# CS162
# Operating Systems and Systems Programming
# Lecture 22

## Client-Server

April 18, 2011
Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

## Distributed Systems are Everywhere!

- We need (want?) to share physical devices (e.g., printers) and information (e.g., files)
- Many applications are distributed in nature (e.g., ATM machines, airline reservations)
- Many large problems can be solved by decomposing smaller problems that run in parallel (e.g., MapReduce, SETI@home)
- Next three lectures: go over three distributed system models
  - Client-server
  - Peer-to-peer
  - Cloud(cluster) computing



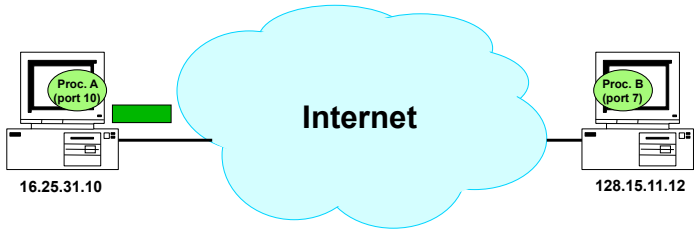
## Client-Server

- One or more clients interacting with one or more servers providing a service, e.g.,
  - Web
  - E-mail, chat
  - Printer
  - Airline reservation
  - On-line shopping
  - Store/streaming video, audio, and/or photos
  - …
- In this lecture
  - End-to-end message communication
  - Remote Procedure Calls
  - Two phase commit transactions
  - World Wide Web

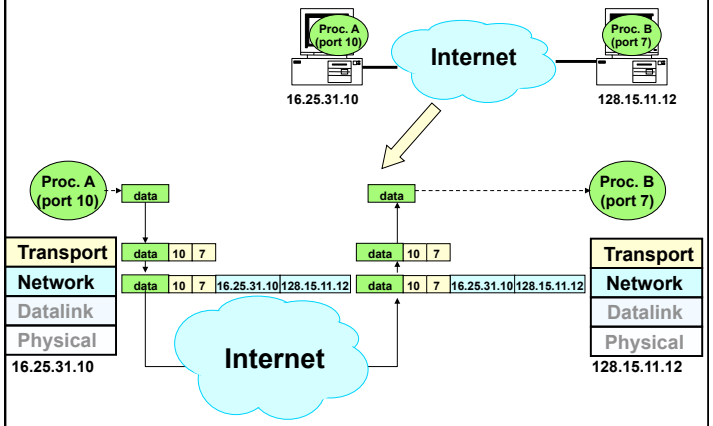


## Message Passing

- Process A (e.g., client) sends a packet to process B (e.g., server)



## Message Passing Details



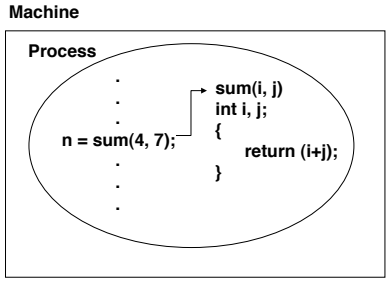



## From Message Passing to Remote Procedure Call

- Raw messaging is a bit too low-level for programming
- Another option: Remote Procedure Call (RPC)
  - Looks like a local procedure call on client
  - Translated automatically into a procedure call on remote machine (server)
- Implementation:
  - Uses request/response message passing "under the covers"



## Example: Local Procedure Call






## Remote Procedure Call

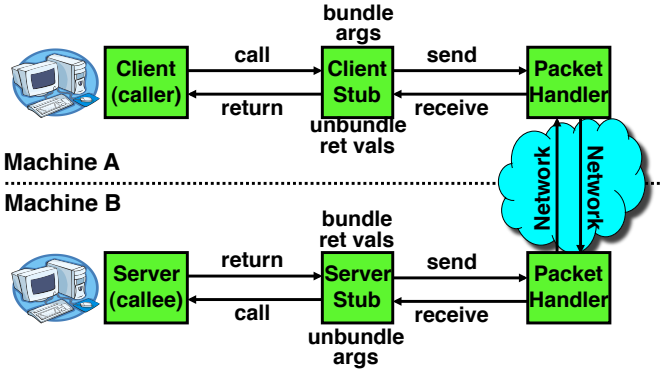
- **Transparently** invoke a procedure (services) implemented in a different address space either on the same machine or a **different** machine
  - Services can be run wherever it's most appropriate
  - Access to local and remote services looks the same
- Challenges:
  - Argument (parameter) passing, potentially across different architectures
  - Discover where the service is located
  - Handle failures **transparently**

## RPC: Argument Passing

- Client and server use "stubs" to glue pieces together
  - Client-side stub is responsible for "marshalling" arguments and "unmarshalling" the return values
  - Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values
- Marshalling involves (depending on system) converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.
  - Needs to account for cross-language and cross-platform issues
- Technique: compiler generated stubs
  - Input: interface definition language (IDL)
    - » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language

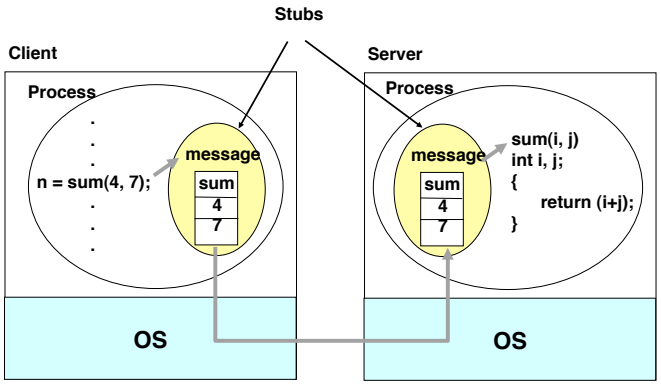


## RPC Information Flow





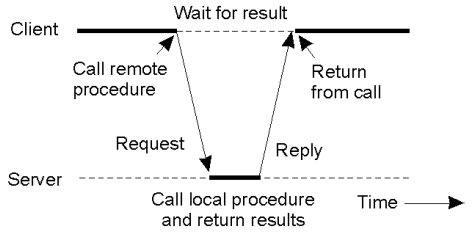

## Example: Remote Procedure Call





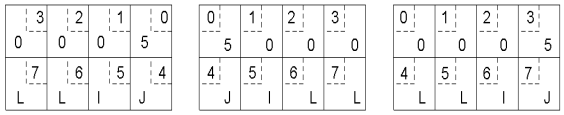

## Client and Server Stubs

- Principle of RPC between a client and server program.



## Encoding

- Server and client may encode arguments differently, e.g.,
  - Big-endian: store from most-to-least significant byte
  - Little-endian: store from least-to-most significant byte




(a) (b) (c)

a) Original message on x86 (e.g., little endian)
b) The message after receipt on the SPARC (e.g., big endian)
c) The message after being inverted. (The little numbers in boxes indicate the address of each byte)

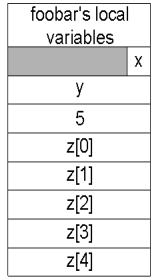


## Parameter Specification and Stub Generation

a) A procedure
b) The corresponding message.

```
foobar( char x; float y; int z[5] )
{
  ....
}
```

(a)




(b)



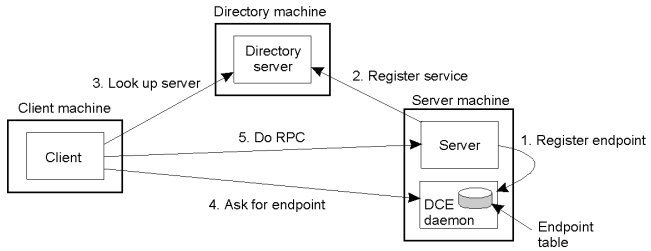
## Service Discovery: RPC Binding

- How does client know which machine to send RPC?
  - Need to translate name of remote service into network endpoint (e.g., host:port)
  - Binding: the process of converting a user-visible name into a network endpoint
    - » Static: fixed at compile time
    - » Dynamic: performed at runtime
- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
  - Why dynamic binding?
    - » Access control: check who is permitted to access service
    - » Fail-over: If server fails, use a different one



## Example of RPC Binding

- Distributed Computing Environment (DCE) framework
- DCE daemon:
  - Allow local services to record their services locally
  - Resolve service name to local end-point (i.e., port)
- Directory machine: resolve service name to DCE daemon (host:port') on machine running the service



## RPC Semantics in the Presence of Failures

- The client is unable to locate the server
- The request message from the client to server is lost
- The reply message from the server is lost
- The server crashes after receiving a request
- The client crashes after sending a request



## Client is Unable to Locate Server

- Causes: server down, different version of server binary, …
- Fixes
  - Return (-1) to indicate failure (in Unix use *errno* to indicate failure type)
    - » What if (-1) is a legal return value?
  - Use exceptions
    - » Transparency is lost



## Lost Request Message

- Easiest to deal with
- Just retransmit the message!
- If multiple message are lost then
  - "client is unable to locate server" error



## Lost Reply Message

- Far more difficult to deal with: client doesn't know what happened at server
  - Did server execute the procedure or not?
- Possible fixes
  - Retransmit the request
    - » Only works if operation is **idempotent**: it's fine to execute it twice
  - What if operation not idempotent?
    - » Assign unique sequence numbers to every request

## 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?



## Client Crashes

- Let's the server computation **orphan**
- Orphans can
  - Waste CPU cycles
  - Lock files
  - Client reboots and it gets the old reply immediately



## Client Crashes: Possible Solutions

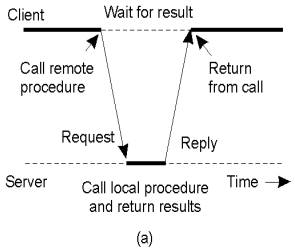
- Extermination:
  - Client keeps a log, reads it when reboots, and kills the orphan
  - Disadvantage: high overhead to maintain the log
- Reincarnation:
  - Divide times in epochs
  - Client broadcasts epoch when reboots
  - Upon hearing a new epoch servers kills the orphans
  - Disadvantage: doesn't solve problem when network partitioned
- Expiration:
  - Each RPC is given a lease T to finish computation
  - If it does not, it needs to ask for another lease
  - If client reboots after T sec all orphans are gone
  - Problem: what is a good value of T?



## RPC Semantics: Discussion

- The original goal: provide the same semantics as a local call
- Impossible to achieve in a distributed system
  - Dealing with remote failures fundamentally affects transparency
- Ideal interface: balance the easy of use with making visible the errors to users

## Asynchronous RPC (1)



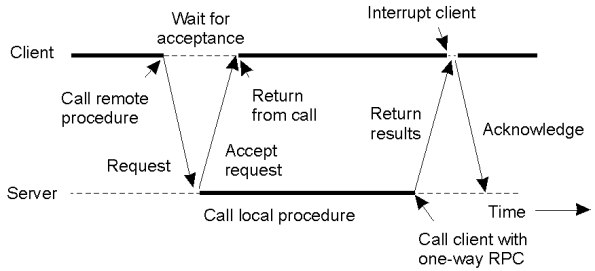

a) The interconnection between client and server in a traditional RPC

b) The interaction using asynchronous RPC



## Asynchronous RPC (2)

- A client and server interacting through two asynchronous RPCs






## Decisions in the Presence of Failures

- How does the client know for sure whether a server has performed an operation on the behalf of the client or not?
- General solution: **two phase commit**:
  - Ensure two or more parties come at the same same decision even if one or more machines fail
  - Only fail stop failures; cannot handle arbitrary failures (e.g., malicious nodes or communication networks). Why?
- Two-Phase Commit Protocol
  - One node plays the role of "coordinator"
  - Phase 1, coordinator sends out a request to commit
    - » each participant responds with yes or no
  - Phase 2
    - » If everyone says yes, coordinator sends out a commit
    - » If someone says no, coordinator sends out an abort

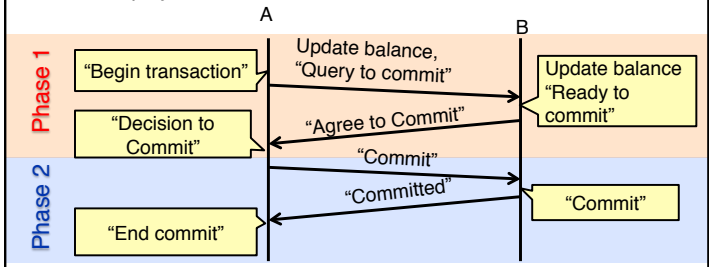


## Two-Phase Commit Details

- Each participant uses a local, persistent, corrupt-free log to keep track of whether a commit has happened
  - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
- Log can be used to complete this process such that all machines either commit or don't commit
- Timeouts can be used to retry if coordinator doesn't hear from all participants

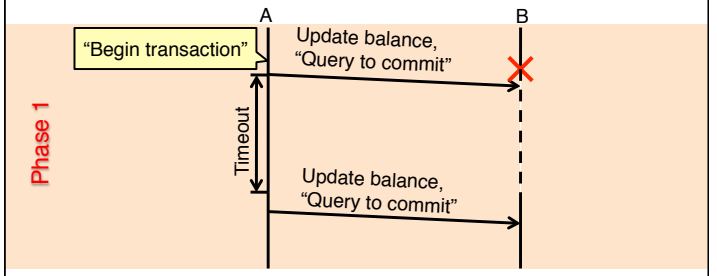
## Two-Phase Commit Example

- Simple Example: A≡Client, B≡Bank
- A wants to make a deposit and then wants to make sure that the bank has indeed updated the balance
  - A plays the coordinator role in this case





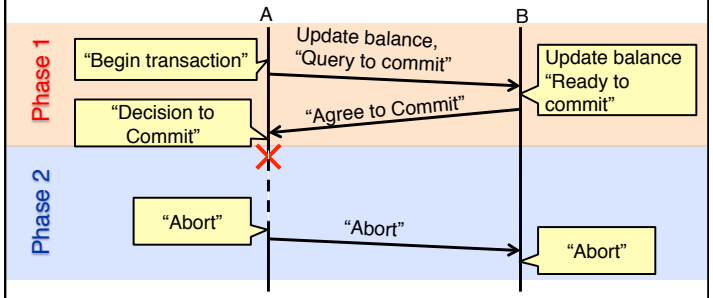

## Two-Phase Commit Example (cont'd)




- What if B crashes at beginning?
  - Wakes up, does nothing; A will timeout and retry



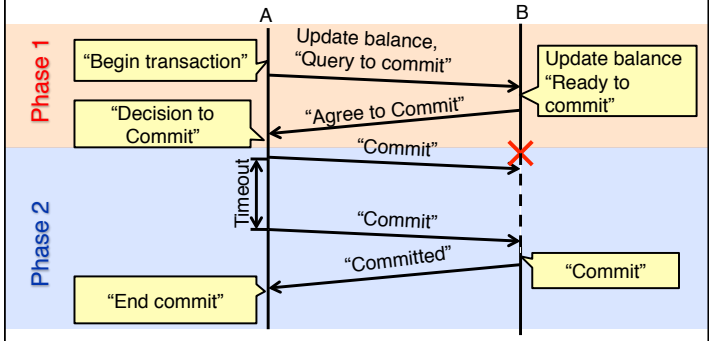
## Two-Phase Commit Example (cont'd)




- What if A crashes at beginning of phase 2?
  - Wakes up, sees that there is a transaction in progress; sends "`Abort`" to B



## Two-Phase Commit Example (cont'd)




- What if B crashes at beginning of phase 2?
  - B comes back up, looks at log; when A (re)sends it "`Commit`" message, it will say, "oh, ok, commit"

## Two-Phase Commit Gotchas

- Undesirable feature of Two-Phase Commit: blocking
  - One machine can be stalled until another site recovers, e.g.,
    - » Coordinator crashes → everyone needs to wait for it to come back;
  - A blocked site holds resources (locks on updated items, pages pinned in memory, etc) until learns fate of update
- Alternatives such as "Three Phase Commit" don't have this blocking problem



## The Web – History (I)

Vannevar Bush (1890-1974)

Memex

- 1945: Vannevar Bush, Memex:

*"a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility"*

**(See http://www.iath.virginia.edu/elab/hfl0051.html)**



## The Web – History (II)

Ted Nelson

- 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 (III)

Tim Berners-Lee

- 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

## 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
```

- Extend the idea of hierarchical namespaces to include anything in a file system
  - ftp://www.cs.berkeley.edu/~istoica/pubs.html
- 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
  - Server side processing can be incorporated in the name

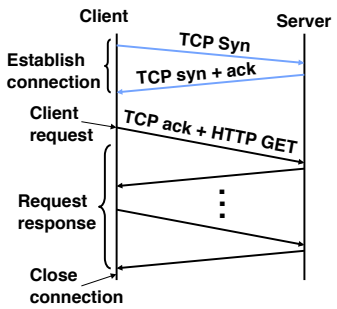


## Hyper Text Transfer Protocol (HTTP)

- Client-server architecture
- Synchronous request/reply protocol
  - Runs over TCP, Port 80
- Stateless



## Big Picture



## 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)



## Response Codes

- 1x informational
- 2x success
- 3x redirection
- 4x client error in request
- 5x server error; can't satisfy the request



## Client Request

- Steps to get the resource:

  http://www.eecs.berkeley.edu/index.html

  1. Use DNS to obtain the IP address of www.eecs.berkeley.edu
  2. Send to an HTTP request:

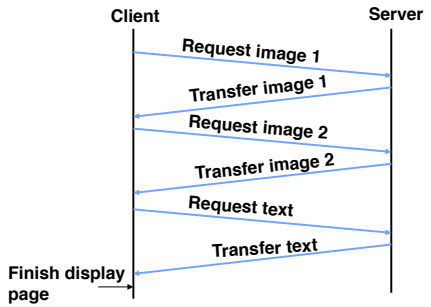
```
GET /index.html HTTP/1.0
```



## Server Response

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 1234
Last-Modified: Mon, 19 Nov
2001 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 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

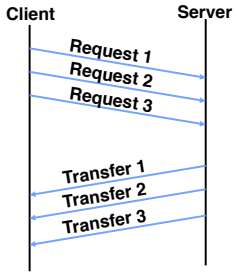
## Persistent Connections

- 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





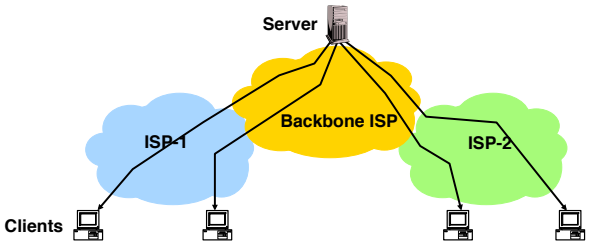

## Caching and Replication

- 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)



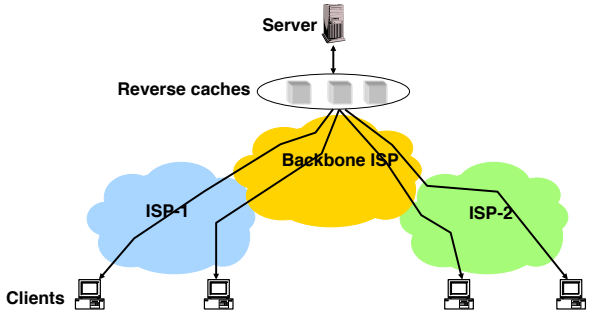
## "Base-line"

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



## Reverse Caches

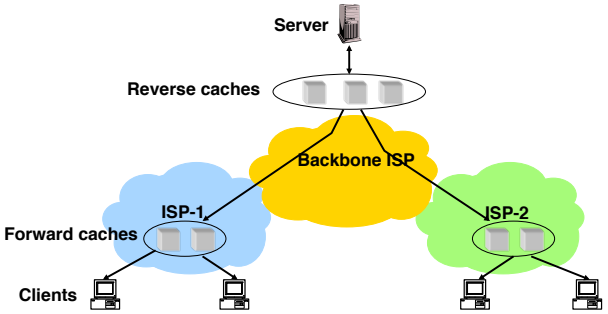
- Cache documents close to server → decrease server load
- Typically done by content providers






## Forward Proxies

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






## Content Distribution Networks (CDNs)

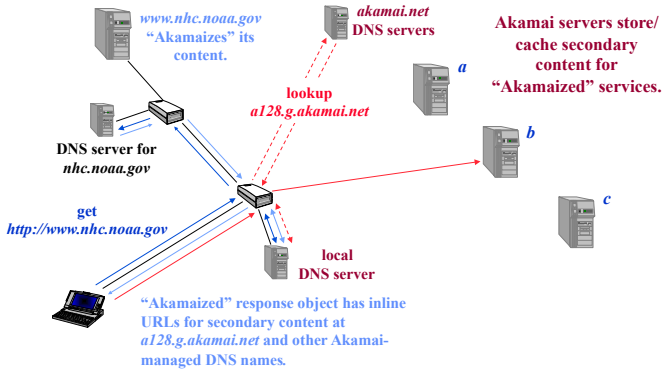
- Integrate forward and reverse caching functionalities into one overlay network (usually) administrated by one entity
  - Example: Akamai
- Documents are cached both
  - As a result of clients' requests (pull)
  - Pushed in the expectation of a high access rate
- Beside caching do processing, e.g.,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






## Summary

- Remote Procedure Call (RPC): Call procedure (service) on remote machine
  - Provides same interface as local procedure call
  - Automatic packing and unpacking of arguments without user programming (in stub)
- Two-phase commit: distributed decision making
  - First, make sure everyone guarantees that everyone is ready to commit
  - Next, ask everyone to commit
- Hypertext Transport Protocol: request-response
  - Use DNS to locate server
  - HTTP 1.1 vs. 1.0: added support for persistent connections and pipeline to improve performance
  - Caching: key to increase scalability