DNS and the Web

EE 122, Fall 2013
Sylvia Ratnasamy

http://inst.eecs.berkeley.edu/~ee122/

Material thanks to Ion Stoica, Scott Shenker, Jennifer Rexford, Nick McKeown, and many other colleagues
Announcements

- Project 3 out on Wednesday, Oct 30
  - Will do a Q&A in class on Monday, Nov 4

- Midterm
  - Pick up graded exams from your TA in section or office hours
  - We’ll do a quick retrospective in class next week
Taking Stock

- Course so far
  - Overall architecture (layers, protocols, inter-networking)
  - Network layer (how we interconnect hosts and networks)
  - Transport layer (E2E commn. that is better than best-effort)

- What’s left?
  - Application layer ← today
  - Lower layers (Link coding, Ethernet, Wireless) ← next
  - Special topics (datacenters, security, SDN) ← last ~2 weeks
Some very special guest lectures

- Oct 30: Anant Sahai ("EE121-in-one-lecture")
- Nov 6: Yahel Ben-David (Wireless)
- Nov 13 and 18: Vern Paxson (Security)
- Dec 2: Scott Shenker (SDN)

(Some dates still tentative.)
Today: Application Layer

- Domain Name System (DNS)
  - What’s behind (e.g.) xyz.cs.berkeley.edu

- HTTP and the Web
  - What happens when you click on a link?
DNS
Internet Names & Addresses

- Machine addresses: *e.g., 169.229.131.109*
  - router-usable labels for machines
  - conforms to network structure (*the “where”*)

- Machine names: *e.g., instr.eecs.berkeley.edu*
  - human-usable labels for machines
  - conforms to organizational structure (*the “who”*)

- The Domain Name System (DNS) is how we map from one to the other
DNS: History

- Initially all host-address mappings were in a hosts.txt file (in /etc/hosts):
  - Maintained by the Stanford Research Institute (SRI)
  - Changes were submitted to SRI by email
  - New versions of hosts.txt periodically FTP’d from SRI
  - An administrator could pick names at their discretion

- As the Internet grew this system broke down:
  - SRI couldn’t handle the load; names were not unique; 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

- No naming conflicts (uniqueness)
- Scalable
  - many names
  - (secondary) frequent updates
- Distributed, autonomous administration
  - Ability to update my own (machines’) names
  - Don’t have to track everybody’s updates
- Highly available
- Lookups are fast
- Non-goal?: perfect consistency
Key idea: Hierarchy

Three intertwined hierarchies

- Hierarchical namespace
  - As opposed to original flat namespace
- Hierarchically administered
  - As opposed to centralized
- (Distributed) 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, eecs.berkeley.edu
- Name is leaf-to-root path
  - instr.eecs.berkeley.edu
- Depth of tree is arbitrary (limit 128)
- Name collisions trivially avoided
  - each domain is responsible
Hierarchical Administration

- A **zone** corresponds to an administrative authority that is responsible for that portion of the hierarchy.
- E.g., UCB controls names: *.berkeley.edu and *.sims.berkeley.edu
- E.g., EECS controls names: *.eecs.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

- Each server stores a (small!) subset of the total DNS database
- An authoritative DNS server stores “resource records” for all DNS names in the domain that it has authority for
- Each server needs to know other servers that are responsible for the other portions of the hierarchy
  - Every server knows the root
  - Root server knows about all top-level domains
DNS Root

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

Verisign, Dulles, VA
DNS Root Servers


![Map of DNS Root Servers](image)
DNS Root Servers

- Replicated via any-casting

A Verisign, Dulles, VA
C Cogent, Herndon, VA (also Los Angeles, NY, Chicago)
D U Maryland College Park, MD
G US DoD Vienna, VA
H ARL Aberdeen, MD
J Verisign (21 locations)

A Verisign, Dulles, VA
C Cogent, Herndon, VA (also Los Angeles, NY, Chicago)
D U Maryland College Park, MD
G US DoD Vienna, VA
H ARL Aberdeen, MD
J Verisign (21 locations)

E NASA Mt View, CA
F Internet Software Consortium, Palo Alto, CA (and 37 other locations)
B USC-ISI Marina del Rey, CA
L ICANN Los Angeles, CA

K RIPE London (plus 16 other locations)
I Autonomica, Stockholm (plus 29 other locations)
M WIDE Tokyo plus Seoul, Paris, San Francisco

Replicated via any-casting
Anycast in a nutshell

- Routing finds shortest paths to destination
- If several locations are given the same address, then the network will deliver the packet to the closest location with that address
- This is called “anycast”
  - Very robust
  - Requires no modification to routing algorithms
Using DNS (Client/App View)

- Two components
  - Local DNS servers
  - Resolver software on hosts

- Local DNS server ("default name server")
  - Usually near the endhosts that use it
  - Hosts configured with local server address (e.g., /etc/resolv.conf) or learn server via a host configuration protocol (e.g., DHCP)

- Client application
  - Obtain DNS name (e.g., from URL)
  - Do `gethostbyname()` to trigger DNS request to its local DNS server
How Does Resolution Happen?

Host at \texttt{cis.poly.edu} wants IP address for \texttt{gaia.cs.umass.edu}

1. \texttt{requesting host} \texttt{cis.poly.edu}
2. \texttt{local DNS server} \texttt{dns.poly.edu}
3. \texttt{root DNS server}
4. \texttt{TLD DNS server}
5. \texttt{authoritative DNS server} \texttt{dns.cs.umass.edu}
6. \texttt{gaia.cs.umass.edu}
Recursive vs. Iterative Queries

- **Recursive** query
  - Ask server to get answer for you
  - E.g., request 1 and response 8

- **Iterative** query
  - Ask server who to ask next
  - E.g., all other request-response pairs
DNS Records

- DNS info. stored as **resource records (RRs)**
  - RR is (name, value, type, TTL)

- **Type = A:** (→ **Address**)
  - name = hostname
  - value = IP address

- **Type = NS:** (→ **Name Server**)
  - name = domain
  - value = name of dns server for domain
DNS Records (cont’d)

- Type = CNAME: (→ Canonical NAME)
  - name = hostname
  - value = canonical name

- Type = MX: (→ Mail eXchanger)
  - name = domain in email address
  - value = canonical name(s) of mail server(s)
Inserting Resource Records into DNS

- Example: just created company “FooBar”
- Get a block of address space from ISP
  - Say 212.44.9.128/25

- Register foobar.com at registrar (e.g., Network Solutions)
  - Provide registrar with names and IP addresses of your authoritative name server (primary and secondary)
  - Registrar inserts RR pairs into the .com TLD server:
    - (foobar.com, dns1.foobar.com, NS)
    - (dns1.foobar.com, 212.44.9.129, A)

- You store appropriate records in your server dns1.foobar.com:
  - e.g., type A record for www.foobar.com
  - e.g., type MX record for foobar.com
DNS Protocol

- Query and Reply messages; both with the same message format
  - header: identifier, flags, etc.
  - plus resource records
  - see text/section for details

- Client--server interaction on UDP Port 53
  - Spec supports TCP too, but not always implemented
Goals -- how are we doing?

- No naming conflicts (uniqueness)
- Scalable
- Distributed, autonomous administration
- Highly available
- Fast lookups
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.cnn.com) 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 www.cnn.comm and www.cnnn.com
  - 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
- Try alternate servers on timeout
  - **Exponential backoff** when retrying same server
- Same identifier for all queries
  - Don’t care which server responds
Important Properties of DNS

Administrative delegation and hierarchy results in:

- Easy unique naming
- “Fate sharing” for network failures
- Reasonable trust model
- Caching lends scalability, performance
DNS provides Indirection

- Addresses can change underneath
  - Move www.cnn.com to 4.125.91.21
  - Humans/Apps should be unaffected

- Name could map to multiple IP addresses
  - Enables
    - Load-balancing
    - Reducing latency by picking nearby servers

- Multiple names for the same address
  - E.g., many services (mail, www, ftp) on same machine
  - E.g., aliases like www.cnn.com and cnn.com

- But, this flexibility applies only within domain!
The Web
The Web – Precursor

- 1967, Ted Nelson, Xanadu:
  - A world-wide publishing network that would allow information to be stored not as separate files but as connected literature
  - Owners of documents would be automatically paid via electronic means for the virtual copying of their documents
  - Coined the term “Hypertext”
The Web – History

- World Wide Web (WWW): a distributed database of “pages” linked through **Hypertext Transport Protocol (HTTP)**
  - First HTTP implementation - 1990
    - Tim Berners-Lee at CERN
  - HTTP/0.9 – 1991
    - Simple GET command for the Web
  - HTTP/1.0 – 1992
    - Client/Server information, simple caching
  - HTTP/1.1 - 1996

Tim Berners-Lee
On inventing a “killer app”

HTML is precisely what we were trying to PREVENT—ever-breaking links, links going outward only, quotes you can't follow to their origins, no version management, no rights management.

– Ted Nelson
Web Components

- Infrastructure:
  - Clients
  - Servers

- Content:
  - URL: naming content
  - HTML: formatting content

- Protocol for exchanging information: HTTP
Uniform Record Locator (URL)

`protocol://host-name[:port]/directory-path/resource`

- Extend the idea of hierarchical hostnames to include anything in a file system
  - [http://www.cs.berkeley.edu/~sylvia/cs268/lecture1.ppt](http://www.cs.berkeley.edu/~sylvia/cs268/lecture1.ppt)

- Extend to program executions as well…
  - [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](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)
  - Server side processing can be incorporated in the name
Uniform Record Locator (URL)

\[ \text{protocol://host-name[:port]/directory-path/resource} \]

- **protocol**: http, ftp, https, smtp, rtsp, etc.
- **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
Web and DNS

- URLs use hostnames
- Thus, content names are tied to specific hosts
- Makes persistence of content difficult!
Hyper Text Transfer Protocol (HTTP)

- Client-server architecture
  - server is “always on” and “well known”
  - clients initiate contact to server

- Synchronous request/reply protocol
  - Runs over TCP, Port 80

- Stateless

- ASCII format
Steps in HTTP Request/Response

Client
- TCP Syn
- TCP syn + ack
- TCP ack + HTTP GET
- Request response
- Close connection

Server
- TCP Syn
- TCP syn + ack
- Request response
- Close connection
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)

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr
(blank line)
```
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**
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

**data**
e.g., requested HTML file
(data data data data data data ...)
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
Performance Issues
Performance Goals

- **User**
  - fast downloads (not identical to low-latency communication)
  - high availability

- **Content provider**
  - happy users (hence, above)
  - cost-effective infrastructure

- **Network (secondary)**
  - avoid overload
Solutions?

- **User**
  - fast downloads (not identical to low-latency communication!)
  - high availability

- **Content provider**
  - happy users (hence, above)
  - cost-effective infrastructure

- **Network (secondary)**
  - avoid overload
Solutions?

- **User**
  - fast downloads (not identical to low-latency commun.)
  - high availability

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

- **Network (secondary)**
  - avoid overload

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

**Caching and Replication**
Solutions?

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

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

- Network (secondary)
  - avoid overload

**Solutions (continued)**

- **User**
  - fast downloads (not identical to low-latency communication)
  - high availability

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

- **Network (secondary)**
  - avoid overload

---

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

**Caching and Replication**

**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!
Improving HTTP Performance: Concurrent Requests & Responses

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

- Client = 🎉
- Content provider = 😊
- Network = 😞 Why?
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
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: $\sim 2$ RTT
- Pipelined/Persistent: $\sim 2$ 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
  * and each TCP connection gets the same bandwidth
- Pipelined and/or persistent: $\sim nF/B$
  * The only thing that helps is getting more bandwidth..
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
  - But many unique requests
Improving HTTP Performance:
Caching: How

● Modifier to GET requests:
  ● If-modified-since – returns “not modified” if resource not modified since specified time

GET /~ee122/fa13/ HTTP/1.1
Host: inst.eecs.berkeley.edu
User-Agent: Mozilla/4.03
If-modified-since: Sun, 27 Oct 2013 22:25:50 GMT

● 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 transfer 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

- Problem: Want to direct client to particular replica
  - Balance load across server replicas
  - Pair clients with nearby servers

- 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