**Peer 2 Peer Networks**

**Client-Server**: Web server always up, send a message, connects, give files / objects that you request at point of time

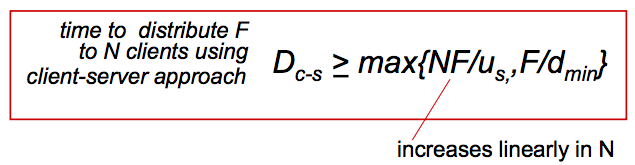
**P2P**: Every host plays client-server, server will not always be on, IP addresses can be changed.

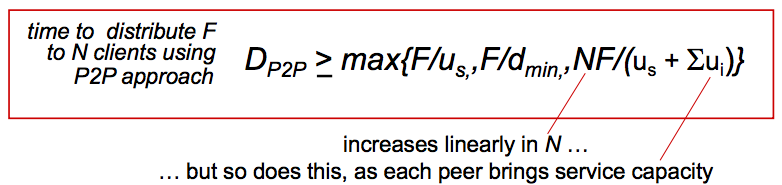
* Examples: BitTorrent (file distribution) and Skype (VoIP / Voice Over IP)

Performance: Client-server VS. P2P

* How much time to distribute file from one server to N peers? Peer upload/download capacity is a limited resource.
* Generally, upload capacity can be a bottleneck.
* **US** = Server upload capacity | **Ui** = Peer *i* upload capacity | **Di** = Peer *i* download capacity

File distribution time: client-server

* **Server transmission**: must sequentially send / upload N file copies.
  + Time to send one copy = **F / US**
  + Time to send N copies = **N\*F / Us**
* **Client:** each client must download file copy.
  + Minimum client download rate = **Dmin**
  + Minimum client download time = **F / Dmin**

File distribution time: P2P

* **Server transmission**: must upload at least one copy
  + Time to send one copy = **F / US**
* **Client**: each client must download file copy
  + Minimum client download time = **F / dmin**
* **Clients**: As aggregate must download **N\*F bist**
  + Max upload rate (limiting max download rate) is **US + SUM( Ui )**

Client-Server vs P2P Graph Example (refer to pg. 7 slides)

* Client-Server: As N clients increase, minimum distribution time F **increases linearly**
* P2P: As N clients increase, minimum distribution time **increases logarithmically** (F slows down)
* **Summary**: In P2P, there is much better performance as # of users increases and you can distribute your files to a large number of users quickly.

Real P2P example: BitTorrent

* File is divided into 256kb chinks / peers in torrent send and receive file chunks
* **Tracker**: keeps track of peers participating in torrent
* **Torrent**: group of peers exchanging chunks of a file
* *Alice arrives 🡪 obtains list of peers from Tracker 🡪 Begins exchanging file chunks with peers in Torrent*

.torrent files

* Contains address of trackers for the file 🡪 Where can I find out other peers?
* Contains a list of file chunks and their cryptographic hashes 🡪 Ensures integrity of files

Peer joining torrent:

* No chunks, but will accumulate them over time from other peers.
* Registers with tracker to get list of peers, connects to a subset of peers “neighbours”
* While downloading, peers upload chunks to other peers.
* Peer may change peers with whom it exchanges chunks / once peer has whole file, they can selfishly leave or remain

BitTorrent: requesting, sending file chunks

* **Requesting Chunks**: At any time, different peers have different subsets of file chunks
  + **Why Rarest First?** Fewer copies around the peers, so if they leave they will be no-one to upload them.
* **Sending Chunks**: Alice sends chunks to those peers currently sending her chunks at highest rate. Other peers are **Choked**
  + Re-evaluate top peers every 10 seconds
  + Every 30 seconds, randomly select another peer, starts sending chunks.
  + I.e. if you are a new peer, you need to wait randomly to be **Un-chocked**
* **Higher Upload Rate = match with better peers, get files faster**

Can you not upload any data to other peers but also download a complete copy of the file?

* If a lot of people in the network have the complete copy of the file, you would be **un-chocked** at some point and can download the file regardless of not uploading.  
  **ANS: Eventually you will be randomly selected by other peers enough times to download the complete copy**

Distribute tracker info using a **Distributed Hash Table (DHT):** Maps keys to a value

* **Centralised HT**: All (key, value) pairs on **1 node**
* **Distributed HT**: Each node as a **section** of (key, value) pairs

DHT is a distributed P2P database.

* Database has (key, value) pairs, e.g:
  + Key: TFN number | Value: human name key: file name | Value: BT tracker peer(s)
* Distribute the key/value pairs over millions of peers
* A peer **queries DHT with the key:** DHT returns values that match the key (value such as IP address etc.)
* Peers can also **insert key/value pairs**

DHT Challenges

* How do we assign Key/Value pairs to nodes? **Run a hash function to generate a number, link it with the value**
  + Convert each key to an int 🡪 assign each int to a peer 🡪 put (key,value) in the peer that is **closest** to the key
  + How to convert key to an int: Use hashing.
* How do we find them again quickly? **Search should be quick.**
* What happens if nodes join/leave? **Need to be resilient. Noes joining/leaving should not collapse system.**

DHT identifiers: **Consistent Hashing**

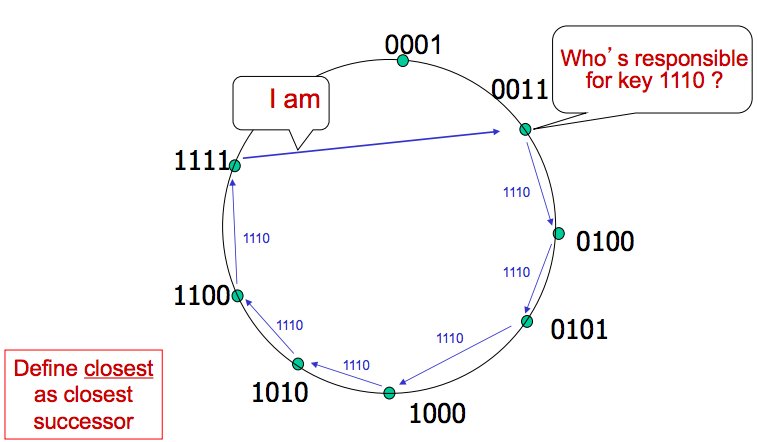
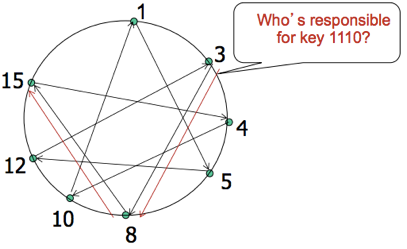
* **Assign integer identifier (node ID) to each peer in rage [0, 2N-1]** **for some n-bit hash function.**
  + E.g. node ID is hash of its IP address
* Require each key to be an integer in the same range
* To get integer key, hash original key. E.g. **key = hash(“House of Cards Season 4”)** 🡪 from there you build the DHT

DHT: Assigning keys to peers

* Rule: assign key to peer that has the **Closest ID** = the **immediate successor of the key**
* E.g. n=4 | peers: 1, 3, 4, 5, 8, 19, 12, 14
  + Key = 13, then successor peer = 14 | key = 15, then successor peer = 1

**Circular DHT**: Each peer is only aware of their immediate successor and predecessor (see pg. 21 of slides for graph)

* **Overlay Network**: Network at the application layer



Note: This is NOT a physical network. It is just a representation of network of peers.

* i.e. 0001 and 0011 are located in Thailand while 0100 is located in SYD

How this works:

1. Connect nearest peer

2. Start with search process for key

3. Searches in that loop until it reaches 1111

4. 111 responds back saying they have the file

* Problem: Inefficient to go through many queries to find who owns the file

Solution to previous problem: **Circular DHT with shortcuts**

* Each peer keeps track of IP addresses of predecessor, successor and short cuts.
* Reduced from 6 to 2 messages to find out who has the file
* Possible to design shortcuts so **O(LogN) neighbours, O(LogN) messages in query.**

**Peer Churn** / handling peer churn

* Each peer knows address of its two successors
* Each peer periodically pings its two successors to check aliveness. **KEEP-ALIVE.**
* If immediate successor leaves, choose next successor as new immediate successor
  + E.g. P4 detects P5 departure, makes P6 its immediate successor.  
    Asks P6 who its immediate successor is, makes P6’s immediate successor its secondary successor

More to learn: **How do nodes join? How does crypto hashing work? How much state does a node store?**

* Look at Open Learning DHT research paper.

**Choking**

Typically, cooperation is achieved when upload bandwidth is exchanged for download bandwidth. Therefore, when a peer in the network is not uploading in return to our own peer uploading, BitTorrent program will choke the connection with the uncooperative peer and allocate this upload slot to a more cooperative peer.

**Optimistic Unchoking**

A peer may allocate an upload slot to a randomly chosen uncooperative peer, allowing search for more cooperating peers and gives a second chance to previous non-cooperating peers.

Another def: A peer periodically selects one of its neighbours at random as a peer for uploading, whether or not this neighbour is uploading data or not. Alice may be chosen at random as a result of this, hence will receive her first chunk.

Note: Many people confuse speed rates with memory. Memories are always powers of 2, so a GigaBYTE of memory would be 1024 MegaBYTES. However, speeds are not powers of 2 and therefore a Gigabit is composed of 1000 Megabits.

**Transport Layer Protocol – PART 1**

Goals:

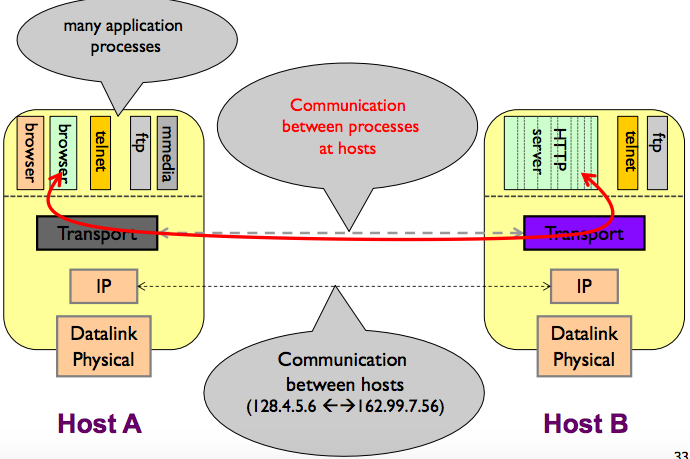
* Understand Principles behind transport layer services
  + Multiplexing / de-multiplexing, reliable data transfer, flow control, congestion control
* Learn about Internet transport layer protocols
  + UDP (connectionless transport), TCP (Connection-Orientated Reliable Trans), TCP congestion control

Network Layer (layer below transport)

* Is responsible for finding a route from one end host to the other
* It doesn’t provide reliable transfer, doesn’t guarantee paths, can’t request particular transfer rates
* For now, assume the network as giving us an “API” with one function: **sendtohost(data, host)**
* The transport layer provides communication between hosts

Transport Services and Protocols

* Provides **logical communication** between app processes running on different hosts
* Transport Protocols run in end-systems:
  + **Send Side:** breaks msgs into segments, passes to network layer
  + **RCV Side**: Re-assembles segments into msgs, passes to app layer.
  + **Exports Services to app that network layer doesn’t provide**
* Network doesn’t provide receiving services (doesn’t guarantee that data will deliver)



Network Layer

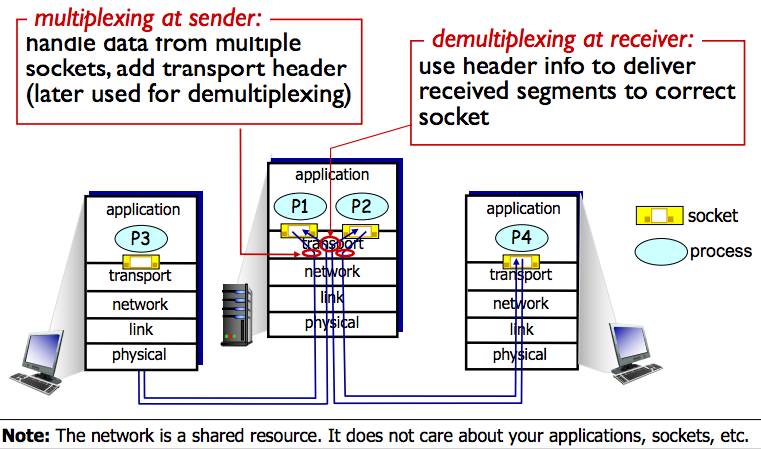
* Communication between two hosts.
* Doesn’t do anything other than deliver from one host to another

Application Layer

* Talking to each other “directly”
* Details are hidden from them

Multiplexing / De-multiplexing

* **Multiplexing:** multiple data streams from diff sources are combined and transmitted over a single shared medium.
* **De-multiplexing:** At the receiving end, the reverse occurs, separating data streams from the single channel and routing to corresponding receivers / destination



**Connectionless De-multiplexing**

Recall: When creating datagram to send to UDP socket, you must specify

* Destination IP address
* Destination Port #

When host receives UDP segment:

* Checks dest port # in segment
* Directs UDP segment to socket with that port #

**IP datagrams with the same destination port # but different source IP addresses and/or source port # will be directed to the SAME SOCKET at dest.**

**Connection-Orientated Demux (TCP)**

TCP socket identified by 4-tuple:

(1) Source IP (2) Source Port # (3) Dest IP (4) Dest Port #

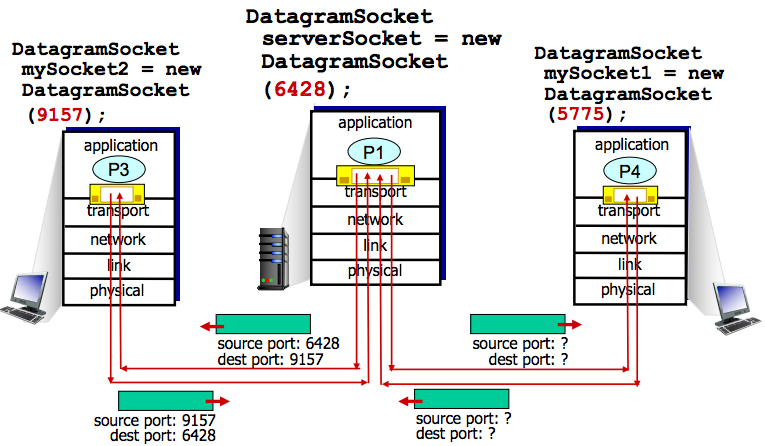
Demux: receiver uses all 4 values to direct segment to appropriate socket.

Server host may support simultaneous TCP sockets

* Each server identified by its own 4-tuple

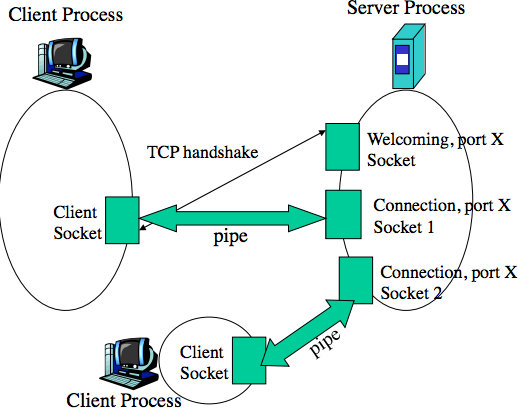
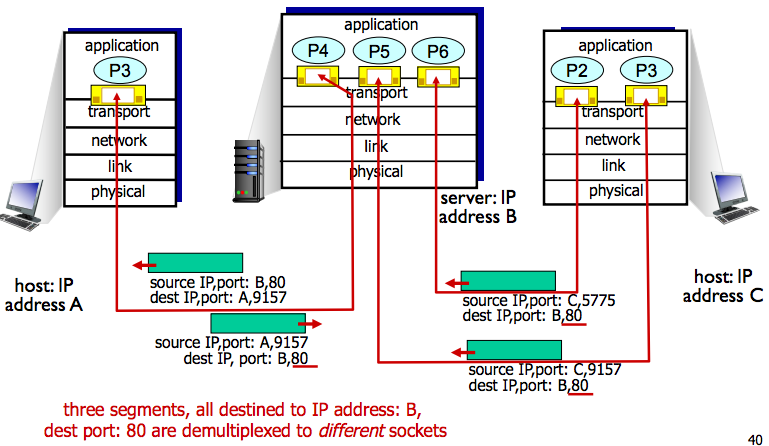
Web servers have different sockets for each client

* Non-persistent HTTP will have different socket for each request.



Revisiting TCP Sockets

* Client Socket 🡪 Handshake 🡪 Server will create new socket for handling communication 🡪 Pipe is created



Quiz: Hosts A and B are sending requests to Web Server Host C

* Are all the requests sent through the same socket at Host?
  + **No. Each request is treated differently** **in TCP, so requests are sent through different sockets at the host**
* If they are being passed through different sockets, do both sockets have port 80?
  + **Yes, both sockets have port 80 for the destination port, but are sent through different sockets.**
  + For each persistent connection, the Web server creates a separate “connection socket”. Each connection socket is identified with a four-tuple: (source IP address, source port number, destination IP address, destination port number). When host C receives an IP datagram, it examines these four fields in the datagram/segment to determine to which socket it should pass the payload of the TCP segment. Thus, the requests from A and B pass through different sockets. **The identifier for both of these sockets has 80 for the destination port; however, the identifiers for these sockets have different values for source IP addresses.** Unlike UDP, when the transport layer passes a TCP segment’s payload to the application process, it does not specify the source IP address, as this is implicitly specified by the socket identifier.