Packet Switching vs Circuit switching

The fundamental question in networking;
How is data transferred through the net?

circuit switching:

- End to end transfer
- all resources used
- Dedicated allocation
- reserved resources

packet switching:

- chunks sent over the line at a time
- resources used in chunks
- If the number of packets arriving exceeds the number that can be serviced, they are dropped
- on-demand allocation

Great for: bursty data $-->$ resource sharing, simpler, no call setup
Bad for: Applications with hard resource requirements
Excessive congestion: packet delay and loss
Need protocols for reliable data transfer, and congestion control

How can we provide circuit-like behavior?
Bandwidth guarantees needed for audio/video apps
Common solution: over-povision

Packet Switching: Statistical Multiplexing

We can use statistics to pick the best option for scheduling of packets. This will lower the amount of time that is wasted while elevating the speed things are transmitted
Packet Switching allows more users to use the network by exploiting the randomness of the users.

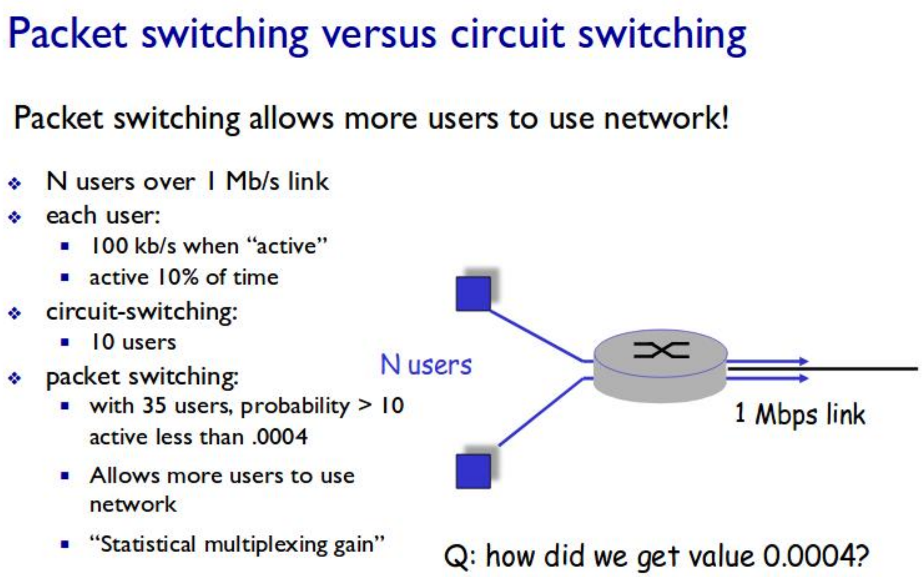

Figure 1: all the math

1 Mbps / Each user needs 100 kbps = 10^6 bps / $100x10^3$ bps
Probability that a user is active: 10%

4 Types of delay

processing $-->$ determine output link, check bit errors

queueing $-->$ biggest portion of delay, it is the slowest
$-->$ time waiting at output link for transmission

transmission
R$-->$ link bandwidth in bits per second
L$-->$ packet length in bits
$D_t = L/R$

propagation
d = length of physical link
s = propagation speed in medium
$D_{prop} = d/s$

EX; Propagation Delay:
Packet of length 1024 bytes
$-->$ $1024 * 8$ bits
$-->$ 8192 bits

Over 27 kbps dial up modem access limit
$-->$ $\frac{8192}{27*10^3}$ sec
$-->$ 0.303 sec
Over 10 Gbps Ethernet
$-->$ $\frac{8192\ bits}{10*10^9\ bits/sec}$
$-->$ 8192 nano sec
$-->$ $8192 * 10^{-7}$

Nodal Delay $= D_N = D_p + D_q + D_t + D_{prop}$

EX; Queueing Delay:
traffic intensity $= \frac{La}{R}$

Overview

Overview

Network apps

Client Server architecture
EX. Google Data Centers

Pure Peer to Peer architecture
no always on servers
Arbitrary end systems directly communicate
peers are intermittently connected and change IP addresses

Pro: Highly scalable
Con: Difficult to manage
Unreliable
Lookup/Discovery

Hybrid of client-server and P2P
Skype: Voice over is p2p
Chat/messages is client-server

Processes communication

Sockets
Process sends/receives messages to and from it's socket
Sockets are like doors; messages can come or go from them

Addressing processes
To receive messages, process must have identifier
host device a 32 bit IP address
Identifier includes both IP address and port numbers associated to process on host
HTTP server: Port 80
Mail server: Port 25

App Layer Protocol defines

What traspoirt services does an app need? Data Loss audio can tolerate some loss file transfers require 100% reliable data transfer

Timing some apps require low delay to be effective EX. Gaming

Throughput

some apps require minimm amount of throughput to be effective
other apps make use of what they get
EX. Tweet; does not require strong through put EX. Video streaming MUST have strong through put

Security
Encryption, data integrity

Transport service requirements of common apps
Jitter: variation in average delay

Application

Internet Transport protocols services
TCP service
connection oriented reliable transport flow control

UDP service

Streaming multimedia −− > TCP, UDP, or SCTP

SCTP −− > TCP friendly congestion control but not reliable data transfer

Application Layer

Web and HTTP
web page consists of objects
base HTML file which includes several referenced objects
Each object is addressable by a URL

HTTP Overview
HTTP: hypertext transfer protocol
client: browser that requests, receives, displays web objects
server: web server sends objects in response to requests
Uses TCP
HTTP is 'stateless'
−− > Protocols that maintain state are complex
Client sends a TCP SYN Server sends an ack, client sends an http request.
non persistent HTTP

At most, one object is sent over TCP connection

Non persistent HTTP without piplining:
Client sends TCP SYN
Server sends TCP SYN and ACK
C sends HTTP Request
s sends object
c sends TCP FIN
c deciphers if it needs to ask for more objects;
C sends TCP SYN
s sends TCP SYN and ACK
c requests object 1
s sends object 1
c sends TCP FIN
c sends TCP SYN
s sends TCPY SYN and ACK
c requests object 2
s sends object 2
c sends TCP FIN
Every time the client recieves it's at 1RTT, so it totaled upto 6RTT for the whole interaction

persistent HTTP
Multiple objects can be sent over single TCP connection between a client and server
Client sends TCP SYN
Server sends TCP SYN and ACK
c requests http
s sends http
c requests object 1
s sends object 1
c requests object 2
s sends object 2
c sends TCP FIN
4 RTT for this interaction

Non persistent HTTP with piplining C sends syn
s sends syn and ack

c sends http request
s sends http
c sends TCP FIN
Client sees there are two objects;
C sends SYN
S sends SYN and ACK
C requests object 1
c requests object 2
s sends object 1
s sends object 2
c sends TCP FIN
4 RTT for this interaction

The best option is persistent with piplining.
With pipelining, the most objects we can request/send is 3 objects

If the objects are varied in size, you have to look at adding the transmission times for all 3 objects seperately

HTTP response status codes:
200 0k
301 Moved Permanently

User Server State: Cookies
Cookies are used to keep state

Overview

NS record; (domain name, authoritative name, NS)
A record; (authoritative name, IP address, A)
MX record; (domain name, authoritative, MX)