

Fall 2018-2019: EE471/CS471/CS573

Computer Networks: Principles & Practice

Slide set 04

Tariq Mahmood Jadoon and Zartash Afzal Uzmi

SBA School of Science and Engineering
LUMS

Material with thanks to Sylvia Ratnasamy

Web and HTTP

Web Components and HTTP

- Web infrastructure:
 - Clients
 - Servers
- Web page
 - Objects (identified by URLs)
 - Base object (html file) and embedded objects
- Protocol for exchanging information: HTTP

Uniform record/resource locator (URL)

protocol://hostname[:port]/directory-path/resource

- Hierarchical hostnames to include anything in the filesystem
 - <http://trend.lums.edu.pk:8080/folder-003/sub/docs/flow.html>
 - <http://venus.lums.edu.pk/cs471/quiz01.pdf>
- Extend to program executions as well...
 - <http://www.cool-translation.com/en-us?sublime>
 - http://www.ebay.com/itm/131524956165?_trksid=p2057872.m2849.l2649&var=430914648976&ssPageName=STRK%3AKEBIDX%3AIT

Uniform record/resource locator (URL)

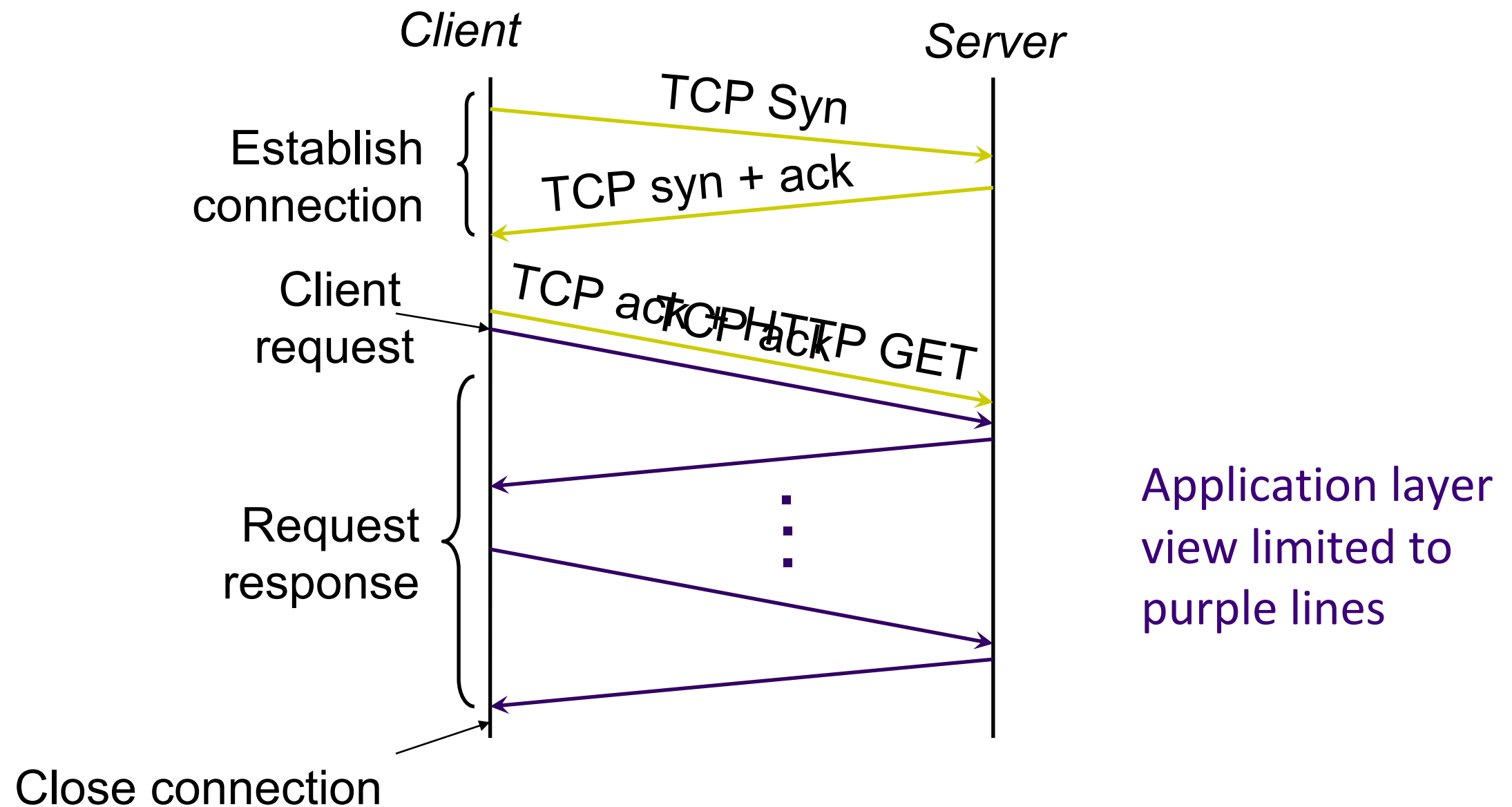
protocol://hostname[:port]/directory-path/resource

- **protocol:** http, ftp, https, smtp, rftp, etc.
- **hostname:** DNS name (or domain name) or IP address
- **port:** default values available, e.g., 80 for http, 443 for https
- **directory-path:** hierarchical, reflecting file system (on the server side)
- **resource:** identifies the desired resource in the filesystem, e.g., an html file, a pdf file, an image file, etc.

Hyper Text Transfer Protocol (HTTP)

- Client-server architecture
 - server is “always on” and “well known”
 - Apache, Nginx
 - clients initiate contact to server
 - Firefox, Chrome/Chromium, Internet Explorer, Safari
- request/reply protocol
 - Runs over TCP, Port 80 (default port)
- Stateless
- ASCII format

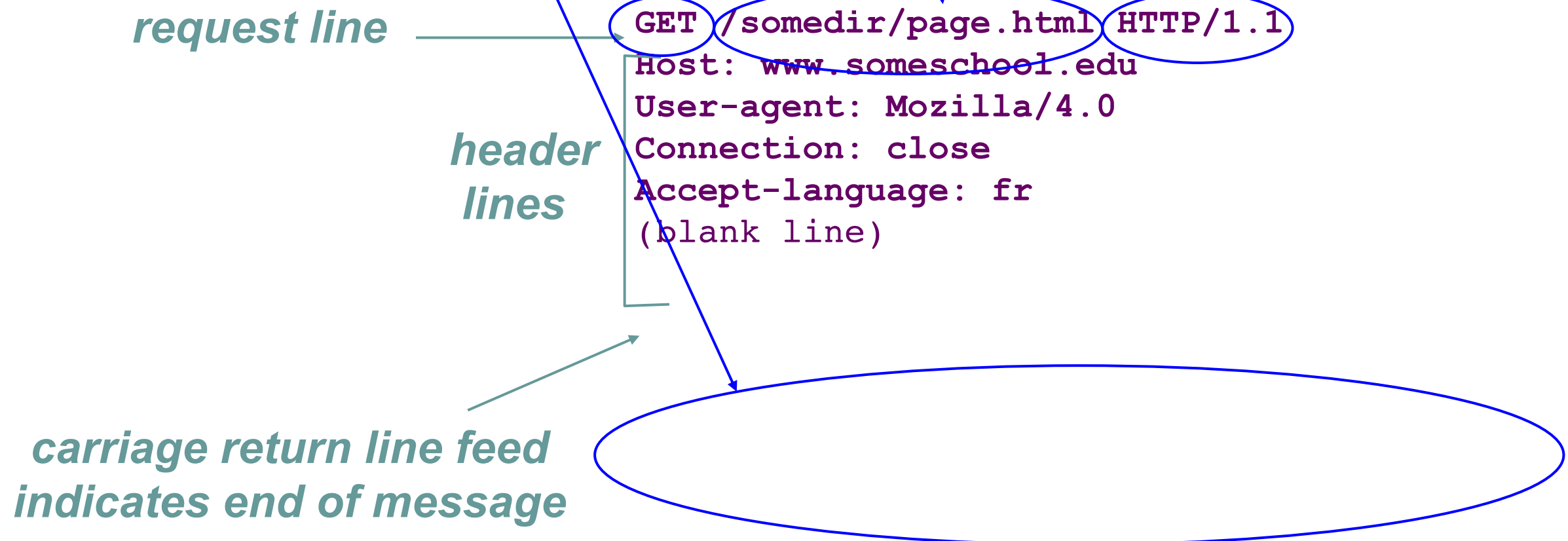
Steps in HTTP Request/Response



Client-to-Server Communication

- HTTP Request Message

- Request line: method, resource, and protocol version
- Request headers: provide information or modify request
- Body: optional data (e.g., to “POST” data to the server)



Server-to-Client Communication

- HTTP Response Message
 - Status line: protocol version, status code, status phrase
 - Response headers: provide information
 - Body: optional data

status line
(protocol, status code,
status phrase)

header lines

data

e.g., requested HTML file

HTTP/1.1 200 OK

Connection: close

Date: Thu, 06 Aug 2006 12:00:15
GMT

Server: Apache/1.3.0 (Unix)

Last-Modified: Mon, 22 Jun 2006

...

Content-Length: 6821

Content-Type: text/html

(blank line)

data data data data data ...

How to view HTTP interaction?

- **Using wireshark**

- Ensure no network activity on your computer
- Use your browser to fetch a page
- Dig down into the packets captured

- **Composing http request yourself**

- Connect to server using command line
 - telnet <server-name, e.g., nova.stanford.edu> 80
- GET <path-to-resource> http/1.0 (or http/1.1)

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, ...

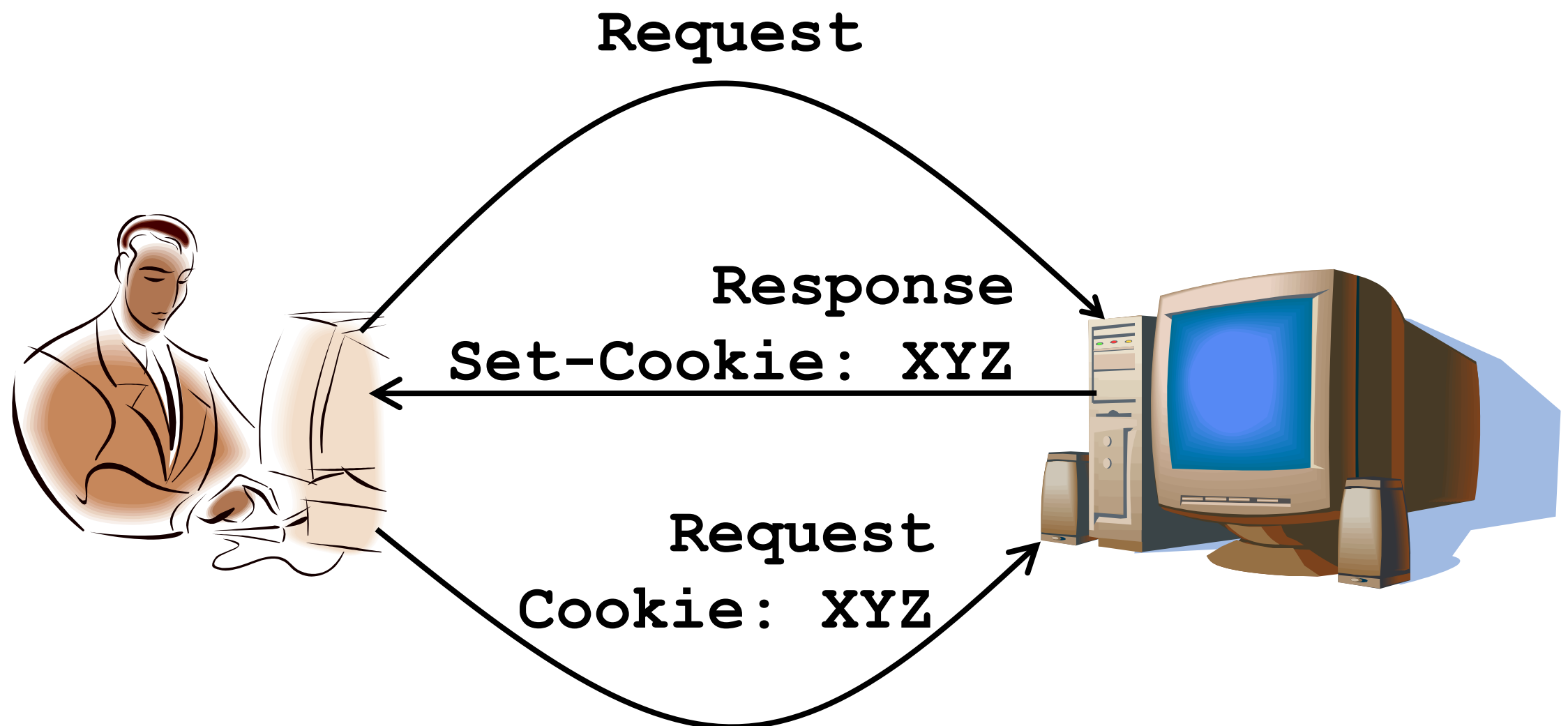
Question

- How does a stateless protocol keep state?

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 Issues


Performance Goals

- User
 - fast downloads (not identical to low-latency commn.!.)
 - high availability
- Content provider
 - happy users (hence, above)
 - cost-effective infrastructure
- Network (secondary)
 - avoid overload

Solutions?

**Improve HTTP to
compensate for
TCP's weak spots**



- **User**
 - fast downloads (not identical to low-latency commn.!)
 - high availability
- **Content provider**
 - happy users (hence, above)
 - cost-effective infrastructure
- **Network (secondary)**
 - avoid overload

Solutions?

- User

- fast downloads (not identical to low-latency commn.!!)
- high availability

Improve HTTP to
compensate for
TCP's weak spots

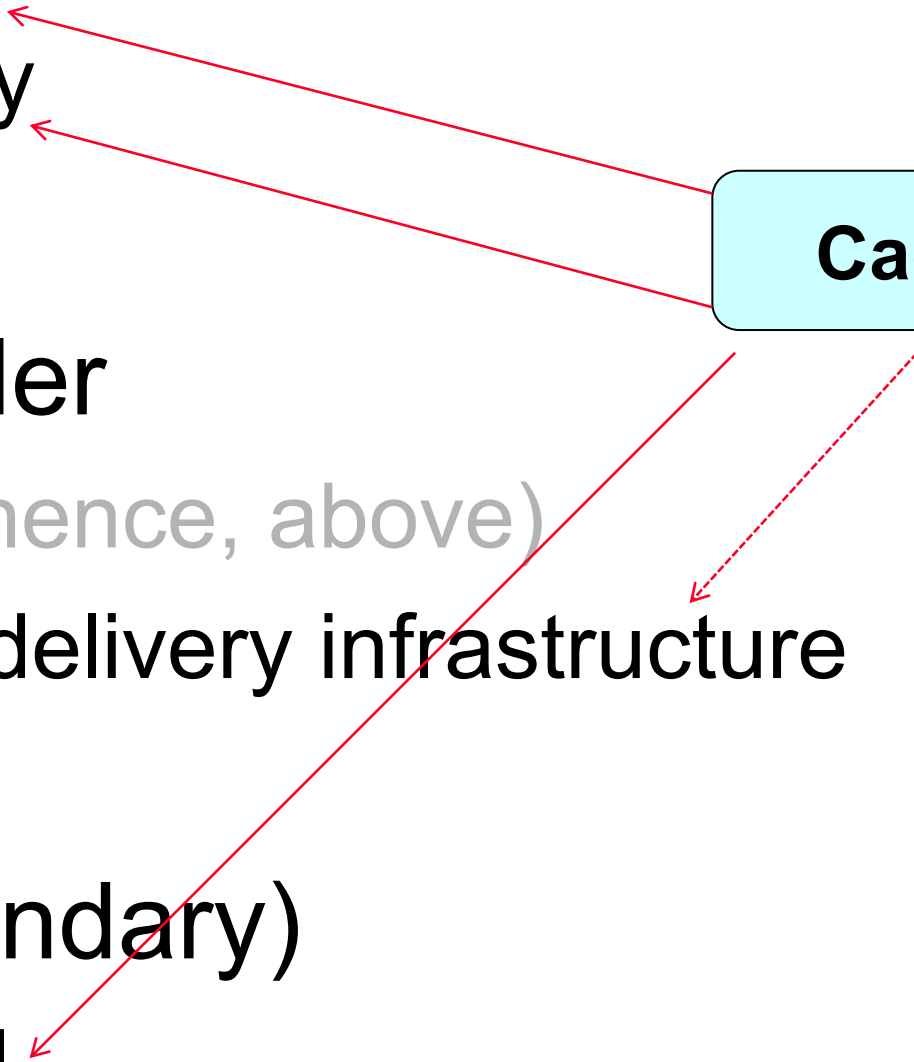
Caching and Replication

- Content provider

- happy users (hence, above)
- cost-effective delivery infrastructure

- Network (secondary)

- avoid overload



Solutions?

- User

- fast downloads (not identical to low-latency commn.!)
 - high availability

**Improve HTTP to
compensate for
TCP's weak spots**

Caching and Replication

- Content provider

- happy users (hence, above)
- cost-effective delivery infrastructure

- Network (secondary)

- avoid overload

**Exploit economies of scale
(Webhosting, CDNs, datacenters)**

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*
- New TCP connection per (small) object!
 - http/1.0

Improving HTTP Performance:

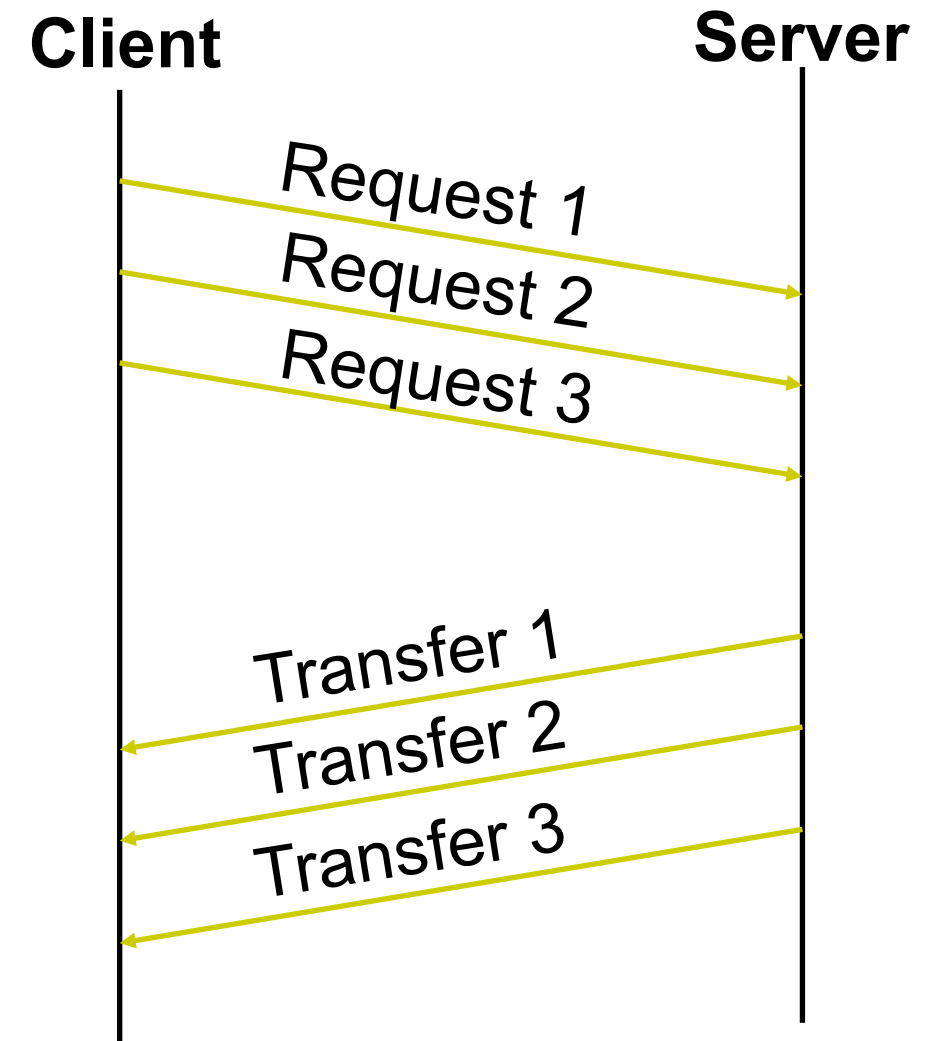
Persistent Connections

- 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

Improving HTTP Performance:

Pipelined Requests & Responses

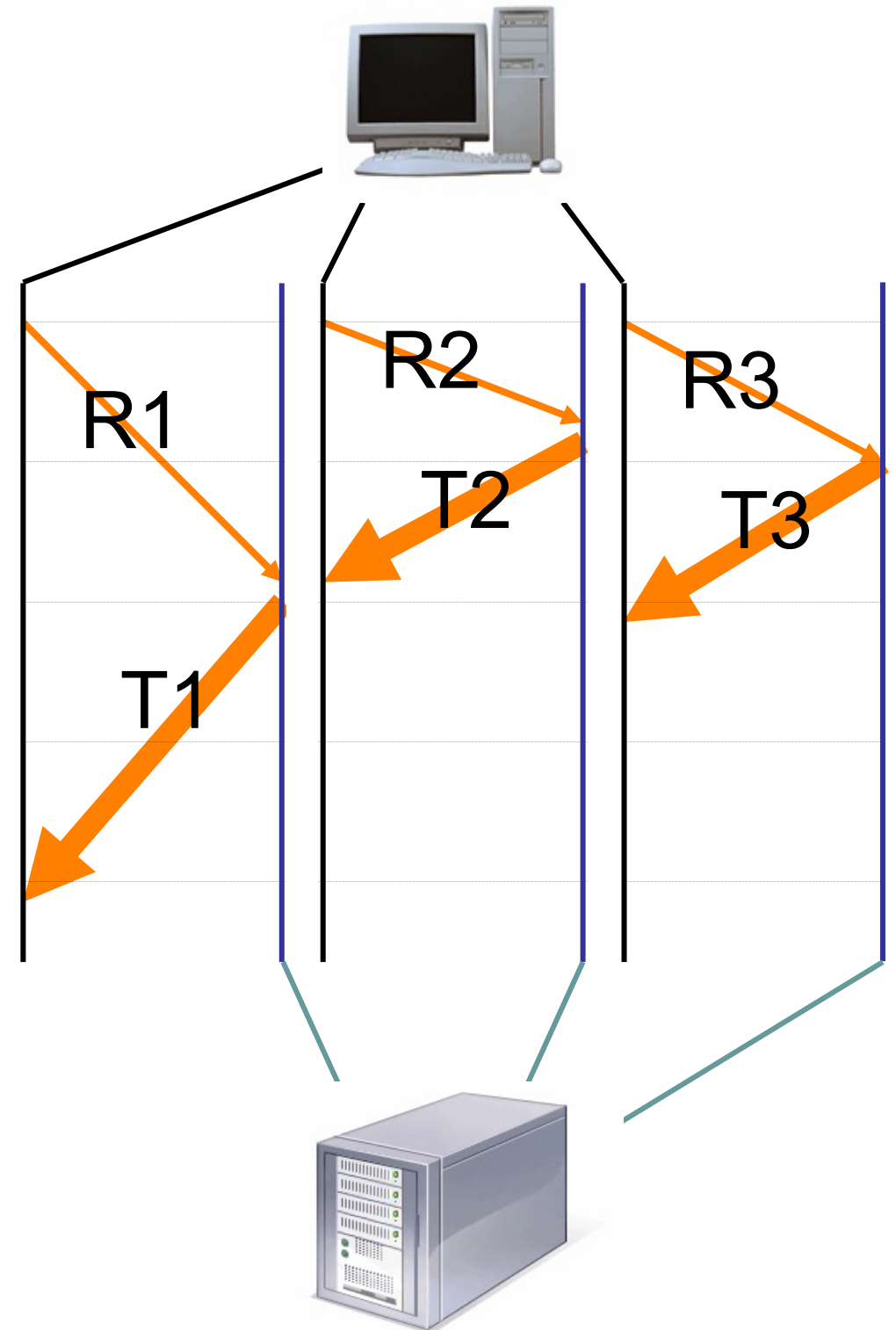
- **Batch** requests and responses to reduce the number of packets
- **Multiple requests** can be contained in one TCP segment



Improving HTTP Performance:

Concurrent Requests & Responses

- Use multiple connections *in parallel*
 - Does not necessarily maintain order of responses
 - Technically, not part of HTTP
-
- Client = 😊
 - Content provider = 😊
 - Network = 😞 Why?



Scorecard: Getting n Small Objects

Time dominated by latency

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

Scorecard: Getting n Large Objects

Time dominated by bandwidth (B)

- One-at-a-time: $\sim nF/B$ (F: size of each object)
- M concurrent: $\sim [n/M] (F/B)$
 - assuming shared with large population of users
 - and each TCP connection gets the same bandwidth
- Pipelined and/or persistent: $\sim nF/B$
 - The only thing that helps is getting more bandwidth..

Solutions?

- User

- fast downloads (not identical to low-latency commn.!!)
- high availability

Improve HTTP to
compensate for
TCP's weak spots

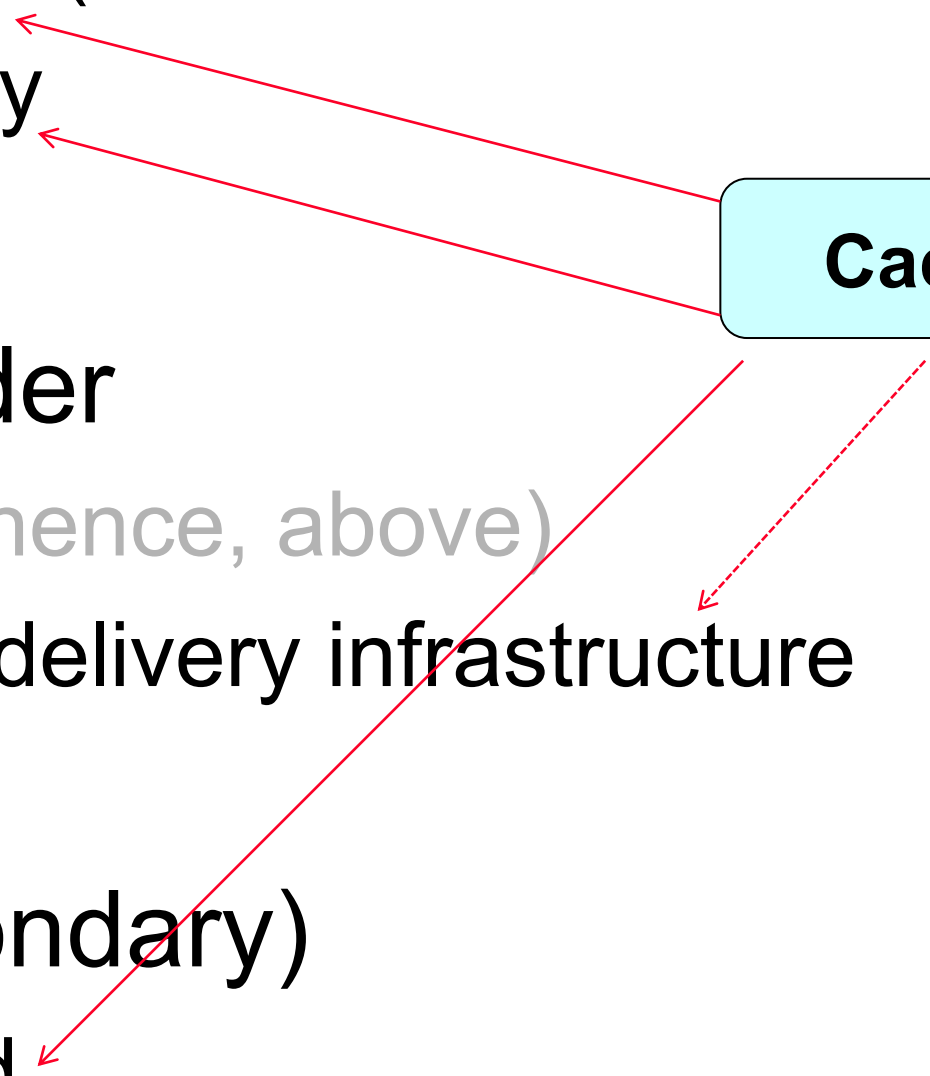
Caching and Replication

- Content provider

- happy users (hence, above)
- cost-effective delivery infrastructure

- Network (secondary)

- avoid overload



Improving HTTP Performance:

Caching

- Why does caching work?
 - Exploits *locality of reference*
- How well does caching work?
 - Very well, up to a limit
 - Large overlap in content

Improving HTTP Performance:

Caching: How

- Modifier to GET requests:
 - `If-modified-since` – returns “not modified” if resource not modified since specified time
- Client specifies “if-modified-since” time in request
 - Server compares this against “last modified” time of resource
 - Server returns “Not Modified” if resource has not changed
 - or a “OK” with the latest version otherwise

Improving HTTP Performance:

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

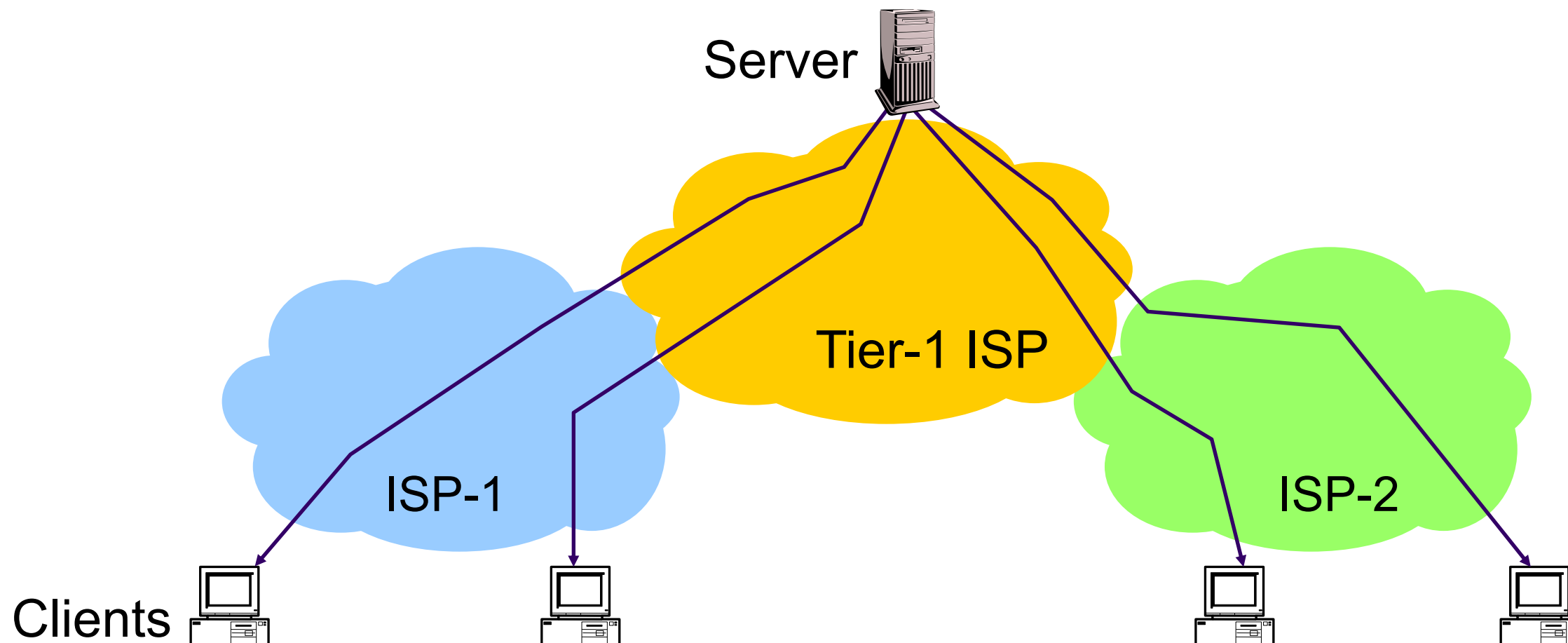
Improving HTTP Performance:

Caching: Where?

- Options
 - Client
 - Forward proxies
 - Reverse proxies
 - Content Distribution Network

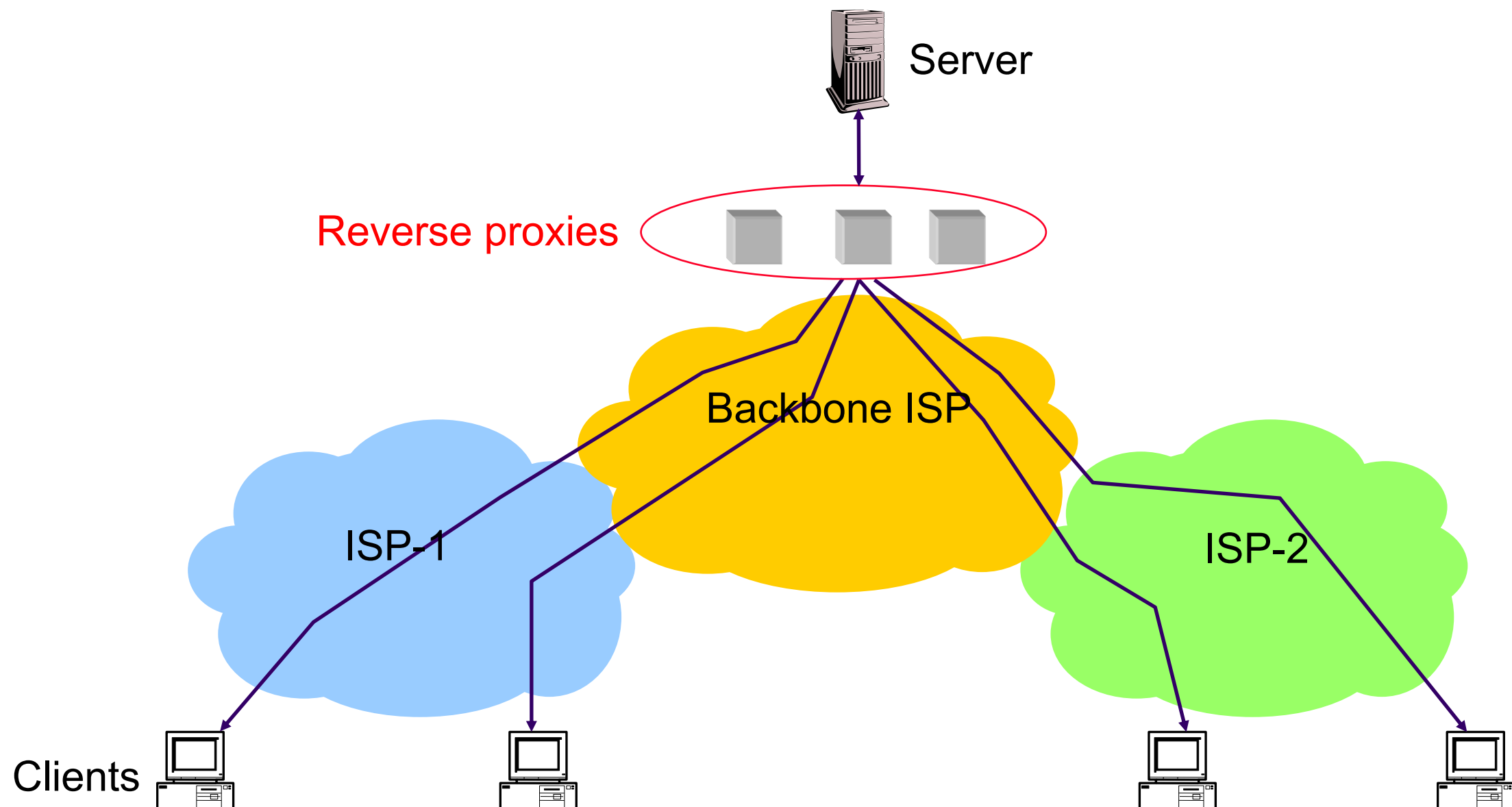
Improving HTTP Performance: Caching: Where?

- Baseline: Many clients access the same information
 - Generate unnecessary server and network load
 - Clients experience unnecessary latency



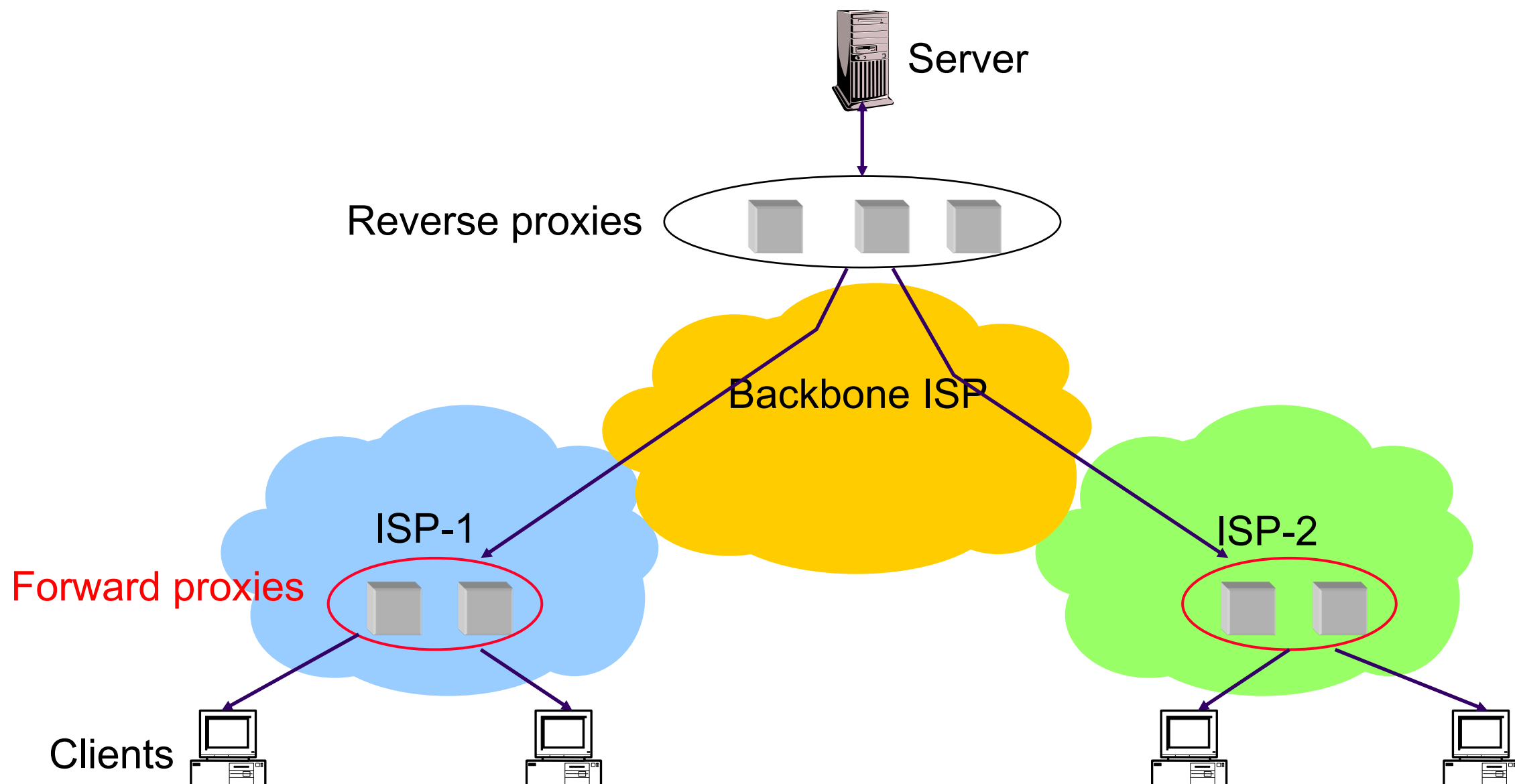
Improving HTTP Performance: Caching with Reverse Proxies

- Cache documents close to **server**
→ **decrease server load**
- Typically done by content provider



Improving HTTP Performance: Caching with Forward Proxies

- Cache documents close to **clients**
 - **reduce network traffic and decrease latency**
- Typically done by ISPs or enterprises



Improving HTTP Performance: Replication

- Replicate popular Web site across many machines
 - Spreads load on servers
 - Places content closer to clients
 - Helps when content isn't cacheable (e.g., dynamic pages)
- Problem: Which client be directed to which replica?
 - Pair clients with nearby servers. Why?
 - Two benefits: load balancing across replicas, lower latency
- How to implement? common solution:
 - DNS returns different addresses based on client's geo location, server load, *etc.*

Improving HTTP Performance: Content Distribution Networks

- Caching and replication as a service
- Large-scale distributed storage infrastructure (usually) administered by one entity
 - *e.g.*, Akamai has servers in 20,000+ locations
- Combination of (pull) caching and (push) replication
 - **Pull:** Direct result of clients' requests
 - **Push:** Expectation of high access rate
- Also do some processing
 - Handle *dynamic* web pages
 - *Transcoding*

Improving HTTP Performance:

CDN Example – Akamai

- Akamai creates new domain names for each client
 - e.g., *a128.g.akamai.net* for *cnn.com*
- 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.cnn.com/image-of-the-day.gif* becomes *http://a128.g.akamai.net/image-of-the-day.gif*
- Requests now sent to CDN's infrastructure...

Cost-Effective Content Delivery

- General theme: multiple sites hosted on shared physical infrastructure
 - efficiency of statistical multiplexing
 - economies of scale (volume pricing, *etc.*)
 - amortization of human operator costs
- Examples:
 - Web hosting companies
 - CDNs
 - Cloud infrastructure

DNS

Host Names & Addresses

- Host addresses: *e.g., 169.229.131.109*
 - a number used by protocols
 - conforms to network structure (the “where”)
- Host names: *e.g., www.usc.edu or venus.lums.edu.pk*
 - mnemonic name usable by humans
 - conforms to organizational structure (the “who”)
- The Domain Name System (DNS) is how we map from one to the other
 - a **directory service** for hosts on the Internet

Why bother?

- Convenience
 - Easier to remember www.google.com than 74.125.239.49
- Provides a level of indirection!
 - Decoupled names from addresses
 - Many uses beyond just naming a specific host

DNS: Early days

- Mappings stored in a hosts.txt file (in /etc/hosts)
 - maintained by the Stanford Research Institute (SRI)
 - new versions periodically copied from SRI (via FTP)
- As the Internet grew this system broke down
 - SRI couldn't handle the load
 - conflicts in selecting names
 - hosts had inaccurate copies of hosts.txt
- The Domain Name System (DNS) was invented to fix this
 - First name server implementation done by 4 UCB students!

Goals?

- Scalable
 - many names
 - many updates
 - many users creating names
 - many users looking up names
- Highly available
- Correct
 - no naming conflicts (uniqueness)
 - consistency
- Lookups are fast

How?

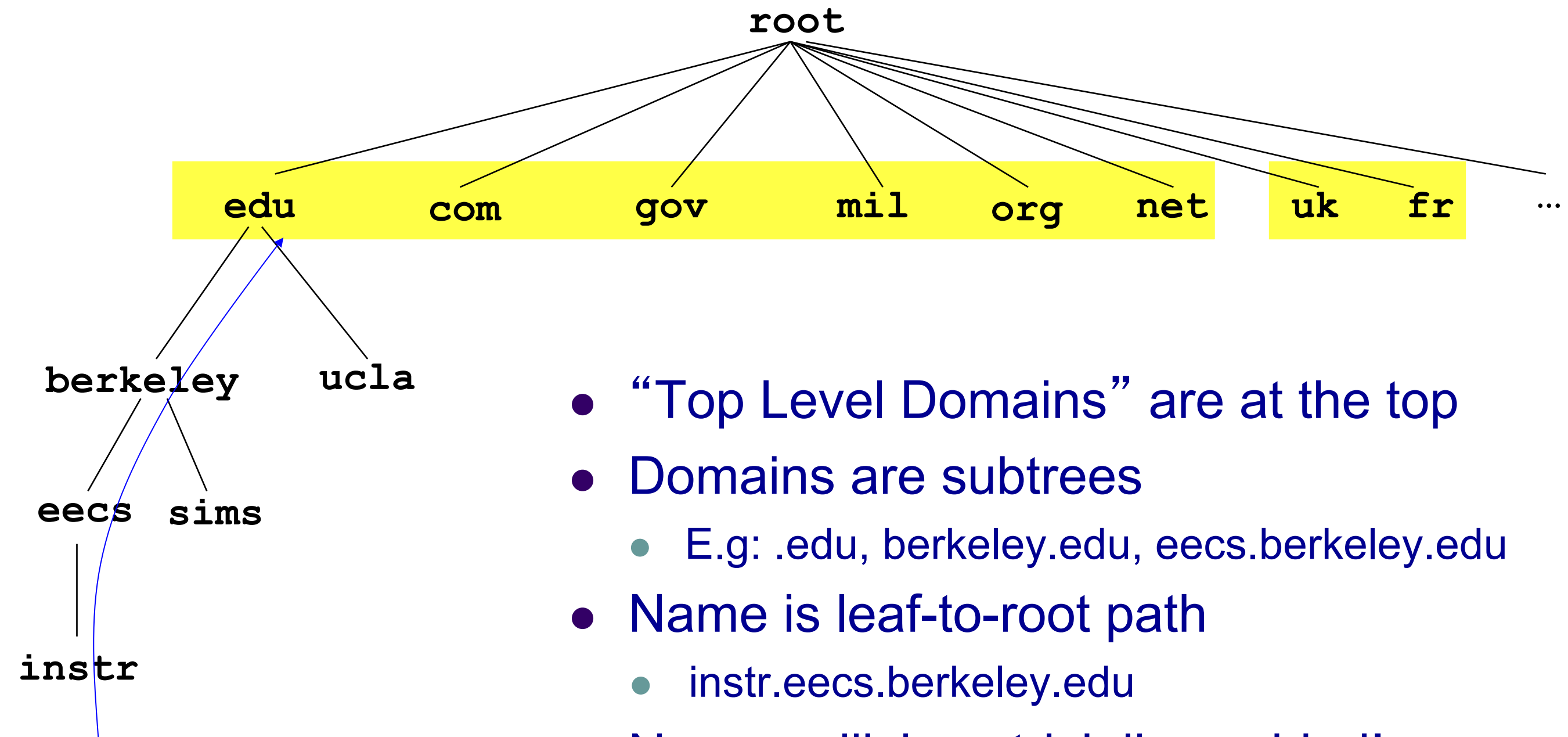
- Partition the namespace
- Distribute administration of each partition
 - Autonomy to update my own (machines') names
 - Don't have to track everybody's updates
- Distribute name resolution for each partition
- *How should we partition things?*

Key idea: hierarchical distribution

Three intertwined hierarchies

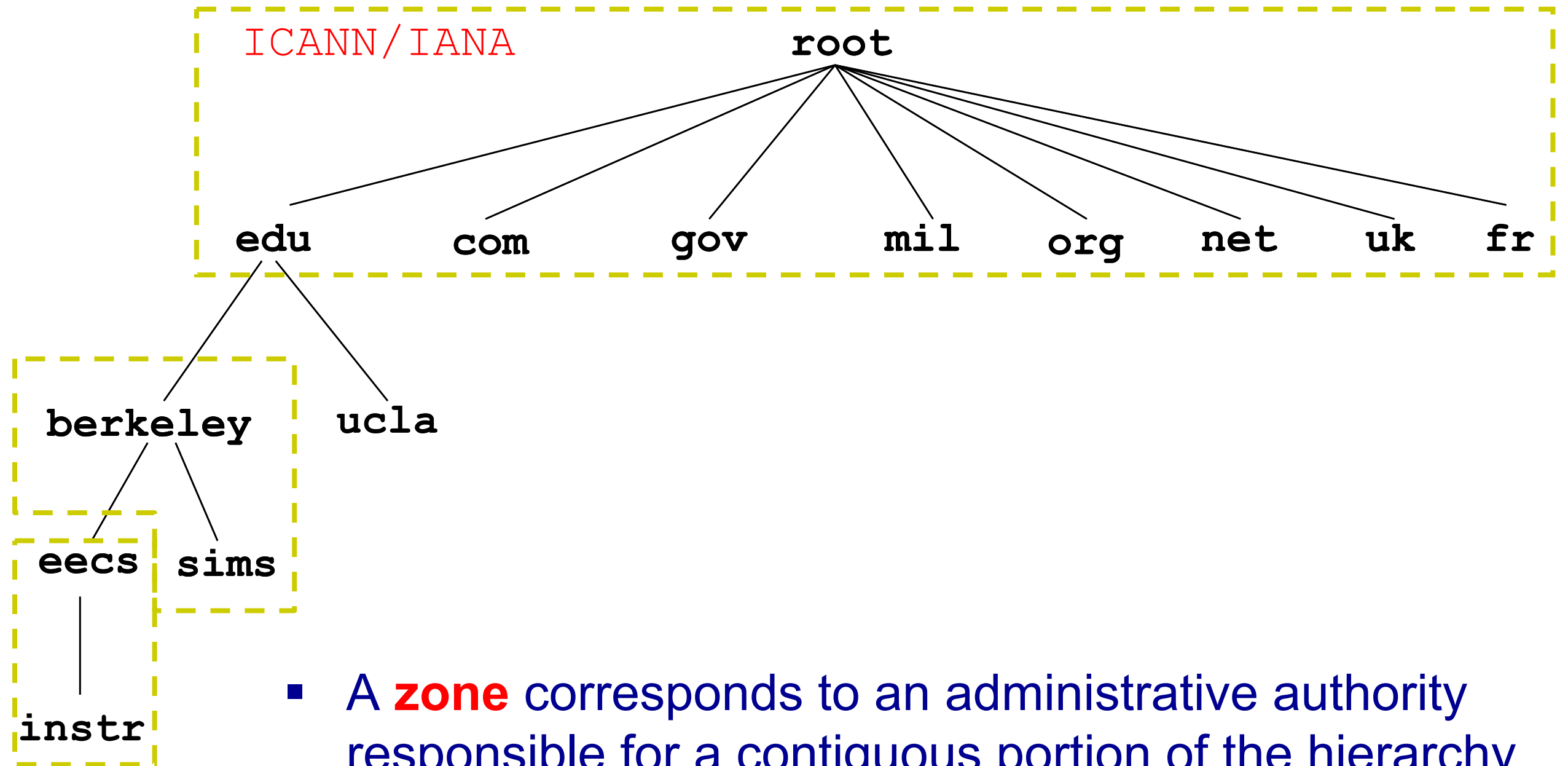
- Hierarchical namespace
 - As opposed to original flat namespace
- Hierarchically administered
 - As opposed to centralized administrator
- Hierarchy of servers
 - As opposed to centralized storage

Hierarchical Namespace



- “Top Level Domains” are at the top
- Domains are subtrees
 - E.g: .edu, berkeley.edu, eeecs.berkeley.edu
- Name is leaf-to-root path
 - instr.eeecs.berkeley.edu
- Name collisions trivially avoided!
 - each domain’s responsibility

Hierarchical Administration



- A **zone** corresponds to an administrative authority responsible for a contiguous portion of the hierarchy
- E.g.: UCB controls **.berkeley.edu* and **.sims.berkeley.edu* while EECS controls **.eeecs.berkeley.edu*

Server Hierarchy

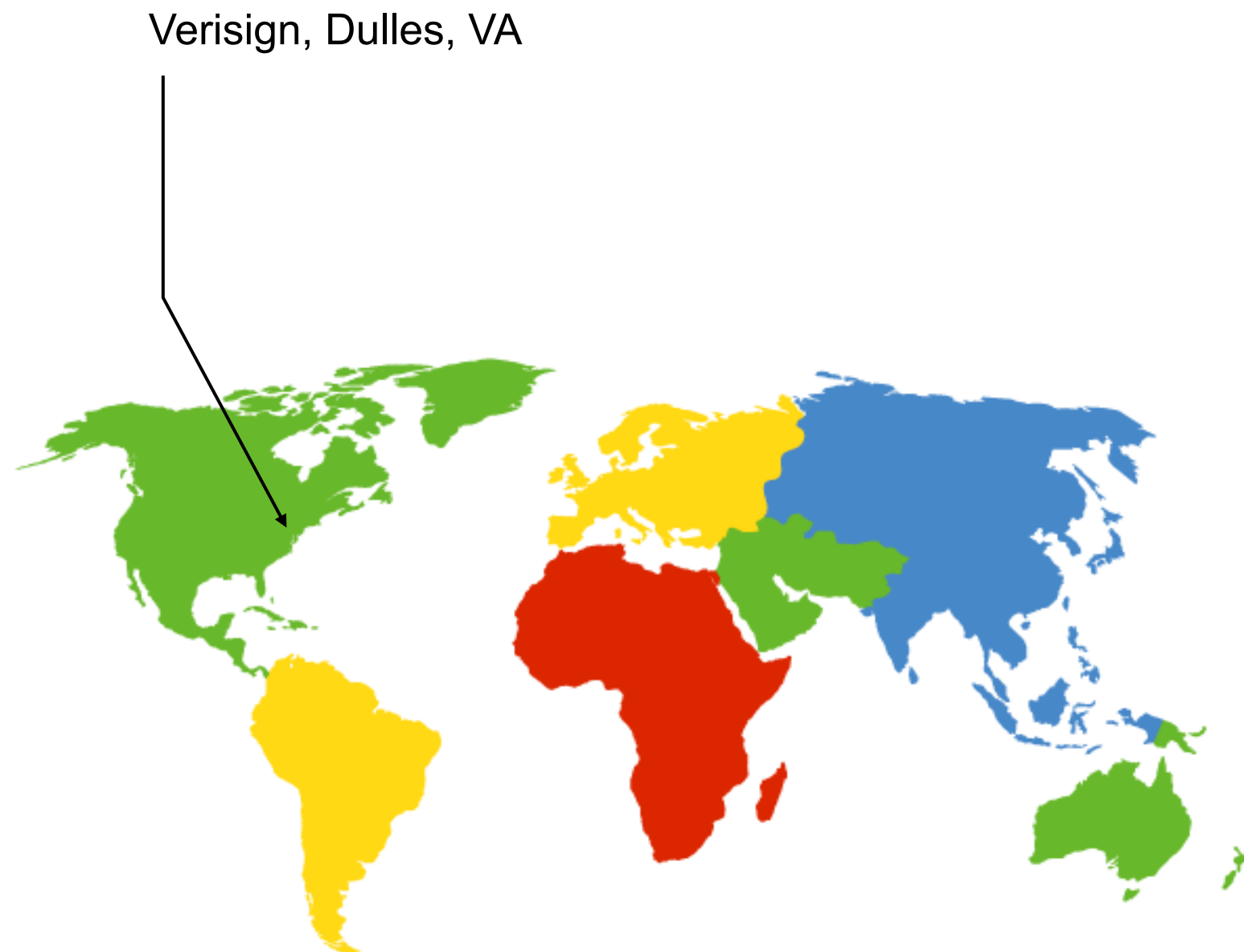
- Top of hierarchy: Root servers
 - Location hardwired into other servers
- Next Level: Top-level domain (TLD) servers
 - .com, .edu, etc.
 - Managed professionally
- Bottom Level: **Authoritative** DNS servers
 - Actually store the name-to-address mapping
 - Maintained by the corresponding administrative authority

Server Hierarchy

- Every server knows the address of the root name server
 - Root servers know the address of all TLD servers
 - ...
 - An authoritative DNS server stores name-to-address mappings (“resource records”) for all DNS names in the domain that it has authority for
-
- Each server stores a subset of the total DNS database
 - Each server can discover the server(s) responsible for any portion of the hierarchy

DNS Root

- Located in Virginia, USA



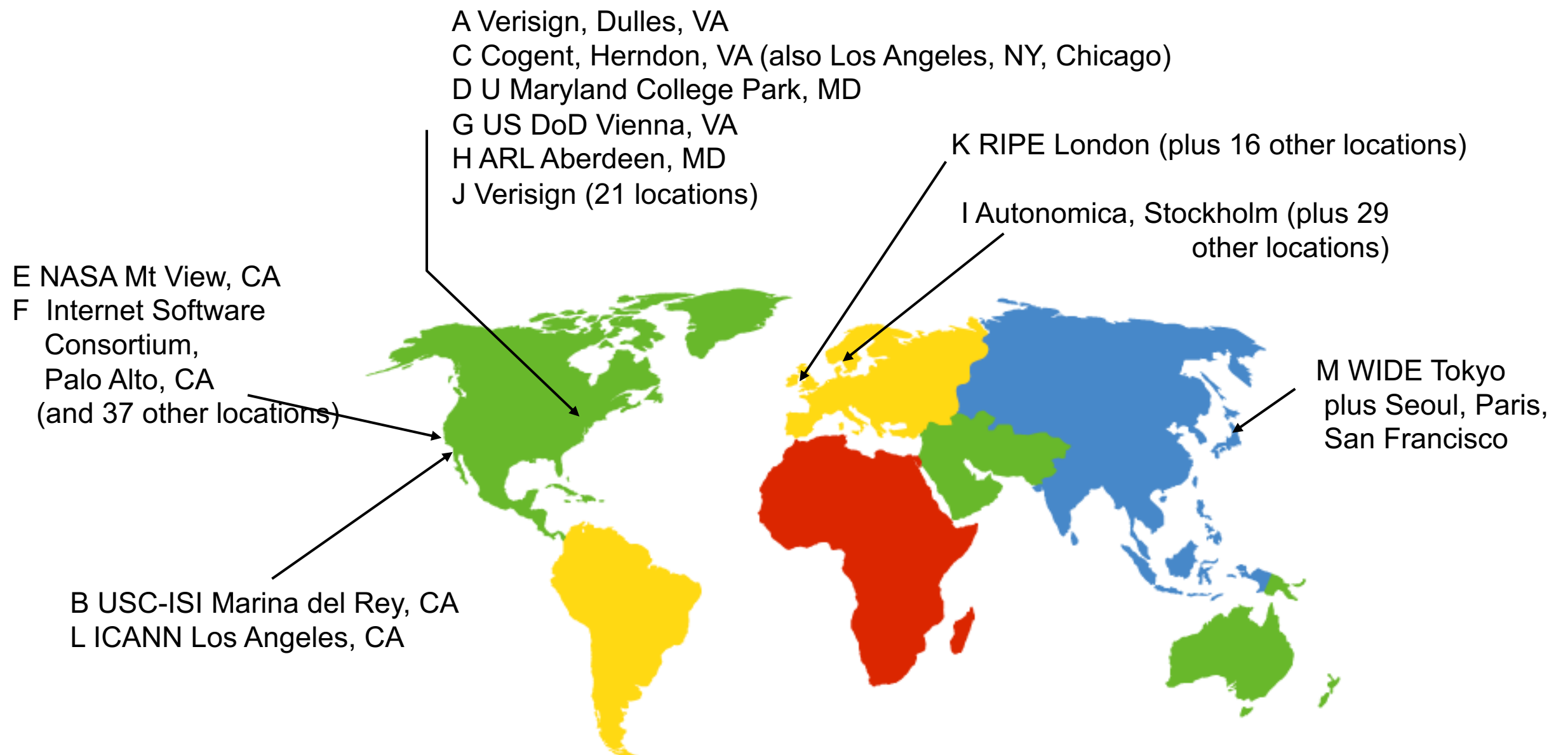
DNS Root Servers

- 13 root servers (labeled A-M; see <http://www.root-servers.org/>)



DNS Root Servers

- 13 root servers (labeled A-M; see <http://www.root-servers.org/>)
- Replicated via **any-casting**



Anycast in a nutshell

- Routing finds shortest paths to destination
- What happens if multiple machines *advertise (i.e., claim to have)* the same address?
- The network will deliver the packet to the closest machine with that address
- This is called “anycast”
 - Very robust
 - Requires no modification to routing algorithms

DNS Records

- DNS servers store **resource records (RRs)**
 - RR is (name, value, type, TTL)
- Type = A: (\rightarrow Address)
 - name = hostname
 - value = IP address
- Type = NS: (\rightarrow Name Server)
 - name = domain
 - value = name of dns server for domain

DNS Records (cont'd)

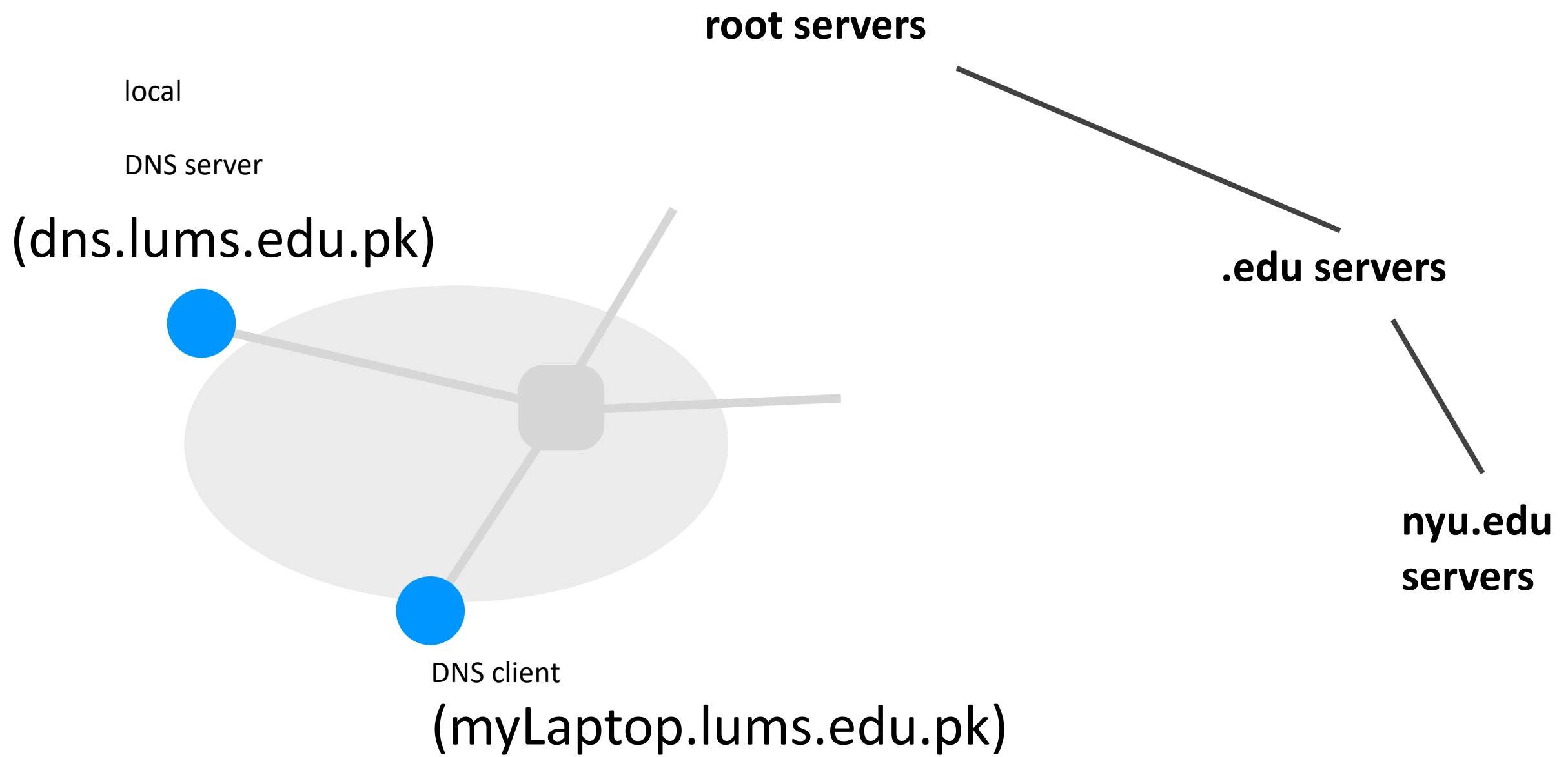
- Type = CNAME
 - name = alias name for a host
 - value = canonical name (often long/difficult)
- Type = MX: (\rightarrow *Mail eXchanger*)
 - name = domain in email address
 - value = name(s) of mail server(s)
 - Example: name is stanford.edu
 - Value: mail03.stanford.edu (hostname of machine)

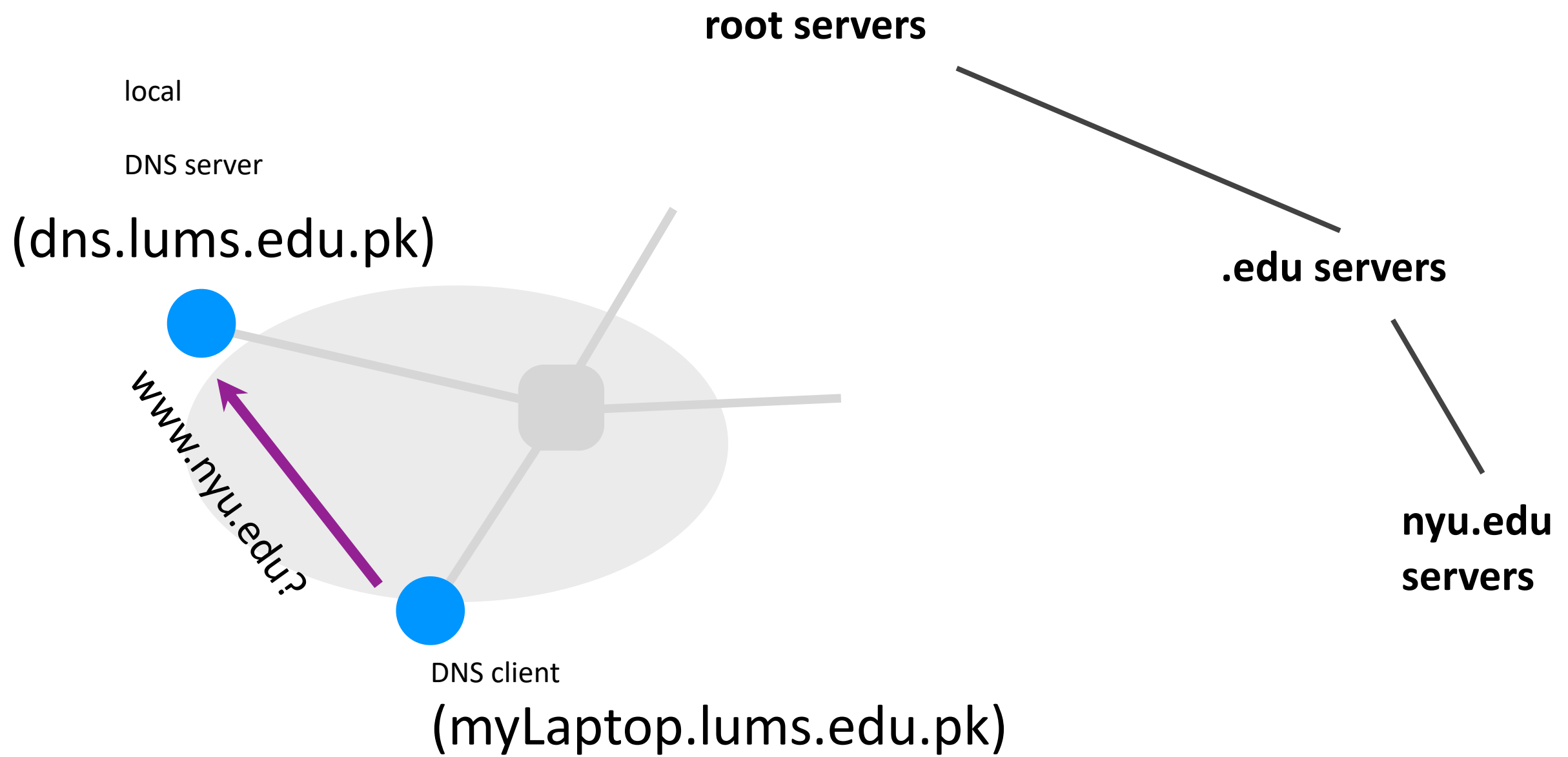
Inserting Resource Records into DNS

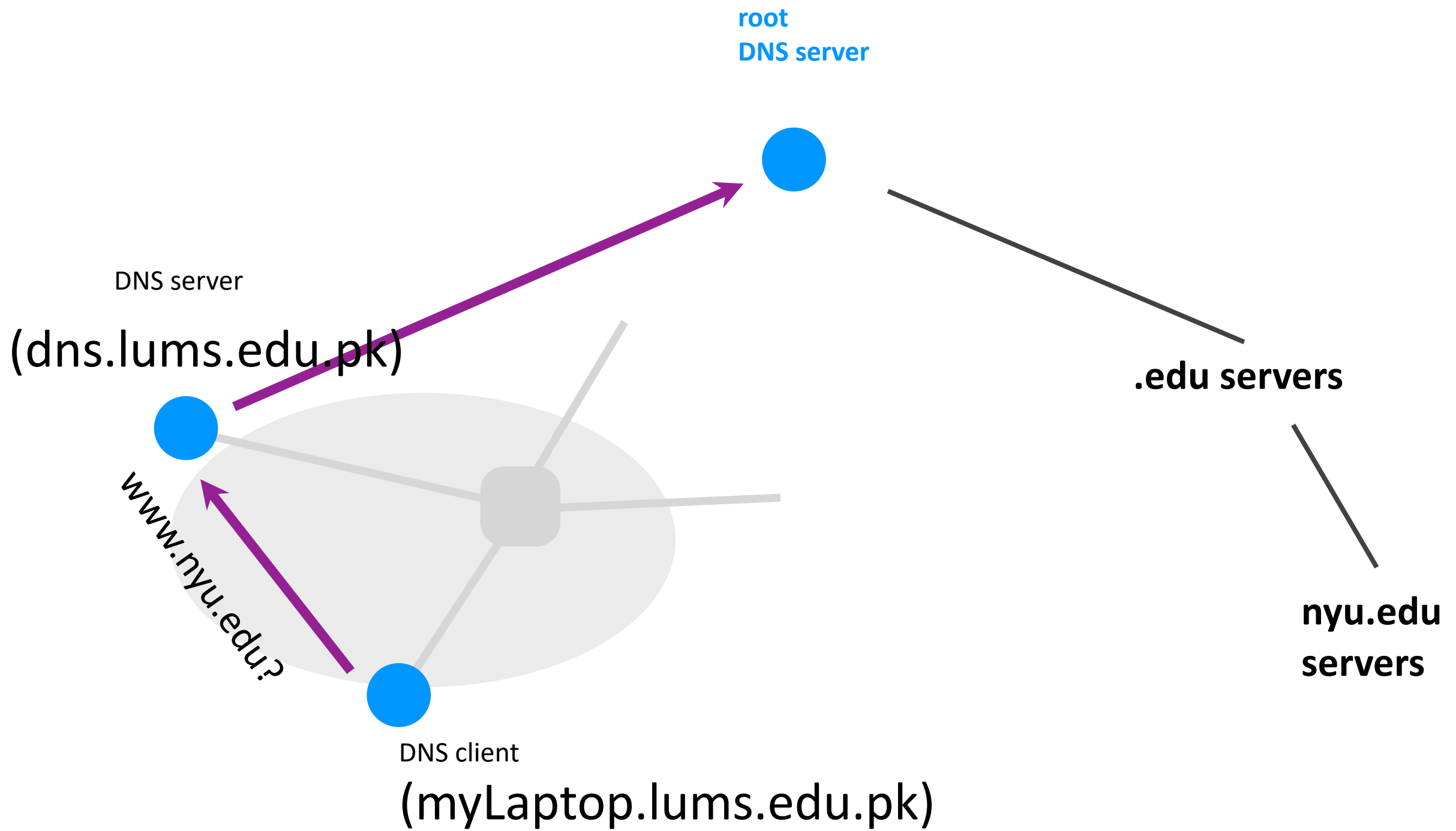
- Example: you just created company “FruitChat”
- You get a block of IP addresses from your ISP
 - say 212.44.9.128/25 (includes **212.44.9.128, 212.44.9.129**, etc.)
- Register **fruitchat.com** at registrar (e.g., Go Daddy)
 - Provide registrar with names and IP addresses of your authoritative name server(s)
 - Registrar inserts RR pairs into the **.com** TLD server:
 - (**fruitchat.com, dns1.fruitchat.com, NS**)
 - (**dns1.fruitchat.com, 212.44.9.129, A**)
- Store resource records in your server **dns1.fruitchat.com**
 - e.g., type A record for **www.fruitchat.com**
 - e.g., type MX record for **fruitchat.com**

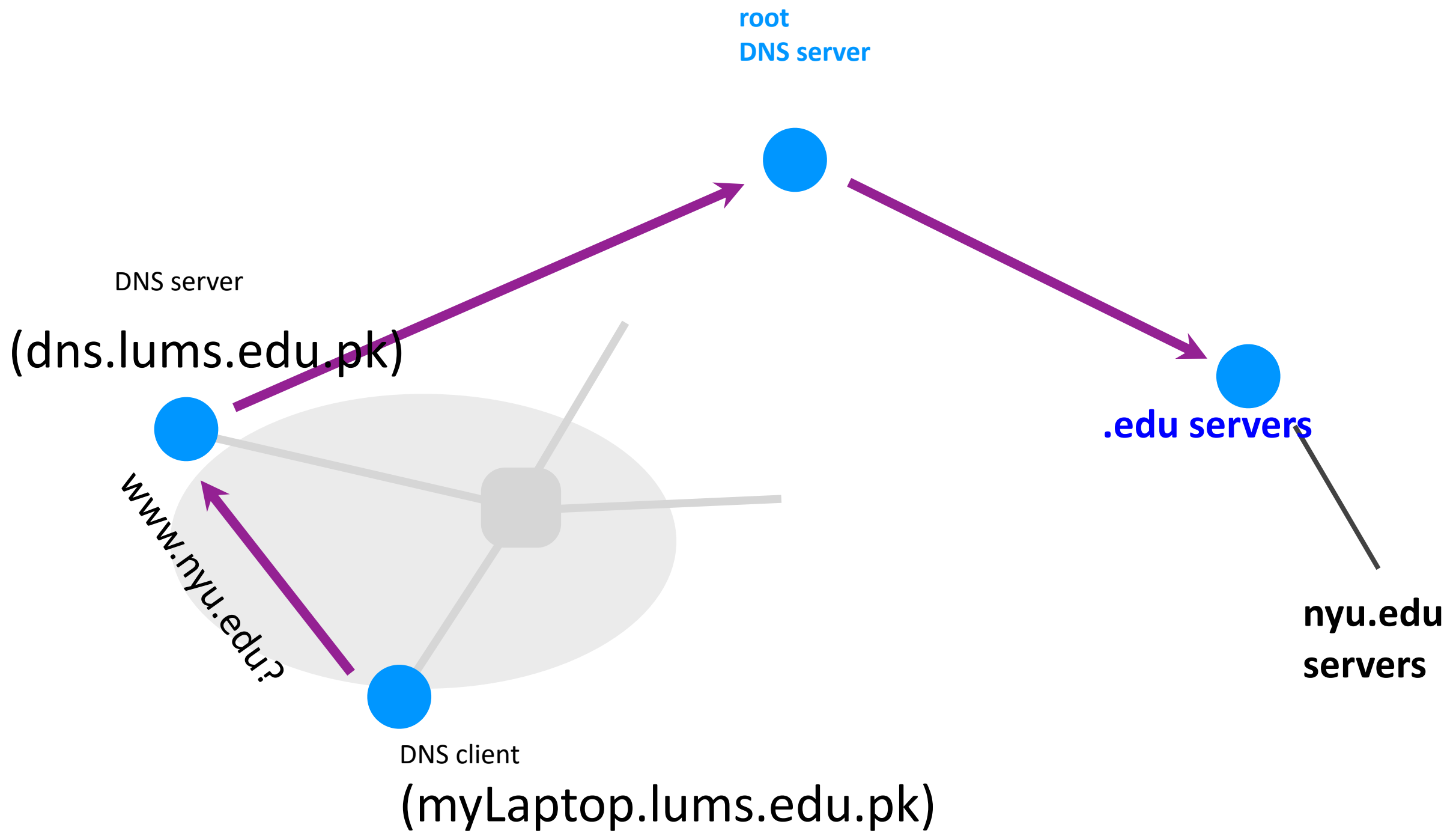
Using DNS (Client/App View)

- Two components
 - Local DNS servers
 - Resolver software on hosts
- Local DNS server (“default name server”)
 - Clients configured with the default server’s address or learn it via a host configuration protocol (e.g., DHCP)
- Client application (e.g., browser on your machine)
 - Obtain DNS name (e.g., from URL)
 - Triggers DNS request to its local DNS server

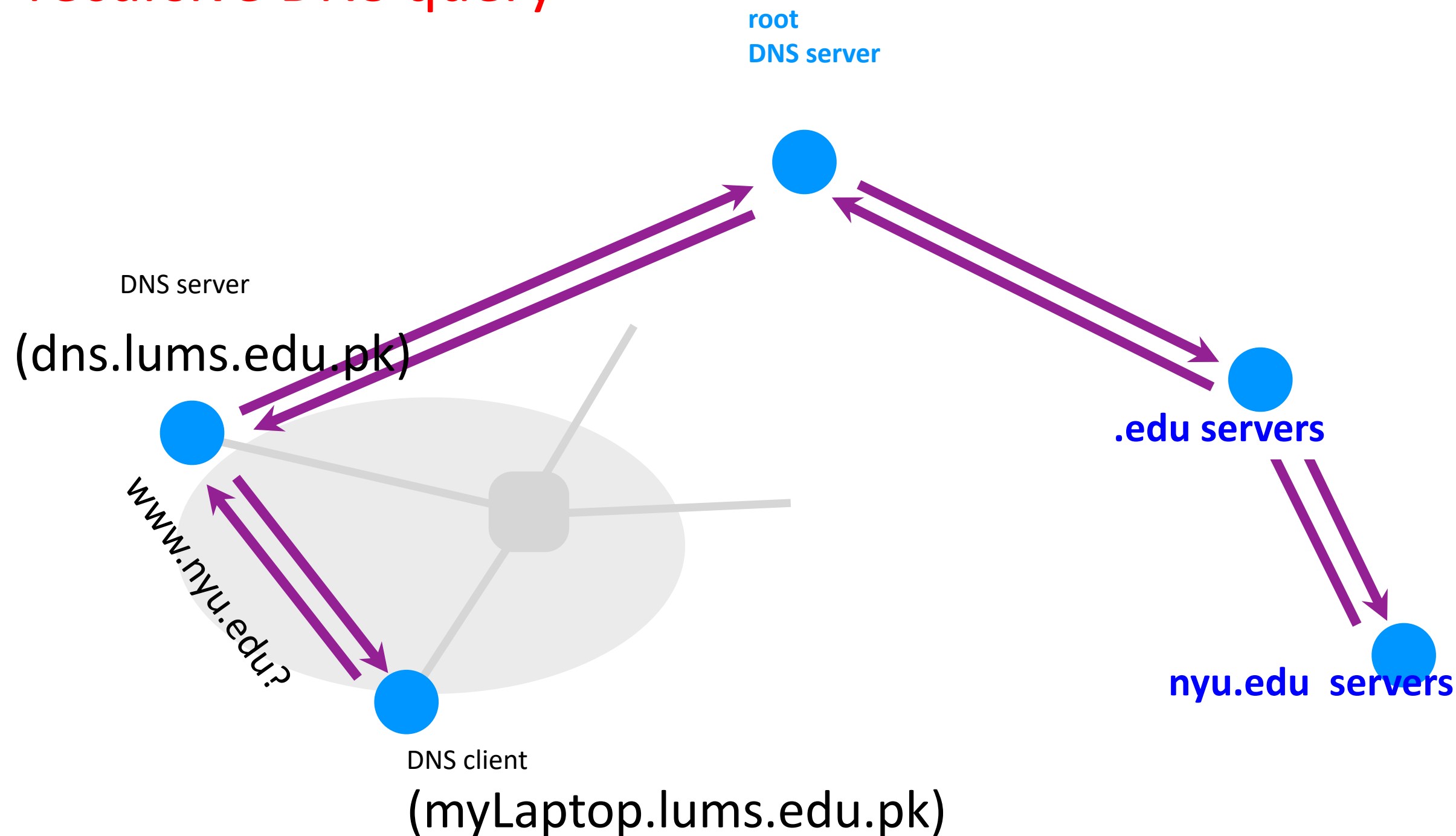


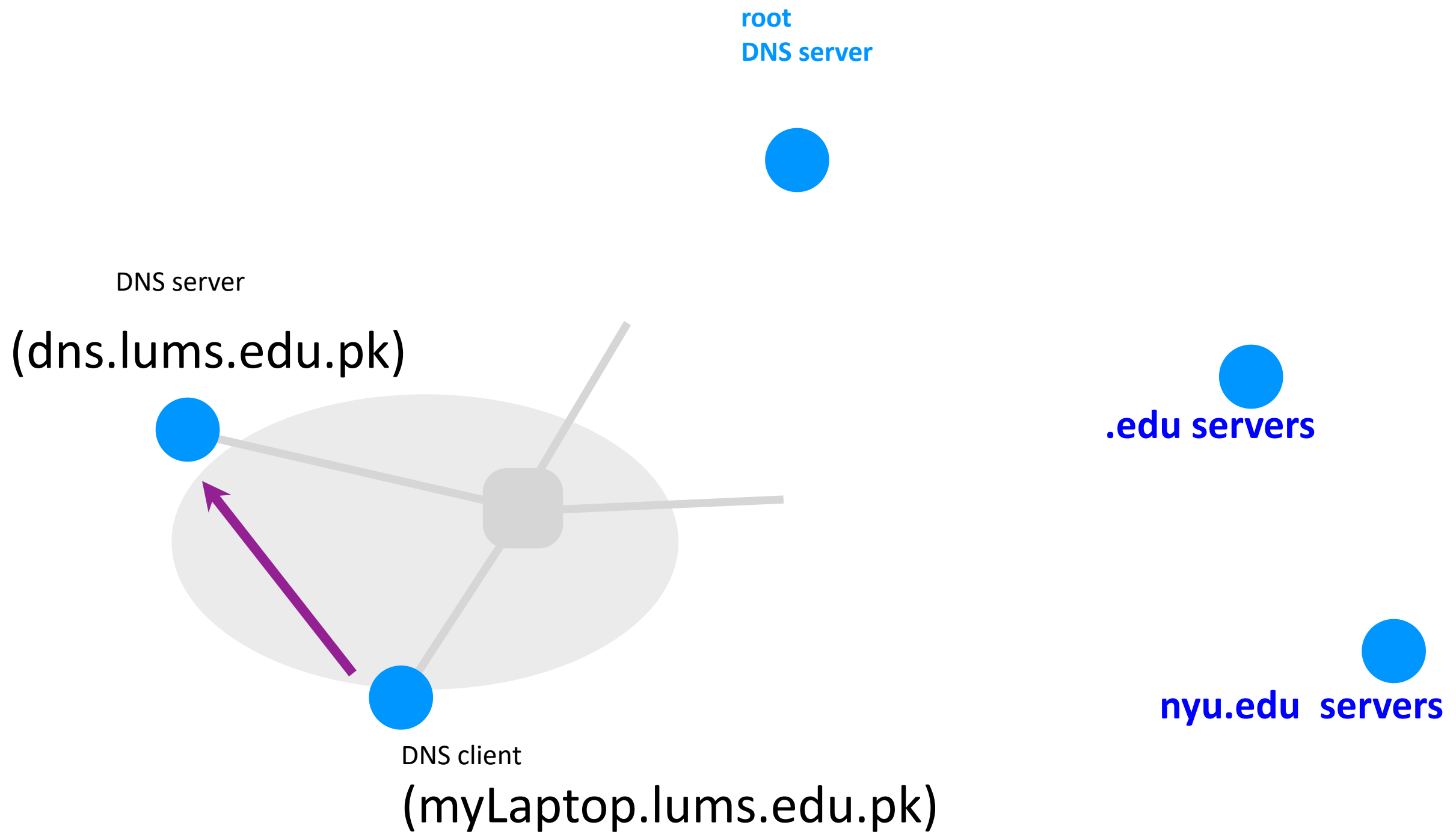




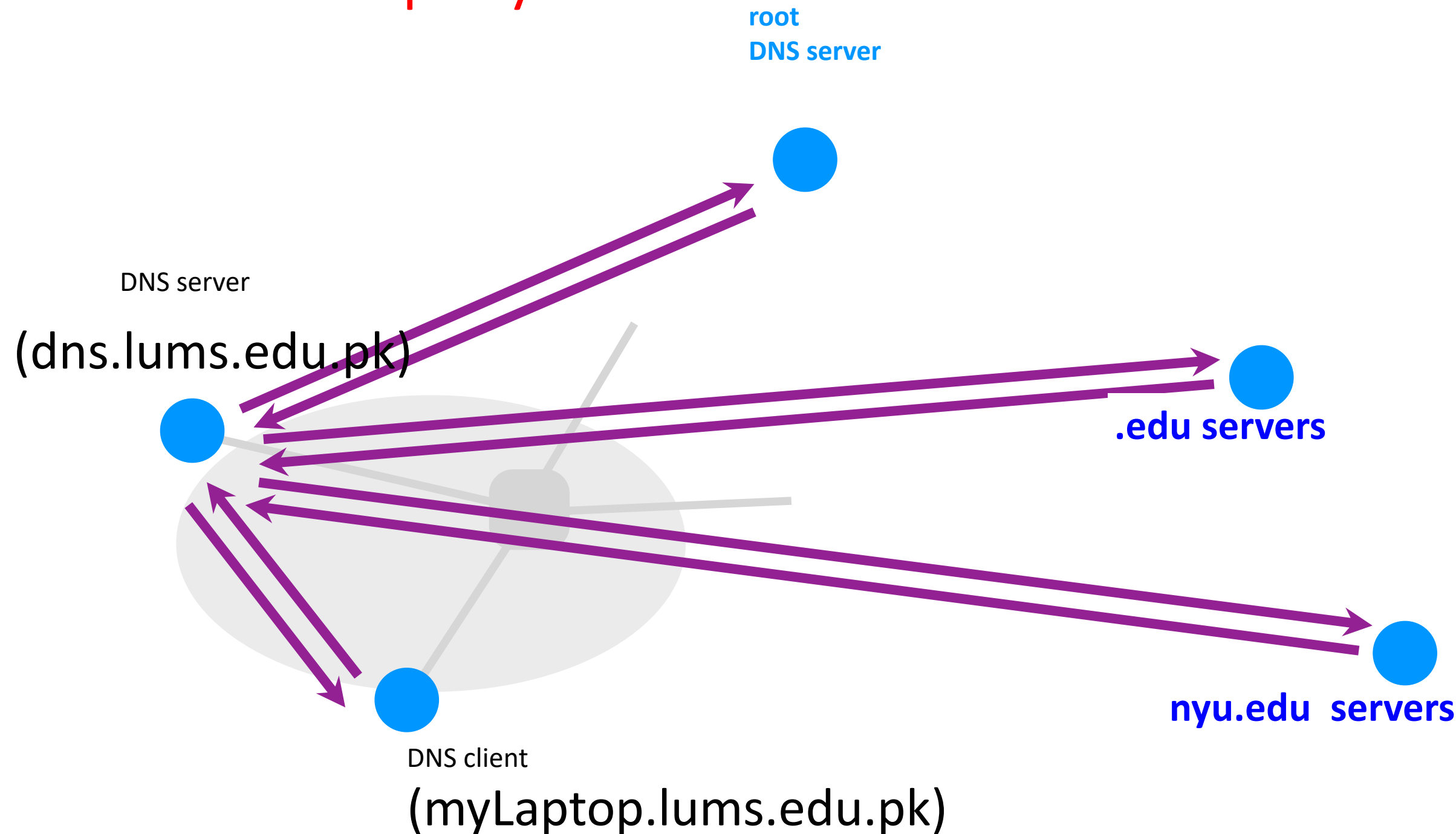


recursive DNS query





iterative DNS query



DNS Protocol

- Query and Reply messages; both with the same message format
 - *see text/RFC for details*
- Client-Server interaction on UDP Port 53
 - Spec supports TCP too, but not always implemented

Goals – how are we doing?

- Scalable
 - many names
 - many updates
 - many users creating names
 - many users looking up names
- Highly available

Per-domain availability

- DNS servers are replicated
 - Primary and secondary name servers required
 - Name service available if at least one replica is up
 - Queries can be load-balanced between replicas
- Try alternate servers on timeout
 - Exponential backoff when retrying same server

Goals – how are we doing?

- Scalable
 - many names
 - many updates
 - many users creating names
 - many users looking up names
- Highly available
- Correct
 - no naming conflicts (uniqueness)
 - consistency
- Lookups are fast

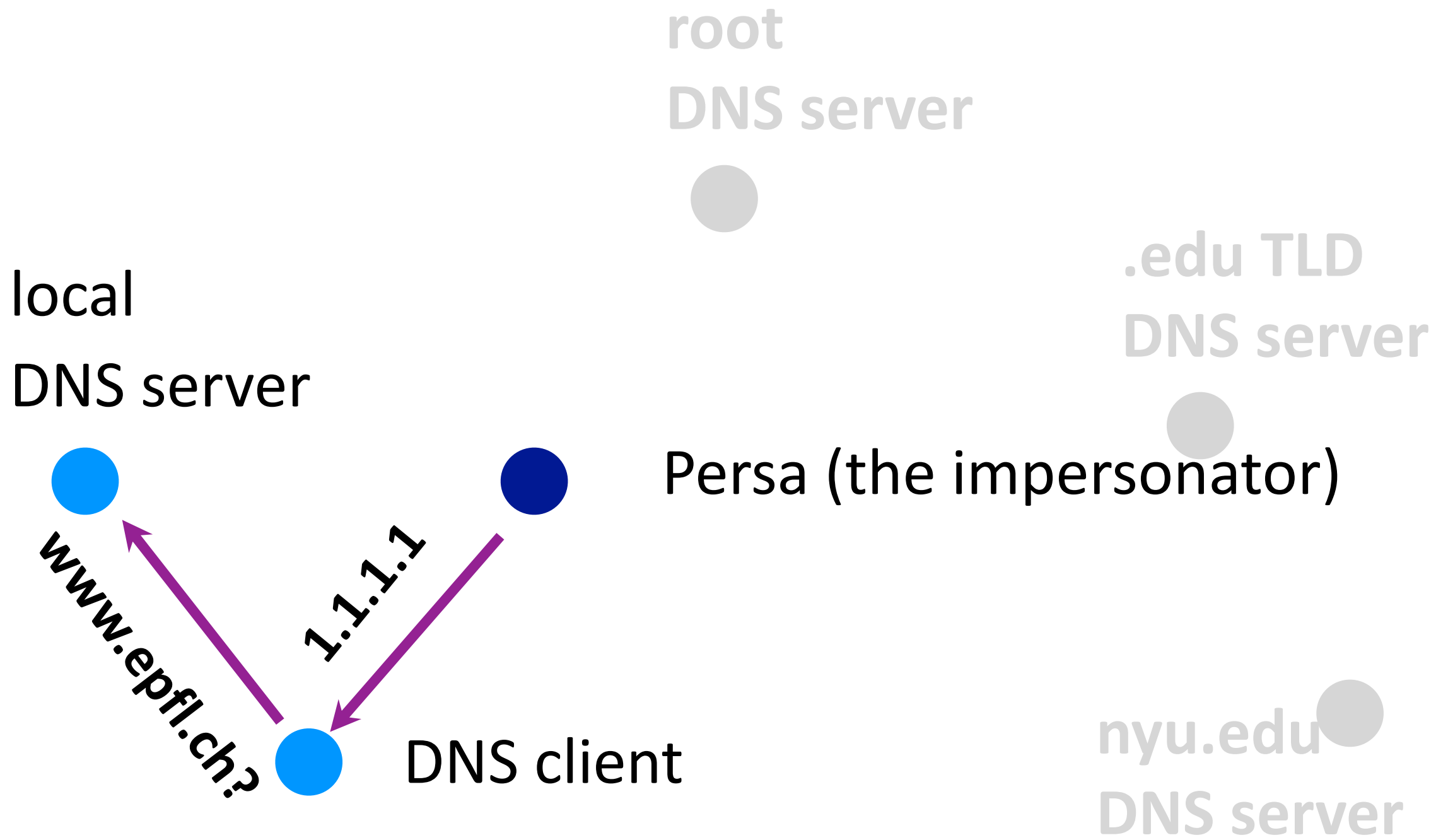
Caching

- Caching of DNS responses at all levels
- Reduces load at all levels
- Reduces delay experienced by DNS client

DNS Caching

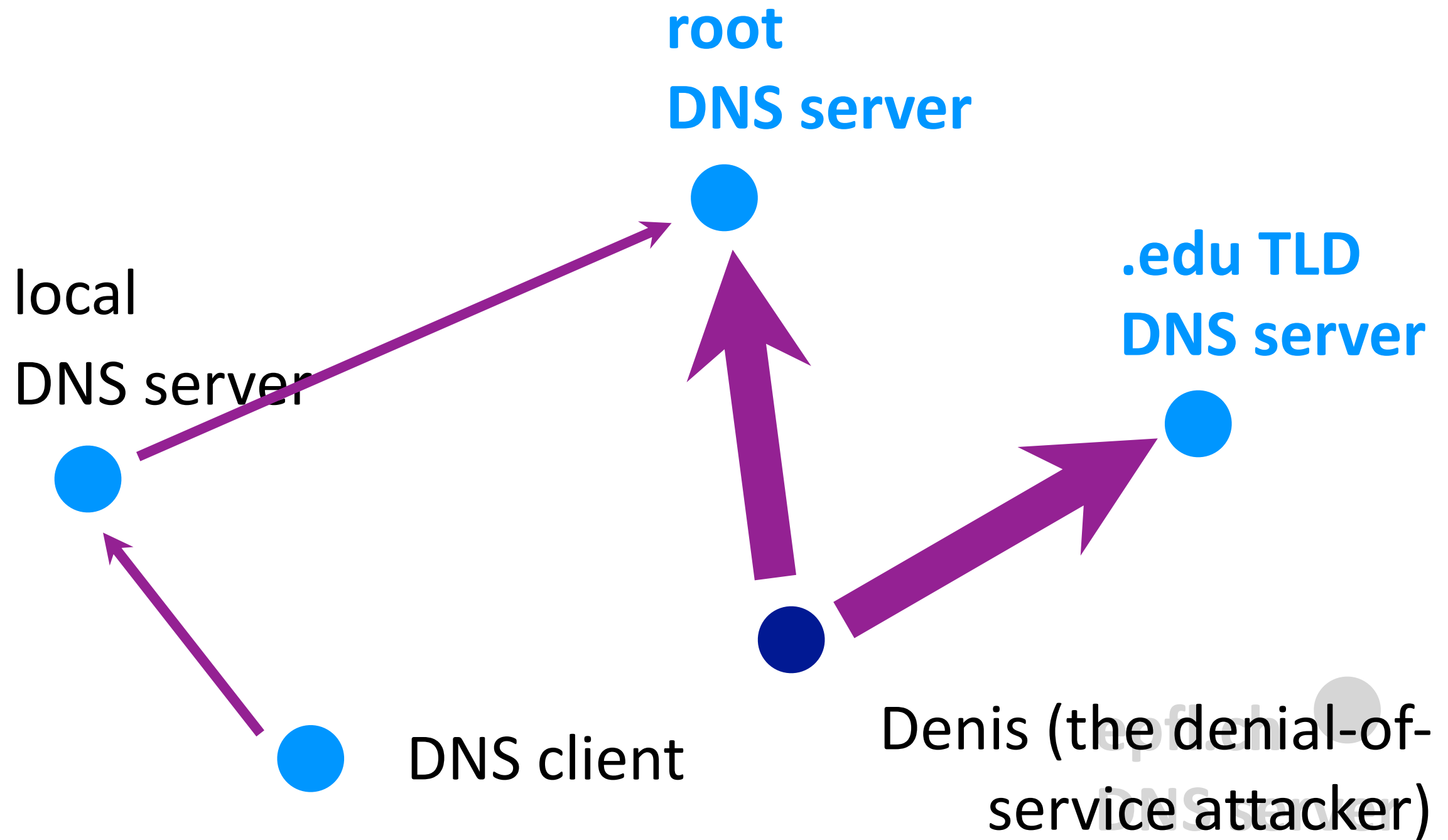
- How DNS caching works
 - DNS servers cache responses to queries
 - Responses include a “time to live” (TTL) field
 - Server deletes cached entry after TTL expires
- Why caching is effective
 - The top-level servers very rarely change
 - Popular sites visited often → local DNS server often has the information cached

How can one attack DNS?



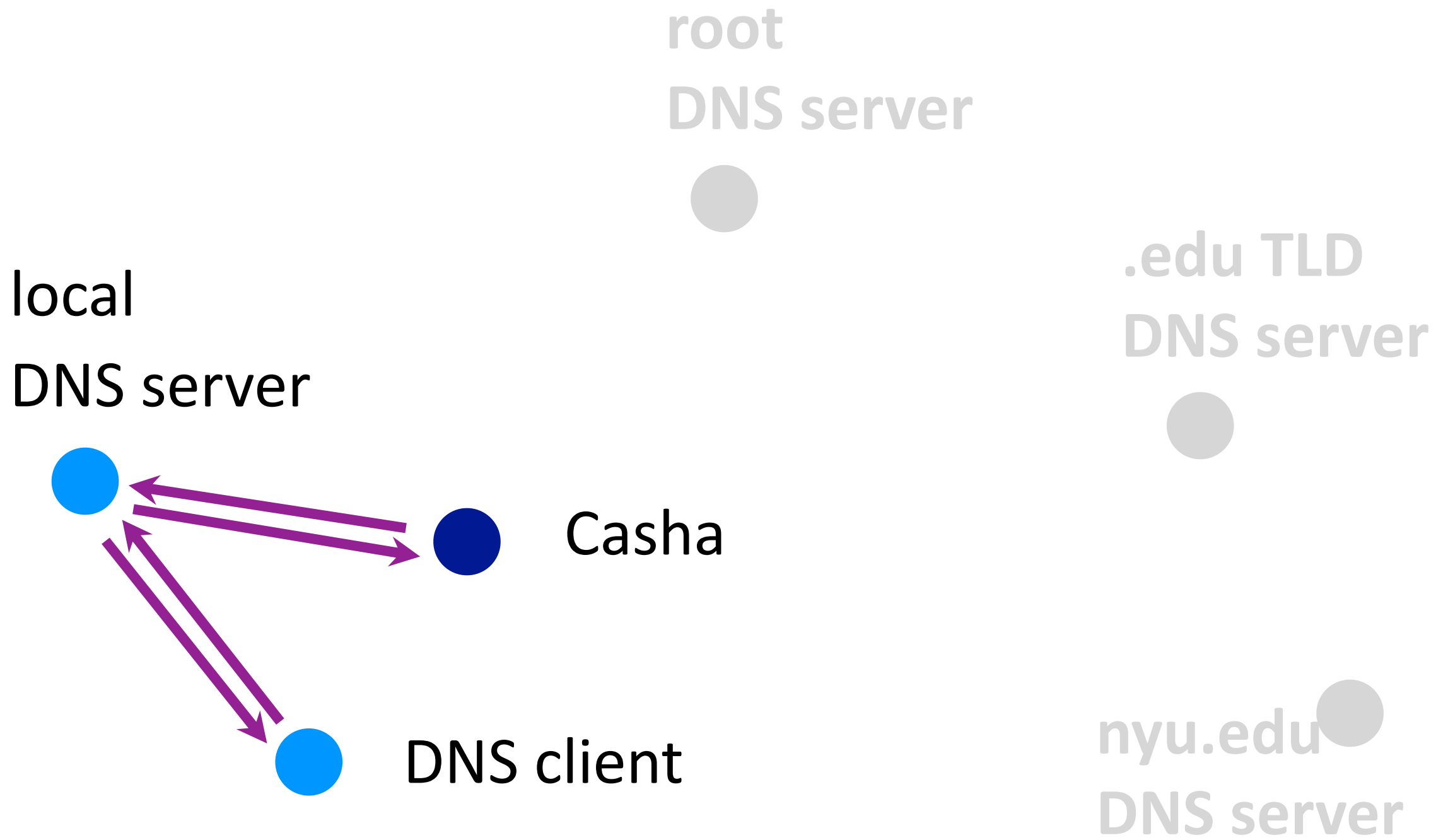
How can one attack DNS?

- Impersonate the local DNS server
 - *give the wrong IP address to the DNS client*



How can one attack DNS?

- Impersonate the local DNS server
 - *give the wrong IP address to the DNS client*
- Denial-of-service the root or TLD servers
 - *make them unavailable to the rest of the world*



How can one attack DNS?

- Impersonate the local DNS server
 - *give the wrong IP address to the DNS client*
- Denial-of-service the root or TLD servers
 - *make them unavailable to the rest of the world*
- Poison the cache of a DNS server
 - *increase the delay experienced by DNS clients*

DNS provides Indirection

- Addresses can **change** underneath
 - Move `www.cnn.com` to a new IP address
 - Humans/apps are unaffected
- Name could map to **multiple** IP addresses
 - Enables load-balancing
- **Multiple names** for the same address
 - E.g., many services (mail, www, ftp) on same machine
- Allowing “host” names to evolve into “service” names

FTP and SMTP

Thank you!