**W1 Quiz + Finish**

1. Packet switching, instead of circuit switching, is generally used to transfer data in the internet. T or F? = **TRUE**

2. Propagation delay depends on the size of the packet. T or F? = **FALSE** (Propagation depends on the type of medium + length)

3. Which of the following delays is significantly affected by load in the network? = **Queuing** **Delay**

* If the load is high, packets will just sit in the queue increasing queuing delay.

4. What is the correct order of delays encountered by a packet until it reaches the next-hop router?

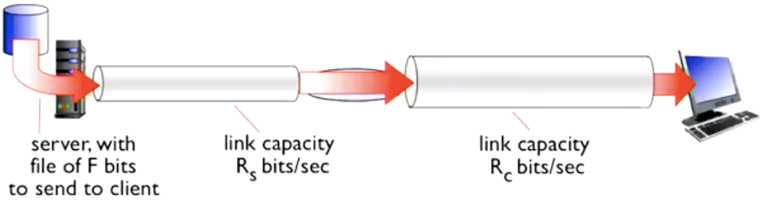
* **Processing, Queuing, Transmission, Propagation**

Packet Loss

* The Queue (buffer) has a finite capacity. A packet which arrives to a full queue will drop.
* A lost packet may be re-transmitted by a previous node, by the source system or not at all.

Throughput

* Throughput is the rate **(bits / time unit)** at which bits are transmitted between the sender / receiver.
  + **Instantaneous**: Rate at a given point in time. (not very useful – may not give the true data)
  + **Average**: Rate over a longer period of time. (more useful – more accurate)



* Different pipe rates:
  + **RS < RC** , what is the average end-end throughput?  
    Average = RS because even if RC is faster, it can’t send any faster than packets arriving at the RS rate.
  + **RS > RC** , what is the average end-end throughput?  
    Average = RC because even of the sane reason above. **(Bottleneck Bandwidth)**
* Throughput: Internet scenario
  + Per-connection end-end throughput: min(RC , RS , R / 10). “10 connections share bottleneck link R bits/sec”
  + Wherever the bottleneck is, you won’t get a faster rate than the bottleneck.

Three networking design steps:

* (1) Break down the problem into tasks (2) Organise these tasks (3) Decide who does what.

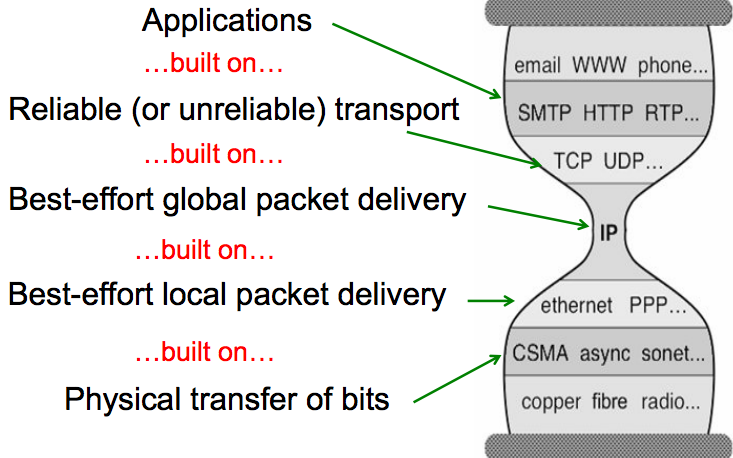
Tasks in networking:

* TASK 1: Send along a single wire
* TASK 2: Stitch these together to go across country

Tasks in networking / Layers

* The table below shows the decomposition of a network (bottom up approach)

|  |  |
| --- | --- |
| **Tasks** | **Layer** |
| 1. Bits on wire | Physical |
| 2. Packets on wire | Physical |
| 3. Deliver frame within local network | Datalink |
| 4. Deliver packets across global network | Network |
| 5. Ensure that packets get to the dest | Transport |
| 6. Do stuff with the data | Application |



* E.g. CEO A sending a letter to CEO B. FedEx will take it to a hub, where it gets sorted then forwarded to the other country.
* CEO doesn’t need to know which hub it goes to, where it will be forwarded to etc.
* Applications in Networks are the “CEO’s” and they need to work with the Transport Layer.

Internet Protocol Stack (stack because they are layered on top of each other)

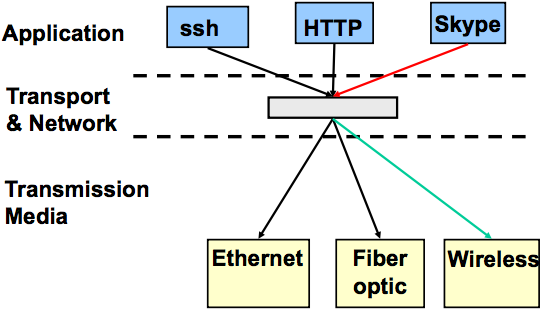
* **Application** **Layer**: Supporting networking applications (FTP, SMTP, HTTP, Skype…)
* **Transport Layer**: Process to process data transfer (TCP, UDP)
* **Network Layer**: (Unifying Protocol) Routing of datagrams from source to destination (IP, Routing Protocols)
* **Link Layer**: Each link that we connect to our own device.
  + Data transfer between neighbouring network elements (Ethernet, 802.111 (WiFi), PPP (Point-to-Point))
* **Physical Layer**: Bits “on the wire”. Can be a physical wire or wireless (WiFi)

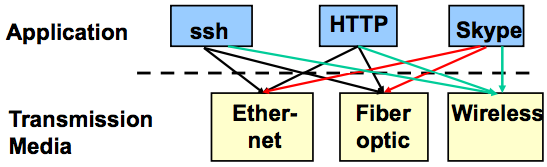
Each layer:

* (1) Depends on the layer below (2) Supports the layer above (3) Independent of others

What are the benefits of layering?

* Abstraction: Applications don’t need to know everything to transmit data to the destination
* API: What hooks are used to provide information to the above layer? What hooks to communicate with layer below?
* **Introduction of an intermediate layer provides a COMMON abstraction for various network technologies**





VS

Can layering be bad?

* Layer N may duplicate lower level functionality (error recovery to re-transmit lost data)
* Information hiding may hurt performance: Packet loss due to corruption vs. congestion
* Headers start to get really big: Typically TCP + IP + Ethernet headers add up to 54 bytes.
* Layer violations when the gains are too great to resist: TCP-over-wireless
* Layer violations when the network doesn’t trust ends: E.g. firewalls

If layering doesn’t exist, each new application (Skype) has to be re-implemented for every network technology = **INEFFICIENT**

Distributing layers across the network

* Layers are simple if only on a single machine: It is just a stack of modules interacting with those above / below
* However, we need to implement layers across machines: Hosts / Routers / Switches

What gets implemented on an end-system (i.e. Host)?

* Bits arrive on wire. All layers must exist at host.

What gets implemented on a router?

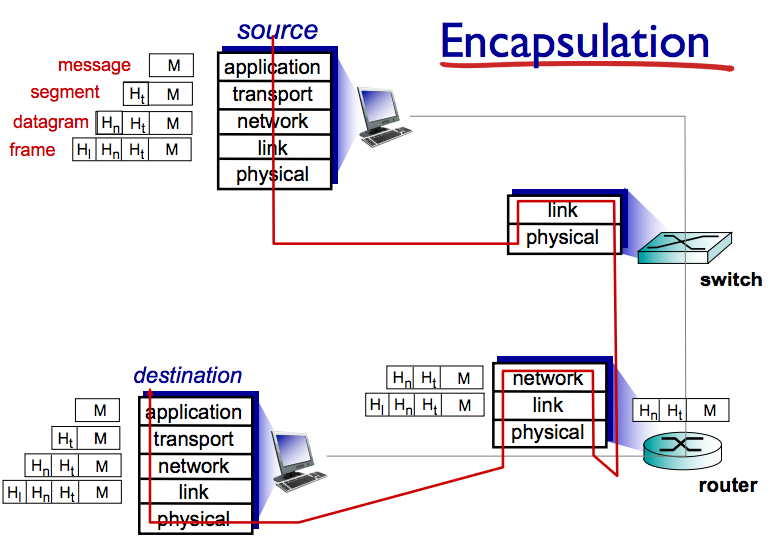
* Bits arrive on wire: Physical layer is unnecessary
* Packets must be delivered to the next-hop: Data-link layer is necessary
* Routers participate in global delivery: Network layer is necessary
* Routers don’t support reliable delivery: Transport Layer (and above) is NOT supported.
* The job of routers/switches is to just forward it to the next-hop. After that, they don’t care about other layers.

Logical Communication: Layers interact with peer’s corresponding layers

* Application Layer, such as HTTP: Two peers (such as a client / server) are interacting with each other. No one else needs to understand anything about HTTP protocol.
* HTTP uses a Transport Protocol (such as TCP). The Transport Layer is done in one end-system and another end-system as an example and they are talking to each other LOGICALLY (not physically, as they are NOT CONNECTED).
* In the Network Layer, the Link Layer and the Physical Layer, the systems are talking to each other.

Physical Communication

* Communication goes downwards to the physical network, then layers get processed, transmitted to another link then communication progresses upwards to the other end’s application layer.



M = bits / file that you want to send

Ht = header / metadata related to T.L

Network layer appends own metadata Hn

= datagram

Link layer appends won metadata Hl

= frame

Finally sent to physical layer.

Switch runs Link + Physical Layer.

Router runs Network + Link + Physical

**The metadata is only treated at their relevant layer.**

When it reaches dest, as it moves up the stack, each header will be stripped off one-at-a-time and passed up the layers.

**Application Layer**

Examples of networked applications: Skype, online games, torrents

Creating a network app:

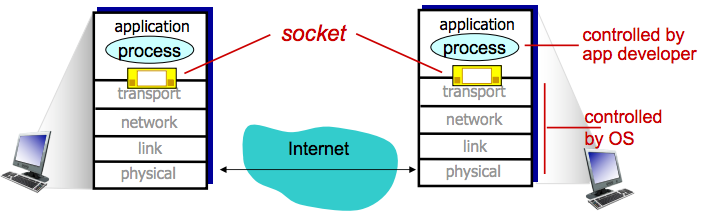
* **Write programs that:** Run on different end systems / communicates over network / E.g. web server software communicates with browser software.
* **Varying degrees of integration:** LOOSE: email, web browsing / MEDIUM: chat, Skype, remote file systems /   
  TIGHT: process migration, distributed file systems
* **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.

**Interprocess Communication (IPC)**

* Processes talk to each other through Interprocess Communication (IPC)
* On a single machine: shared memory / Across machines: we need other abstractions (message passing)

**Sockets**

* Sockets allow communication between two different processes on the same / different machines.
* The Application or Processes sends / receives messages to / from its sockets.
* Sockets = similar to doors:
  + Sending process shoves message out the door
  + Sending process relies on transport infrastructure on the other side of the door to deliver the message to the socket at the receiving process.



**Addressing Processes**

* To receive messages, processes must have an identifier. Host devices have a unique 32-bit IP address.
* QN: Is the IP address of the host on which the process is running enough for identifying the process?
  + ANS: No, many processes can be running on the same host.
* **Identifier = IP Address + Port Number** associated with the process on the host.
  + E.g. HTTP server: port 80 / mail server: port 25
  + E.g. To send HTTPmessage to cse.unsw.edu.au web server: IP address = 129.94.242.51 / Port = 80

**Client-Server Architecture**

* **Server**
  + Exports well-defined requests / respond interface.
  + Long-lived process that waits for requests
  + Upon receiving a request, carries out the request.
* **Clients**
  + Short-lived process that makes requests. It is the “User-side” of the application
  + Initiates the connection

**Client vs. Server**

|  |  |
| --- | --- |
| **Client** | **Serve**r |
| Always-on host  Permanent IP address  Static host conventions (http: 80 | email: 25 | ssh: 25)  Data centres for scaling  May communicate with other servers to respond | May be irregularly connected  May have a dynamic IP address  Do not communicate directly with each other |

**P2P Architecture**

* There is no “always-on” server. No permanent address. There is no single server
* It is a dynamic system, people can join and become part of the peer system, so that can provide and receive a service from other people.
* Arbitrary end systems (peers) directly communicate with each other
* There is symmetric responsibility (unlike client / server)
* Often used for: File sharing (torrents), Gaming, Video Distribution, Video Chat, in general: “distributed systems”

Pros and Cons of P2P

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| **Self-scalability:** new peers bring new service capability, as well as new service demands.  **Speed**: Parallelism, less disagreements  **Reliability**: Redundancy, fault tolerance  **Geographic Distribution** | Fundamental problems of decentralised control:  **Synchronisation**: If one thing changes, everyone else has to update with the new changes too, otherwise it won’t work with new and old copies of the file everywhere. |

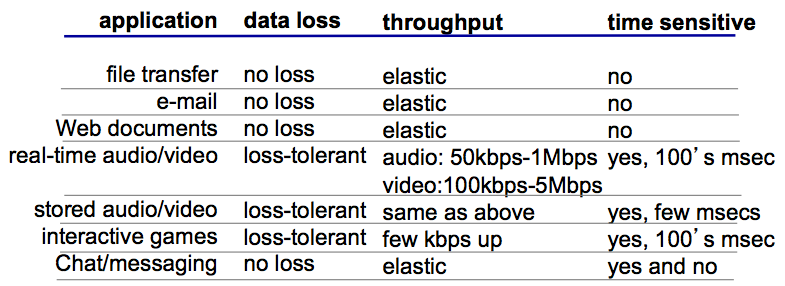
**Application-Layer Protocols**

* **Types of messages**: Request, response
* **Message syntax**: What fields in messages and how fields are delineated.
* **Message semantics**: How do I interpret the information in the fields? What actions do I take?
* **Rules**: For when and how processes send and respond to messages.
* **Open Protocols**: Defined in RFC’s (Request for Comments) / allows for interoperability / HTTP, SMTP.
  + Purpose of these standard protocols: Anyone can develop off these protocols
* **Proprietary Protocols**: Skype, Google etc. who may have developed more efficient technology

**What transport service does an app need? (application layer makes use of the transport layer)**

* **Data Integrity**: No one has tampered with information. Some apps require 100% reliable data transfer.
* **Timing**: Some apps require low delay to be “effective”. E.g. voice apps, online games.
* **Throughput *(units of info processed in a given amount of time)***: Some apps require minimum amount of throughput to be effective. Other apps make use of whatever throughput they get.
* **Security**: Encryption, data integrity

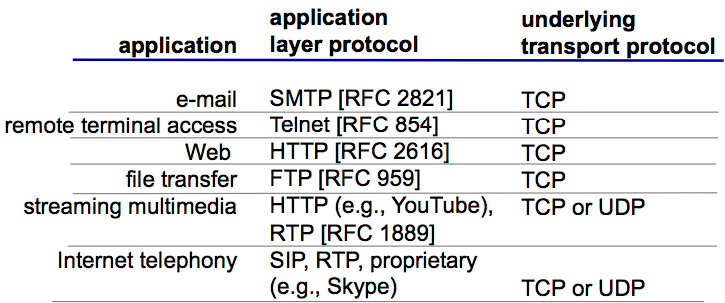
**Transport Service Requirements: Common Apps**



**Internet transport protocol services**

Transmission Control Protocol Service (TCP) vs. User Datagram Protocol Service (UDP) – More on this later.

**Internet apps: Applications, transport protocols**



**NOTE: Applications and Application Layer Protocols are DIFFERENT** **THINGS**

* Applications have a lot more components involved, but they are using standard application layer protocols which make them useable, so even if you use different mail clients (Gmail, Outlook etc.) or servers, you are still able to communicate.
* Applications USE Application Layer Protocols.
* Applications are usually NOT-standardised, but Application Layer Protocols are.

**Web and HTTP**

* A web page consists of objects: HTML files, JPEG images, Java applets, audio files etc.
* It consists of a base HTML file which includes several references objects
* [www.someschool.edu/someDept/pic.gif](http://www.someschool.edu/someDept/pic.gif)  
  hostname | pathname /someDept/pic.gif

**Uniform Resource Locator (URL)**

* FORMAT: Protocol: //host-name[:port]/directory-path/resources
* Protocol: http / ftp / https / smtp / rtsp etcc.
* Hostname: DNS name, IP address
* Port: defaults to protocol’s standard port: e.g. http: 80 https: 443
* Directory path: hierarchical, reflecting file system
* Resource: Identifies the desired resource

**HTTP Overview**

* Web’s application layer protocol
* Client / Server model:
  + Client: browser requests, receives and displays web objects
  + Server: Web server sends objects in response to requests
* It uses TCP (see LAB #2)
  + 1. Client initiates TCP connection (creates socket) to server, port 80
  + 2. Server accepts TCP connection from client
  + HTTP messages exchanged between browser and web server
  + TCP connection closed
* HTTP is a **“stateless**” protocol: Server maintains no info about past client requests
  + E.g. while downloading file, if you connection breaks, you will have to restart the download from the beginning. It doesn’t keep track of what you have already downloaded.
  + Good because it keeps things simple
  + Bad because it can be inefficient such as the example above
* Protocols that maintain **“state”** are complex
  + Packet switching is a protocol that maintains state
  + Past history (state) must be maintained and if the server/client crashes, their views of the state may be inconsistent, so they must be reconciled.

**HTTP request / response messages**

* See the W2 slides. Not examined.

**HTTP is all text**

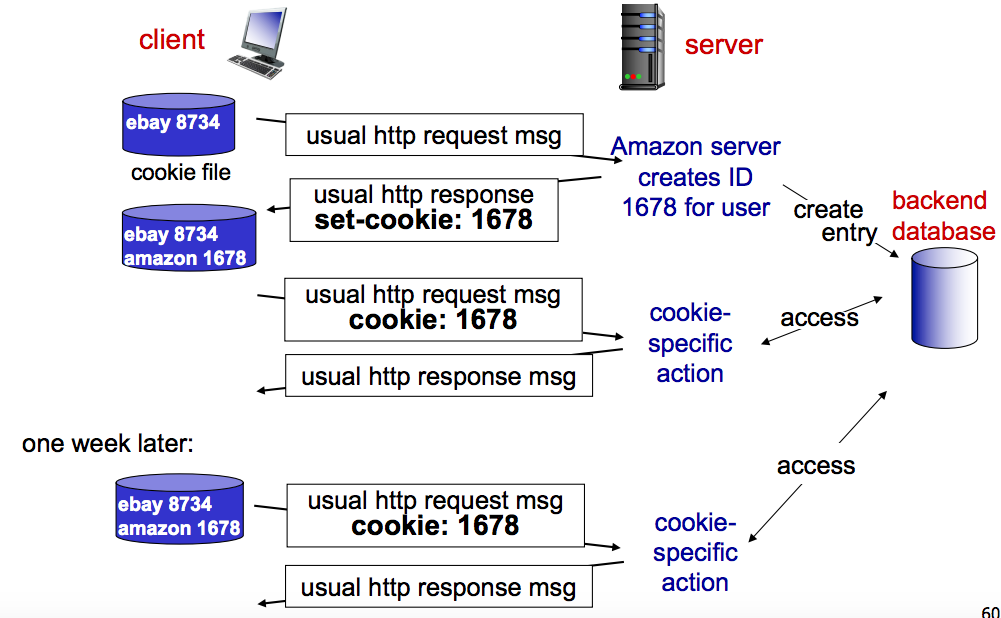
* Makes the protocol simple: easy to delineate message (\r\n), human-readable, no issues with encoding, variable len
* Not the most efficient:
  + Many protocols use binary fields, sending “12345678” as a string is 8-bytes. As an integer it only needs 4 bytes
* Headers may come in any order
* Requires string parsing / processing

**Request Method types**

* HTTP/1.0: GET / POST HEAD
* HTTP/1.1: GET / POST / HEAD / PUT / **DELETE / TRACE, OPTIONS, CONNECT, PATCH (not examined)**
* **POST**: Web page often includes form input (capturing info from user). Input is uploaded to server in <body>
* **GET:** Input is uploaded in URL field of request line: [www.some.com/animalsearch?monkeys&banana](http://www.some.com/animalsearch?monkeys&banana)

**User-server state: COOKIES (Stateful)**

* A cookie is a sort of unique ID.
* Four components of cookies:
  + 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 website.
* E,g, Amazon ecommerce shopping, adding products to cart, keeping track of these products so when you go to check-out all the products will be there.



Bad things about cookies

* Cookies permit sites to learn a lot about you
* You may supply name / email to sites (and more)
* 3rd party cookies (from ad networks etc.) can follow you across multiple sites. (even if you haven’t visited them before)
* You can turn off cookies, but it will reduce the experience of the web a lot more

HTTP Performance Goals

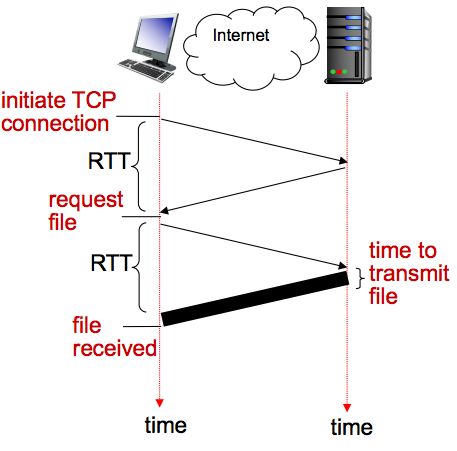
* As a user: you want fast downloads, high availability
* As a content provider: You want happy users (hence above) and a cost-effective infrastructure
* As a network: avoid overload

Solutions to Performance Goals

* **Caching and replication** (replicate content, don’t need to go to origin server every time)
* **Improve HTTP** to achieve faster downloads + compensate for TCP’s weak spots
* **Exploit economies of scale** (webhosting, CDN’s, datacentres)

HTTP Performance

* Most web pages have multiple objects
* How do you retrieve those objects? *One at a time* = **New TCP connection for each small object** = bad
* **Non-persistent HTTP**
  + At most one object sent over TPC connection, connection is then closed.
  + Downloading multiple objects requiring multiple downloads

****Non-persistent HTTP: Response time

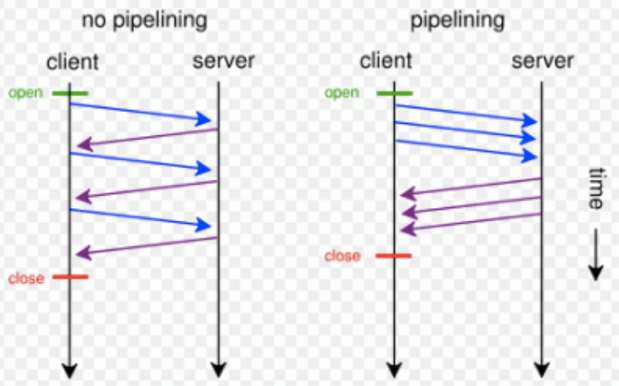
* **Round Trip Time (RTT):** time for small packet to travel from  
  client to server and back
* **HTTP response time**
  + One RTT to initiate TCP connection
  + One RTT for HTTP request and few bytes  
    of HTTP response to return
  + File transmission time
  + **Non-persistent HTTP response time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

Concurrent requests and responses

* Use multiple connections in parallel
* Does not necessarily maintain order of responses.
* Issues with this:
  + **Performance issues.**
  + **Every-time you open a connection, you will be filling the sockets and filling memory very quickly**

Persistent HTTP

* Server leaves TCP connection open after sending response
* Subsequent HTTP messages between same client / server are sent over the same TCP connection
* **Persistent without pipelining:** Client issues new request only when previous response has been received.
  + One RTT for each referenced object
* **Persistent with pipelining**: Default in HTTP/1.1. Client sends requests as soon as it encounters a referenced object
  + As little as one RTT for ALL referenced objects



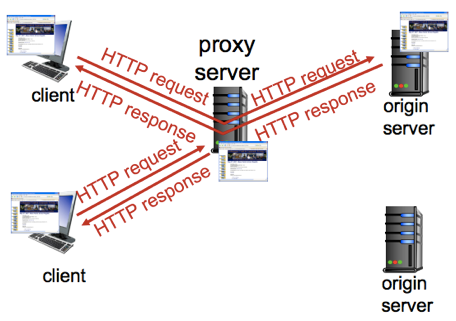
HTTP/1.1: response time (work through the quantitative comparison between persistence vs. non-persistence on OpenLearning)

Improving HTTP Performance: Caching

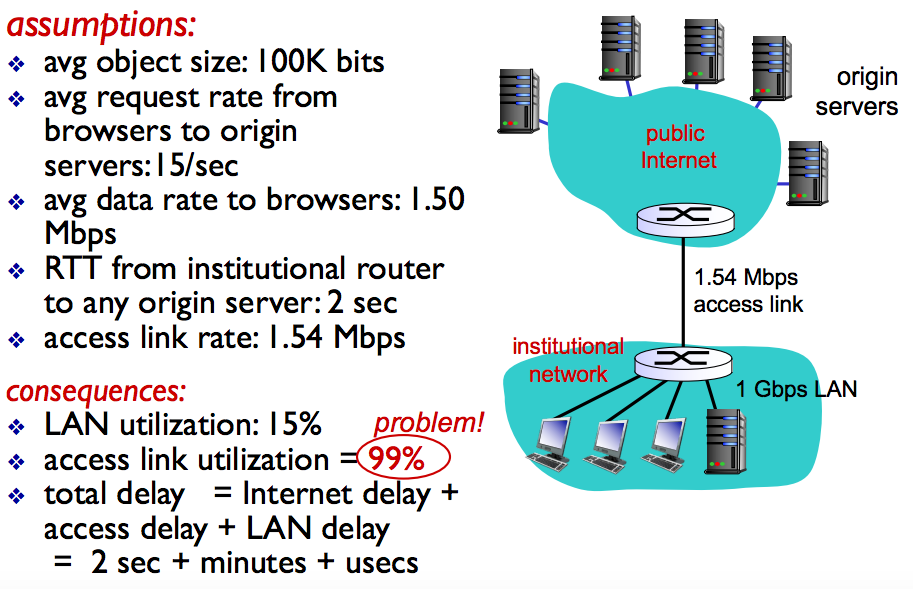
* Why does caching work?
  + **Locality of reference**: Scenario where the same values / related storage locations are frequently accessed
  + Systems with strong locality of reference are great for performance optimisation.
* How well does caching work? Very well, up to a limit because there can be large overlap in content
  + **But if there are many unique requests, then the benefits start to go down and costs up.**

Web caches (proxy server)

* **Proxy Server**: Install server locally so you don’t have to go to the origin server everytime
* GOAL: Satisfy client request without involving origin server
* User sets browser: web accesses via. Cache
* Browser sends all HTTP requests to cache: cache returns object, else cache requests object from origin server, then returns object to client.



**Caching Example (THIS PART IS IMPORTANT – EXAMINABLE)**



15 requests per second

100k (100 \* 103) bits per request

**A = 15 x 100 \* 103 bits**

1.54 Mbps Link Speed  
**B = 1.54 x 106 bits**(1 megabit = 106 bits)

A / B = 0.974 **= 97% access link utilisation**

Access delay is large because we have ~99% utilisation

When utilisation is high, queuing is high (therefore will be in minutes)

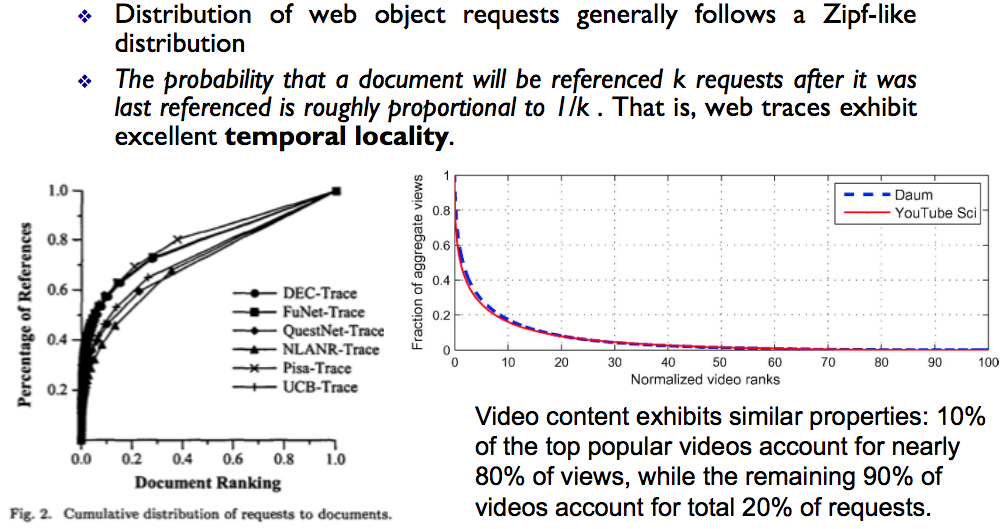
**Solutions to this problem**:

* Have a better access link: Low utilisation, no delay and things can be sent much faster. Downside = really expensive
* Caching: Put in a local cache = **Suppose Cache hit rate is 0.4: 40% requests satisfied at cache / 60% satisfied at origin**

**Re-calculating problem with Caching solution**

* Access link utilisation: 60%
  + Data rate to browsers over access link = 0.6 \* 1.50mbps = 0.9mbps
  + Utilisation = 0.9 / 1.54 = **0.58 (much better than ~99% previously)**
* Total delay = **0.6 \* delay from origin servers + 0.4 \* delay when satisfied at cache**
  + = 0.6 (2.01) + 0.4 (~millisecs)
  + = ~ 1.2 seconds
  + **better performance than with a 154 Mbps link and cheaper**

What is the likehood of cache hits?



Improving HTTP Performance: CDN

* Caching and replication as a service
* Integrate forward and reverse caching functionality
* Large-scale distributed storage infrastructure (usually) administered by one entity
* Combination of PULL caching and PUSH replication
  + **PULL**: Someone loads info on a web server and users use HTTP to pull info from their server at their convenience. TCP connection is initiated by the machine that wants to receive the file.
  + **PUSH**: The sending email server pushes the file to the receiving mail server. TCP is initiated by the machine that wants to send the file.
* Also does some processing
  + Handles dynamic web pages
  + Transcoding
  + Possibly do some security functions – watermarking IP

What about HTTPS?

* HTTP is insecure with basic authentication: password sent using base64 encoding (easily converted to plaintext)
* HTTPS: HTTP over a connection encrypted by Transport Layer Security (TLS)
* Provides: Auth + Bidirectonal encryption
* Widely used in place of plain vanilla HTTP