

Computer Networking

Lent Term

M/W/F 11:00-12:00

LT1 in Gates Building

Handout 4 (Topic 6)

Andrew W. Moore

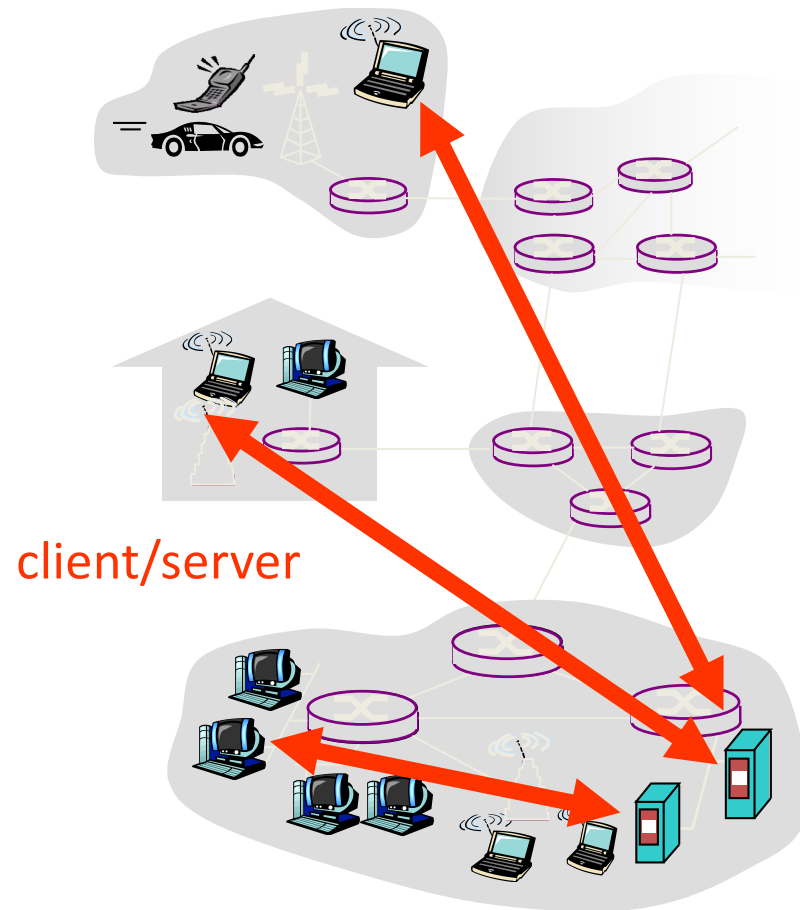
Andrew.Moore@cl.cam.ac.uk

2018-2019

Topic 6 – Applications

- Overview
- Infrastructure Services (DNS)
- Traditional Applications (web)
- Multimedia Applications (SIP)
- P2P Networks

Client-server paradigm

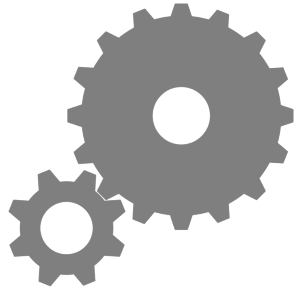


server:

- always-on host
- permanent IP address
- server farms for scaling

clients:

- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other



Relationship Between Names&Addresses

- Addresses can **change** underneath
 - Move `www.bbc.co.uk` to `212.58.246.92`
 - Humans/Apps should be unaffected
- Name could map to **multiple** IP addresses
 - `www.bbc.co.uk` to multiple replicas of the Web site
 - Enables
 - Load-balancing
 - Reducing latency by picking nearby servers
- **Multiple names** for the same address
 - E.g., aliases like `www.bbc.co.uk` and `bbc.co.uk`
 - Mnemonic stable name, and dynamic canonical name
 - Canonical name = actual name of host

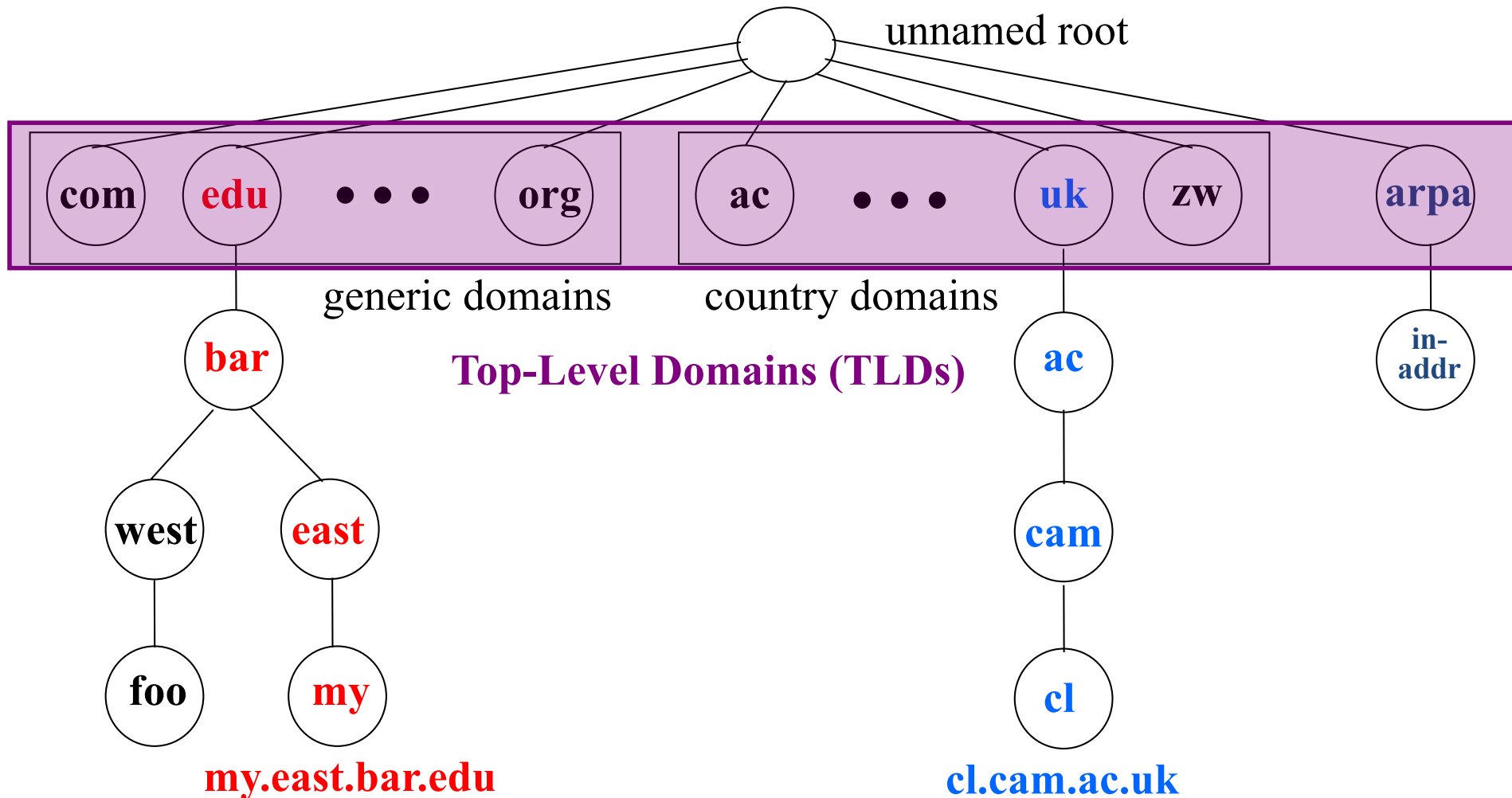
Mapping from Names to Addresses

- Originally: per-host file /etc/hosts
 - SRI (Menlo Park) kept master copy
 - Downloaded regularly
 - Flat namespace
- Single server not resilient, doesn't scale
 - Adopted a distributed hierarchical system
- Two intertwined hierarchies:
 - Infrastructure: hierarchy of DNS servers
 - Naming structure: `www.bbc.co.uk`

Domain Name System (DNS)

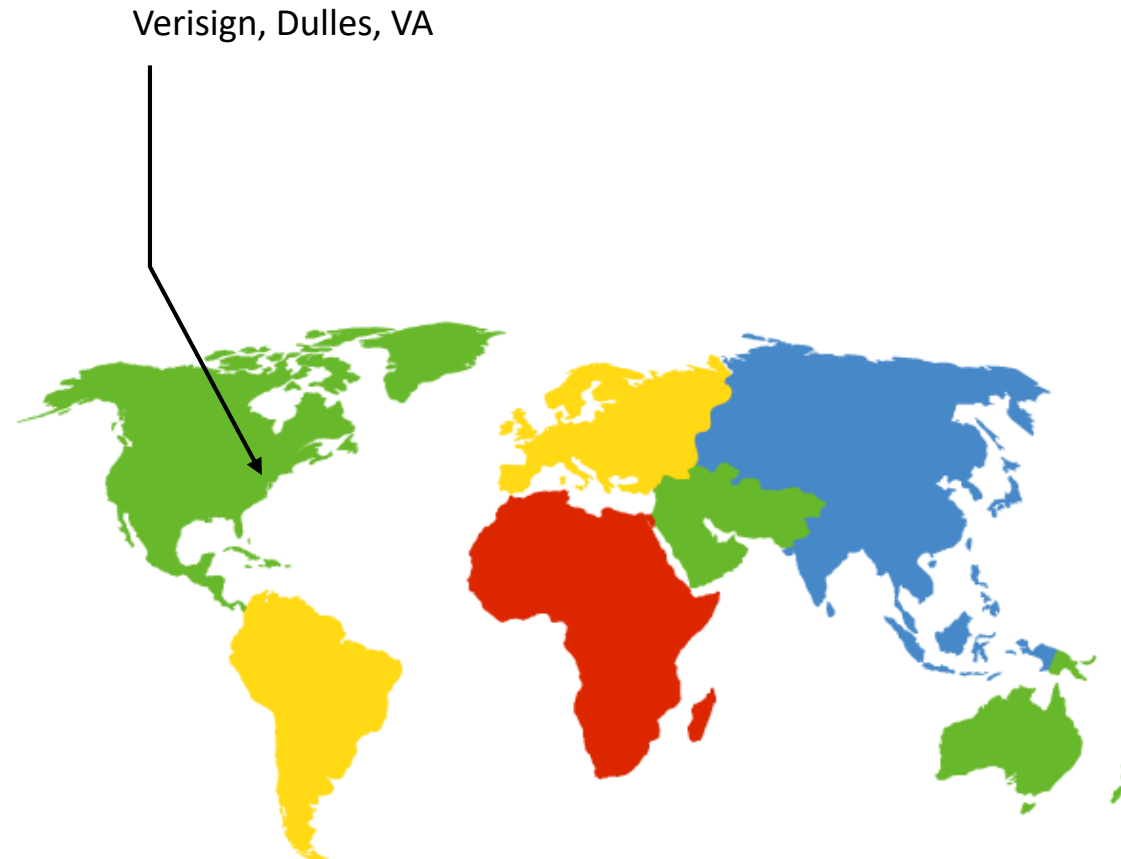
- Top of hierarchy: Root
 - Location hardwired into other servers
- Next Level: Top-level domain (TLD) servers
 - .com, .edu, etc.
 - .uk, .au, .to, etc.
 - Managed professionally
- Bottom Level: Authoritative DNS servers
 - Actually do the mapping
 - Can be maintained locally or by a service provider

Distributed Hierarchical Database



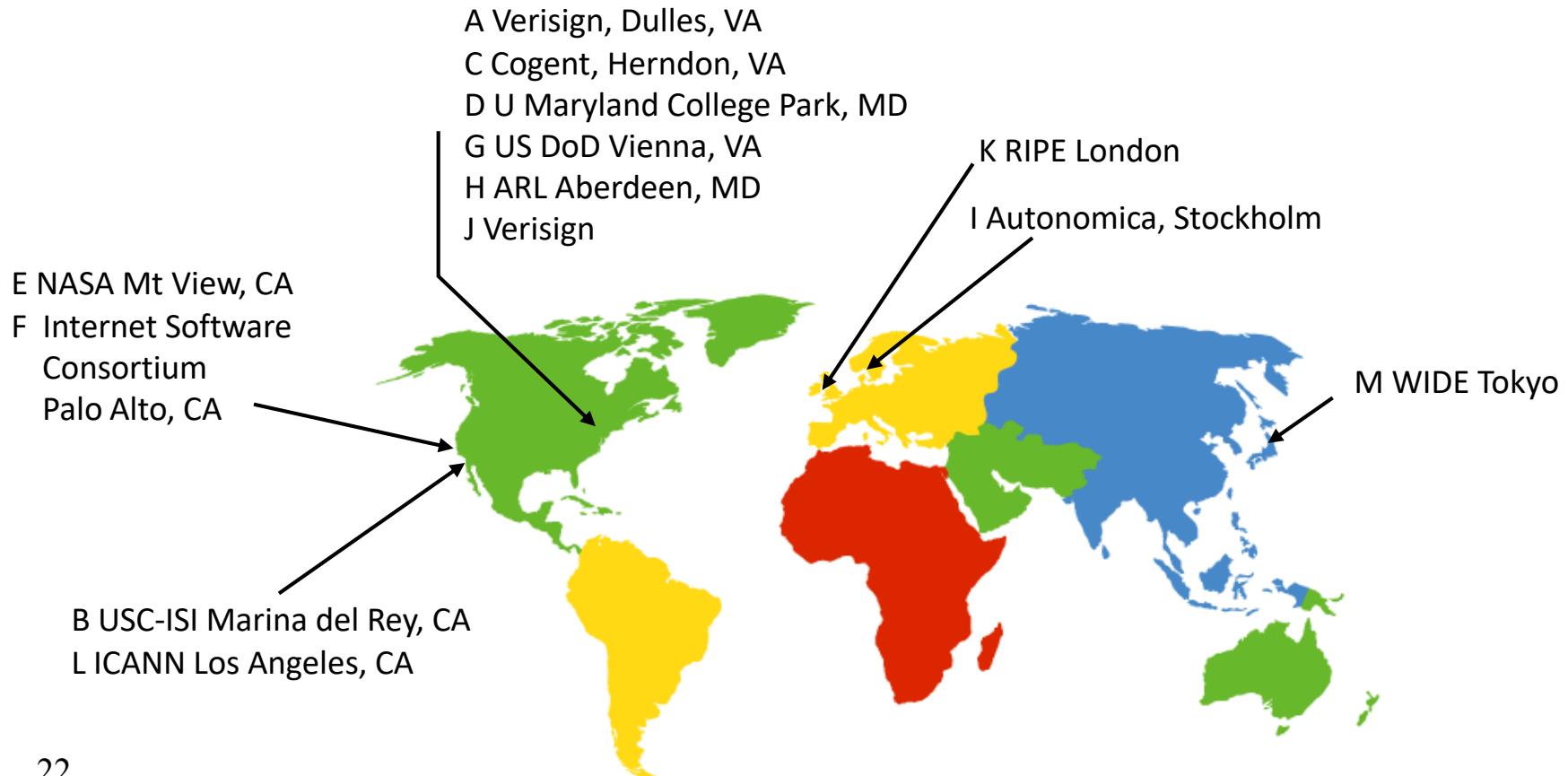
DNS Root

- Located in Virginia, USA
- How do we make the root scale?



DNS Root Servers

- 13 root servers (see <http://www.root-servers.org/>)
 - Labeled A through M
- Does **this** scale?



DNS Root Servers

- 13 root servers (see <http://www.root-servers.org/>)
 - Labeled A through M
- Replication via **any-casting** (localized routing for addresses)



Using DNS

- Two components
 - Local DNS servers
 - Resolver software on hosts
- Local DNS server (“default name server”)
 - Usually near the endhosts that use it
 - Local hosts configured with local server (e.g., `/etc/resolv.conf`) or learn server via DHCP
- Client application
 - Extract server name (e.g., from the URL)
 - Do `gethostbyname()` to trigger resolver code

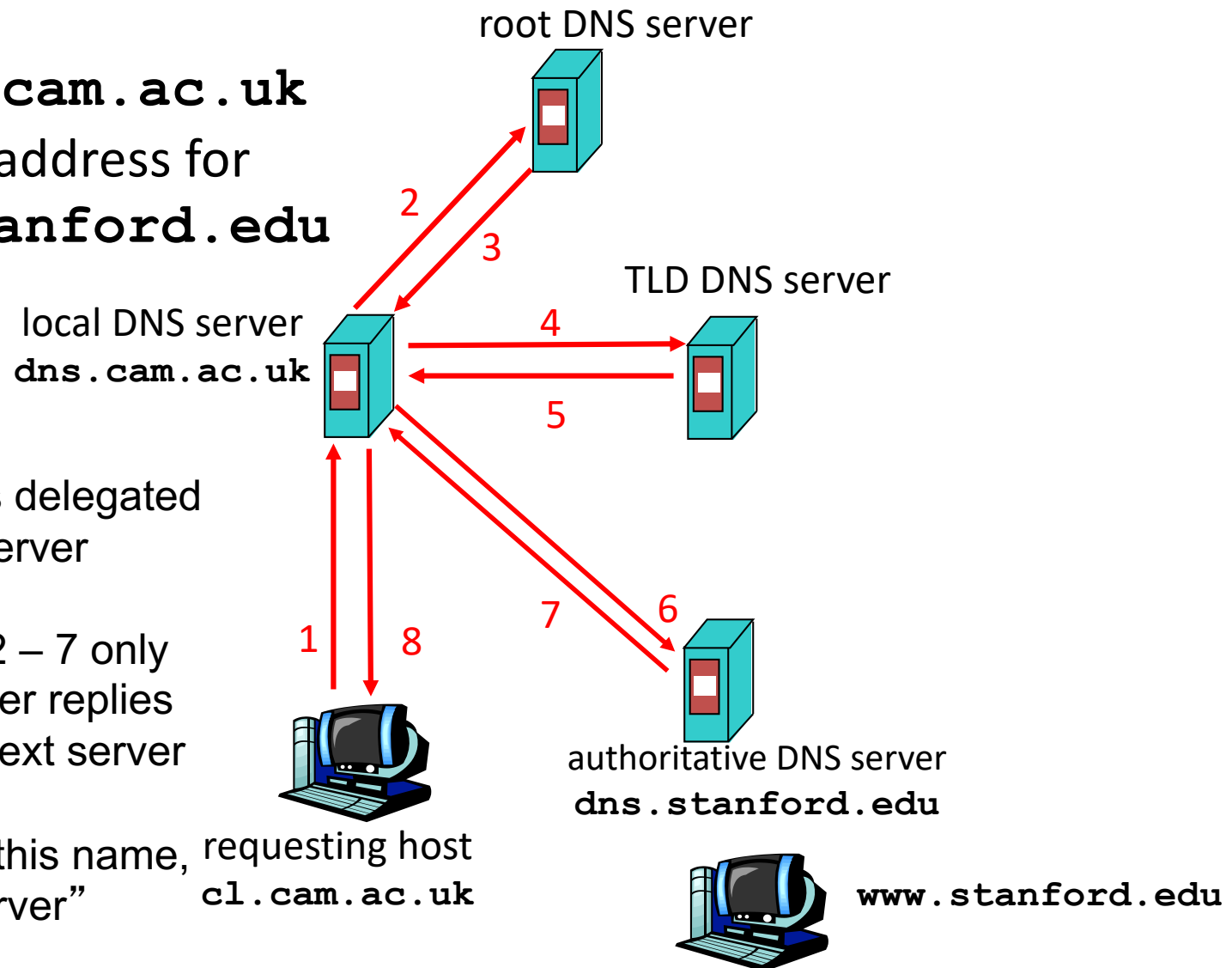
How Does Resolution Happen?

(Iterative example)

Host at **cl.cam.ac.uk**
wants IP address for
www.stanford.edu

iterated query:

- r Host enquiry is delegated to local DNS server
- r Consider transactions 2 – 7 only
- r contacted server replies with name of next server to contact
- r “I don’t know this name, but ask this server”

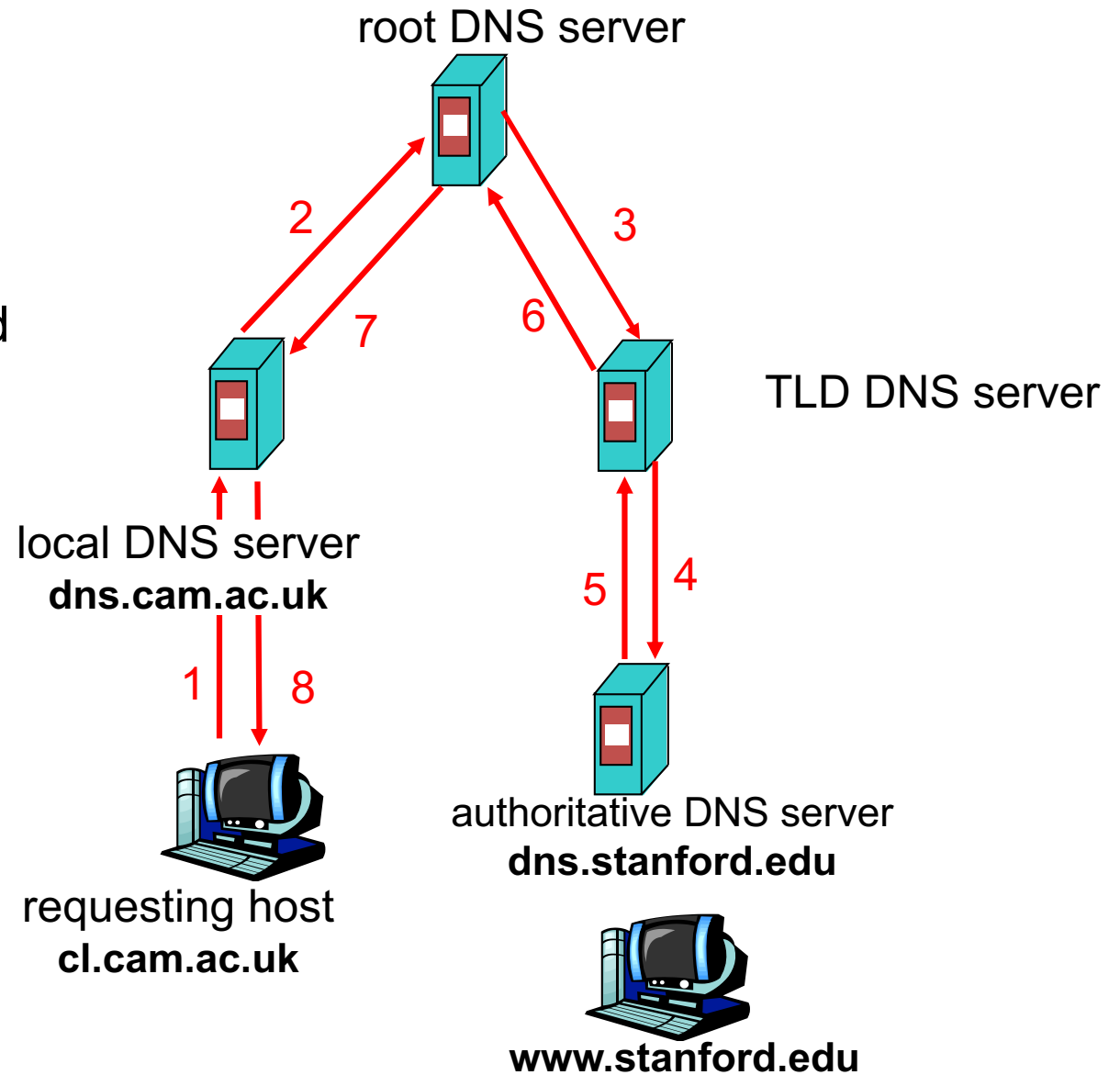


DNS name resolution **recursive** example

recursive query:

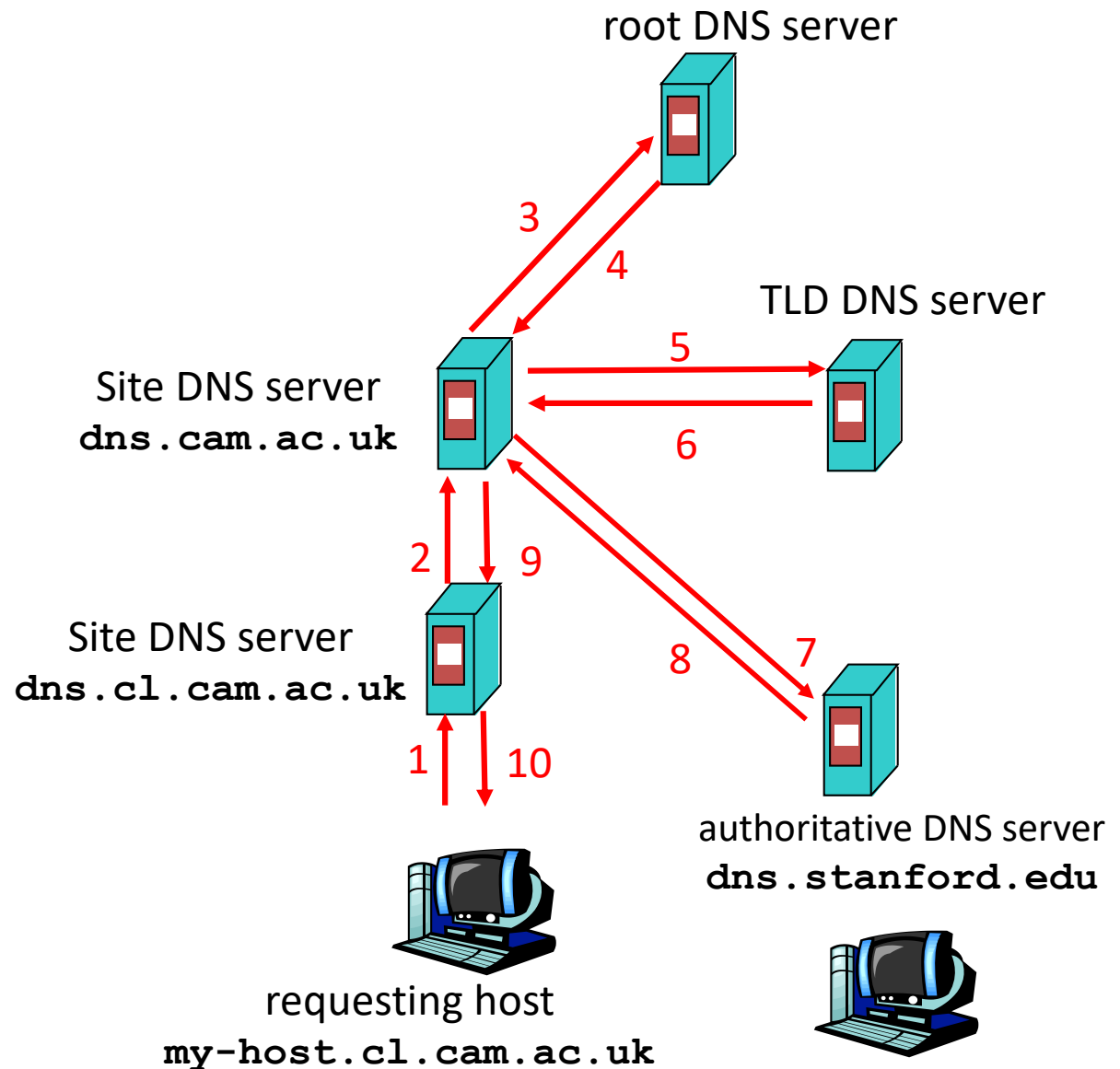
r puts burden of name resolution on contacted name server

r heavy load?



Recursive and Iterative Queries - **Hybrid** case

- **Recursive** query
 - Ask server to get answer for you
 - E.g., requests 1,2 and responses 9,10
- **Iterative** query
 - Ask server who to ask next
 - E.g., all other request-response pairs



DNS Caching

- Performing all these queries takes time
 - And all this **before** actual communication takes place
 - E.g., 1-second latency before starting Web download
- **Caching** can greatly reduce overhead
 - The top-level servers very rarely change
 - Popular sites (e.g., www.bbc.co.uk) visited often
 - Local DNS server often has the information cached
- How DNS caching works
 - DNS servers cache responses to queries
 - Responses include a “**time to live**” (TTL) field
 - Server deletes cached entry after TTL expires

Negative Caching

- Remember things that don't work
 - Misspellings like *bbcc.co.uk* and *www.bbc.com.uk*
 - These can take a long time to fail the first time
 - Good to remember that they don't work
 - ... so the failure takes less time the next time around
- But: negative caching is optional
 - And not widely implemented

Reliability

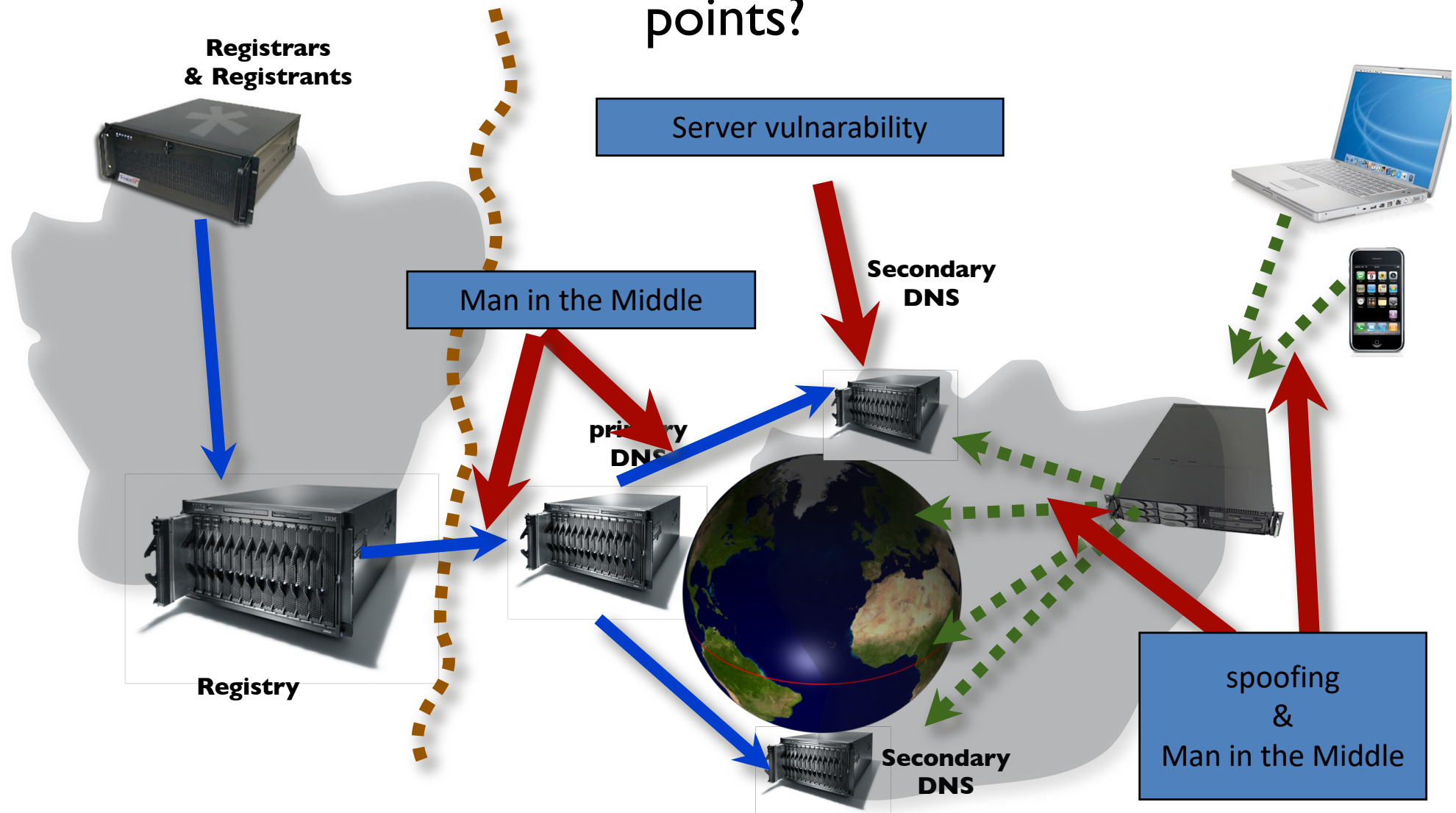
- DNS servers are **replicated** (primary/secondary)
 - Name service available if at least one replica is up
 - Queries can be load-balanced between replicas
- Usually, UDP used for queries
 - Need reliability: must implement this on top of UDP
 - Spec supports TCP too, but not always implemented
- Try alternate servers on timeout
 - **Exponential backoff** when retrying same server
- Same identifier for all queries
 - Don't care which server responds

DNS and Security

- No way to verify answers
 - Opens up DNS to many potential attacks
 - DNSSEC fixes this
- Most obvious vulnerability: recursive resolution
 - Using recursive resolution, host must trust DNS server
 - When at Starbucks, server is under their control
 - And can return whatever values it wants
- More subtle attack: Cache poisoning
 - Those “additional” records can be anything!

Data flow through the DNS

Where are the vulnerable points?



DNSSEC protects all these end-to-end

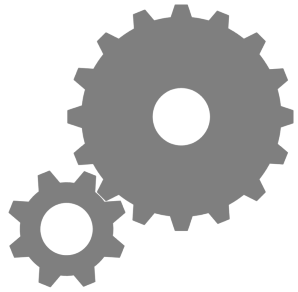
- provides message authentication and integrity verification through cryptographic signatures
 - You know who provided the signature
 - No modifications between signing and validation
- It does **not** provide authorization
- It does **not** provide confidentiality
- It does **not** provide protection against DDOS

DNSSEC in practice

- Scaling the key signing and key distribution

Solution: Using the DNS to Distribute Keys

- Distributing keys through DNS hierarchy:
 - Use one trusted key to establish authenticity of other keys
 - Building chains of trust from the root down
 - Parents need to sign the keys of their children
- Only the root key needed in ideal world
 - Parents always delegate security to child



Why is the web so successful?

- What do the web, youtube, facebook, twitter, instagram, have in common?
 - The ability to self-publish
- Self-publishing that is easy, independent, *free*
- No interest in collaborative and idealistic endeavor
 - People aren't looking for Nirvana (or even Xanadu)
 - People also aren't looking for technical perfection
- Want to make their mark, and find something neat
 - Two sides of the same coin, creates synergy
 - “Performance” more important than dialogue....

Web Components

- Infrastructure:
 - Clients
 - Servers
 - Proxies
- Content:
 - Individual objects (files, etc.)
 - Web sites (coherent collection of objects)
- Implementation
 - HTML: formatting content
 - URL: naming content
 - HTTP: protocol for exchanging content

Any content not just HTML!

HTML: HyperText Markup Language

- *A Web page* has:
 - Base HTML file
 - Referenced objects (*e.g.*, images)
- HTML has several functions:
 - Format text
 - Reference images
 - Embed *hyperlinks* (HREF)

URL Syntax

protocol : //hostname[:port]/directorypath/resource

<i>protocol</i>	http, ftp, https, smtp, rtsp, etc.
-----------------	------------------------------------

<i>hostname</i>	DNS name, IP address
-----------------	----------------------

<i>port</i>	Defaults to protocol's standard port e.g. http: 80 https: 443
-------------	--

<i>directory path</i>	Hierarchical, reflecting file system
-----------------------	--------------------------------------

<i>resource</i>	Identifies the desired resource
-----------------	---------------------------------

Can also extend to program executions:

```
http://us.f413.mail.yahoo.com/ym/ShowLetter?box=%40B%40Bulk&MsgId=2604_1744106_29699_1123_1261_0_28917_3552_1289957100&Search=&Nhead=f&YY=31454&order=down&sort=date&pos=0&view=a&head=b
```

HyperText Transfer Protocol (HTTP)

- Request-response protocol
- Reliance on a global namespace
- Resource *metadata*
- *Stateless*
- ASCII format (ok this changed....)

```
$ telnet www.cl.cam.ac.uk 80
GET /win HTTP/1.0
<blank line, i.e., CRLF>
```

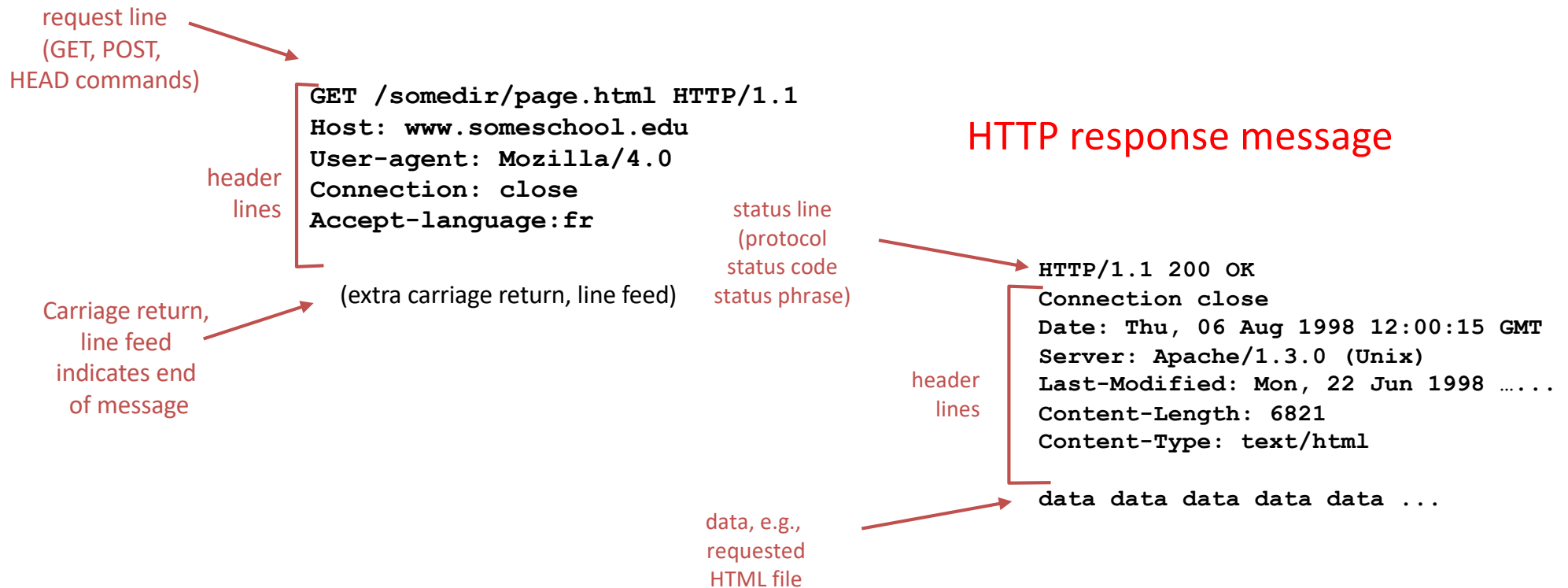
Steps in HTTP Request

- HTTP Client initiates TCP connection to server
 - SYN
 - SYNACK
 - ACK
- Client sends HTTP request to server
 - Can be piggybacked on TCP's ACK
- HTTP Server responds to request
- Client receives the request, terminates connection
- TCP connection termination exchange

How many RTTs for a single request?

Client-Server Communication

- two types of HTTP messages: *request, response*
- HTTP request message: (GET POST HEAD)



Different Forms of Server Response

- Return a file
 - URL matches a file (*e.g.*, `/www/index.html`)
 - Server returns file as the response
 - Server generates appropriate response header
- Generate response dynamically
 - URL triggers a program on the server
 - Server runs program and sends output to client
- Return meta-data with no body

HTTP Resource Meta-Data

- Meta-data
 - Info *about* a resource, stored as a separate entity
- Examples:
 - Size of resource, last modification time, type of content
- Usage example: Conditional GET Request
 - Client requests object “**If-modified-since**”
 - If unchanged, “**HTTP/1.1 304 Not Modified**”
 - No body in the server’s response, only a header

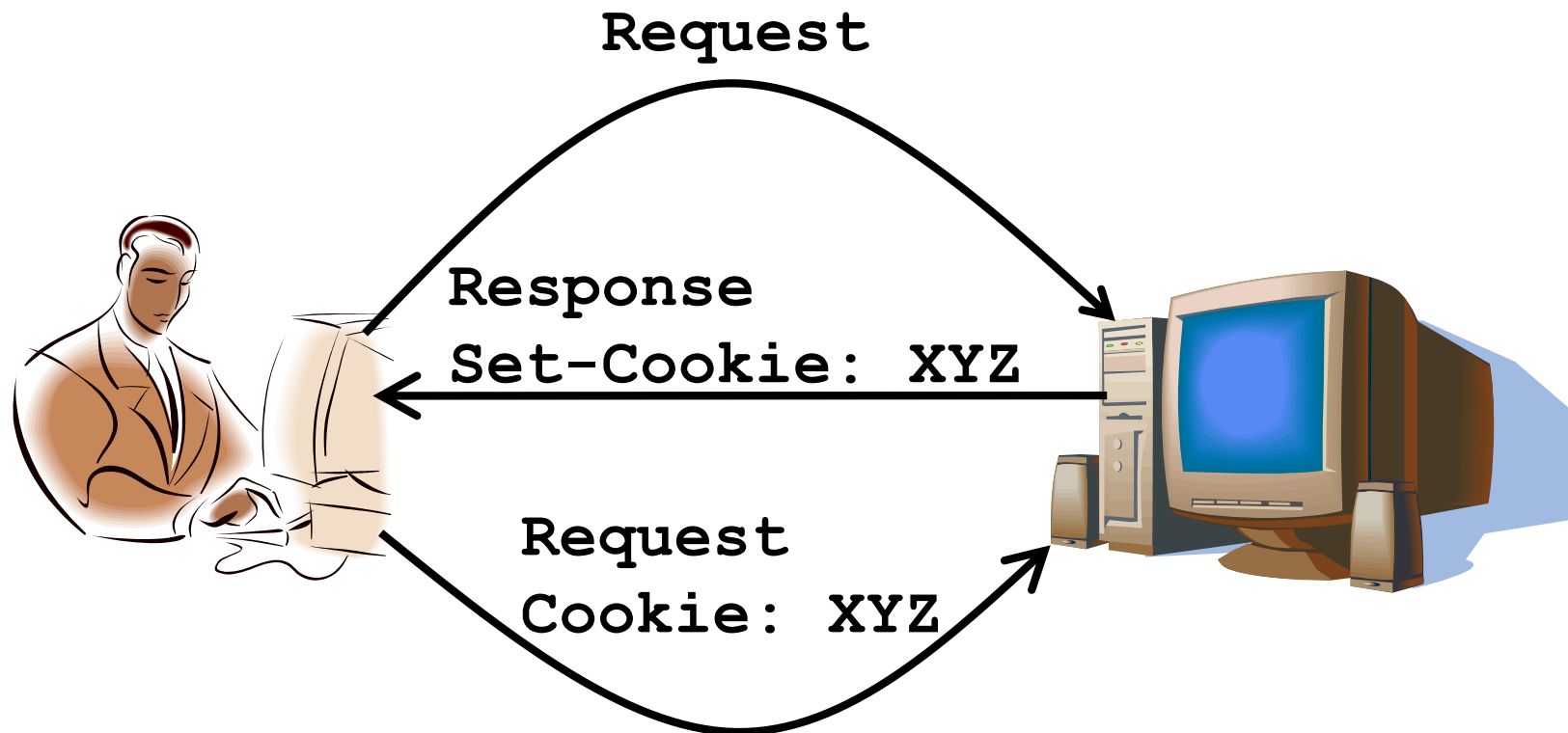
HTTP is *Stateless*

- Each request-response treated independently
 - Servers *not* required to retain state
- **Good:** Improves scalability on the server-side
 - Failure handling is easier
 - Can handle higher rate of requests
 - Order of requests doesn't matter
- **Bad:** Some applications **need** persistent state
 - Need to uniquely identify user or store temporary info
 - *e.g.*, Shopping cart, user profiles, usage tracking, ...

State in a Stateless Protocol:

Cookies

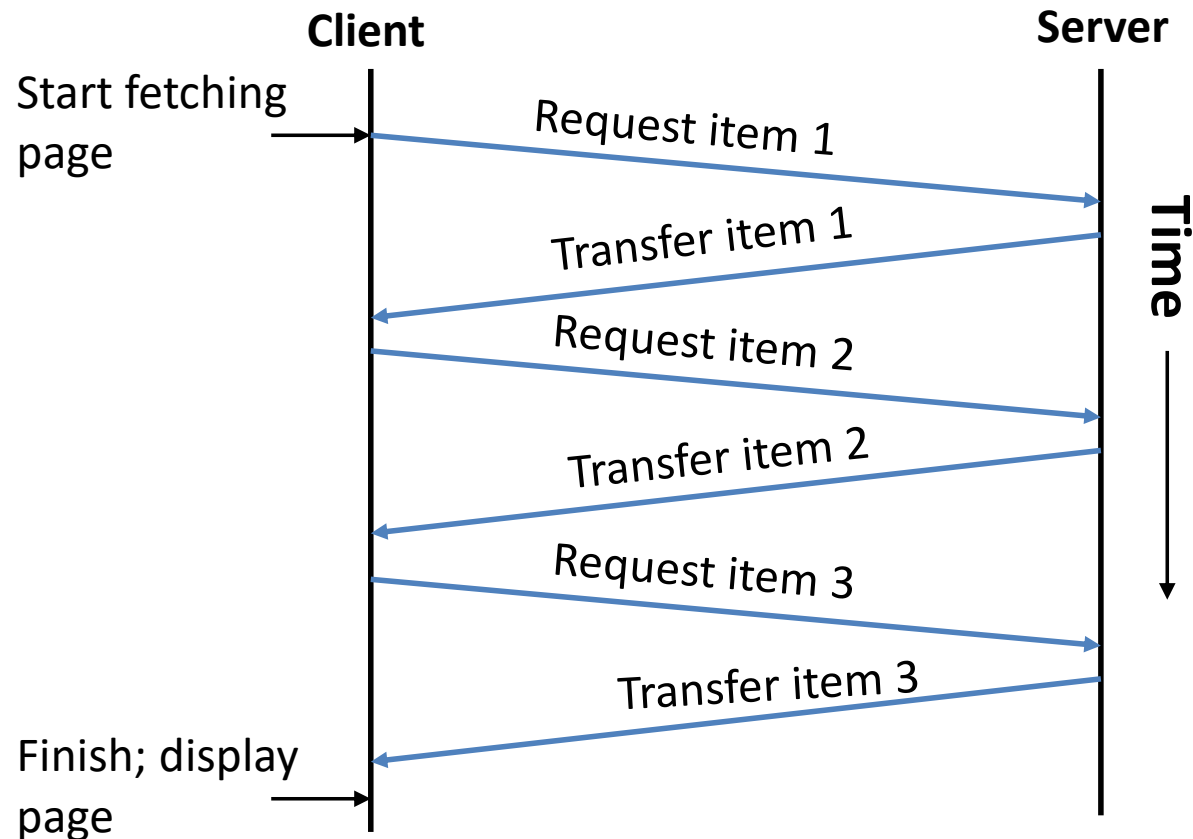
- *Client-side* state maintenance
 - Client stores small(?) state on behalf of server
 - Client sends state in future requests to the server
- Can provide authentication



HTTP Performance

- Most Web pages have multiple objects
 - *e.g.*, HTML file and a bunch of embedded images
- How do you retrieve those objects (naively)?
 - *One item at a time*
- Put stuff in the optimal place?
 - *Where is that precisely?*
 - ***Enter the Web cache and the CDN***

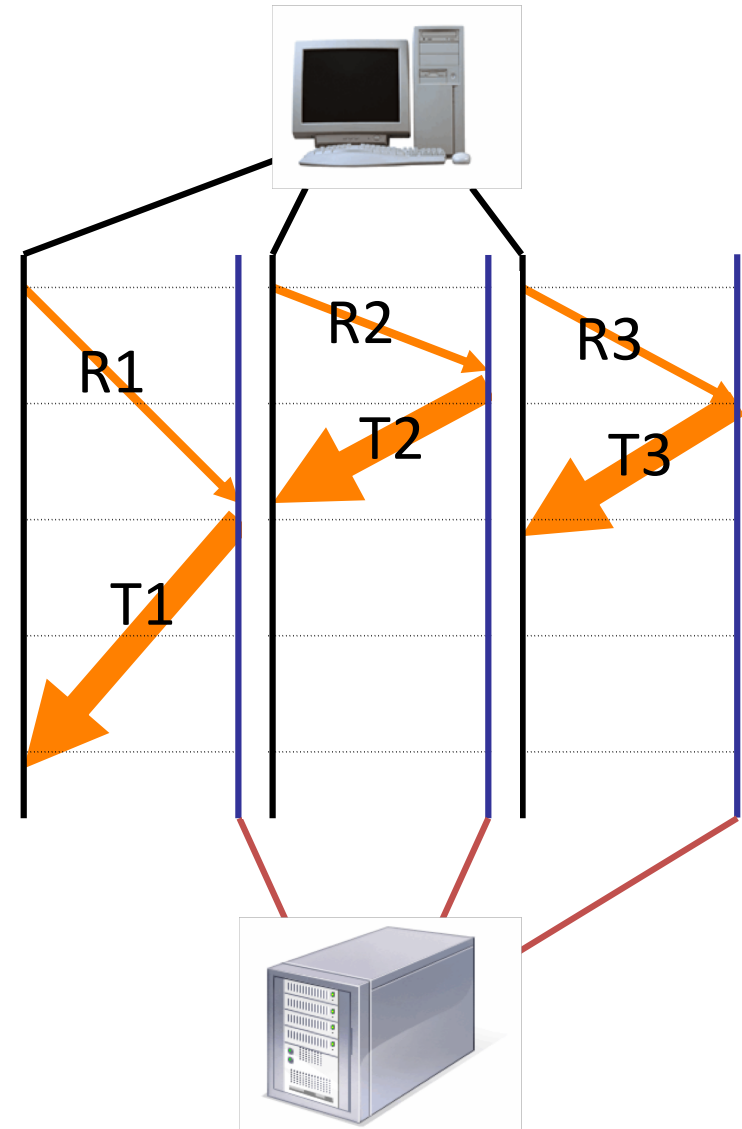
Fetch HTTP Items: Stop & Wait



Improving HTTP Performance: Concurrent Requests & Responses

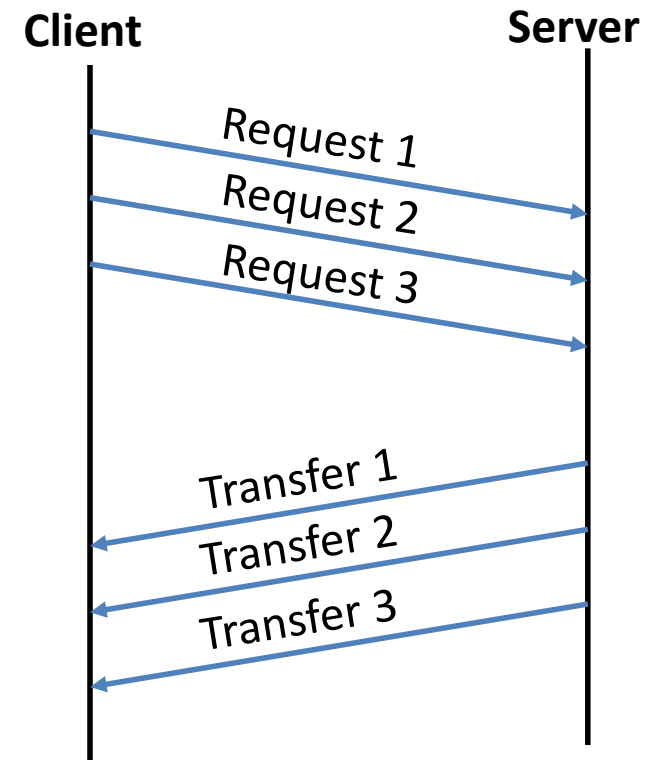
- Use multiple connections *in parallel*
- Does not necessarily maintain order of responses

- Client = 😊
- Server = 😊
- Network = 😞 Why?



Improving HTTP Performance: Pipelined Requests & Responses

- *Batch* requests and responses
 - Reduce connection overhead
 - Multiple requests sent in a single batch
 - Maintains order of responses
 - Item 1 always arrives before item 2
- How is this different from concurrent requests/responses?
 - Single TCP connection



Improving HTTP Performance: Persistent Connections

- Enables multiple transfers per connection
 - Maintain TCP connection across multiple requests
 - Including transfers subsequent to current page
 - Client or server can tear down connection
- Performance advantages:
 - Avoid overhead of connection set-up and tear-down
 - Allow TCP to learn more accurate RTT estimate
 - Allow TCP congestion window to increase
 - i.e., leverage previously discovered bandwidth
- Default in HTTP/1.1

HTTP *evolution*

- 1.0 – one object per TCP: simple but **slow**
- Parallel connections - multiple TCP, one object each: **wastes b/w, may be svr limited, out of order**
- 1.1 pipelining – aggregate retrieval time: ordered, multiple objects sharing single TCP
- 1.1 persistent – aggregate TCP overhead: lower overhead in time, increase overhead at ends (**e.g., when should/do you close the connection?**)

Scorecard: Getting n Small Objects

Time dominated by latency

- One-at-a-time: $\sim 2n$ RTT
- Persistent: $\sim (n+1)$ RTT
- M concurrent: $\sim 2\lceil n/m \rceil$ RTT
- Pipelined: ~ 2 RTT
- Pipelined/Persistent: ~ 2 RTT first time, RTT later

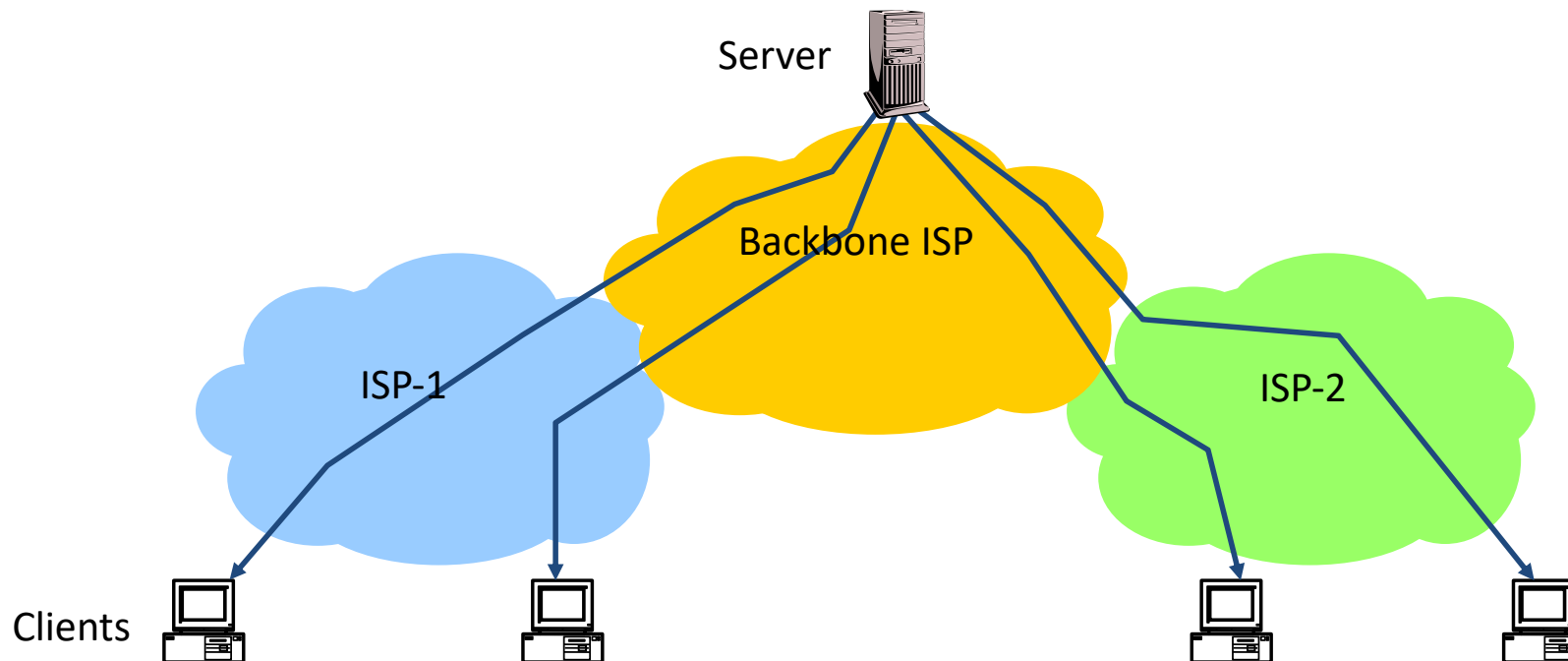
Scorecard: Getting n Large Objects

Time dominated by bandwidth

- One-at-a-time: $\sim nF/B$
- M concurrent: $\sim [n/m] F/B$
 - assuming shared with large population of users
- Pipelined and/or persistent: $\sim nF/B$
 - The only thing that helps is getting more bandwidth..

Improving HTTP Performance: Caching

- Many clients transfer the **same information**
 - Generates **redundant** server and network load
 - Clients experience **unnecessary** latency



Caching: How

- Modifier to GET requests:
 - `If-modified-since` – returns “not modified” if resource not modified since specified time
- Response header:
 - `Expires` – how long it’s safe to cache the resource
 - `No-cache` – ignore all caches; always get resource directly from server

Caching: Why

- Motive for placing content closer to client:
 - User gets better response time
 - Content providers get happier users
 - Time is money, really!
 - Network gets reduced load
- Why does caching work?
 - Exploits *locality of reference*
- How well does caching work?
 - Very well, up to a limit
 - Large overlap in content
 - But many unique requests

Improving HTTP Performance: Caching on the Client

Example: Conditional GET Request

- Return resource only if it has changed at the server

– Save server resources!
Request from client to server:

```
GET /~awm22/win HTTP/1.1
Host: www.cl.cam.ac.uk
User-Agent: Mozilla/4.03
If-Modified-Since: Sun, 27 Aug 2006 22:25:50 GMT
<CRLF>
```

- HOW?
 - Client specifies “if-modified-since” time in request
 - Server compares this against “last modified” time of desired resource
 - Server returns “304 Not Modified” if resource has not changed
 - or a “200 OK” with the latest version otherwise

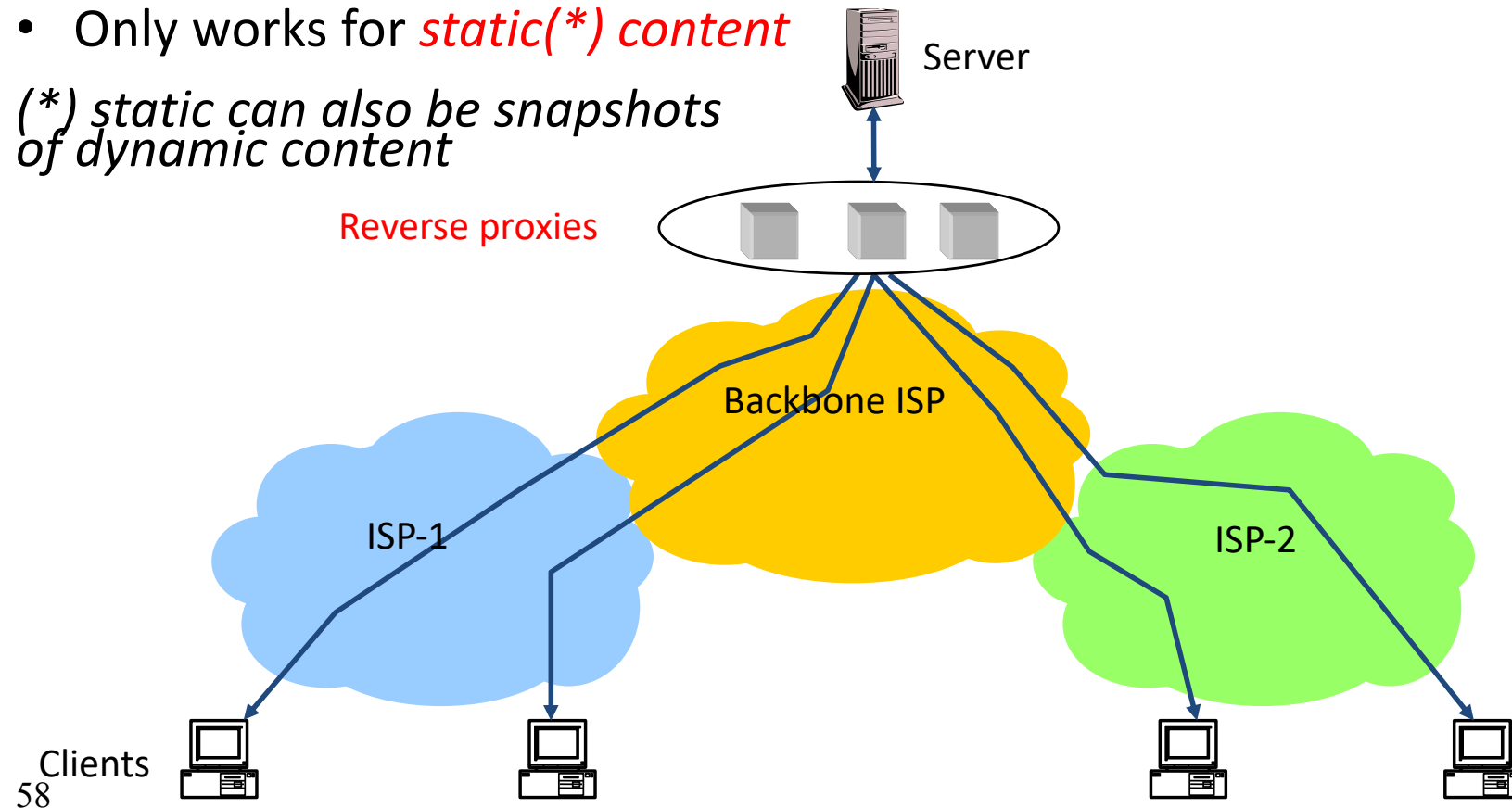
Improving HTTP Performance: Caching with Reverse Proxies

Cache documents close to **server**

→ decrease server load

- Typically done by content providers
- Only works for *static(*) content*

() static can also be snapshots of dynamic content*

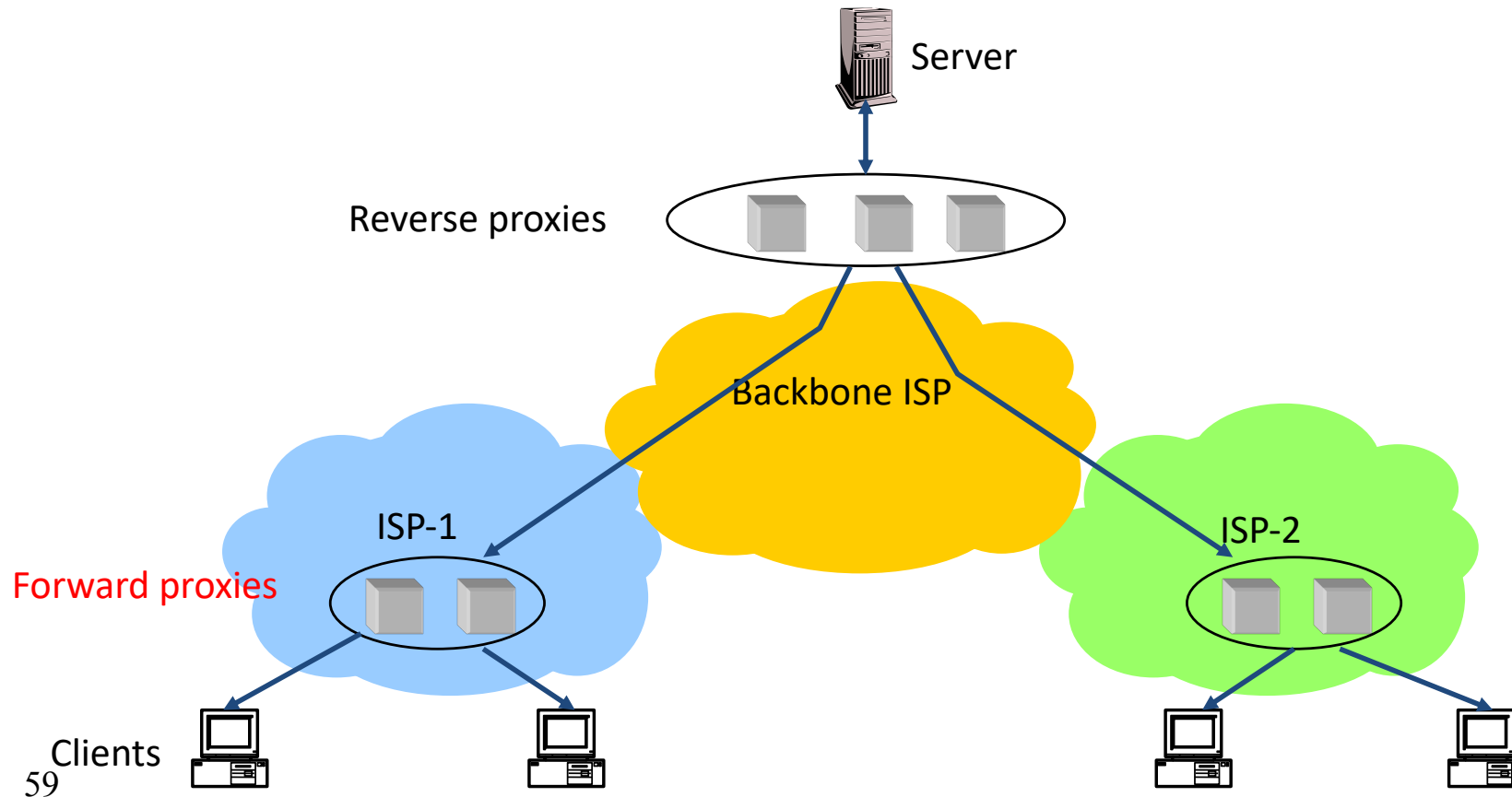


Improving HTTP Performance: Caching with Forward Proxies

Cache documents close to **clients**

→ reduce network traffic and decrease latency

- Typically done by ISPs or corporate LANs

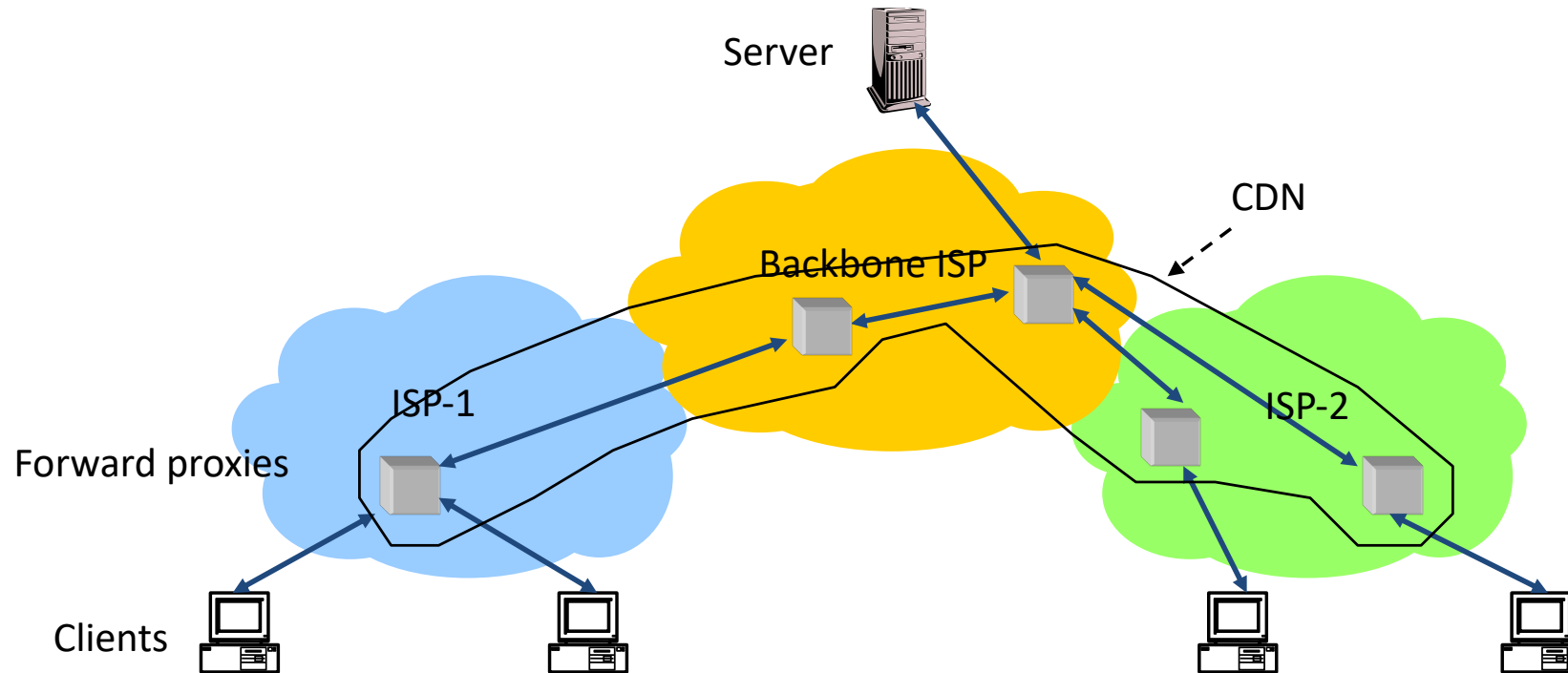


Improving HTTP Performance:

Caching w/ Content Distribution Networks

- Integrate forward and reverse caching functionality
 - One overlay network (usually) administered by one entity
 - *e.g.*, Akamai
- Provide document caching
 - **Pull**: Direct result of clients' requests
 - **Push**: Expectation of high access rate
- Also do some processing
 - Handle *dynamic* web pages
 - *Transcoding*
 - *Maybe do some security function – watermark IP*

Improving HTTP Performance: Caching with CDNs (cont.)



Improving HTTP Performance:
CDN Example – Akamai

- Akamai creates new domain names for each client content provider.
 - e.g., a128.g.akamai.net
- The CDN's DNS servers are authoritative for the new domains
- The client content provider modifies its content so that embedded URLs reference the new domains.
 - “Akamaize” content
 - e.g.: <http://www.bbc.co.uk/popular-image.jpg> becomes <http://a128.g.akamai.net/popular-image.jpg>
- *Requests now sent to CDN's infrastructure...*

Hosting: Multiple Sites Per Machine

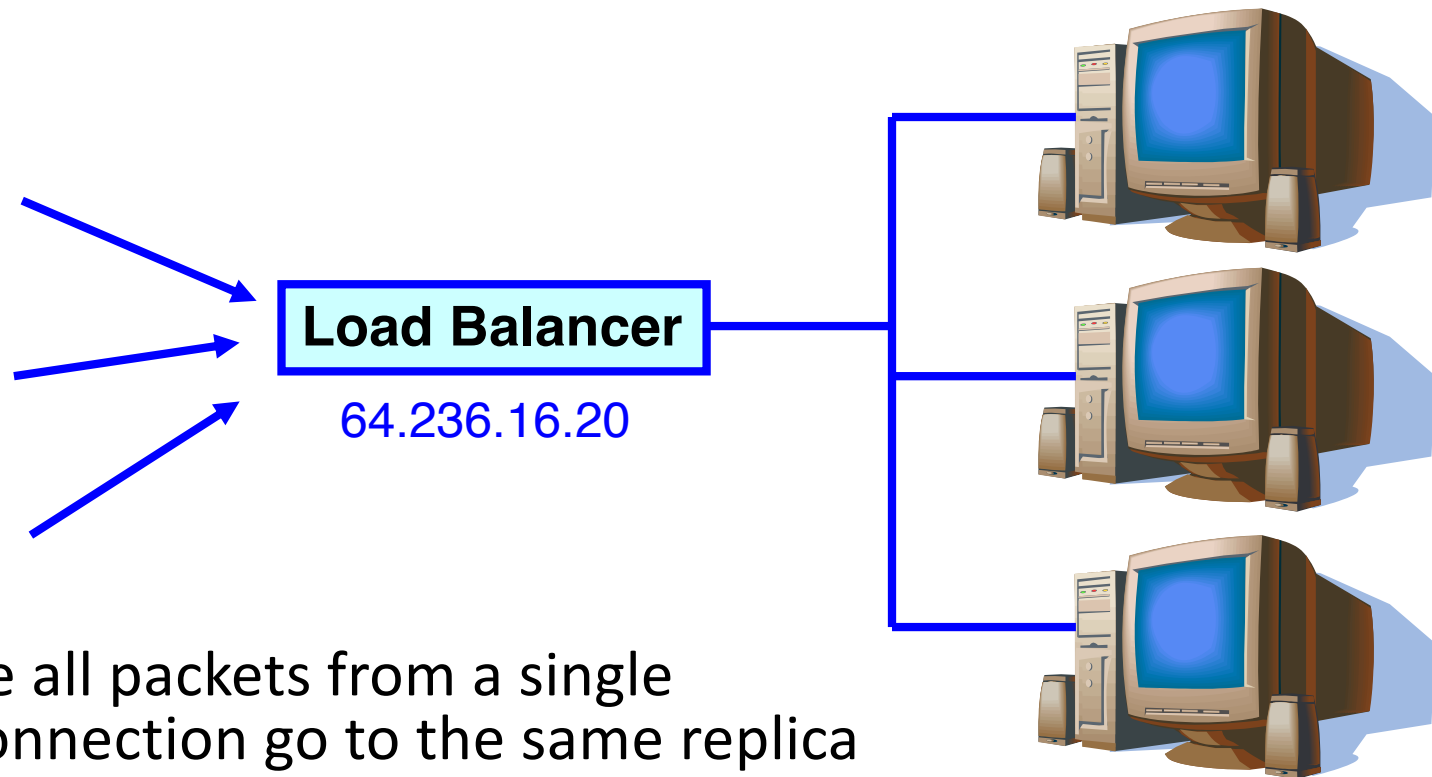
- Multiple Web sites on a single machine
 - Hosting company runs the Web server on behalf of multiple sites (*e.g.*, `www.foo.com` and `www.bar.com`)
- Problem: `GET /index.html`
 - `www.foo.com/index.html` Or `www.bar.com/index.html`?
- Solutions:
 - Multiple server processes on the same machine
 - Have a separate IP address (or port) for each server
 - Include site name in HTTP request
 - Single Web server process with a single IP address
 - Client includes “Host” header (*e.g.*, `Host: www.foo.com`)
 - *Required header* with HTTP/1.1

Hosting: Multiple Machines Per Site

- Replicate popular Web site across many machines
 - Helps to handle the load
 - Places content closer to clients
- Helps when content isn't cacheable
- Problem: Want to direct client to particular replica
 - Balance load across server replicas
 - Pair clients with nearby servers

Multi-Hosting at Single Location

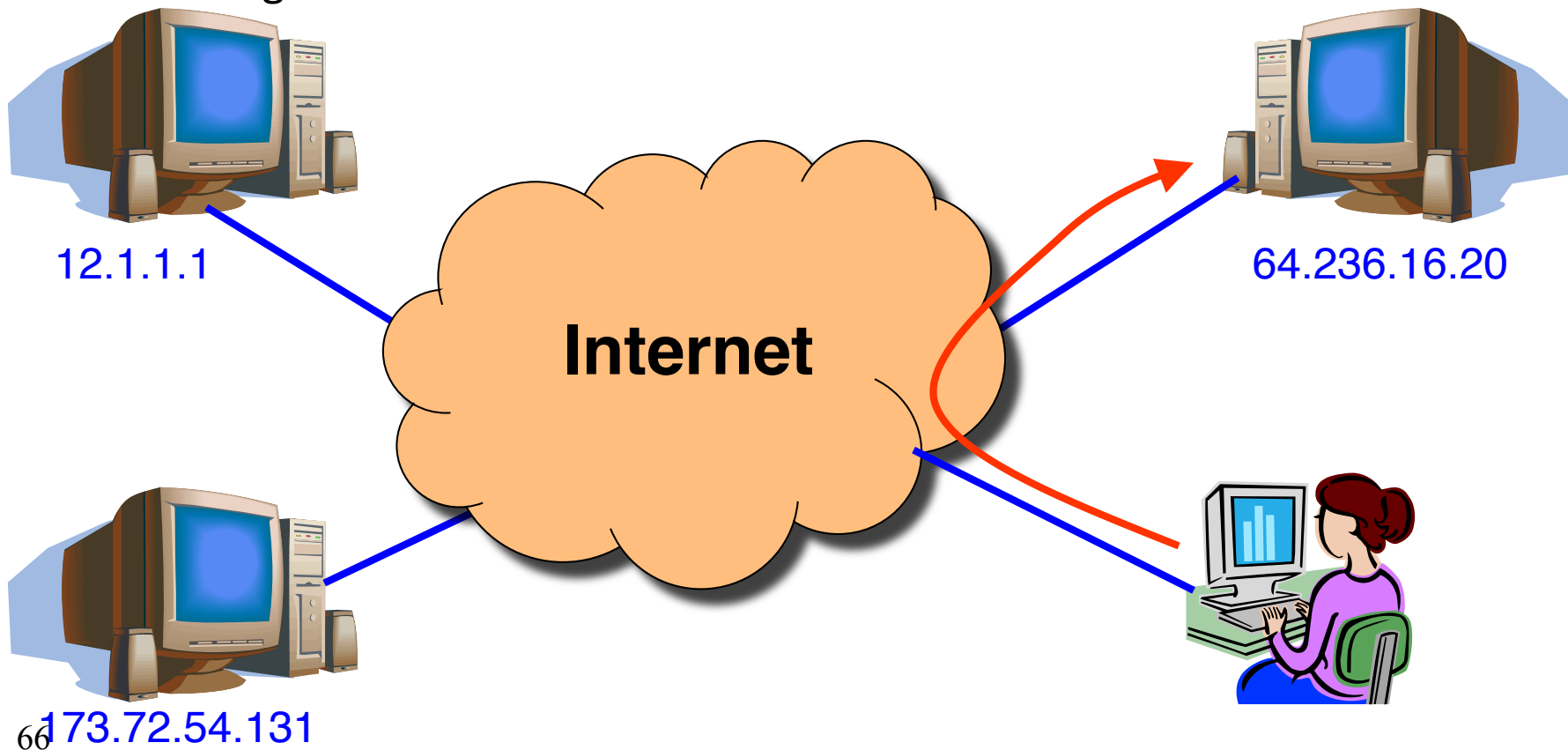
- Single IP address, multiple machines
 - Run multiple machines behind a single IP address



- Ensure all packets from a single TCP connection go to the same replica

Multi-Hosting at Several Locations

- Multiple addresses, multiple machines
 - Same name but different addresses for all of the replicas
 - Configure DNS server to return *closest* address



CDN examples round-up

- CDN using DNS
DNS has information on loading/distribution/location
- CDN using anycast
same address from DNS name but local routes
- CDN based on rewriting HTML URLs
(akami example just covered – akami uses DNS too)

After HTTP/1.1

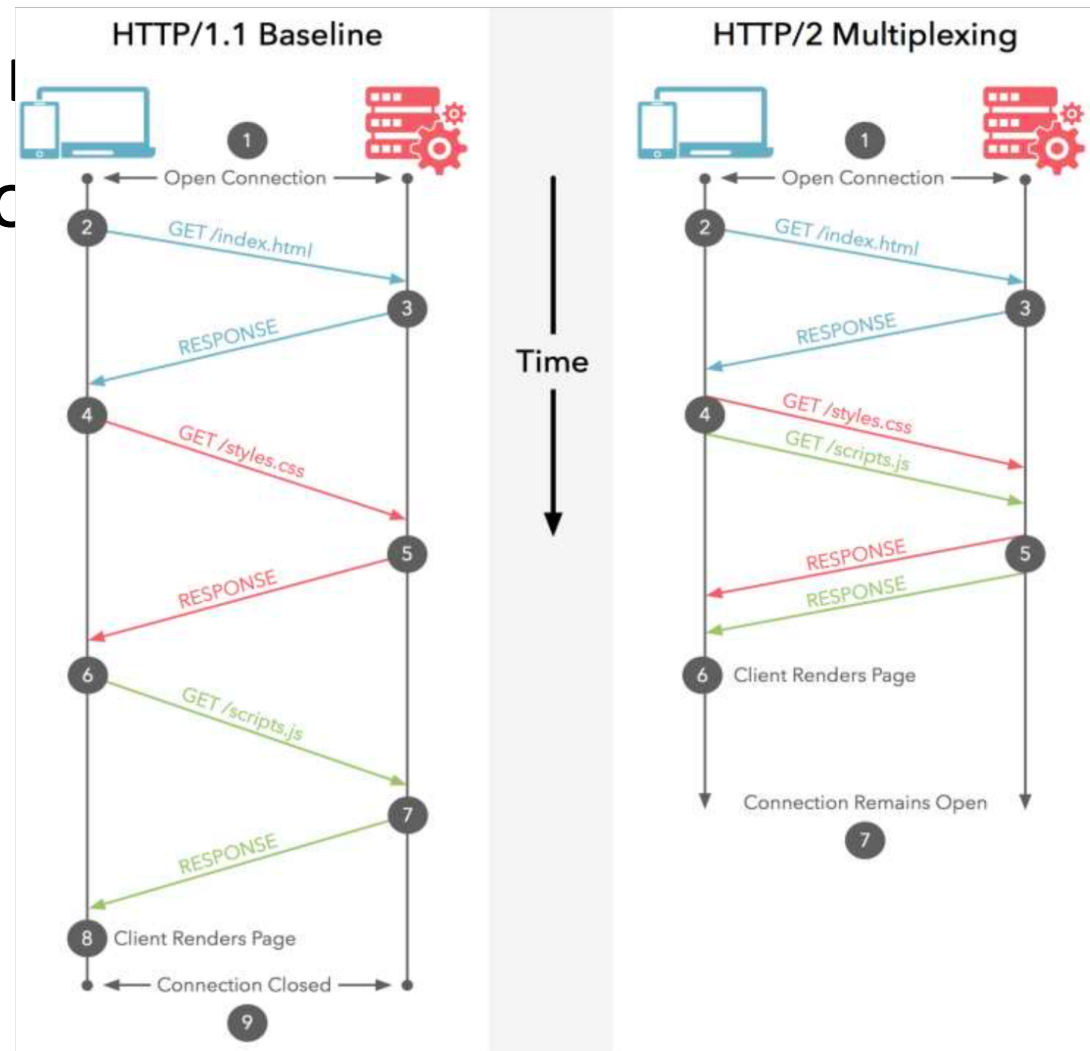
SPDY (speedy) and its moral successor HTTP/2

- Binary protocol
 - More efficient to parse
 - More compact on the wire
 - Much less error prone as compared
 - to textual protocols

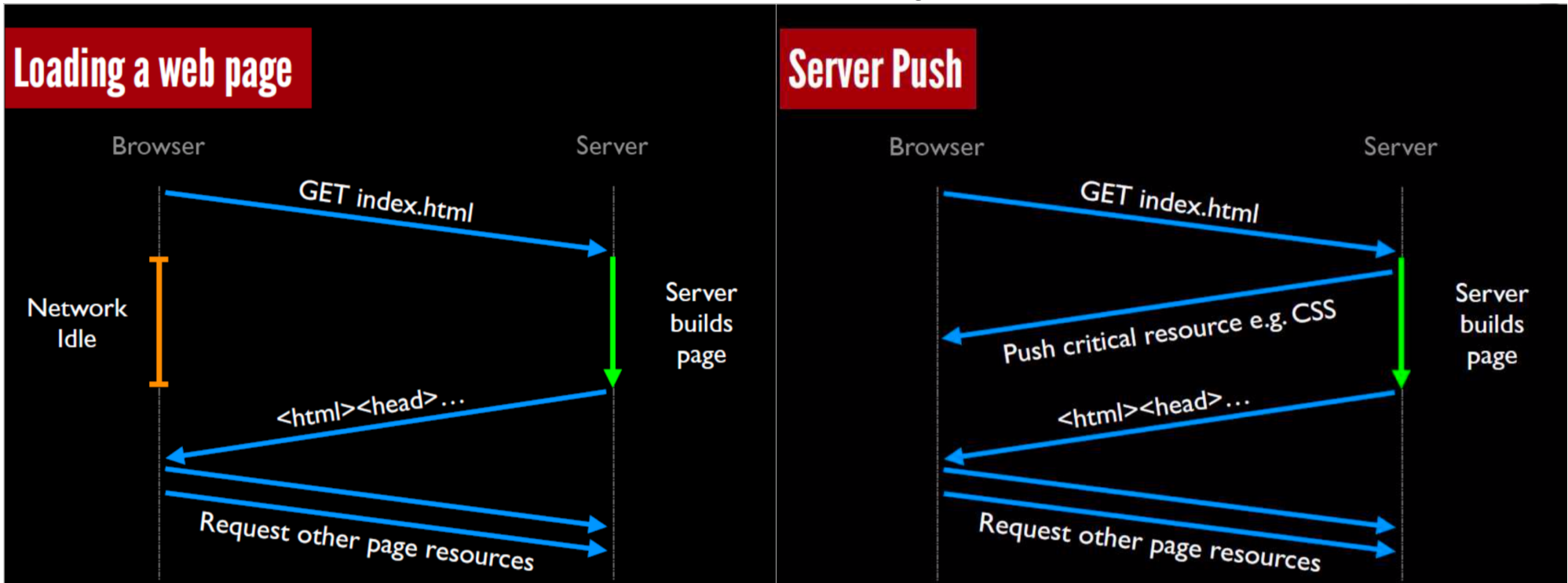
After HTTP/1.1

SPDY (speedy) and

- Binary protocol
- Multiplexing
 - Interleaved



After HTTP/1.1



- Server Push
 - Proactively push stuff to client that it will need

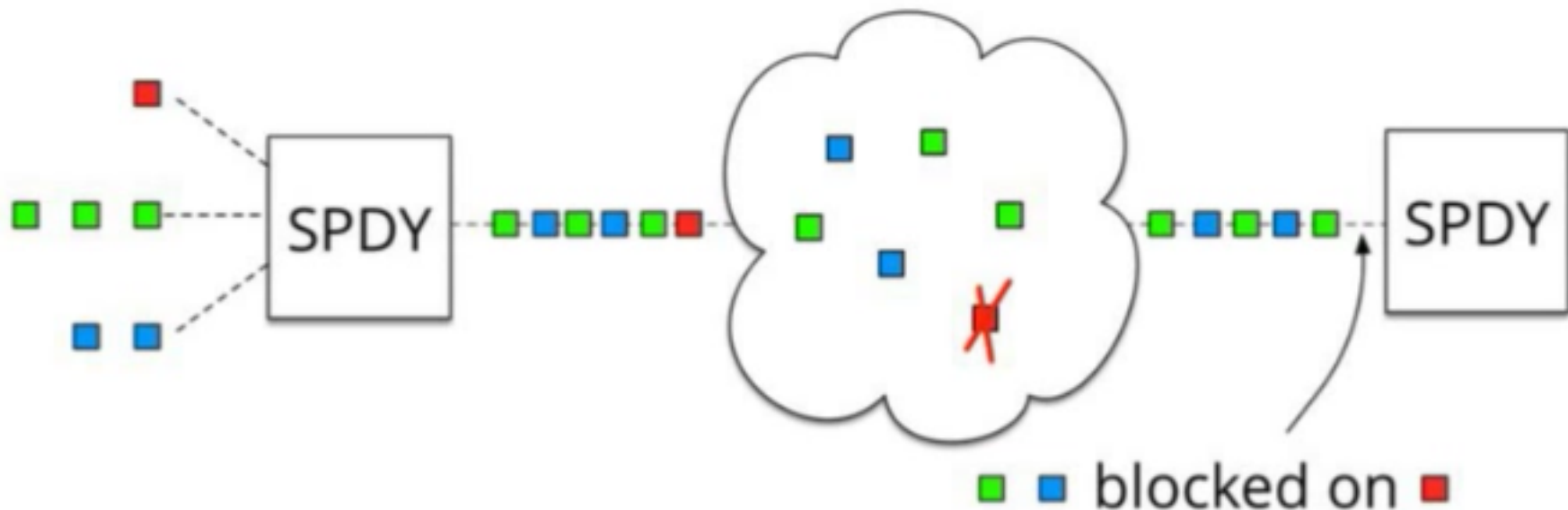
After HTTP/1.1

SPDY (speedy) and its moral successor HTTP/2

- Binary protocol
- Multiplexing
- Priority control over Frames
- Header Compression
- Server Push

SPDY

- SPDY + HTTP/2: One single TCP connection instead of multiple
- Downside: Head of line blocking
- In TCP, packets need to be processed in



Add QUIC and stir...

Quick UDP Internet Connections

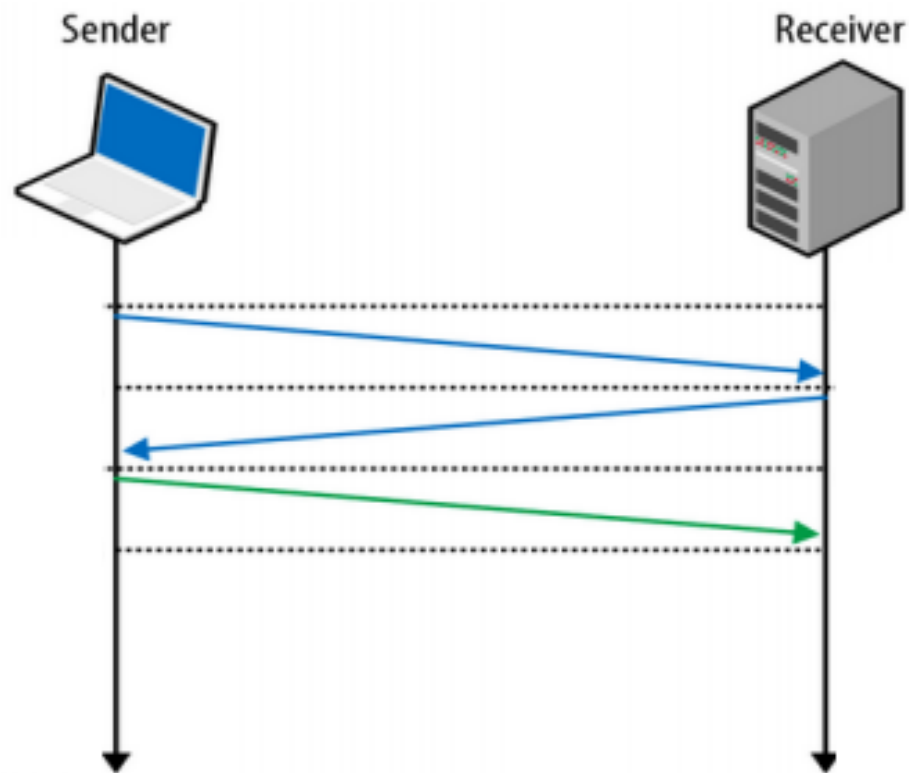
Objective: Combine speed of UDP protocol with TCP's reliability

- Very hard to make changes to TCP
- *Faster to implement new protocol on top of UDP*
- Roll out features in TCP if they prove theory

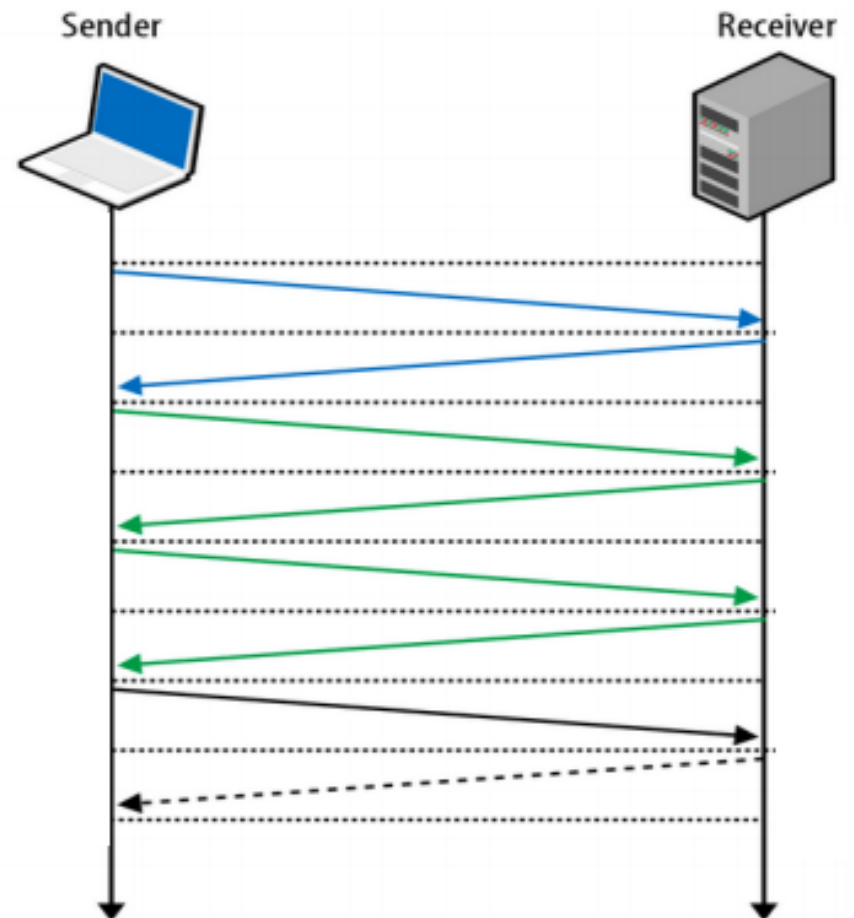
QUIC:

- Reliable transport over UDP (seriously)
- Uses FEC
- Default crypto
- Restartable connections

3-Way Handshake



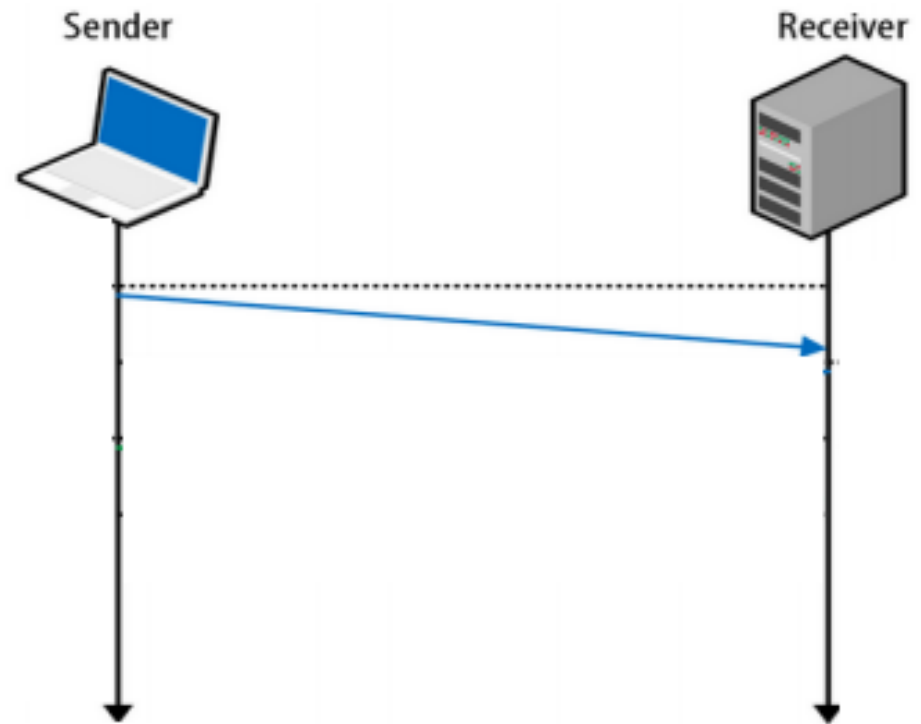
Without TLS



With TLS

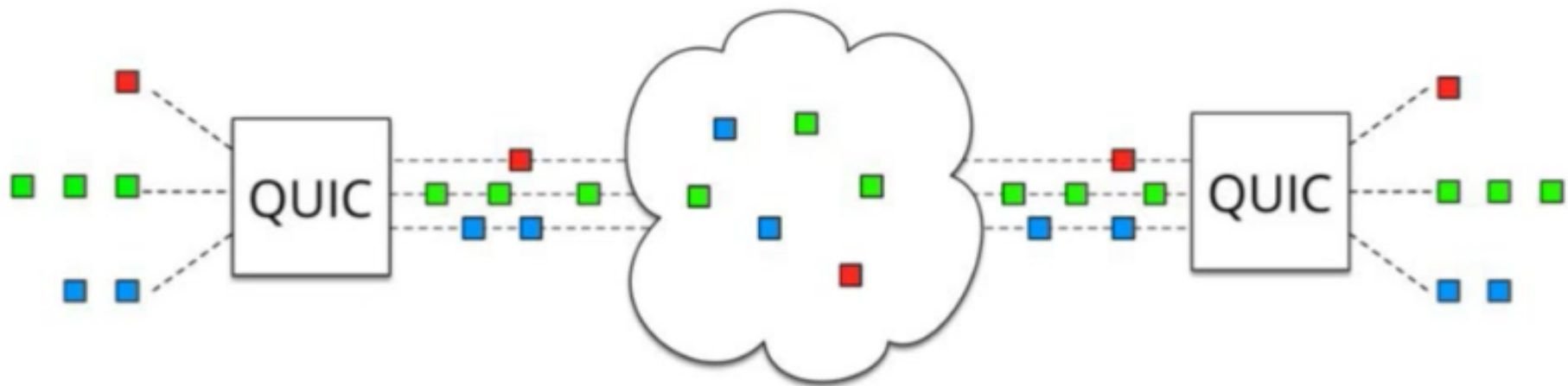
UDP

- Fire and forget
 - Less time spent to validate packets
 - Downside - no reliability, has to be built on top of UDP



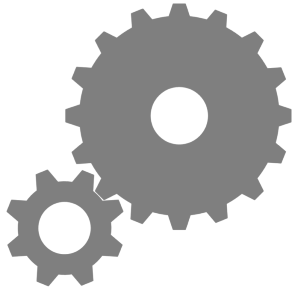
QUIC

- UDP does NOT depend on order of arriving packets
- Lost packets will only impact an individual resource, e.g., CSS or JS file.
- QUIC is combining best parts of HTTP/2 over UDP:
 - Multiplexing on top of non-blocking transport protocol



QUIC – more than just UDP

- QUIC outshines TCP under poor network conditions, shaving a full second off the Google Search page load time for the slowest 1% of connections.
- These benefits are even more apparent for video services like YouTube. Users report 30% fewer rebuffers when watching videos over QUIC.

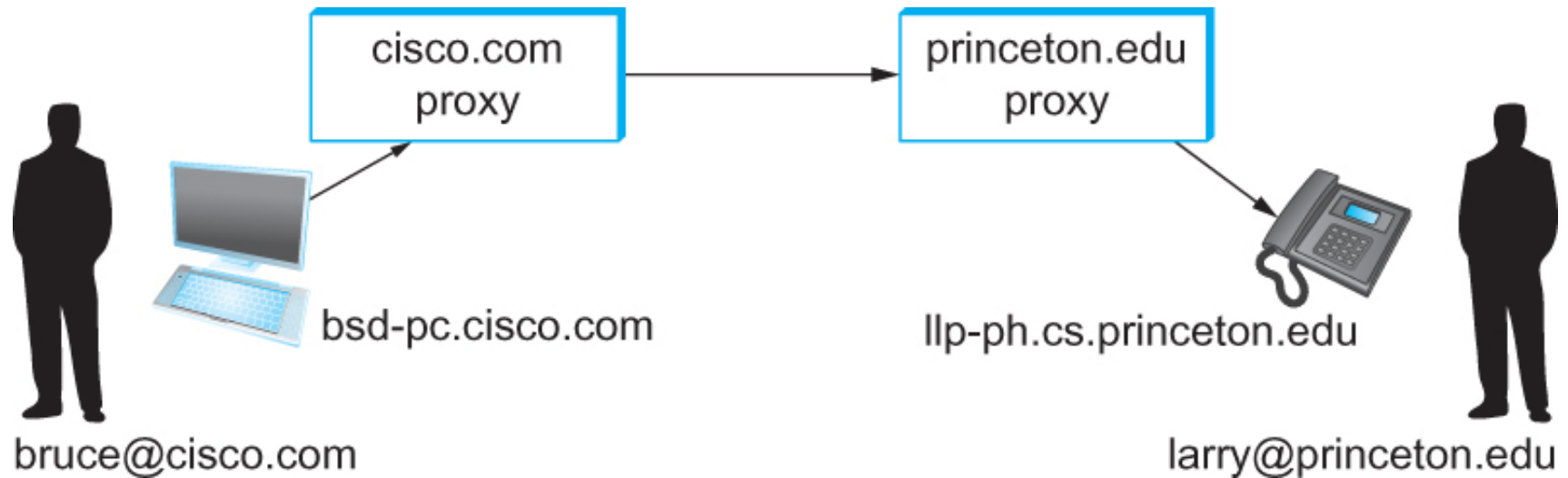


SIP – Session Initiation Protocol

Session?

Anyone smell an OSI / ISO standards document burning?

SIP - VoIP



Establishing communication
through SIP proxies.



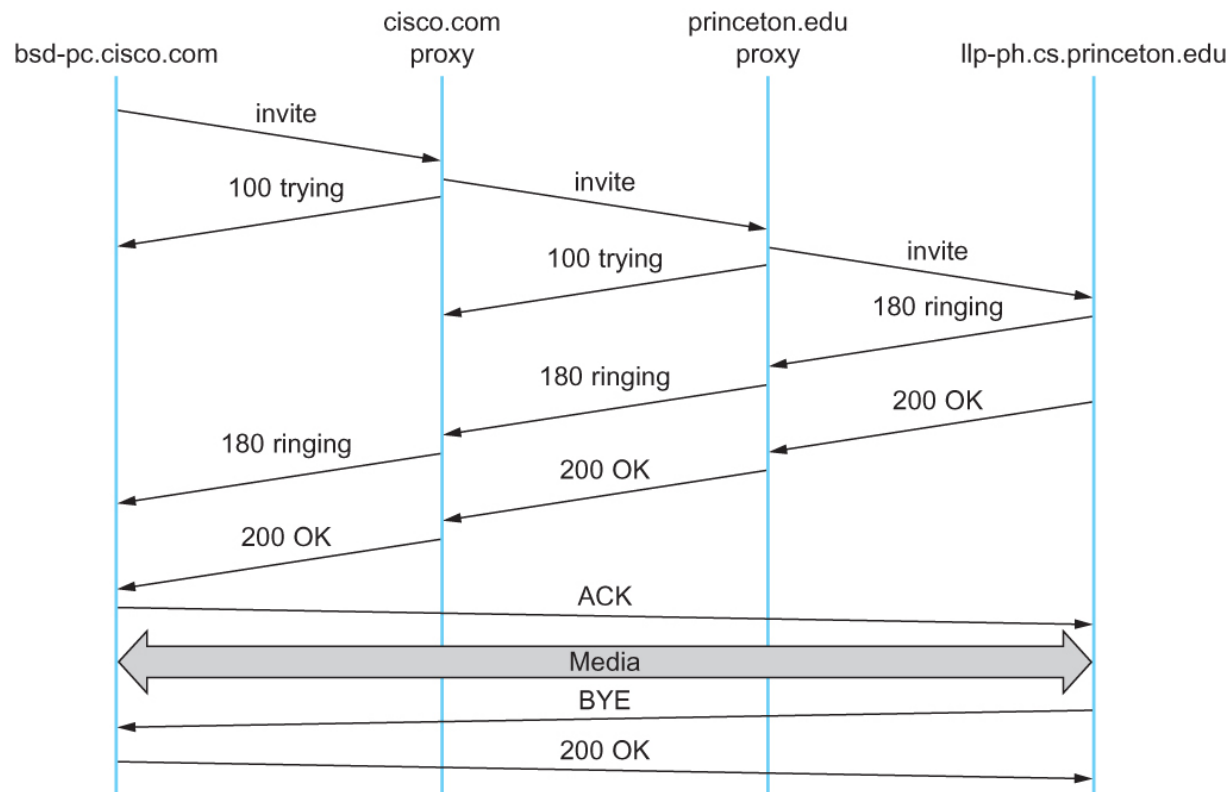
SIP?

- SIP – bringing the fun/complexity of telephony to the Internet
 - User location
 - User availability
 - User capabilities
 - Session setup
 - Session management
 - (e.g. “call forwarding”)

H.323 – ITU

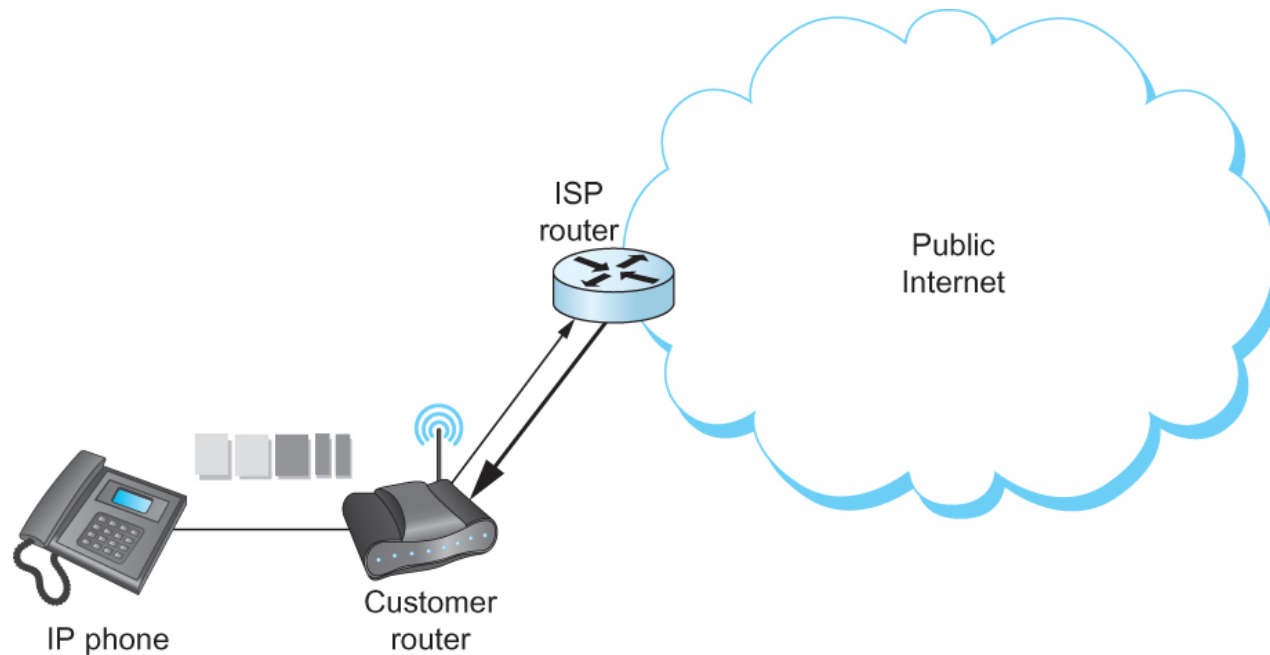
- Why have one standard when there are at least two....
- The full H.323 is hundreds of pages
 - The protocol is known for its complexity – an ITU hallmark
- SIP is not much better
 - IETF grew up and became the ITU....

Multimedia Applications



Message flow for a basic SIP session

The (still?) missing piece: Resource Allocation for Multimedia Applications

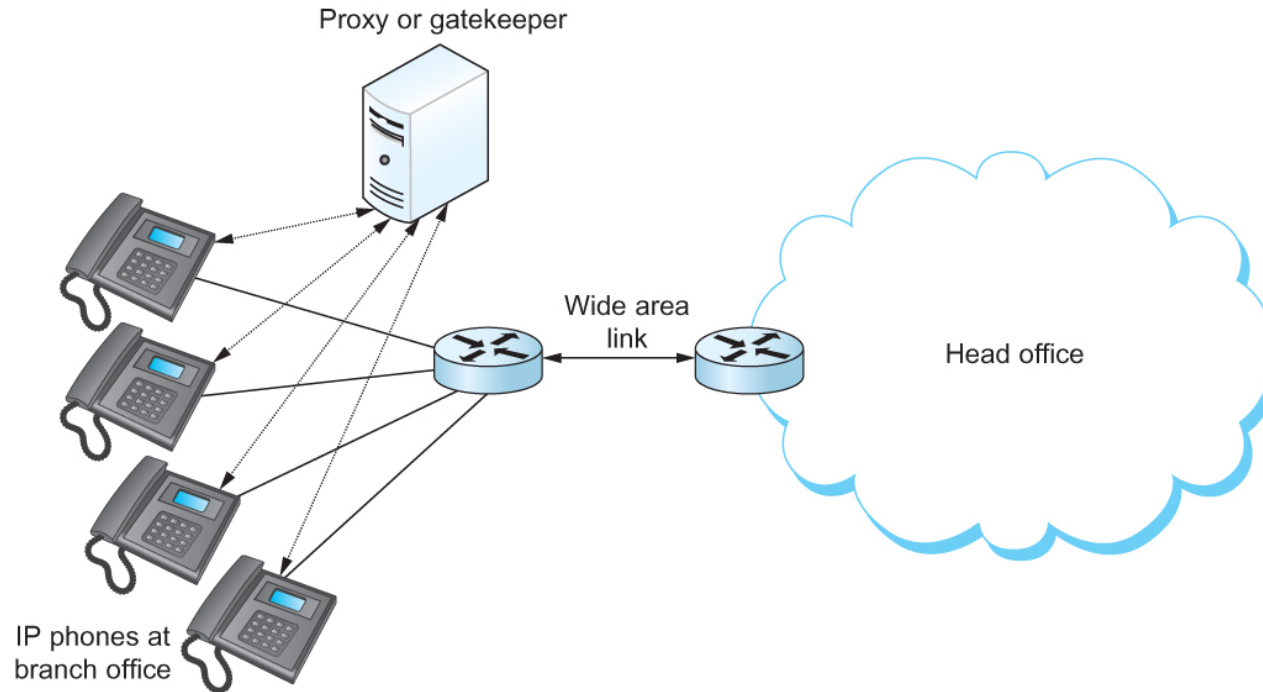


I can 'differentiate' VoIP from data but...

I can only control data going into the Internet

Multimedia Applications

- Resource Allocation for Multimedia Applications



Admission control using session control protocol.

Resource Allocation for Multimedia Applications

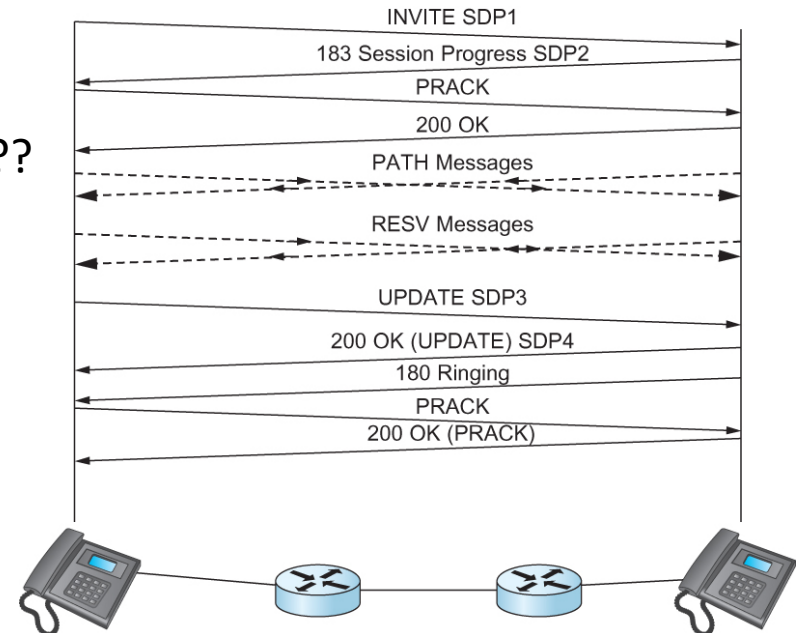
Coming soon... ~~1995~~

~~2000~~

~~2010~~

who are we kidding??

Co-ordination of SIP signaling and
resource reservation.



So where does it happen?

Inside single institutions or *domains of control*.....

(Universities, Hospitals, big corp...)

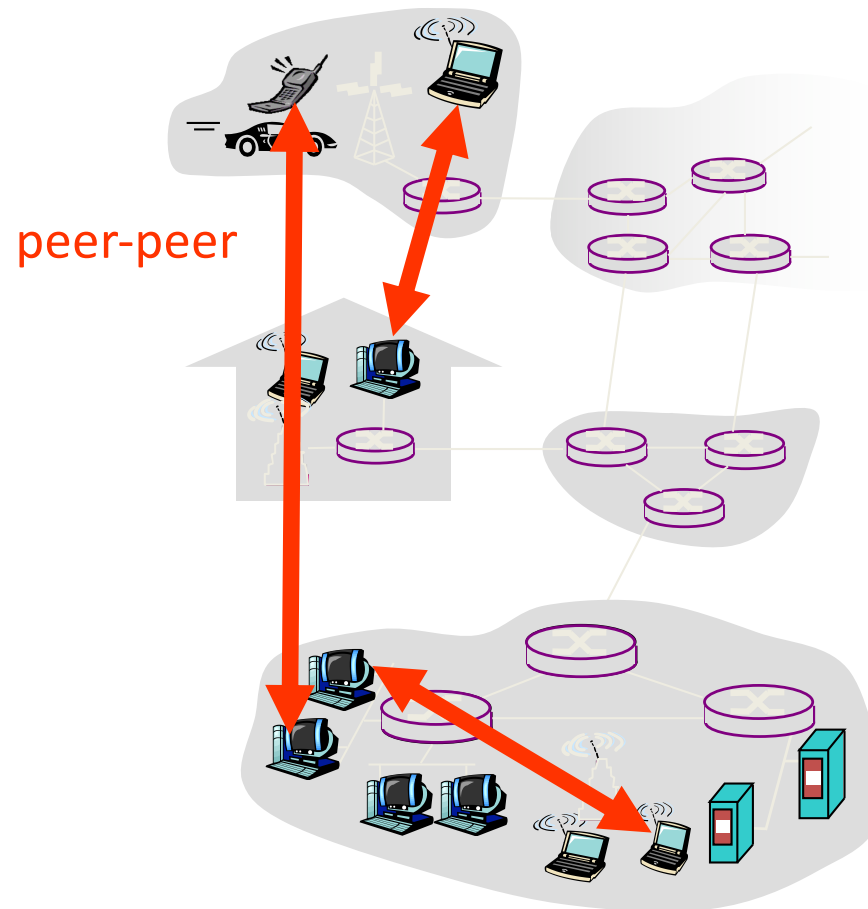
What about my aDSL/CABLE/etc it combines voice and data?

Phone company **controls** the multiplexing on the line
and throughout their own network too.....

P2P – efficient network use that
annoys the ISP

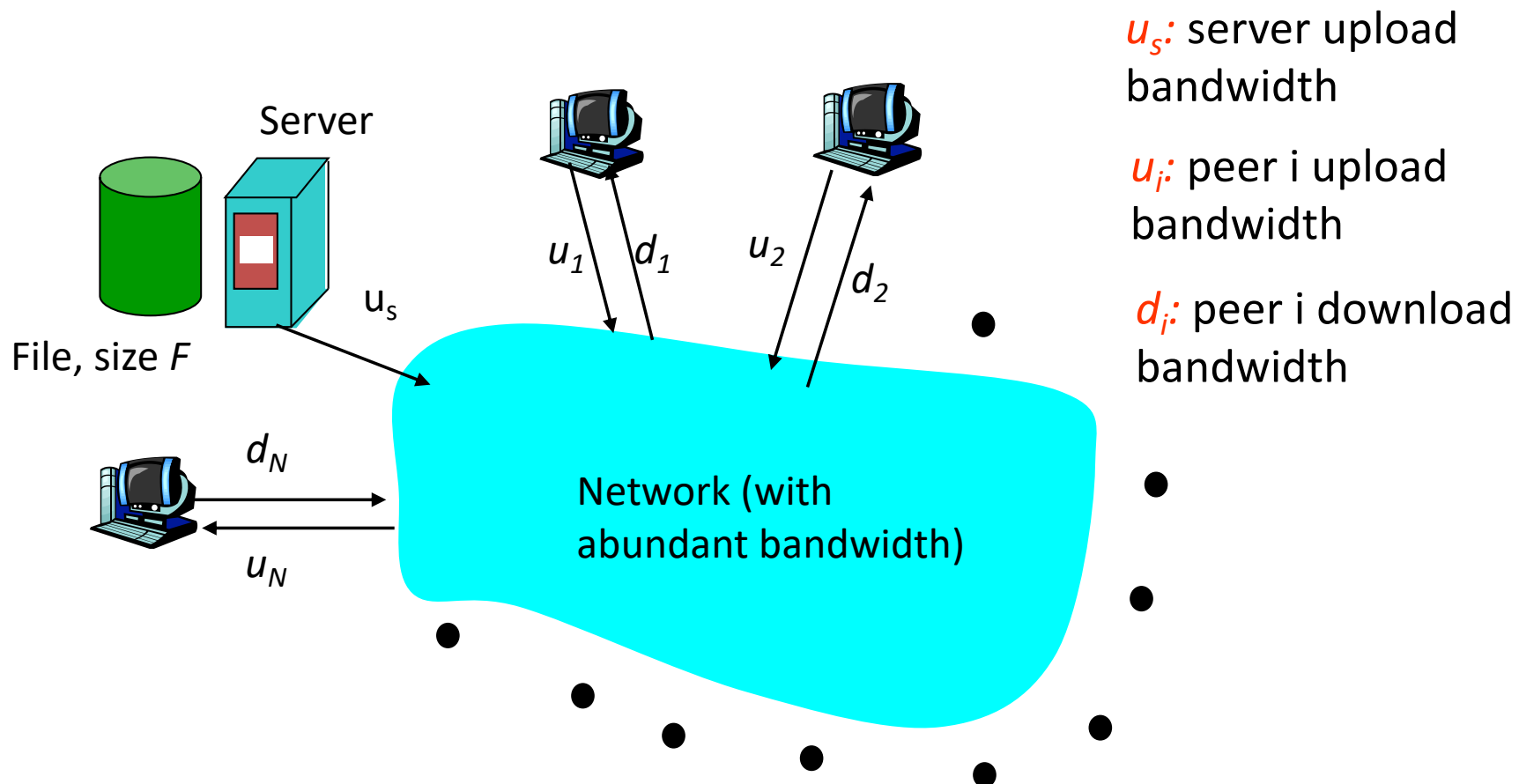
Pure P2P architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses
- Three topics:
 - File distribution
 - Searching for information
 - Case Study: Skype



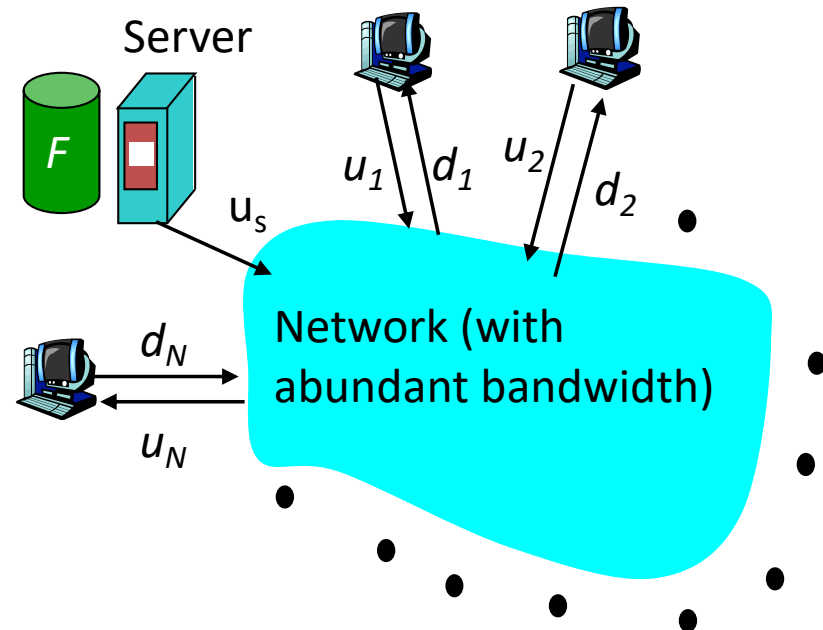
File Distribution: Server-Client vs P2P

Question : How much time to distribute file from one server to N peers?



File distribution time: server-client

- server sequentially sends N copies:
 - NF/u_s time
- client i takes F/d_i time to download

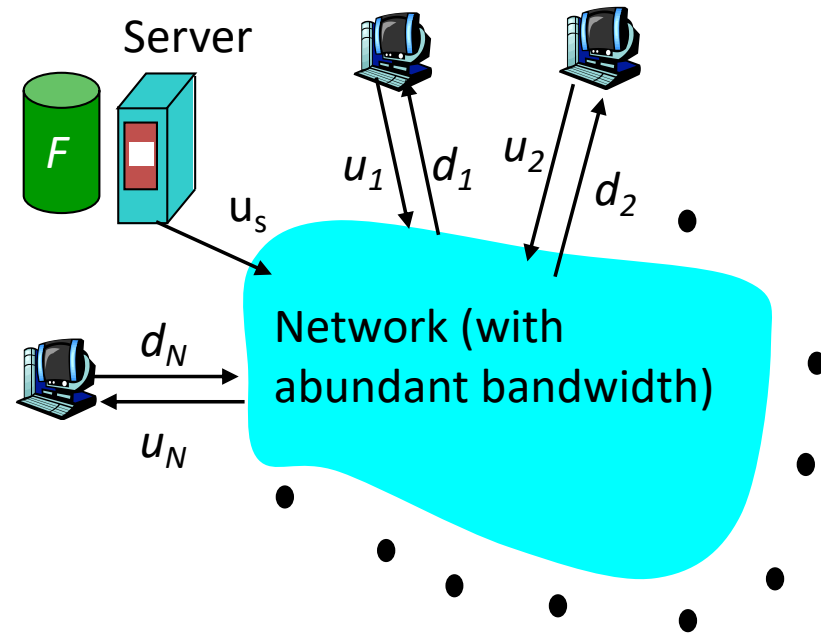


Time to distribute F
to N clients using
client/server approach $= d_{cs} = \max \{ NF/u_s, F/\min(d_i) \}$

increases linearly in N
(for large N)

File distribution time: P2P

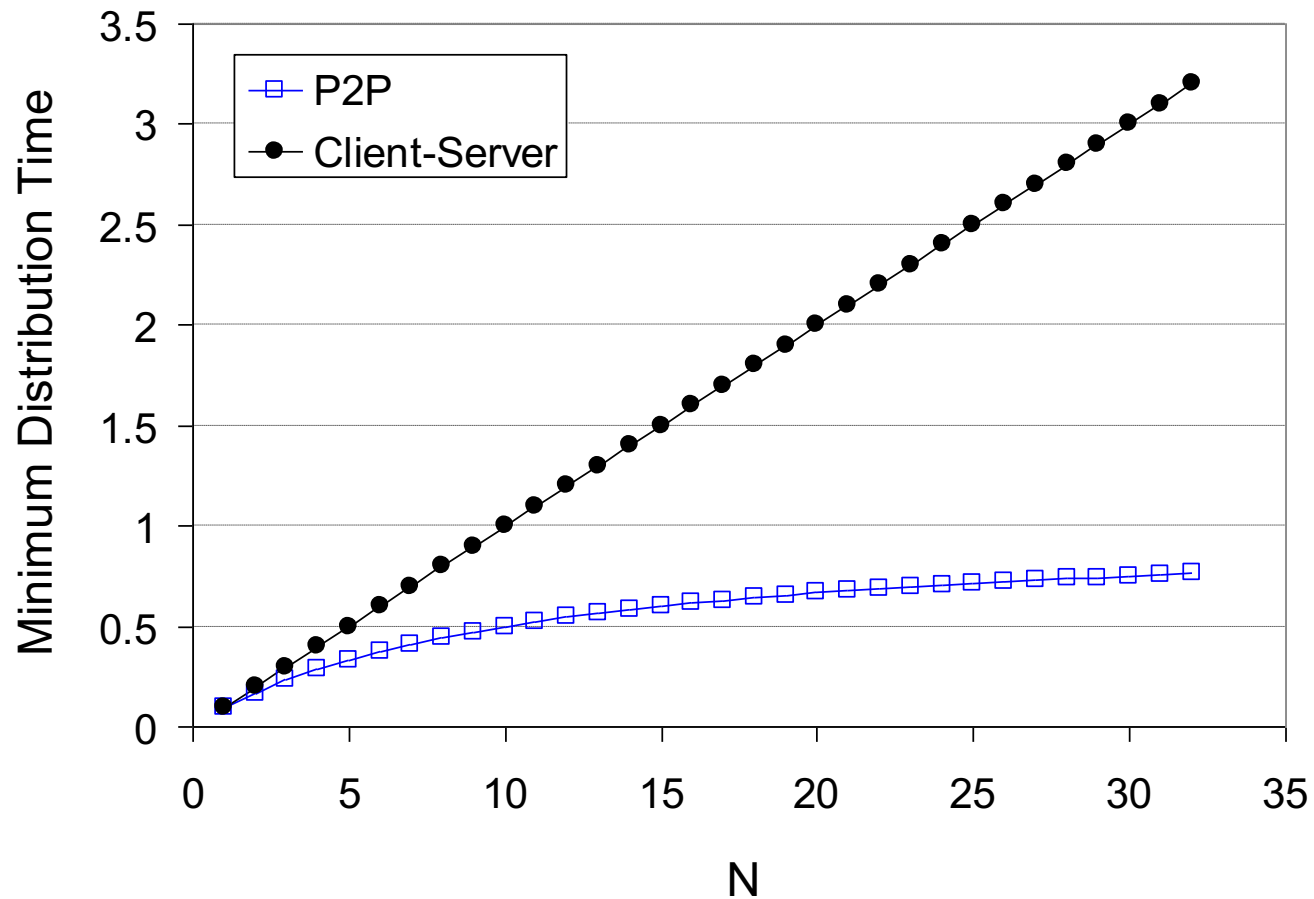
- server must send one copy:
 F/u_s time
- client i takes F/d_i time to download
- NF bits must be downloaded (aggregate)
r fastest possible upload rate: $u_s + \sum u_i$



$$d_{\text{P2P}} = \max \left\{ F/u_s, F/\min(d_i), NF/(u_s + \sum u_i) \right\}$$

Server-client vs. P2P: example

Client upload rate = u , $F/u = 1$ hour, $u_s = 10u$, $d_{\min} \geq u_s$



Distributed Hash Table (DHT)

- DHT = distributed P2P database
- Database has (key, value) pairs;
 - key: ss number; value: human name
 - key: content type; value: IP address
- Peers query DB with key
 - DB returns values that match the key
- Peers can also insert (key, value) peers

Distributed Hash Table (DHT)

- DHT = distributed P2P database
- Database has (key, value) pairs;
 - key: ss number; value: human name
 - key: content type; value: IP address
- Peers query DB with key
 - DB returns values that match the key
- Peers can also insert (key, value) peers

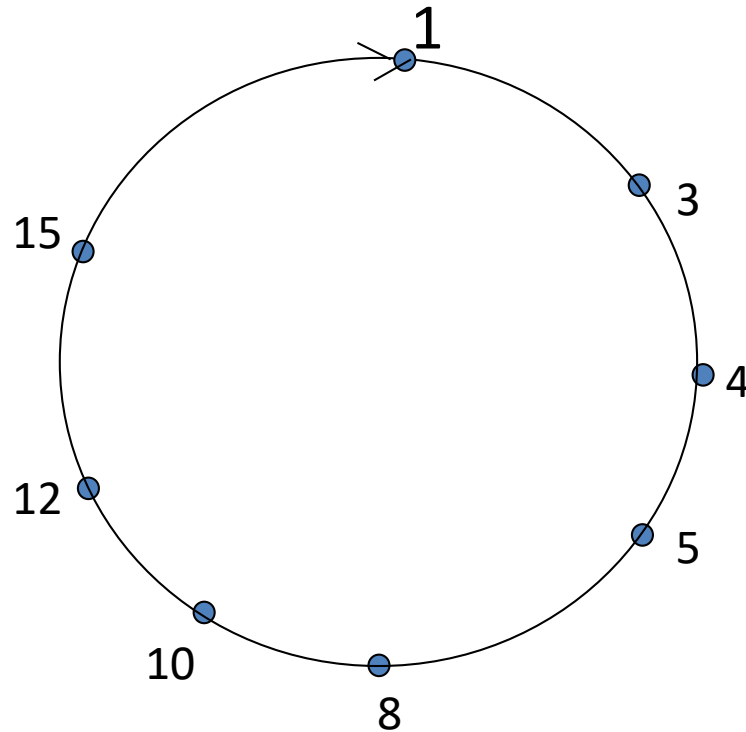
DHT Identifiers

- Assign integer identifier to each peer in range $[0, 2^n - 1]$.
 - Each identifier can be represented by n bits.
- Require each key to be an integer in **same range**.
- To get integer keys, hash original key.
 - eg, $\text{key} = h(\text{"Game of Thrones season 29"})$
 - This is why they call it a distributed “hash” table

How to assign keys to peers?

- Central issue:
 - Assigning (key, value) pairs to peers.
- Rule: assign key to the peer that has the **closest** ID.
- Convention in lecture: closest is the **immediate successor** of the key.
- Ex: $n=4$; peers: 1,3,4,5,8,10,12,14;
 - key = 13, then successor peer = 14
 - key = 15, then successor peer = 1

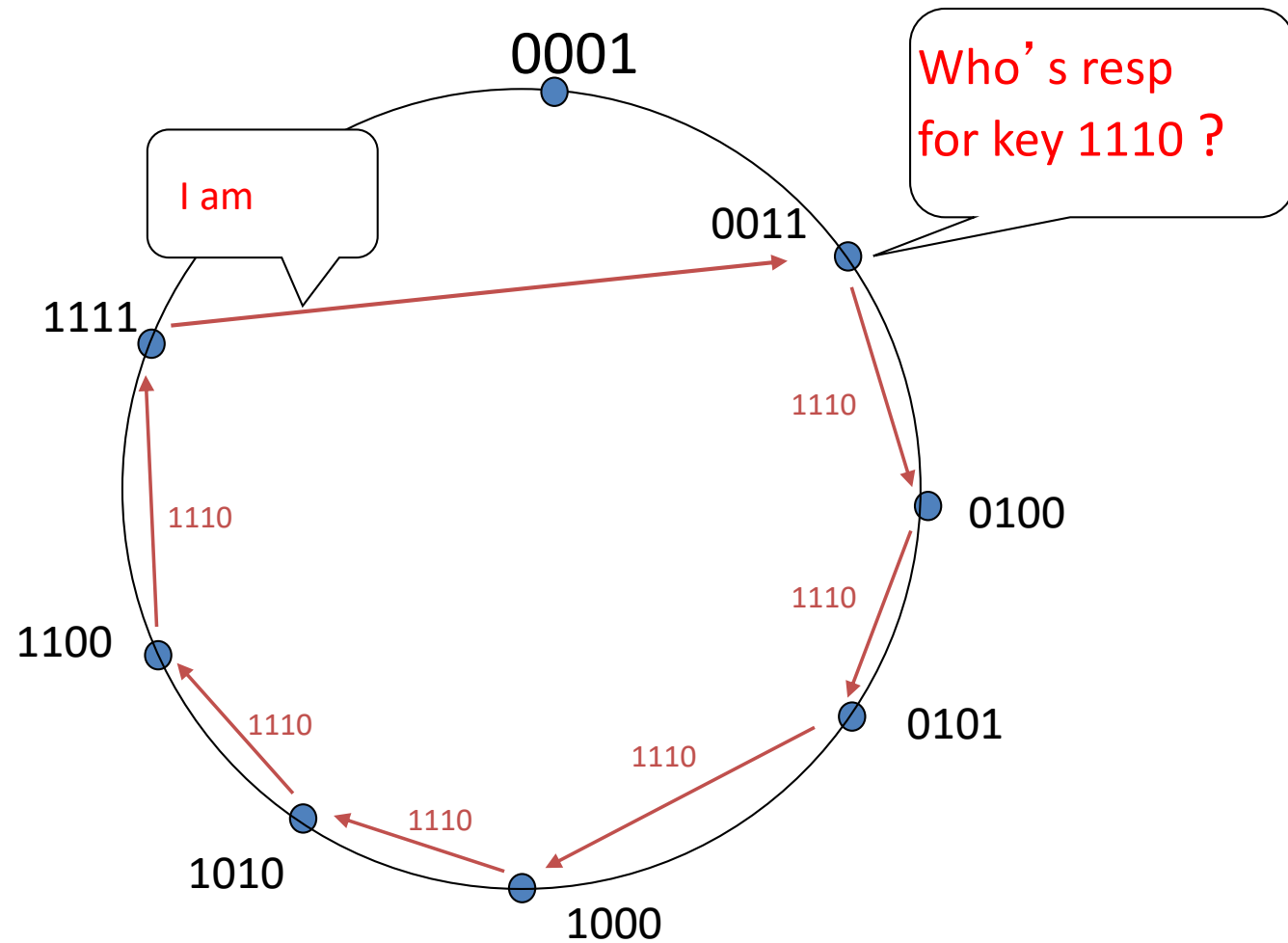
Circular DHT (1)



- Each peer *only* aware of immediate successor and predecessor.
- “Overlay network”

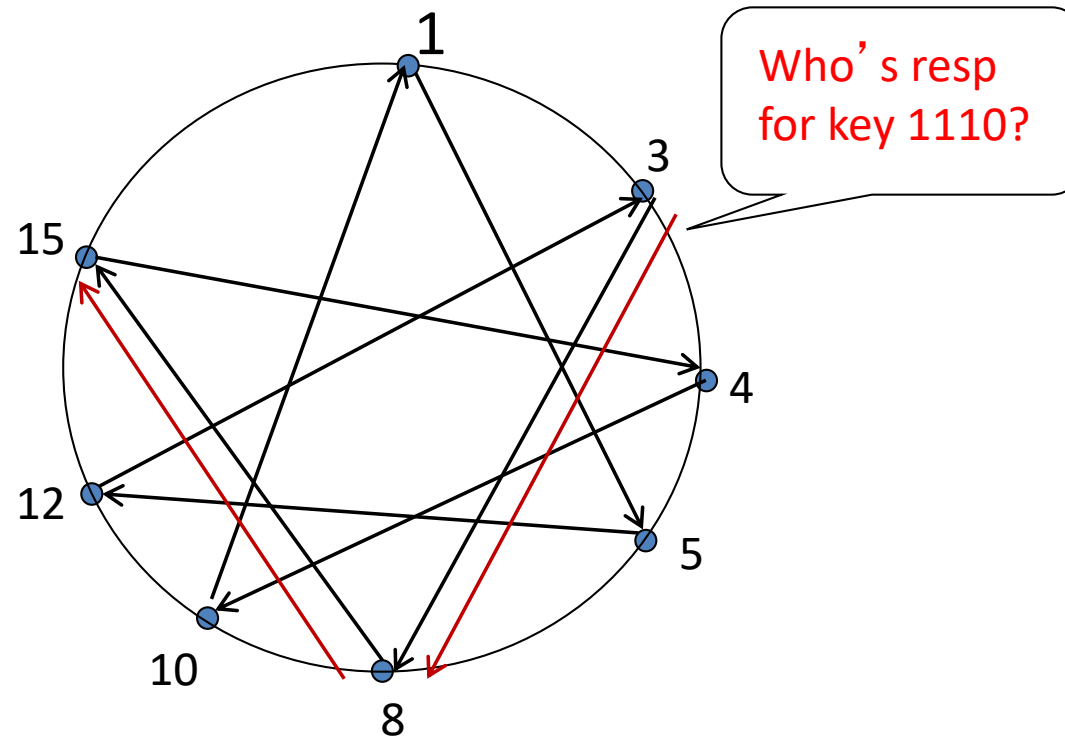
Circle DHT (2)

$O(N)$ messages
on avg to resolve
query, when there
are N peers



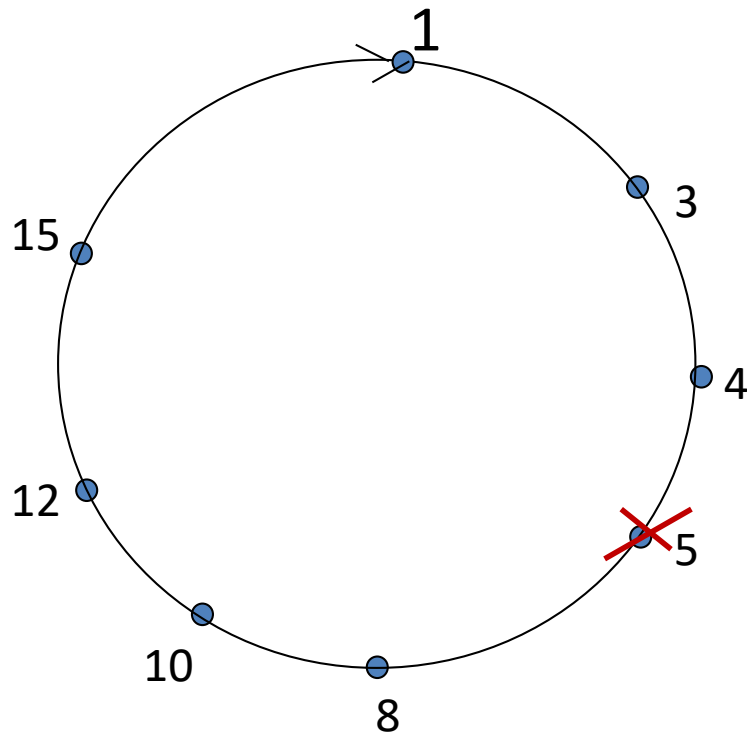
Define closest
as closest
successor

Circular DHT with Shortcuts



- Each peer keeps track of IP addresses of predecessor, successor, short cuts.
- Reduced from 6 to 2 messages.
- Possible to design shortcuts so $O(\log N)$ neighbors, $O(\log N)$ messages in query

Peer Churn



- To handle peer churn, require each peer to know the IP address of its two successors.
- Each peer periodically pings its two successors to see if they are still alive.

- Peer 5 abruptly leaves
- Peer 4 detects; makes 8 its immediate successor; asks 8 who its immediate successor is; makes 8's immediate successor its second successor.
- What if peer 13 wants to join?

Summary.

- Apps need protocols too
- We covered examples from
 - Traditional Applications (web)
 - Scaling and Speeding the web (CDN/Cache tricks)
- Infrastructure Services (DNS)
 - Cache and Hierarchy
- Multimedia Applications (SIP)
 - Extremely hard to do better than worst-effort
- P2P Network examples