Outgoing email - SMTP
Retrieving email - POP3, IMAP

```
S: 220 mr1.its.yale.edu
C: HELO cyndra.yale.edu
S: 250  Hello cyndra.cs.yale.edu, pleased to meet you
C: MAIL FROM: <spoof@cs.yale.edu>
S: 250 spoof@cs.yale.edu... Sender ok
C: RCPT TO: <yry@yale.edu>
S: 250 yry@yale.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Date: Wed, 23 Jan 2008 11:20:27 -0500 (EST)
C: From: "Y. R. Yang" <yry@cs.yale.edu>
C: To: "Y. R. Yang" <yry@cs.yale.edu>
C: Subject: This is subject
C:
C: This is the message body!
C: Please don' t spoof!
C:
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 mr1.its.yale.edu closing connection
```

Pros of email design

❍ separate protocols for different functions
• email retrieval (e.g., POP3, IMAP)
• email transmission (SMTP)
❍ simple/basic requests to implement basic control; finegrain control through ASCII header and message body
• make the protocol easy to read/debug/extend (analogy with end-to-end layered design?)
❍ status code in response makes message easy to parse

Email security
Sender Policy Frame (SPF) - check in channel
DomainKeys Identified Mail (DKIM) - check in sender and receiver
Frequency division multiplexing(FDM)
Time division multiplexing(TDM)
Code division multiplexing(CDM)
Queue theory
Client requests arrival rate lambda/second
Service rate: each call takes on average 1/mu second
Circuit switching

#(transitions k → k+1) = #(transitions k+1 → k)

$$p_k\lambda = p_{k+1}(k+1)\mu$$

$$p_{k+1} = \frac{1}{k+1}\frac{\lambda}{\mu}p_k = \frac{1}{(k+1)!}\left(\frac{\lambda}{\mu}\right)^{k+1}p_0$$

$$p_0 = \frac{1}{1+\frac{1}{1!}\frac{\lambda}{\mu}+\frac{1}{2!}\left(\frac{\lambda}{\mu}\right)^2+\ldots+\frac{1}{N!}\left(\frac{\lambda}{\mu}\right)^N}$$

Packet switching

at equilibrium (time reversibility) in one unit time:
#(transitions k → k+1) = #(transitions k+1 → k)

$$p_k\lambda = p_{k+1}\mu$$

$$p_{k+1} = \frac{\lambda}{\mu}p_k = \left(\frac{\lambda}{\mu}\right)^{k+1}p_0 = \rho^{k+1}p_0 \qquad p_0 = 1-\rho$$

$$\rho = \frac{\lambda}{\mu}$$

$$\rho = \frac{\lambda}{\mu} \qquad S = \frac{1}{\mu}$$

$$\text{average queueing delay: } w = S\frac{\rho}{1-\rho}$$

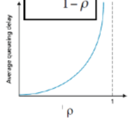
$$queueing + trans = S\frac{\rho}{1-\rho} + S = S\frac{1}{1-\rho}$$

Assume:
R = link bandwidth (bps)
L = packet length (bits)
S = L / R
a = average packet arrival rate (pkt/sec)

$$utilization: \ \rho = \frac{a}{1/S} = aS$$

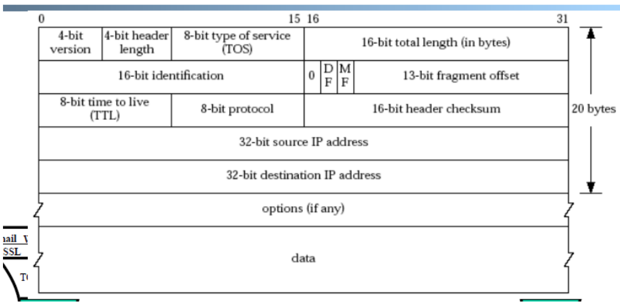
$$w = S\frac{\rho}{1-\rho}$$

❐ ρ ~ 0: average queueing delay small
❐ ρ -> 1: delay becomes large
❐ ρ > 1: more "work" arriving than can be serviced, average delay infinite !

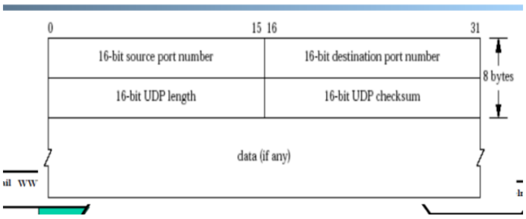


pros of packet switching:
Statistical multiplexing, more efficient bandwidth usage.
Cons: Congestion, packet header overhead, per packet processing overhead.

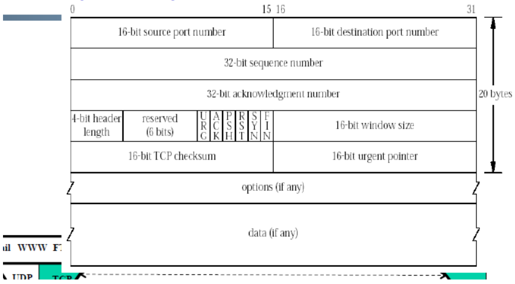
Network Layer: IPv4 Header



Transport Layer: UDP Header

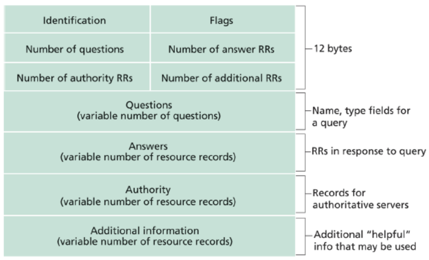


Transport Layer: TCP Header



Key questions to ask about a C-S application
Extensible? Scalable? Handle failures(robust)? Security?
DNS: Port 53, over UDP

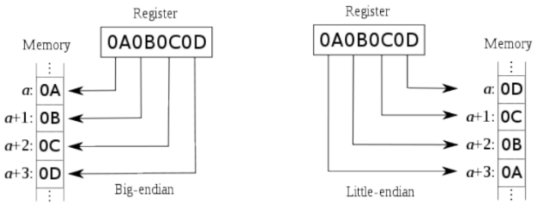
DNS protocol : typically over UDP (can use TCP); *query* and *reply* messages, both with the same *message format*



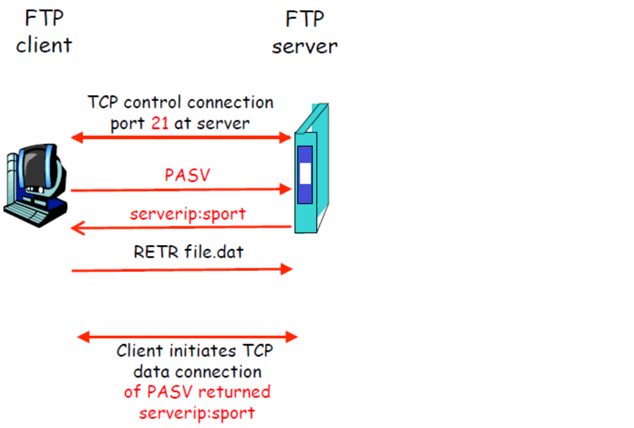
Pros of DNS design
Hierarchical delegation avoids central control, improving manageability and scalability
❐ Redundant servers improve robustness
❐ Caching reduces workload and improves robustness

Cons:
Domain names may not be the best way to name other resources, e.g. files
❐ Simple query model makes it hard to implement advanced query
❐ Relatively static resource types make it hard to introduce new services or handle mobility
❐ Although theoretically you can update the values of the records, it is rarely enabled
❐ Early binding (separation of DNS query from application query) does not work well in mobile, dynamic environments

TCP_UDP
Network endian: big-endian
Intel x86 - little-endian



FTP
Use telnet for the control channel
use nc (NetCat) to receive/send data
FTP PASV: Server Specifies Data Port,Client Initiates Connection



HTTP
Http is stateless, comparing to FTP
Speed up http protocol
❒ Reduce the number of objects fetched [Browser cache] - if-modified-since
Response 304, not modified
❒ Reduce data volume [Compression of data]
❒ Reduce the latency to the server to fetch the content [Proxy cache]
Forward proxy server - behave like client
Reverse proxy server - behave like server
Keep consistency
❍ pull
• Web caches periodically pull the web server to see if a document is modified
❍ push
• whenever a server gives a copy of a web page to a web cache, they sign a lease with an expiration time; if the web page is modified before the lease, the server notifies the cache
❒ Increase concurrency [Multiple TCP connections]
❒ Remove the extra RTTs to fetch an object [Persistent HTTP, aka HTTP/1.1]
❒ Asynchronous fetch (multiple streams) using a single TCP [HTTP/2]
❒ Server push [HTTP/2]
❒ Header compression [HTTP/2]

Background: Little's Law (1961)

- For any system with no or (low) loss.
- Assume
  - mean arrival rate λ, mean time R at system, and mean number Q of requests at system
- Then relationship between Q, λ, and R:

$$Q = \lambda R$$

Asyn server
In addition to management threads, a system may still need multiple threads for performance (why?)
-FSM code can never block, but page faults, file io, garbage collection may still force blocking
-CPU may become the bottleneck and there maybe multiple cores supporting multiple threads (typically 2 n threads)

Operational Quantities

- T: observation interval
- Bi: busy time of device i
- i = 0 denotes system
- Ai: # arrivals to device i
- Ci: # completions at device i

$$\text{arrival rate } \lambda_i = \frac{A_i}{T}$$

$$\text{Throughput } X_i = \frac{C_i}{T}$$

$$\text{Utilization } U_i = \frac{B_i}{T}$$

$$\text{Mean service time } S_i = \frac{B_i}{C_i}$$

Utilization $U_i = \frac{B_i}{T} = \frac{C_i}{T} * \frac{B_i}{C_i} = X_i S_i$

- Assume flow balanced (arrival=throughput), Little's Law:

$$Q = \lambda R = XR$$

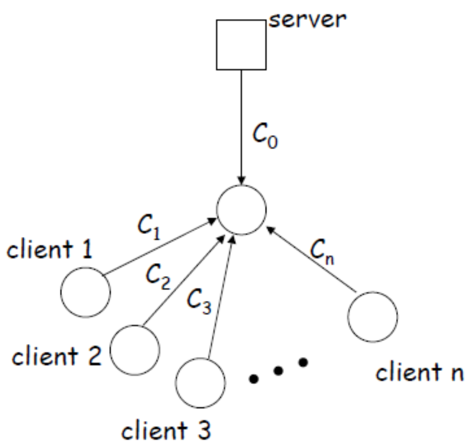
- Assume PASTA (Poisson arrival--memory-less arrival--sees time average), a new request sees Q ahead of it, and FIFO

$$R = S + QS = S + XRS$$

- According to utilization law, U = XS

$$R = S + UR \longrightarrow R = \frac{S}{1-U}$$

Why Multiple Servers?
❒ Scalability
❍ Scaling beyond single server capability
• There is a fundamental limit on what a single server can
– process (CPU/bw/disk throughput)
– store (disk/memory)
❍ Scaling beyond geographical location capability
• There is a limit on the speed of light
• Network detour and delay further increase the delay
LB/NAT
Pro: Only one public IP address is needed for the load balancer; real servers can use private IP Addresses; Real servers need no change and are not aware of load balancing
cons:The load balancer must be on the critical path and hence may become the bottleneck due to load to rewrite request and response packets. Typically, rewriting responses has more load because there are more response packets
So we configure real servers and load balancer to use same ip address.
Configure address resolution protocol



R = min{C0, (C0+ΣCi)/n}
Commands
Telnet
Traceroute
Dig <type> <domain> Type: MX, A, TXT, NS, CNAME