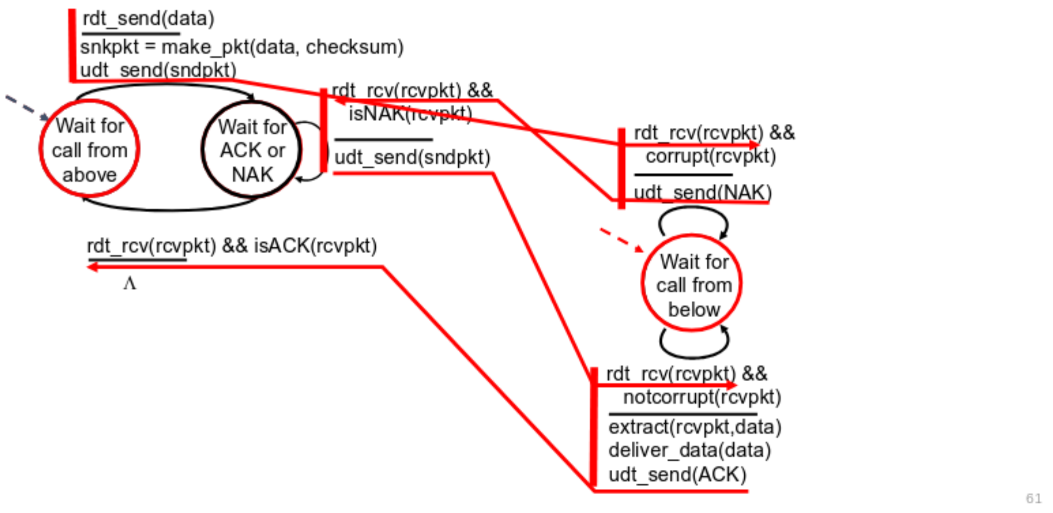
Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
- But control info will flow on both directions!
- Use finite state machines (FSM) to specify sender, receiver

rdt1.0: reliable transfer over a reliable channel
- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
- Separate FSMs for sender, receiver:
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channe
- sender: wait for call from above: loop rdt_send(data): packet = make_pkg(data); udt_send(packet)
- receiver: wait for call from below: loop rdt_rcv(packet): data = extract(packet); deliver_data(data)

rdt2.0: channel with bit errors
- Underlying channel may flip bits in packet: Checksum to detect bit errors
- 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

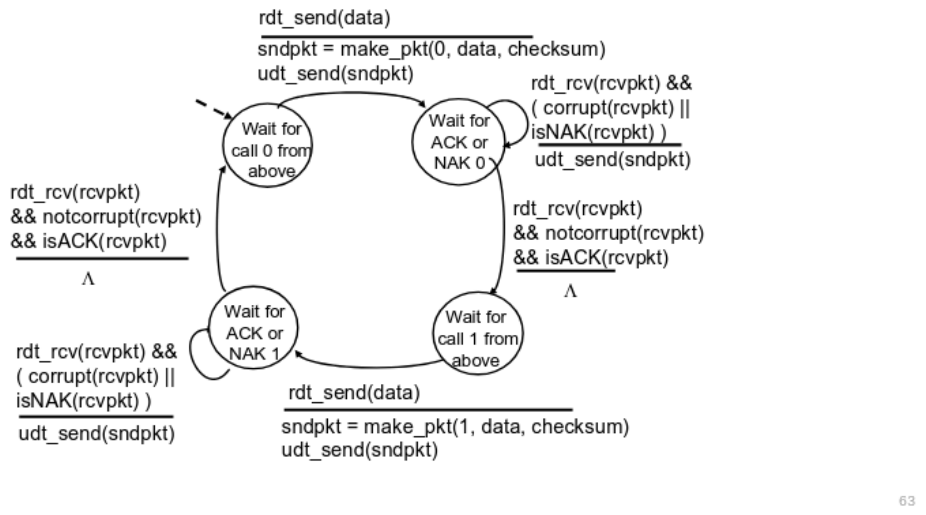
rdt2.0: Error scenario



rdt2.0 has a fatal flaw! What happens if ACK/NAK corrupted?
- Sender doesn't know what happened at receiver!
- Can't just retransmit -> possible duplicate

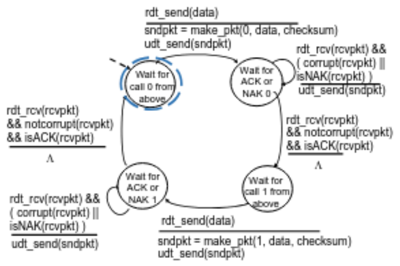
Handling duplicates:
- Sender retransmits current pkt if ACK/NAK corrupted
- Sender adds sequence number to each pkt
- Receiver discards (doesn't deliver up) duplicate pkt

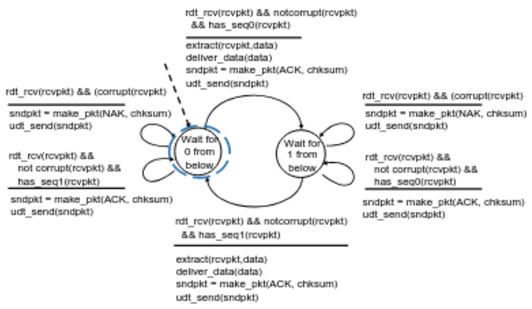
rdt2.1: sender, handles garbled ACK/NAKs



rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK or NAK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
Wait for ACK or NAK 1
Wait for call 1 from above
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
63

rdt2.1: sender vs receiver: no error


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK or NAK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
Wait for ACK or NAK 1
Wait for call 1 from above
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

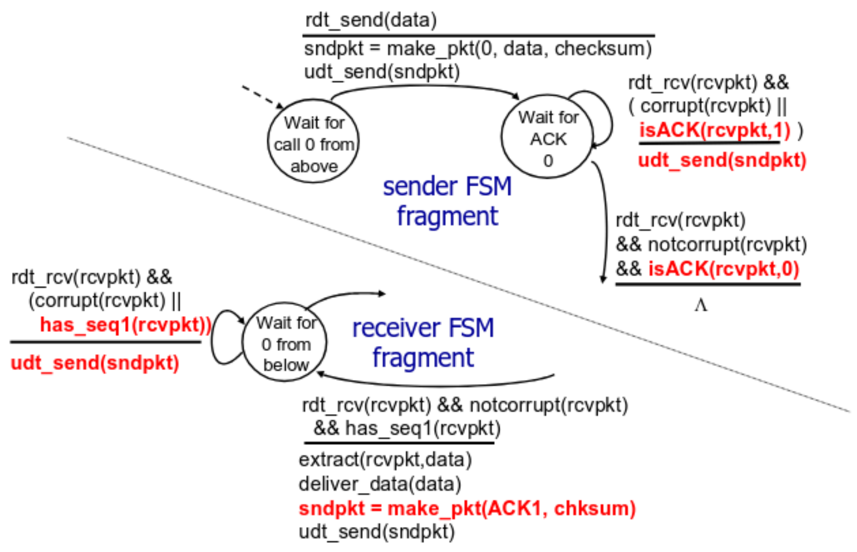

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq0(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

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 #
- Note: receiver can not know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol
- 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
- Duplicate ACK at sender results in same action as NAK: retransmit current pkt

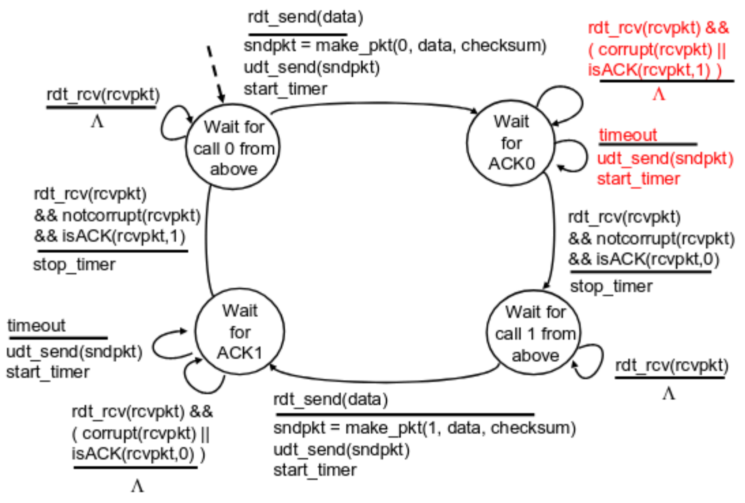
rdt2.2: Sender, receiver fragments


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
udt_send(sndpkt)
sender FSM fragment
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
Λ
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))
udt_send(sndpkt)
Wait for 0 from below
receiver FSM fragment
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)

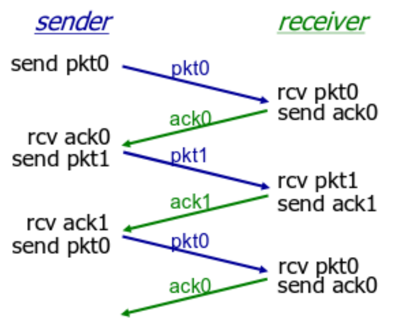
rdt3.0: Channels with errors and loss New assumption: Underlying channel can also lose packets (data, ACKs): Checksum, seq. #, ACKs, retransmissions will be of help … but not enough Approach: Sender waits "reasonable" amount of time for ACK

- 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

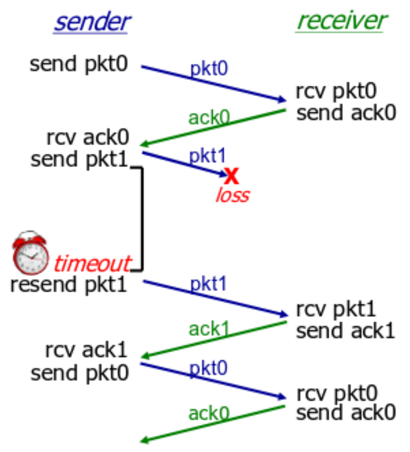
rdt3.0 sender


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
Λ
Wait for call 0 from above
Wait for ACK0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
Λ
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
timeout
udt_send(sndpkt)
start_timer
Wait for ACK1
Wait for call 1 from above
rdt_rcv(rcvpkt)
Λ
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

rdt3.0 in action

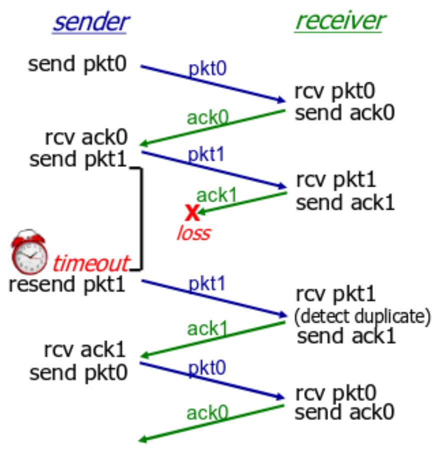


(a) no loss

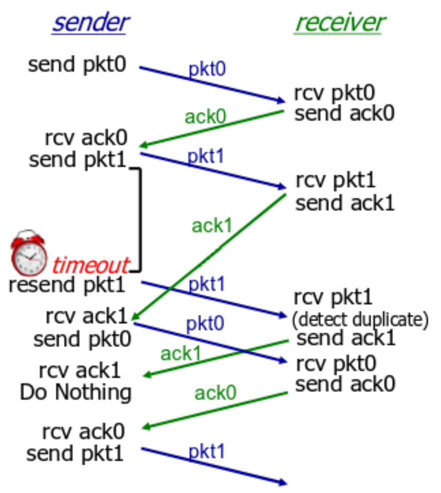


(b) packet loss

rdt3.0 in action

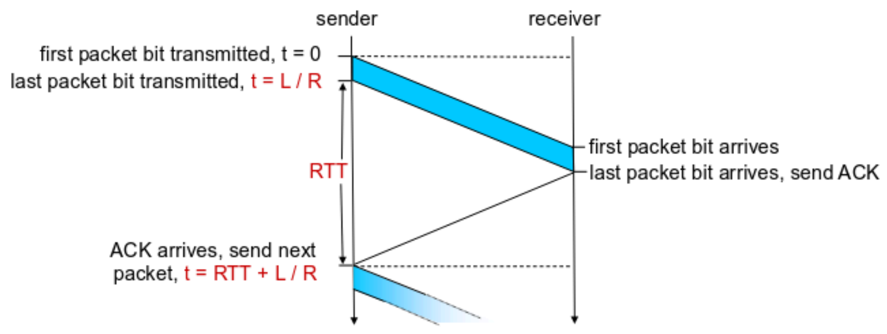


(c) ACK loss



(d) premature timeout/ delayed ACK

rdt3.0: stop-and-wait operation


sender
receiver
first packet bit transmitted, t = 0
last packet bit transmitted, t = L / R
RTT
first packet bit arrives
last packet bit arrives, send ACK
ACK arrives, send next packet, t = RTT + L / R

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

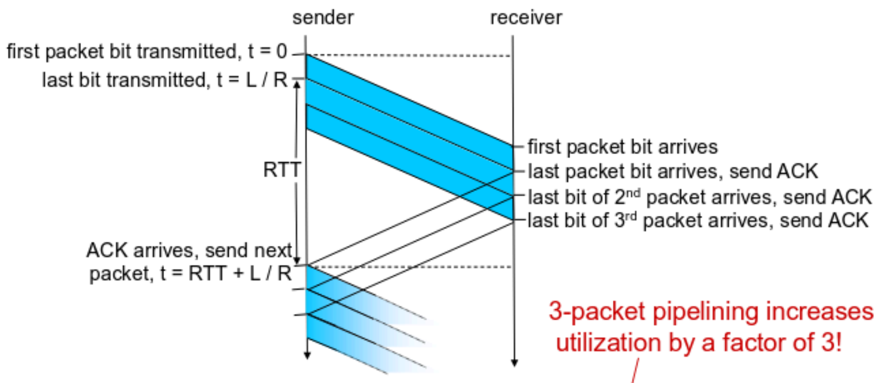
Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:
- Usender: utilization – fraction of time sender busy sending
- if RTT=30ms, 1KB pkt every 30 msec: 33kB/sec throughput over 1 Gbps link
- Network protocol limits use of physical resources!

Pipelined protocols

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts • Range of sequence numbers must be increased • Buffering at sender and/or receiver

Pipelining: increased utilization


sender
receiver
first packet bit transmitted, t = 0
last bit transmitted, t = L / R
RTT
first packet bit arrives
last packet bit arrives, send ACK
last bit of 2nd packet arrives, send ACK
last bit of 3rd packet arrives, send ACK
ACK arrives, send next packet, t = RTT + L / R
3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + 3L / R} = \frac{.024}{30.024} = 0.0008$$

Two forms: Go-Back-N and Selective repeat

Go-back-N (GBN):

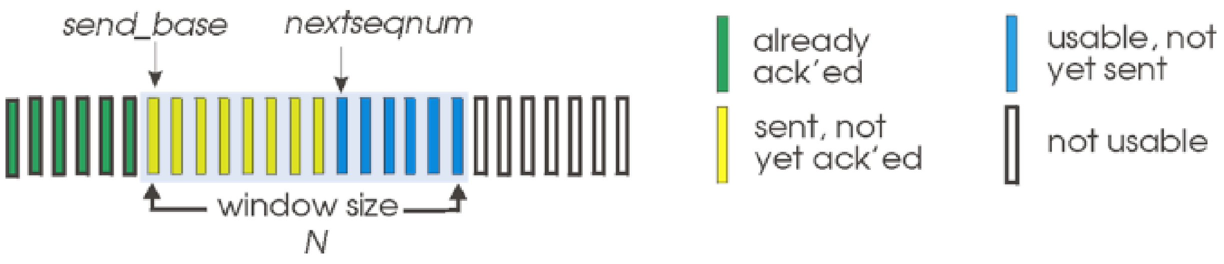
- Sender can have up to N unacked packets in pipeline
- Receiver only sends cumulative ACK: Doesn't ACK packet if there's a gap
- Sender has timer for oldest unACKed packet: When timer expires, retransmit all unacked packets

Selective Repeat (SR):

- Sender can have up to N unacked packets in pipeline
- Rcvr sends individual ack for each packet
- Sender maintains timer for each unacked packet: When timer expires, retransmit only that unacked packet

Go-Back-N: sender

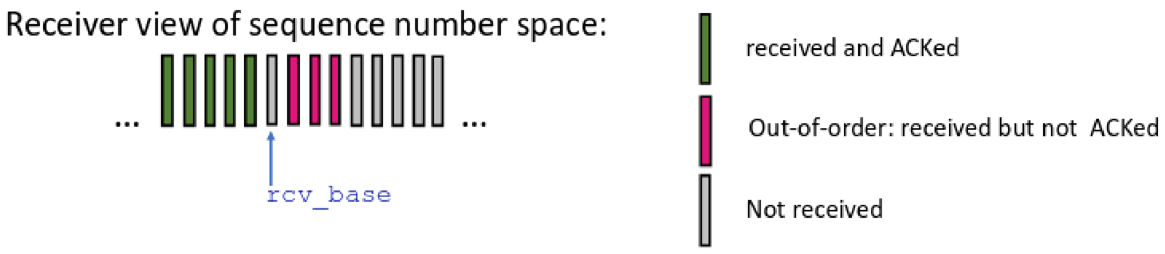
- sender: "window" of up to N, consecutive transmitted but unACKed pkts: k-bit seq # in pkt header



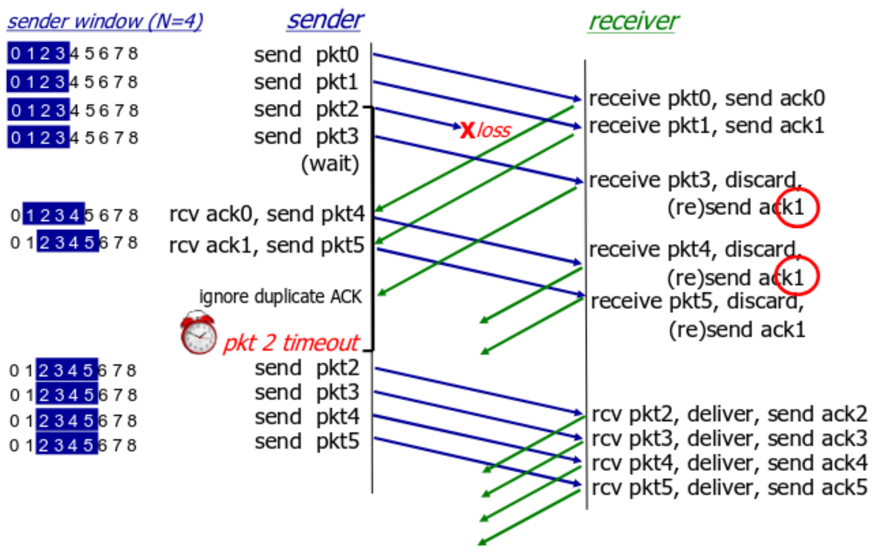
- cumulative ACK: ACK(n): ACKs all packets up to, including seq # n: on receiving ACK(n): move window forward to begin at n+1
- timer for oldest in-flight packet
- timeout(n): retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest in-order seq #
  - may generate duplicate ACKs
  - need only remember rcv_base
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #


Receiver view of sequence number space:
...
...
rcv_base
received and ACKed
Out-of-order: received but not ACKed
Not received

Go-Back-N in action



Selective repeat: the approach

- pipelining: multiple packets in flight
- receiver individually ACKs all correctly received packets: buffers packets, as needed, for in-order delivery to upper layer

sender:

- maintains (conceptually) a timer for each unACKed pkt: timeout: retransmits single unACKed packet associated with timeout
- maintains (conceptually) "window" over N consecutive seq #s: limits pipelined, "in flight" packets to be within this window

Sender:

data from above:

- If next available seq # in window, send pkt timeout(n):
- Resend pkt n, restart timer

ACK(n) in [sendbase, sendbase+N-1]:

- Mark pkt n as received
- If n smallest unACKed pkt, advance window base to next unACKed seq #

Receiver:

pkt n in [rcvbase, rcvbase+N-1]

- Send ACK(n)
- Out-of-order: buffer
- In-order: deliver (also deliver buffered, in-order pkts), advance window to next not- yet-received pkt

pkt n in [rcvbase-N,rcvbase-1]: ACK(n) otherwise: Ignore

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

- Point-to-point: one sender, one receiver
- Reliable, in-order byte stream: no "message boundaries"
- Pipelined: TCP congestion and flow control set window size

- Full duplex data: bi-directional data flow in same connection
- Connection-oriented: handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- Flow controlled: sender will not overwhelm receiver

```
 0                             15 16                              31
|        source port (16)        |      destination port (16)     | \
|                      Sequence number (32)                       |  |
|                    Acknowledge number (32)                      |  |
| head |    reserve    |U|A|P|R|S|F|                              |  20 Bytes
|      |               |R|C|S|S|Y|I|        Window size (16)      |  |
| (4)  |      (6)      |G|K|H|T|N|N|                              |  |
|         check sum (16)         |           pointer (16)         | /
~                                                                 ~
|                                                                 |
|                            options                              |
```

- ACK: seq # of next expected byte; A bit: this is an ACK
- head: length (of TCP header) Internet checksum
- C, E: congestion notification
- RST, SYN, FIN: connection management
- window size: flow control: # bytes receiver willing to accept
- seqence number: segment seq #: counting bytes of data into bytestream (not segments!)

Sequence numbers: byte stream “number” of first byte in segment’s data

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

TCP round trip time, timeout

Q: How to set TCP timeout value? Longer than RTT: But RTT varies
- Too short: premature timeout, unnecessary retransmissions
- Too long: slow reaction to segment loss

Q: How to estimate RTT?
- SampleRTT: measured time from segment transmission until ACK receipt: Ignore retransmissions
- SampleRTT will vary, want estimated RTT “smoother”: Average several recent measurements, not just current SampleRTT

TCP round trip time, timeout

$\text{EstimatedRTT}_n = (1 - \mathtt{a}) * \text{EstimatedRTT}_{n-1} + \alpha * \text{SampleRTT}$

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: a = 0.125

TCP round trip time, timeout • Timeout interval: EstimatedRTT plus “safety margin” • large variation in EstimatedRTT -> larger safety margin • Estimate SampleRTT deviation from EstimatedRTT:

$$\text{DevRTT} = (1 - \beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}| \ \ (\text{typically}, \beta = 0.25)$$

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
                          ↑            ↑
                 estimated RTT  “safety margin”
```

(Dev = Deviation)

TCP reliable data transfer:
- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- Retransmissions triggered by:
  - timeout events
  - duplicate acks

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 fast retransmit: if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #: likely that unACKed segment lost, so don't wait for timeout

Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!

Flow control: receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast
- TCP receiver "advertises" free buffer space in rwnd field in TCP header
  - RcvBuffer size set via socket options (typical default is 4096 bytes)
  - many operating systems auto-adjust RcvBuffer
- sender limits amount of unACKed ("in-flight") data to received rwnd
- guarantees receive buffer will not overflow
- TCP receiver "advertises" free buffer space in rwnd field in TCP header
  - RcvBuffer size set via socket options (typical default is 4096 bytes)
  - many operating systems auto-adjust RcvBuffer
- sender limits amount of unACKed ("in-flight") data to received rwnd
- guarantees receive buffer will not overflow

Connection Management Before exchanging data, sender/receiver "handshake": Agree to establish connection (each knowing the other willing to establish connection) Agree on connection parameters

closing:

- Client, server each close their side of connection: Send TCP segment with FIN bit = 1
- Respond to received FIN with ACK: On receiving FIN, ACK can be combined with own FIN
- Simultaneous FIN exchanges can be handled

Congestion:

"too many sources sending too much data too fast for network to handle"(informally)

Manifestations:

- long delays (queueing in router buffers)
- packet loss (buffer overflow at routers)

Different from flow control!

Causes and Costs of Network Congestion

- Load increases: Many hosts send data at the same time, increasing the number of packets routers must process.
- Queues build up: If packets arrive faster than they can be forwarded, they start to queue in router buffers.
- Buffers overflow: When buffers are full, new packets are dropped.
- Retransmissions add load: Dropped packets trigger retransmissions, adding more traffic.
- => Result: Higher delay, lower throughput, wasted bandwidth.
- Congestion occurs when the input rate exceeds the network's capacity.

Approaches towards congestion control:

End-end congestion control:

- no explicit feedback from network
- congestion inferred from observed loss, delay
- approach taken by TCP

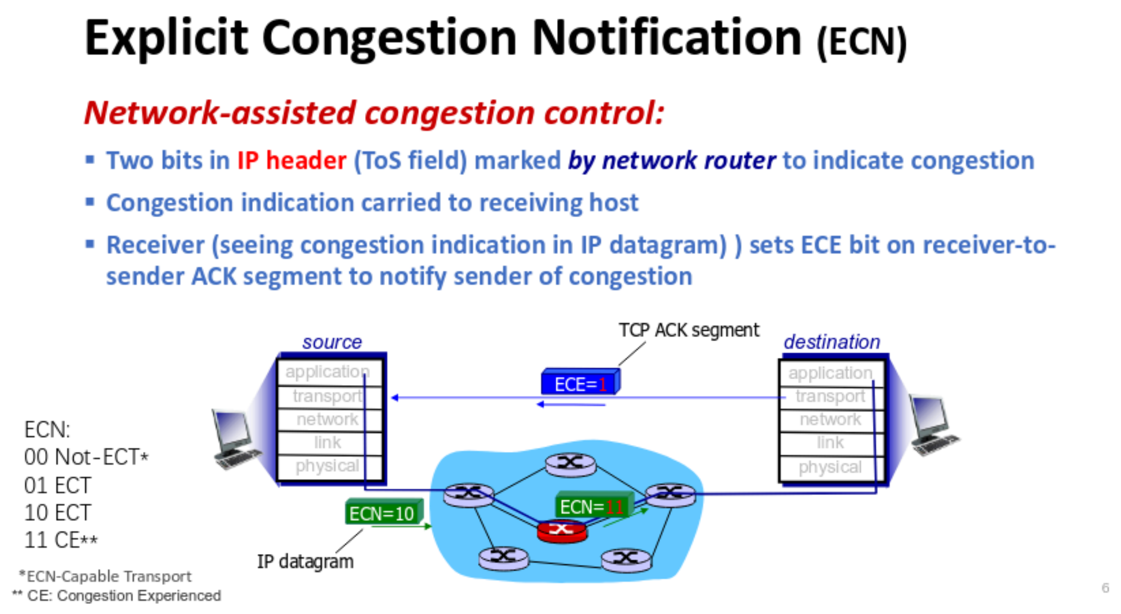
Network-assisted congestion control:

- routers provide direct feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate
- TCP ECN …

Explicit Congestion Notification (ECN)

Network-assisted congestion control:

- Two bits in IP header (ToS field) marked by network router to indicate congestion
- Congestion indication carried to receiving host
- Receiver (seeing congestion indication in IP datagram)) sets ECE bit on receiver-to- sender ACK segment to notify sender of congestion



Explicit Congestion Notification (ECN)

Network-assisted congestion control:

- Two bits in IP header (ToS field) marked ***by network router*** to indicate congestion
- Congestion indication carried to receiving host
- Receiver (seeing congestion indication in IP datagram)) sets ECE bit on receiver-to-sender ACK segment to notify sender of congestion

ECN:
00 Not-ECT*
01 ECT
10 ECT
11 CE**

*ECN-Capable Transport
** CE: Congestion Experienced

AIMD: sawtooth behavior: probing for bandwidth

approach: senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

- Additive Increase: increase sending rate by 1 maximum segment size every RTT until loss detected
- Multiplicative Decrease: cut sending rate in half at each loss event

Multiplicative decrease detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD? AIMD – a distributed, asynchronous algorithm – has been shown to:

- optimize congested flow rates network wide
- have desirable stability properties

TCP sending behavior:

▪ roughly: send cwnd bytes, wait RTT for ACKS, then send more bytes TCP $\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}}$ bytes/sec

TCP sender limits transmission: `LastByteSent - LastByteAcked ⩽ cwnd`: cwnd is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

TCP slow start

when connection begins, increase rate exponentially until first loss event:

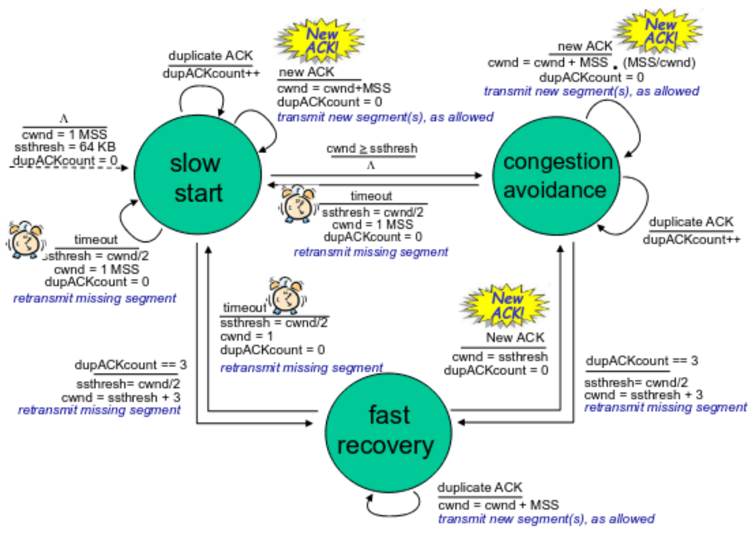
- initially cwnd = 1 MSS
- double cwnd every RTT
- done by incrementing cwnd for every ACK received

Summary: initial rate is slow, but ramps up exponentially fast

TCP: from slow start to congestion avoidance Q: when should the exponential increase switch to linear? A: when cwnd gets to 1/2 of its value before timeout. Implementation:

- variable ssthresh
- on loss event, ssthresh is set to 1/2 of cwnd just before loss event

Summary: TCP congestion control



TCP throughput

- avg. TCP throughput as function of window size, RTT? ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
- avg. window size (# in-flight bytes) is ¾ W
- avg. thruput is 3/4W per RTT

$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{\text{RTT}} \text{ bytes/sec}$$

Evolving transport-layer functionality

TCP, UDP: principal transport protocols for 40 years

different “flavors” of TCP developed, for specific scenarios:

▪ moving transport–layer functions to application layer, on top of UDP: HTTP/3: QUIC

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets “in flight”; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, “background” TCP flows

QUIC: Quick UDP Internet Connections

application-layer protocol, on top of UDP

- increase performance of HTTP
- deployed on many Google servers, apps (Chrome, mobile YouTube app)

adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- error and congestion control: "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- connection establishment: reliability, congestion control, authentication, encryption, state established in one (or even zero) RTT (recall HTTP3 discussion in Chapter 2)
- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

Principles behind transport layer services:
- Multiplexing, demultiplexing
- Reliable data transfer
- Flow control
- Congestion control

Implementation in the Internet
- UDP
- TCP

Network Layer

Goal: Understand principles behind network layer services, focusing on data plane:
- Network layer service models
- Forwarding versus routing
- How a router works
- Generalized forwarding

Instantiation, implementation in the Internet

Network-layer services and protocols

Transport segment from sending to receiving host
- sender: encapsulates segments into datagrams, passes to link layer
- receiver: delivers segments to transport layer protocol

Network layer protocols in every Internet device: hosts, routers

Routers:
- examines header fields in all IP datagrams passing through it
- moves datagrams from input ports to output ports to transfer datagrams along end-end path

Two key network-layer functions

Network-layer functions:
- Forwarding: move packets from router's input to appropriate router output
- Routing: determine route taken by packets from source to destination: Routing algorithm

Analogy: taking a trip
- forwarding: process of getting through single interchange
- routing: process of planning trip from source to destination

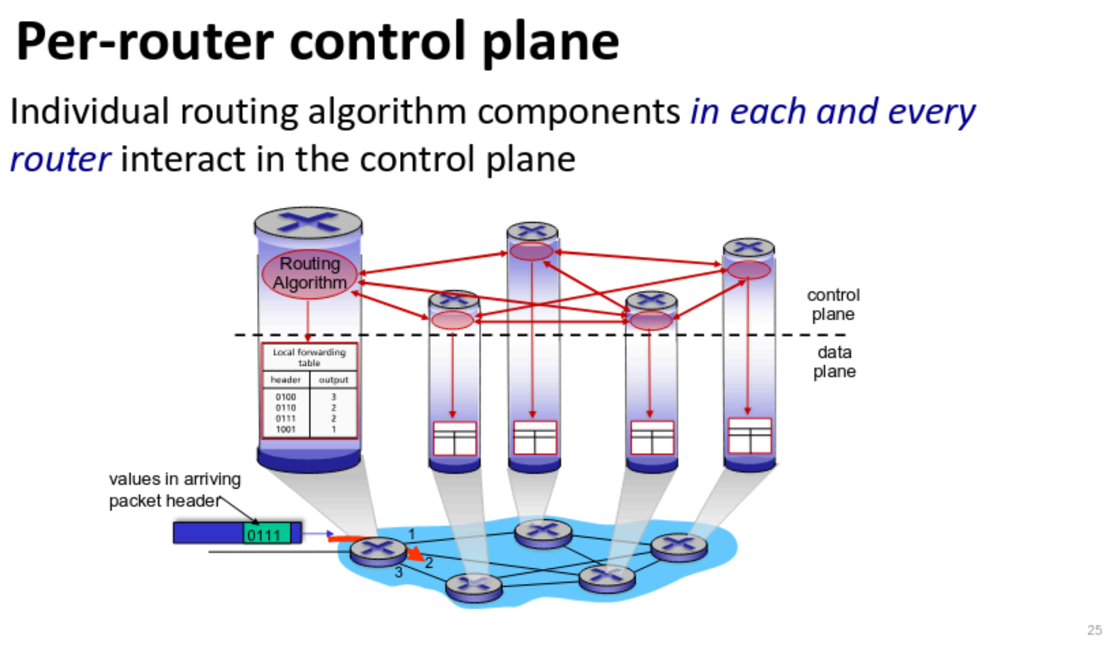
Network layer: data plane, control plane

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

Per-router control plane: Individual routing algorithm components in each and every router interact in the control plane



Per-router control plane

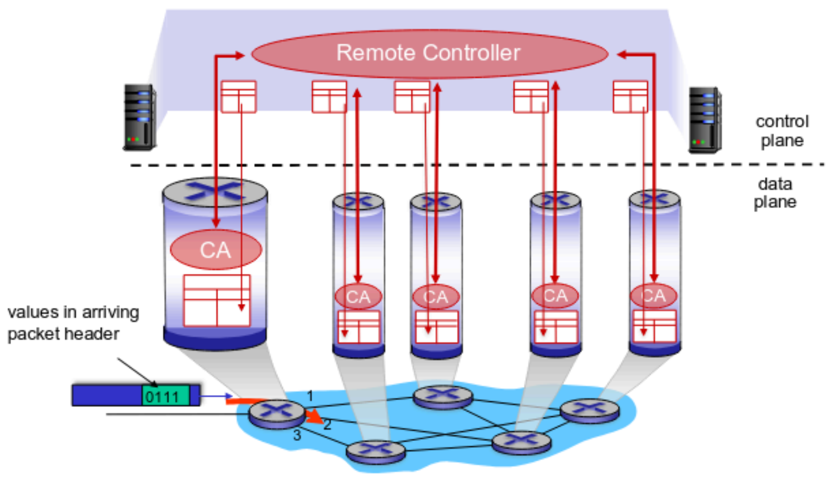
Individual routing algorithm components *in each and every router* interact in the control plane

Local forwarding table	
header	output
0100	3
0110	2
0111	2
1001	1

Software-Defined Networking (SDN) control plane: Remote controller computes, installs forwarding tables in routers

Software-Defined Networking (SDN) control plane

Remote controller computes, installs forwarding tables in routers



Network service model

example services for individual datagrams:

- guaranteed delivery
- guaranteed delivery with less than 40 msec delay

example services for a flow of datagrams:

- in-order datagram delivery
- guaranteed minimum bandwidth to flow
- restrictions on changes in inter-packet spacing

Internet “best effort” service model: No guarantees on:

- successful datagram delivery to destination
- timing or order of delivery
- bandwidth available to end-end flow

		Quality of Service (QoS) Guarantees			
Network Architecture	Service Model	Bandwidth	Loss	Order	Timing
Internet	best effort	none	no	no	no
ATM	Constant Bit Rate	Constant rate	yes	yes	yes
ATM	Available Bit Rate	Guaranteed min	no	yes	no
Internet	Intserv Guaranteed (RFC 1633)	yes	yes	yes	yes
Internet	Diffserv (RFC 2475)	possible	possibly	possibly	no

Reflections on best-effort service:

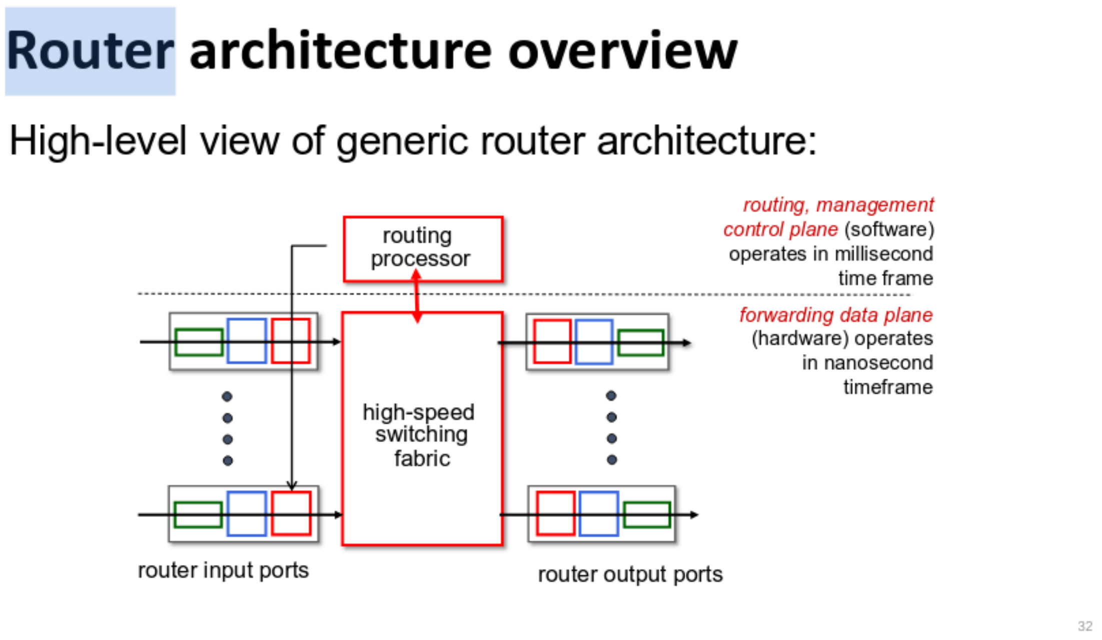
- simplicity of mechanism has allowed Internet to be widely deployed adopted
- sufficient provisioning of bandwidth allows performance of real-time applications (e.g., interactive voice, video) to be “good enough” for “most of the time”
- replicated, application-layer distributed services (datacenters, content distribution networks) connecting close to clients’ networks, allow services to be provided from multiple locations

- congestion control of “elastic” services helps

It’s hard to argue with success of best-effort service model

Router architecture overview

High-level view of generic router architecture:


Router architecture overview
High-level view of generic router architecture:
routing processor
routing, management control plane (software) operates in millisecond time frame
forwarding data plane (hardware) operates in nanosecond timeframe
high-speed switching fabric
router input ports
router output ports
32

Input port functions:

```
  +--------------------------------------------------+
  |                  +------------+   +------------+  |
  | +------------+   |            |   |   lookup   |  |
  | |            |   |  link      |   | forwarding |  |
  | |    line    |   | layer      |   |  +-+++++   |  |
──→|             |──→| protocol   |──→|  | |||||   |--+──→ switch fabric
  | |termination|   | (receive)  |   |  +-+++++   |  |
  | +------------+   |            |   |  queueing  |  |
  |                  +------------+   + -----------+  |
  +--------------------------------------------------+                +
```

line: termination: bit-level reception

link layer: e.g., Ethernet

decentralized switching: using header field values, lookup output port using forwarding table in input port memory (“match plus action”)

- goal: complete input port processing at ‘line speed’
- input port queuing: if datagrams arrive faster than forwarding rate into switch fabric
- destination-based forwarding: forward based only on destination IP address (traditional)
- generalized forwarding: forward based on any set of header field values

longest prefix match: when looking for forwarding table entry for given destination address, use longest address prefix that matches destination address.

- longest prefix matching: often performed using ternary content addressable memories (TCAMs)

- content addressable: present address to TCAM: retrieve address in one clock cycle, regardless of table size
- Cisco Catalyst: ~1M routing table entries in TCAM

Switching fabrics

transfer packet from input link to appropriate output link

switching rate: rate at which packets can be transfer from inputs to outputs

- often measured as multiple of input/output line rate
- N inputs: switching rate N times line rate desirable

Switching via memory: first generation routers:

- traditional computers with switching under direct control of CPU
- packet copied to system's memory
- speed limited by memory bandwidth (2 bus crossings per datagram)

Switching via a bus

- datagram from input port memory to output port memory via a shared bus
- bus contention: switching speed limited by bus bandwidth
- 32 Gbps bus, Cisco 5600: sufficient speed for access routers

Switching via interconnection network

- Crossbar, Clos networks, other interconnection nets initially developed to connect processors in multiprocessor
- multistage switch: nxn switch from multiple stages of smaller switches
- exploiting parallelism:
  - fragment datagram into fixed length cells on entry
  - switch cells through the fabric, reassemble datagram at exit
- scaling, using multiple switching "planes" in parallel: speedup, scaleup via parallelism

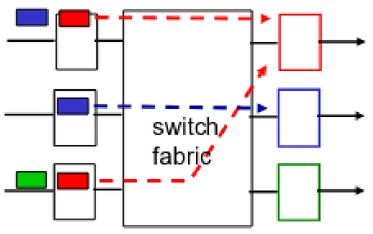
Cisco CRS router:

- basic unit: 8 switching planes
- each plane: 3-stage interconnection network
- up to 100's Tbps switching capacity

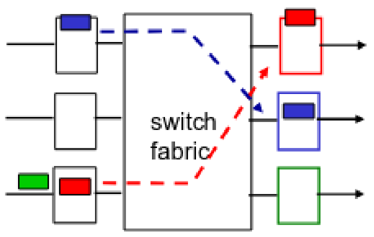
Input port queuing

If switch fabric slower than input ports combined -> queueing may occur at input queues: queueing delay and loss due to input buffer overflow!

Head-of-the-Line (HOL) blocking: queued datagram at front of queue prevents others in queue from moving forward



output port contention: only one red datagram can be transferred. lower red packet is *blocked*



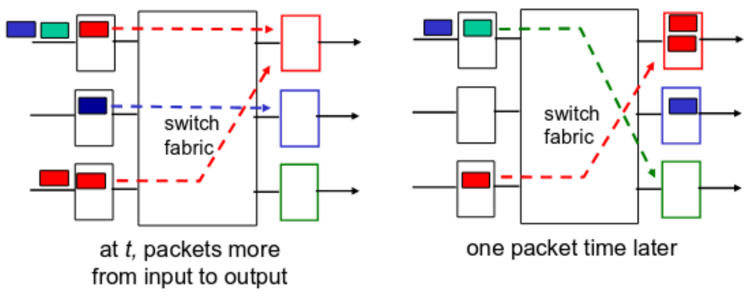
one packet time later: green packet experiences HOL blocking

Output port queuing

Buffering required when datagrams arrive from fabric faster than link transmission rate. Drop policy: which datagrams to drop if no free buffers?
Datagrams can be lost due to congestion, lack of buffers.

Scheduling discipline chooses among queued datagrams for transmission
Priority scheduling – who gets best performance, network neutrality

Output port queuing



- buffering when arrival rate via switch exceeds output line speed
- *queueing (delay) and loss due to output port buffer overflow!*

50

Buffer Management
buffer management:
- drop: which packet to add, drop when buffers are full
  - tail drop: drop arriving packet
  - priority: drop/remove on priority basis
- marking: which packets to mark to signal congestion (ECN, RED)

Packet Scheduling: FCFS packet scheduling: deciding which packet to send next on link
- first come, first served
- priority
- round robin
- weighted fair queueing

FCFS: packets transmitted in order of arrival to output port
- also known as: First-in-first-out (FIFO)
- real world examples?

Priority scheduling:
- arriving traffic classified, queued by class
- any header fields can be used for classification
- send packet from highest priority queue that has buffered packets
- FCFS within priority class

Round Robin (RR) scheduling:

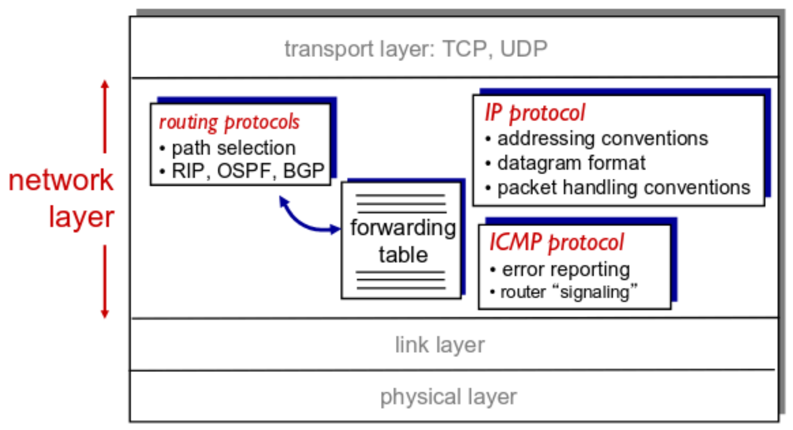
- arriving traffic classified, queued by class
- any header fields can be used for classification
- server cyclically, repeatedly scans class queues, sending one complete packet from each class (if available) in turn

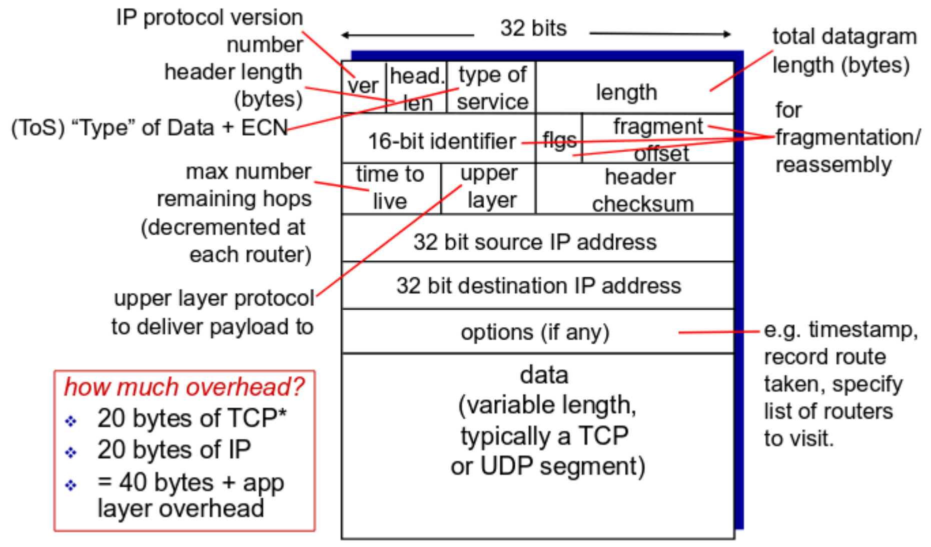
Weighted Fair Queuing (WFQ):
- generalized Round Robin
- each class, i, has weight, wi, and gets weighted amount of service in each cycle: $\frac{W_i}{\Sigma_j w_j}$
- minimum bandwidth guarantee (per-traffic-class)

The Internet network layer

The Internet network layer

Host, router network layer functions:




IP protocol version number
header length (bytes)
(ToS) "Type" of Data + ECN
max number remaining hops (decremented at each router)
upper layer protocol to deliver payload to
32 bits
ver
head. len
type of service
length
16-bit identifier
flgs
fragment offset
time to live
upper layer
header checksum
32 bit source IP address
32 bit destination IP address
options (if any)
data (variable length, typically a TCP or UDP segment)
total datagram length (bytes)
for fragmentation/ reassembly
e.g. timestamp, record route taken, specify list of routers to visit.

how much overhead?
- 20 bytes of TCP*
- 20 bytes of IP
- = 40 bytes + app layer overhead

IP datagram format

*sometimes the optional header will be used, the overhead of TCP will be more.

IP fragmentation, reassembly

- Network links have MTU (max.transfer size): largest possible link-level frame: Different link types, different MTUs
- Large IP datagram divided (“fragmented”) within net
  - One datagram becomes several datagrams
  - “Reassembled” only at final destination
  - IP header bits used to identify, order related fragments