CS162
Operating Systems and Systems Programming
Lecture 23

HTTP and Peer-to-Peer Networks

April 20, 2011

Ion Stoica

http://inst.eecs.berkeley.edu/~cs162

Recap: RPC Server Crashes

- Three cases
  - Crash after execution
  - Crash before execution
  - Crash during the execution

- Three possible semantics
  - At least once semantics
    - » Client keeps trying until it gets a reply
  - At most once semantics
    - » Client gives up on failure
  - Exactly once semantics
    - » Can this be correctly implemented?

Why Not Use Logging?

- Assume
  - Server can log either before starting or after executing the operation
  - Server restarts after crashing

- First case:
  - Server execute operation first, then logs “done”
  - What semantics does this implement?

- Second case:
  - Server logs “start”, and then execute operation
  - What semantics does this implement?

- So, can you ensure “exactly once” semantics?

Today's Lecture

- Web
  - Hypertext Transport Protocol
- Peer-to-Peer networks
  - Distributed Hash Tables (DHTs)

The Web

- Core components:
  - Servers: store files and execute remote commands
  - Browsers: retrieve and display “pages”
  - Uniform Resource Locators (URLs): way to refer to pages
- A protocol to transfer information between clients and servers
  - HTTP

Uniform Record Locator (URL)

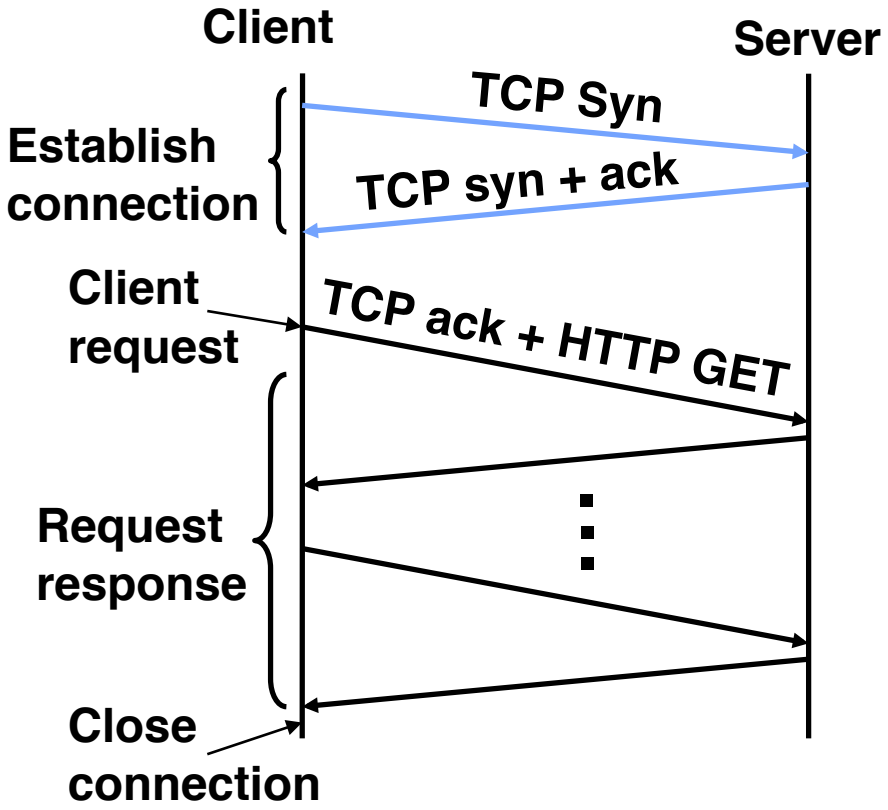
```
protocol://host-name:port/directory-path/resource
```

- E.g., http://www-inst.eecs.berkeley.edu/~cs162/sp11/
- Extend to program executions as well…
  - http://www.google.com/#sclient=psy&hl=en&source=hp&q=cs162+berkeley&aq=0&aqi=g5&aql=&oq=&pbx=1&bav=on.2,or.r_gc.r_pw.&fp=1ef120049c3f5a29

Hyper Text Transfer Protocol (HTTP)

- Client-server architecture
- Synchronous request/reply protocol
  - Runs over TCP, Port 80
- Stateless
  - Server does not keep state about client across requests, i.e., after each request the web server forgets about client
  - Why is this good?

Big Picture


Client
Server
TCP Syn
Establish connection
TCP syn + ack
Client request
TCP ack + HTTP GET
Request response
Close connection

Hyper Text Transfer Protocol Commands

- GET – transfer resource from given URL
- HEAD – GET resource metadata (headers) only
- PUT – store/modify resource under given URL
- DELETE – remove resource
- POST – provide input for a process identified by the given URL (usually used to post CGI parameters)

Client Request

- Steps to get the resource:

  http://www-inst.eecs.berkeley.edu/~cs162/sp11/

  1. Use DNS to obtain the IP address of www-inst.eecs.berkeley.edu

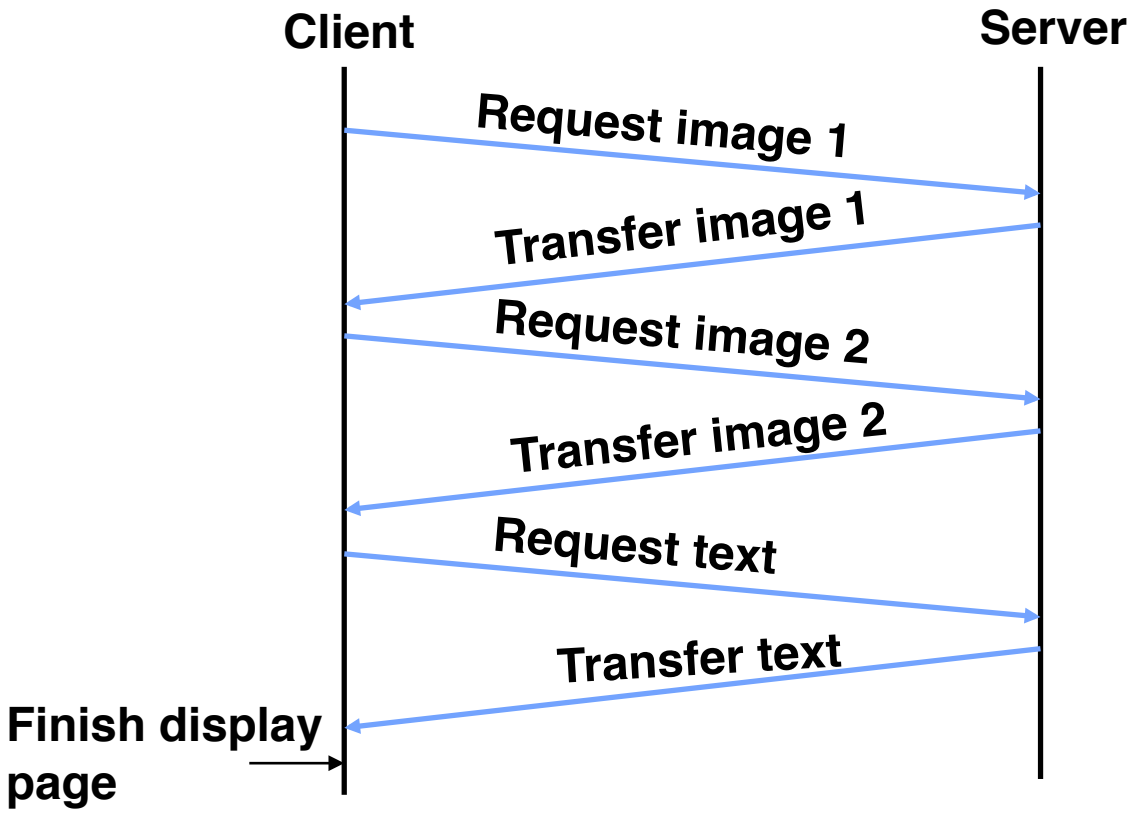
  2. Send an HTTP request to IP address and port 80:

     ```
     GET /~cs162/sp11 HTTP/1.0
     ```

Server Response

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 1234
Last-Modified: Mon, 19 Nov
2010 15:31:20 GMT
<HTML>
<HEAD>
<TITLE>EECS Home Page</TITLE>
</HEAD>
…
</BODY>
</HTML>
```

HTTP/1.0 Example



HTTP/1.0 Performance

- Create a new TCP connection for each resource
  - Large number of embedded objects in a web page
  - Many short lived connections
- TCP transfer
  - Too slow for small object
  - It takes time to establish a connection and ramp-up (i.e., exit slow-start phase)
- Connections may be set up in parallel (5 is default in most browsers)

HTTP/1.0 Caching Support

- A modifier to the GET request:
  - If-modified-since – return a “not modified” response if resource was not modified since specified time
- A response header:
  - Expires – specify to the client for how long it is safe to cache the resource
- A request directive:
  - No-cache – ignore all caches and get resource directly from server
- These features can be best taken advantage of with HTTP proxies
  - Locality of reference increases if many clients share a proxy

HTTP/1.1 (1996)

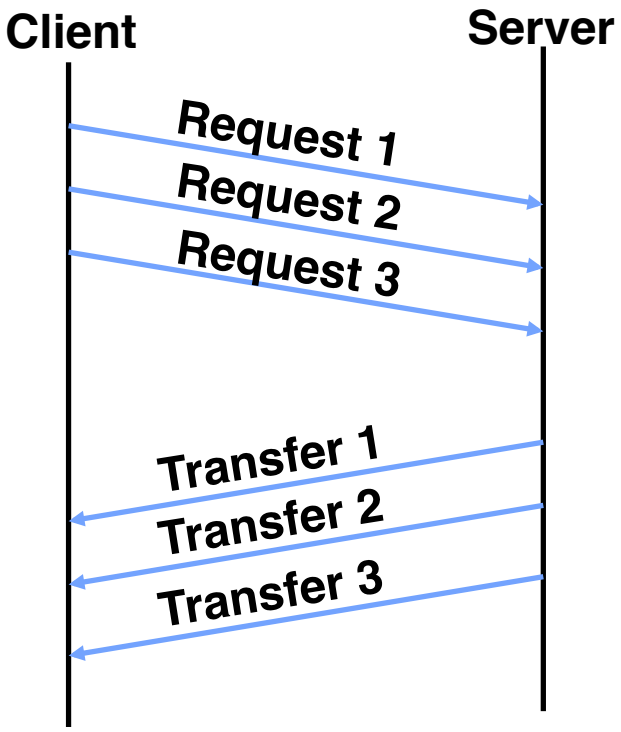
- Performance:
  - Persistent connections
  - Pipelined requests/responses
  - …
- Efficient caching support
  - Network Cache assumed more explicitly in the design
  - Gives more control to the server on how it wants data cached
- Support for virtual hosting
  - Allows to run multiple web servers on the same machine

Persistent Connections

- Allow multiple transfers over one connection
- Avoid multiple TCP connection setups
- Avoid multiple TCP slow starts (i.e., TCP ramp ups)

Pipelined Requests/Responses

- Buffer requests and responses to reduce the number of packets
- Multiple requests can be contained in one TCP segment
- Note: order of responses has to be maintained



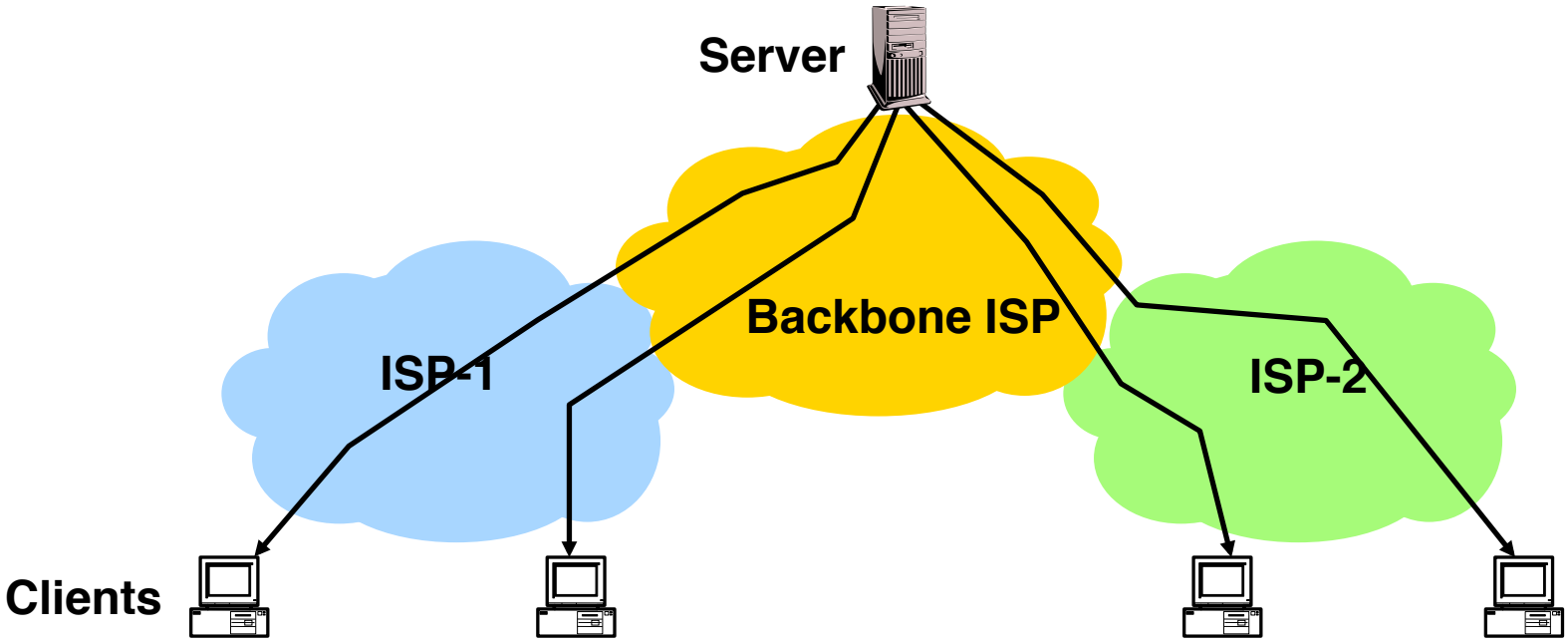
Achieving Scale and Availability

- Problem: You are a web content provider
  - How do you handle millions of web clients?
  - How do you ensure that all clients experience good performance?
  - How do you maintain availability in the presence of server and network failures?

- Solutions:
  - Add more servers at different locations → If you are CNN this might work!
  - Caching
  - Content Distribution Networks (Replication)

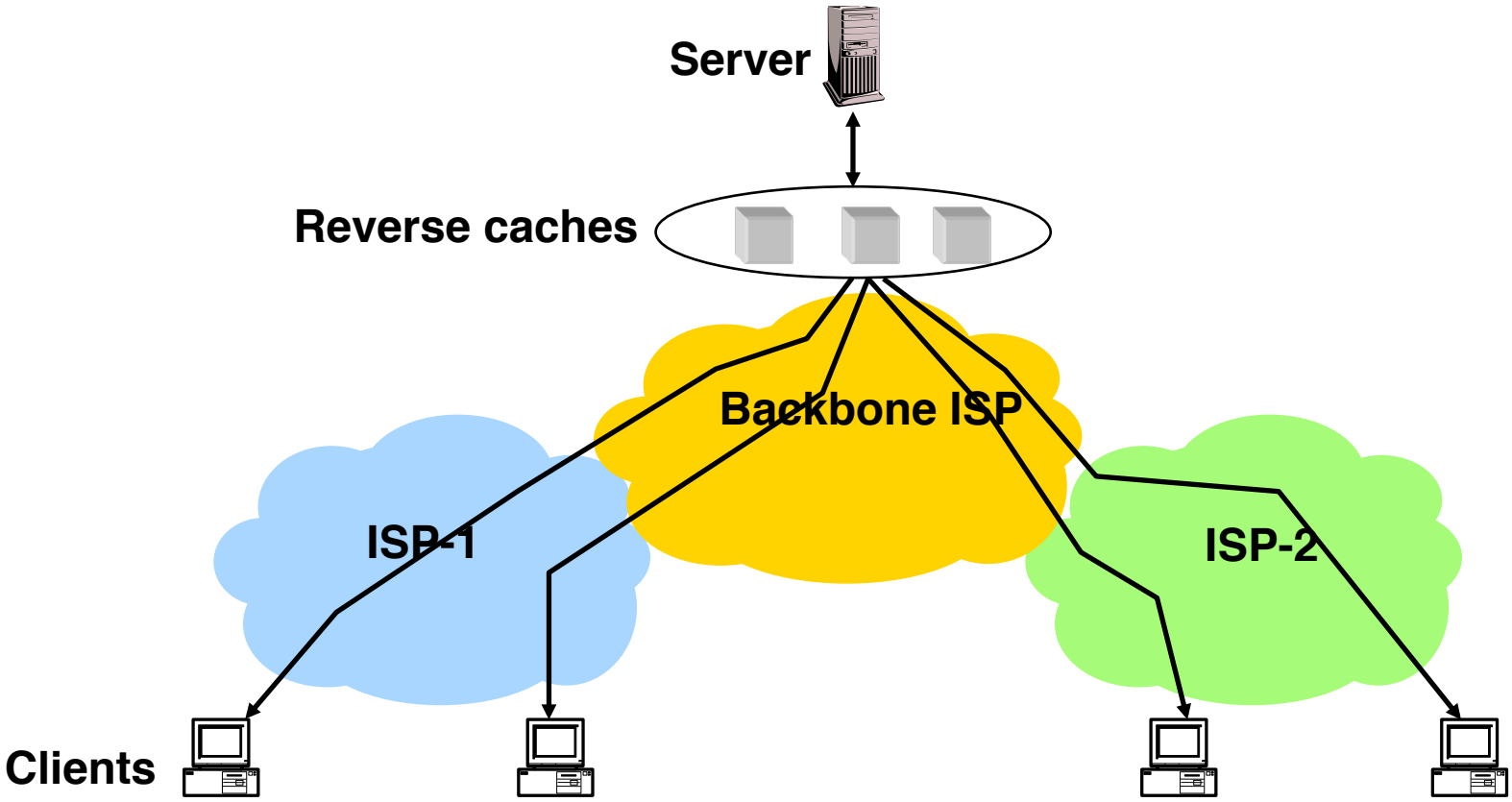
"Base-line"

- Many clients transfer same information
  - Generate unnecessary server and network load
  - Clients experience unnecessary latency



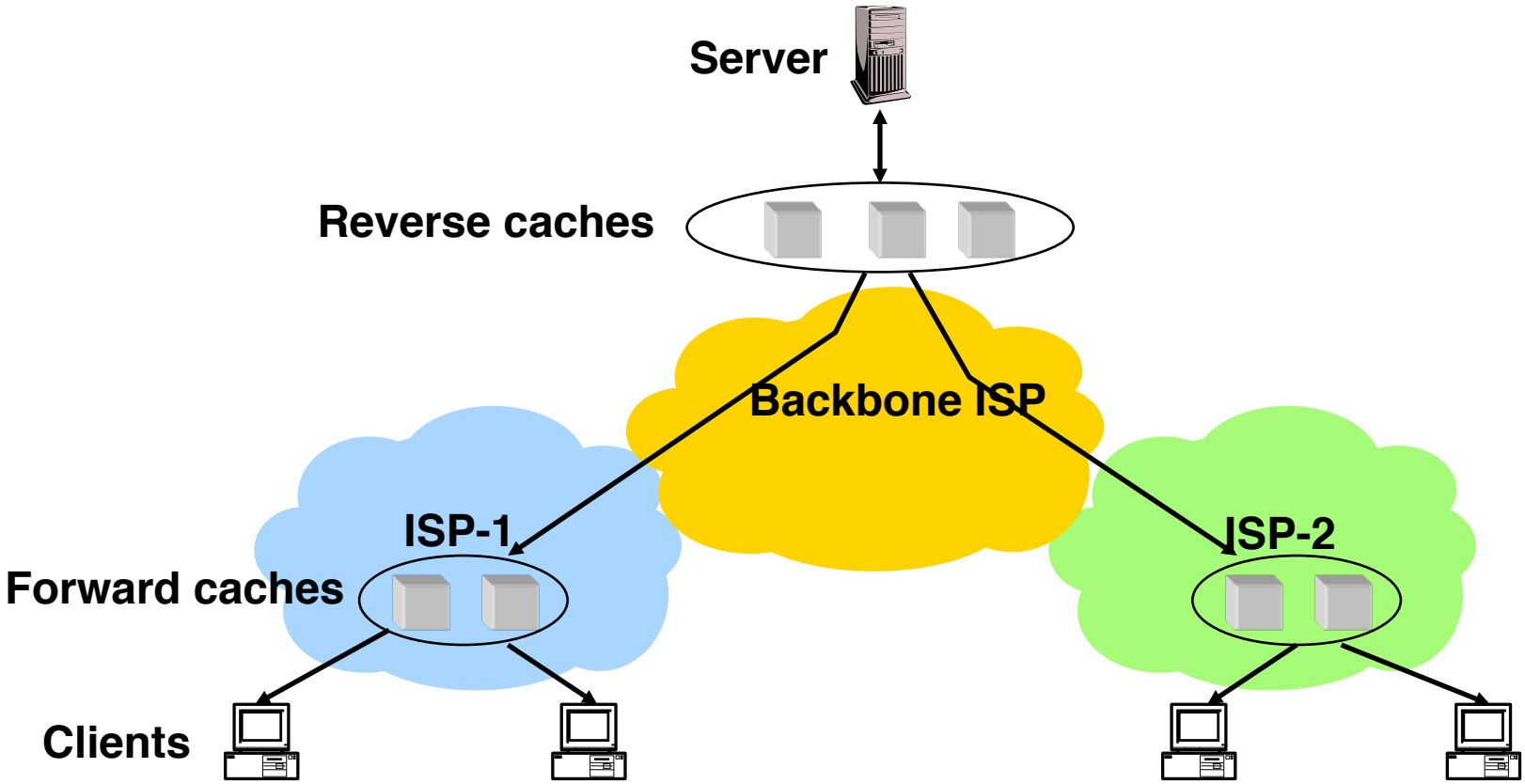
Reverse Caches

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



Forward Proxies

- Cache documents close to clients → reduce network traffic and decrease latency
- Typically done by ISPs or corporate LANs



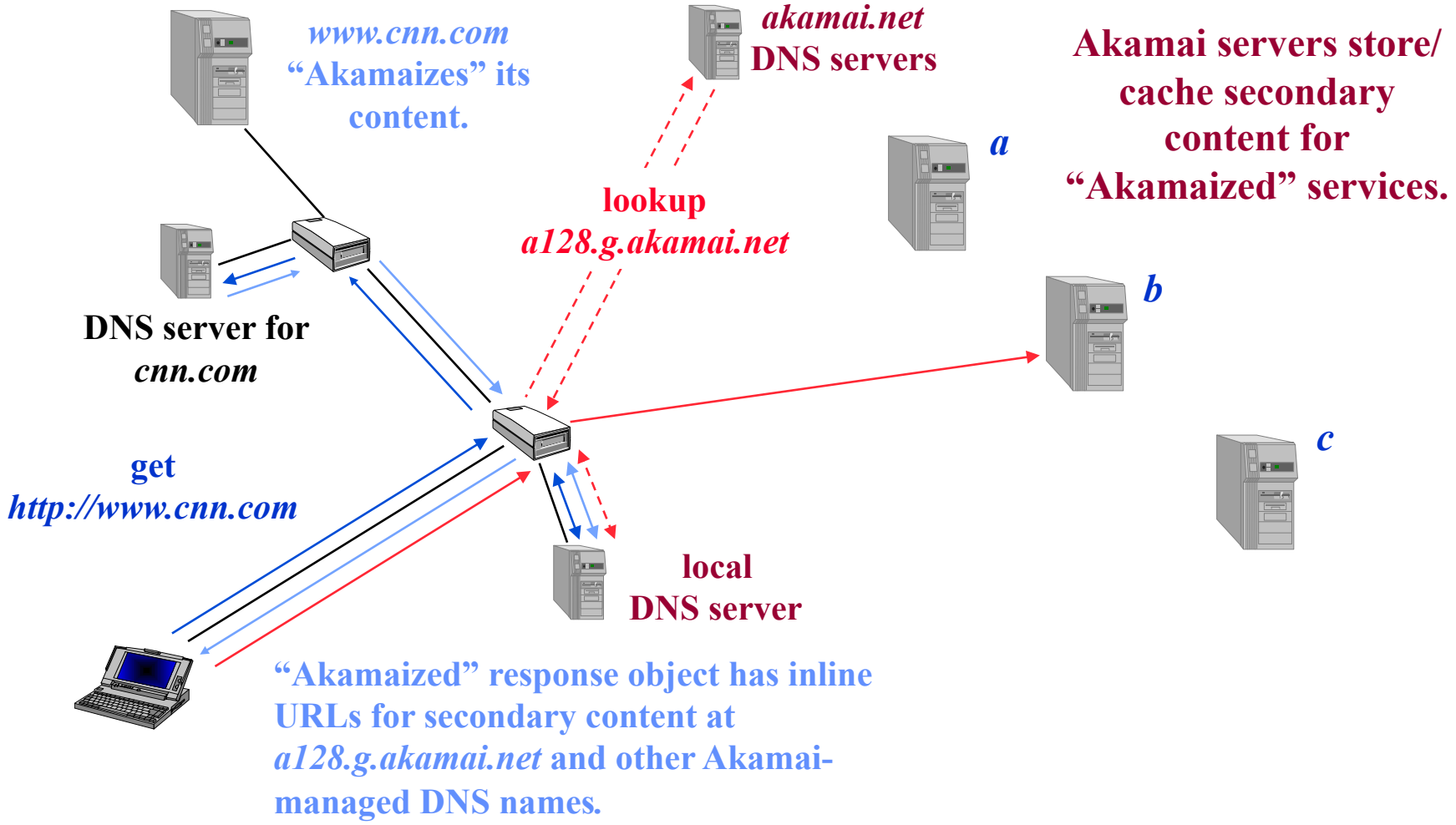
Content Distribution Networks (CDNs)

- Integrate forward and reverse caching functionalities into one overlay network (usually) administrated by one entity
  - Example: Akamai
- Documents are cached both
  - As a result of clients' requests (pull)
  - Pushed in the expectation of a high access rate
- Beside caching do processing, e.g.,
  - Handle dynamic web pages
  - Transcoding

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.cnn.com/image-of-the-day.gif* becomes *http://a128.g.akamai.net/image-of-the-day.gif*.

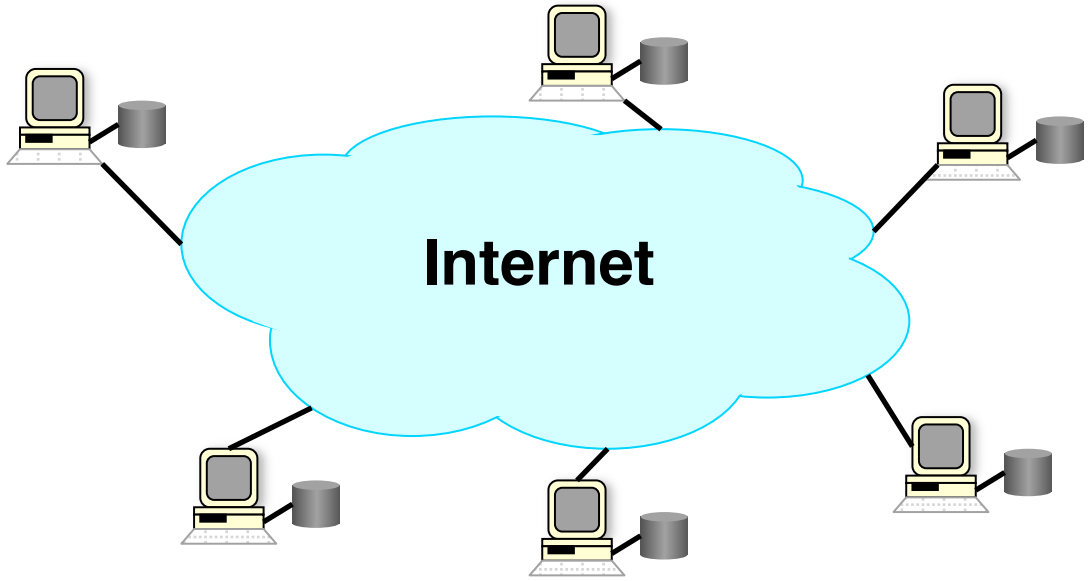
Example: Akamai



Peer-to-Peer Networks & Distributed Hash Tables

How Did it Start?

- A killer application: Naptser
  - Free music over the Internet
- Key idea: share the storage *and* bandwidth of individual (home) users

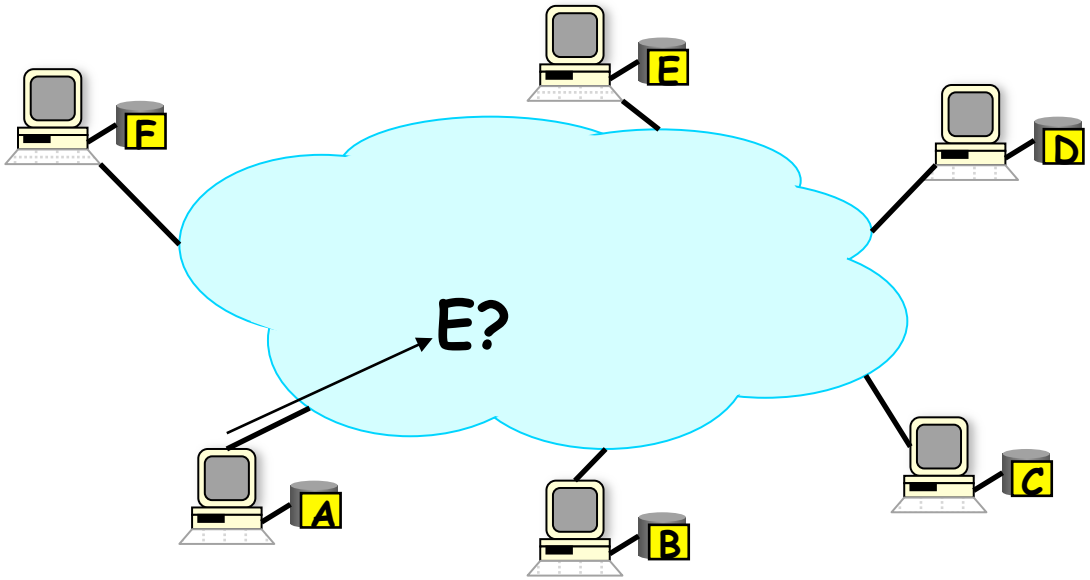


Model

- Each user stores a subset of files
- Each user has access (can download) files from all users in the system

Main Challenge

- Find where a particular file is stored
  - Note: problem similar to finding a particular page in web caching (what are the differences?)



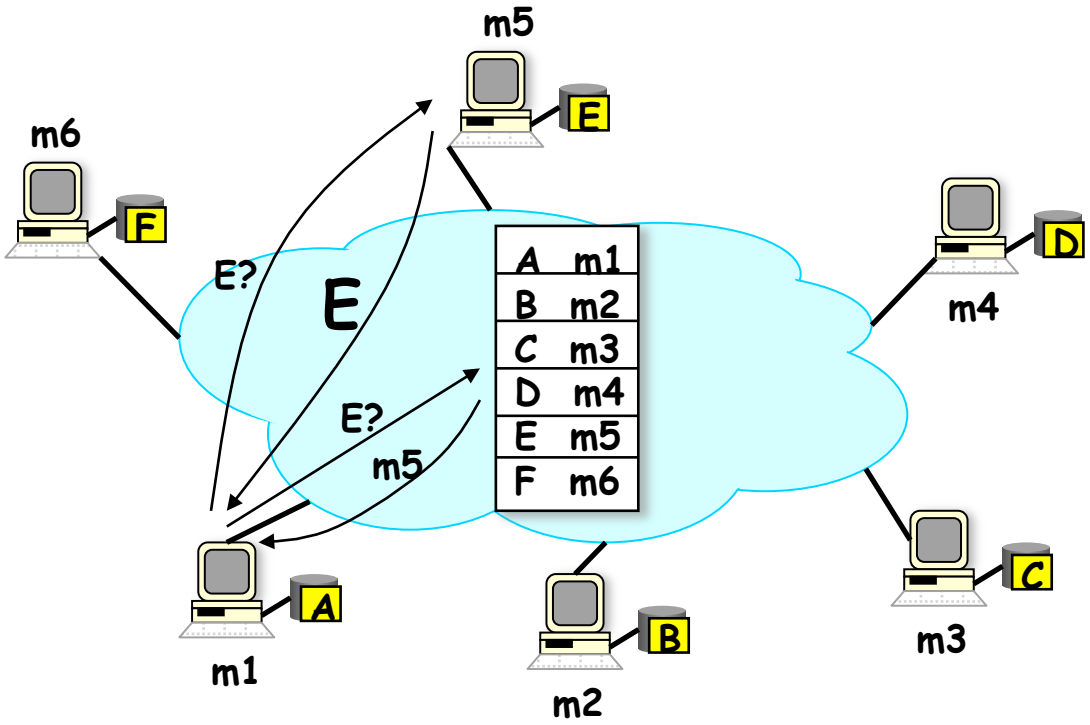
Other Challenges

- Scale: up to hundred of thousands or millions of machines
- Dynamicity: machines can come and go any time

Napster

- Assume a centralized index system that maps files (songs) to machines that are alive
- How to find a file (song)
  - Query the index system → return a machine that stores the required file
    - » Ideally this is the closest/least-loaded machine
  - ftp the file
- Advantages:
  - Simplicity, easy to implement sophisticated search engines on top of the index system
- Disadvantages:
  - Robustness, scalability (?)

Napster: Example

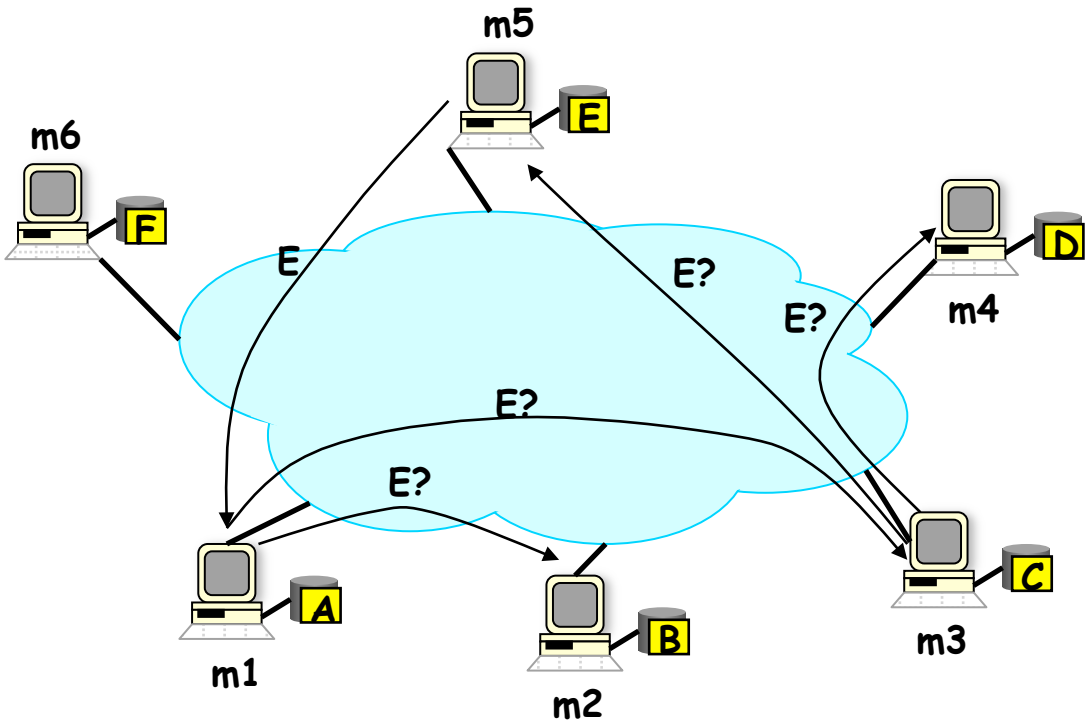


Gnutella

- Distribute file location
- Idea: broadcast the request
- How to find a file?
  - Send request to all neighbors
  - Neighbors recursively multicast the request
  - Eventually a machine that has the file receives the request, and it sends back the answer
- Advantages:
  - Totally decentralized, highly robust
- Disadvantages:
  - Not scalable; the entire network can be swamped with requests (to alleviate this problem, each request has a TTL)

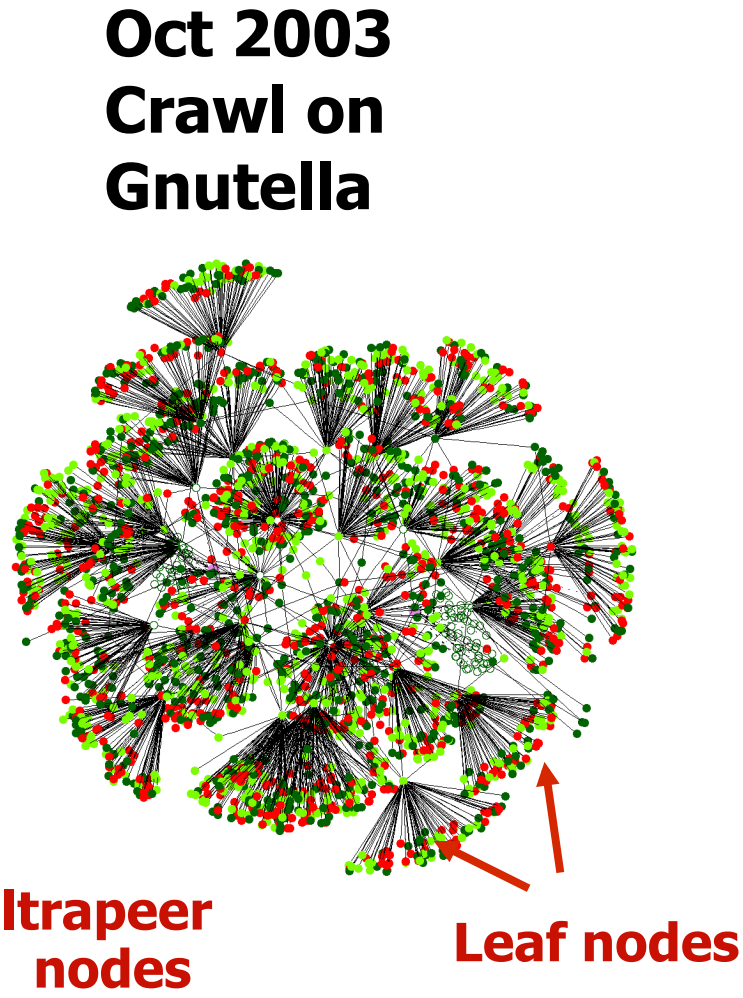
Gnutella: Example

- Assume: m1's neighbors are m2 and m3; m3's neighbors are m4 and m5;…



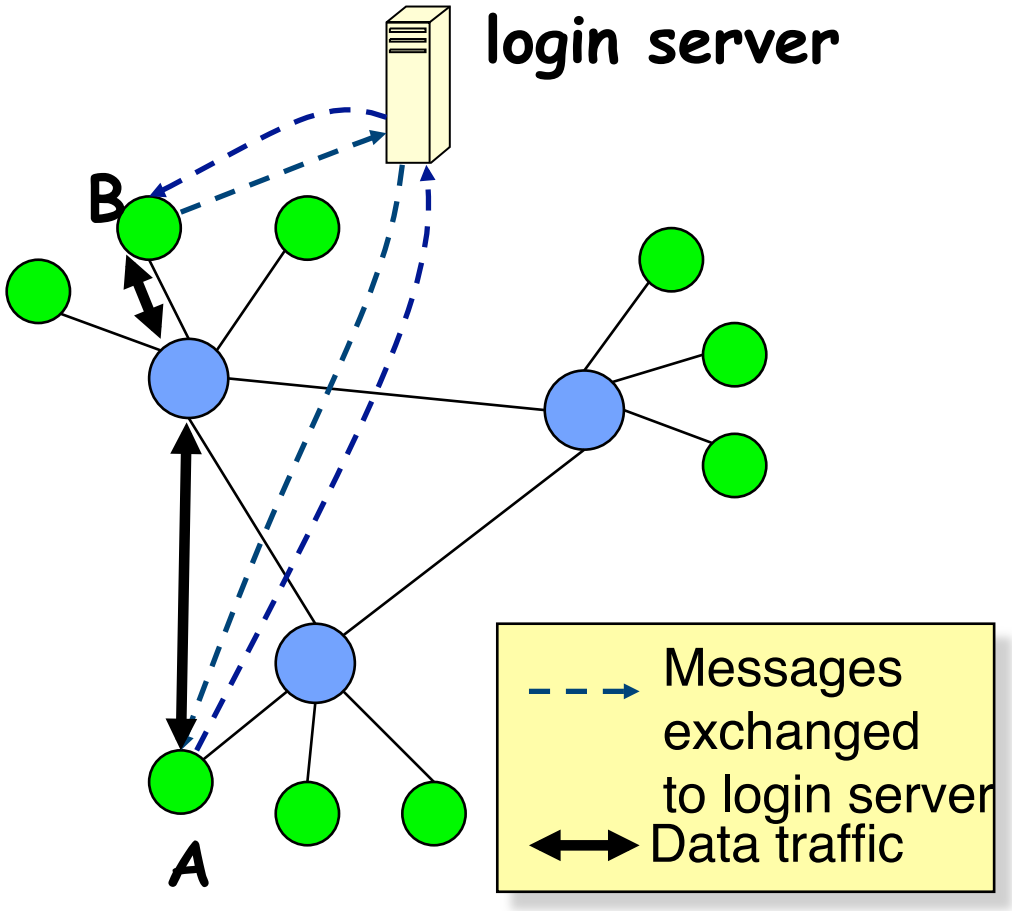
Two-Level Hierarchy

- Current Gnutella implementation, KaZaa
- Leaf nodes are connected to a small number of ultrapeers (suppernodes)
- Query
  - A leaf sends query to its ultrapeers
  - If ultrapeers don't know the answer, they flood the query to other ultrapeers
- More scalable:
  - Flooding only among ultrapeers



Skype

- Peer-to-peer Internet Telephony
- Two-level hierarchy like KaZaa
  - Ultrapeers used mainly to route traffic between NATed end-hosts (see next slide)…
  - … plus a login server to
    - » authenticate users
    - » ensure that names are unique across network



(Note*: probable protocol; Skype protocol is not published)

Conclusions

- Hypertext Transport Protocol: request-response
  - Use DNS to locate web server
  - HTTP 1.1 vs. 1.0: added support for persistent connections and pipeline to improve performance
  - Caching: key to increase scalability

- The key challenge of building wide area P2P systems is a scalable and robust directory service

- Solutions covered in this lecture
  - Naptser: centralized location service
  - Gnutella: broadcast-based decentralized location service