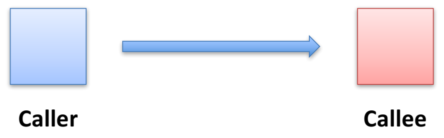
# Naming and layering

# Replicated storage, consistency

COS 518: *Advanced Computer Systems*
Lecture 2

Mike Freedman

## Naming and system components



- How to design interface between components?
- Many interactions involve naming things
  - Naming objects that caller asks callee to manipulate
  - Naming caller and callee together

## Potential Name Syntax

- Human readable?
  - If users interact with the names
- Fixed length?
  - If equipment processes at high speed
- Large name space?
  - If many nodes need unique names
- Hierarchical names?
  - If the system is very large and/or federated
- Self-certifying?
  - If preventing “spoofing” is important

## Properties of Naming

- Enabling sharing in applications
  - Multiple components or users can name a shared object.
  - Without names, client-server interface pass entire object by value
- Retrieval
  - Accessing same object later on, just by remembering name
- Indirection mechanism
  - Component A knows about name N
  - Interposition: can change what N refers to without changing A
- Hiding
  - Hides impl. details, don’t know where google.com located
  - For security purposes, might only access resource if know name (e.g., dropbox or Google docs URL –> knowledge gives access)

## Names all around…

- Registers: LD R0, 0x1234
- IP addresses: 128.112.132.86
- Host names: www.cs.princeton.edu
- Path names: /courses/archive/spring17/cos518/syllabus.html vs. "syllabus.html"
- ".." (to parent directory)
- URLs: http://www.cs.princeton.edu/courses/archive/spring17/cos518/
- Email addresses
- Function names: ls
- Phone numbers: 609-258-9169 vs. x8-9179
- SSNs

5

## High-level view of naming

- Set of possible names
- Set of possible values that names map to
- Lookup algorithm that translates name to value
- Optional context that affects the lookup algorithm

6

## Helping to understand naming system

- Name syntax?
- Values?
- Context used to resolve name?
- Who supplies context?
- Global (context-free) or local names?

7

## Different Kinds of Names

- **Host names:** www.cs.princeton.edu
  - Mnemonic, variable-length, appreciated *by humans*
  - Hierarchical, based on organizations
- **IP addresses:** 128.112.7.156
  - Numerical 32-bit address appreciated *by routers*
  - Hierarchical, based on organizations and topology
- **MAC addresses:** 00-15-C5-49-04-A9
  - Numerical 48-bit address appreciated *by adapters*
  - Non-hierarchical, unrelated to network topology

8

## Hierarchical Assignment Processes

- **Host names:** www.cs.princeton.edu
  - Domain: registrar for each top-level domain (eg, .edu)
  - Host name: local administrator assigns to each host
- **IP addresses:** 128.112.7.156
  - Prefixes: ICANN, regional Internet registries, and ISPs
  - Hosts: static configuration, or dynamic using DHCP
- **MAC addresses:** 00-15-C5-49-04-A9
  - Blocks: assigned to vendors by the IEEE
  - Adapters: assigned by the vendor from its block

## Case Study: Domain Name System (DNS)

Computer science concepts underlying DNS

- Indirection: names in place of addresses
- Hierarchy: in names, addresses, and servers
- Caching: of mappings from names to/from addresses

## Strawman Solution #1: Local File

- Original name to address mapping
  - Flat namespace
  - /etc/hosts
  - SRI kept main copy
  - Downloaded regularly
- Count of hosts was increasing: moving from a machine per domain to machine per user
  - Many more downloads
  - Many more updates

## Strawman Solution #2: Central Server

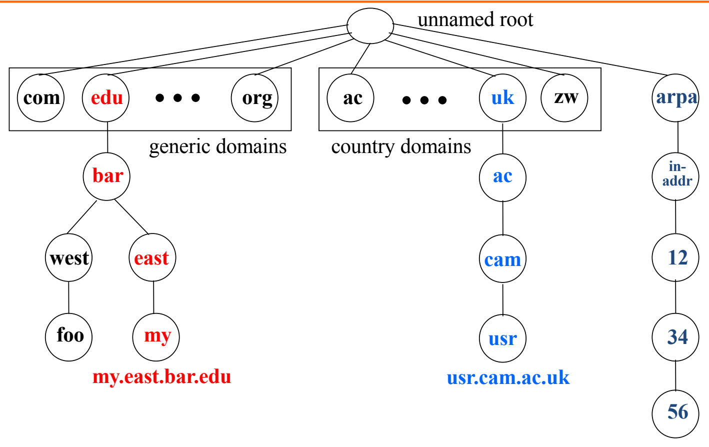
- Central server
  - One place where all mappings are stored
  - All queries go to the central server
- Many practical problems
  - Single point of failure
  - High traffic volume
  - Distant centralized database
  - Single point of update
  - Does not scale

**Need a distributed, hierarchical collection of servers**

## Domain Name System (DNS)

- Properties of DNS
  - Hierarchical name space divided into zones
  - Distributed over a collection of DNS servers
- Hierarchy of DNS servers
  - Root servers
  - Top-level domain (TLD) servers
  - Authoritative DNS servers
- Performing the translations
  - Local DNS servers and client resolvers

## Distributed Hierarchical Database

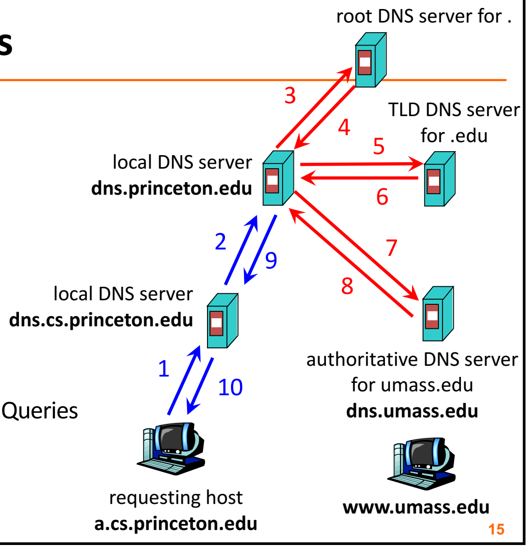


## DNS Queries

**a.cs.princeton.edu**

wants IP address for

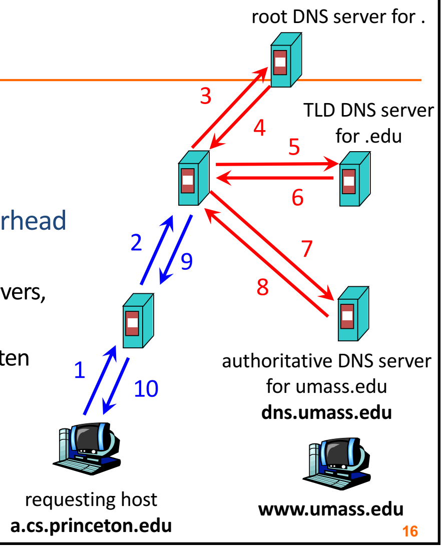
**www.umass.edu**


root DNS server for .
TLD DNS server for .edu
local DNS server dns.princeton.edu
local DNS server dns.cs.princeton.edu
authoritative DNS server for umass.edu dns.umass.edu
requesting host a.cs.princeton.edu
www.umass.edu
1
2
3
4
5
6
7
8
9
10

Recursive vs. Iterative Queries

## DNS Queries

- DNS query latency:
  - e.g., 1 second
- Caching to reduce overhead and delay
  - Small # of top-level servers, that change rarely
  - Popular sites visited often
- Where to cache?
  - Local DNS server
  - Browser


root DNS server for .
TLD DNS server for .edu
authoritative DNS server for umass.edu dns.umass.edu
requesting host a.cs.princeton.edu
www.umass.edu
1
2
3
4
5
6
7
8
9
10

## Reliability

- DNS servers are replicated
  - Name service available if at least one replica is up
  - Queries can be load balanced between replicas
- 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

17

## DNS Cache Consistency

- Goal: Ensuring cached data is up to date
- DNS design considerations
  - Cached data is "read only"
  - Explicit invalidation would be expensive
    - Server would need to keep track of all resolvers caching
- Avoiding stale information
  - Responses include a "time to live" (TTL) field
  - Delete the cached entry after TTL expires
- Perform negative caching (for dead links, misspellings)
  - So failures quick and don't overload gTLD servers

18

## Layering

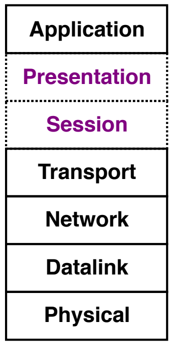
19

## Layering

- Partition the system
  - Each layer solely relies on services from layer below
  - Each layer solely exports services to layer above
- Interface between layers defines interaction
  - Hides implementation details
  - Layers can change without disturbing other layers

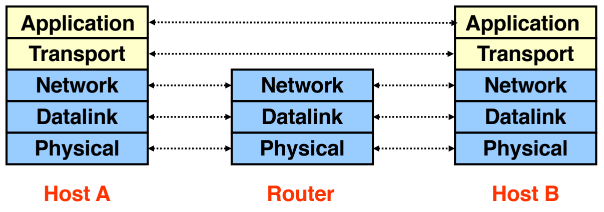
## OSI Layering Model

- Open Systems Interconnection (OSI)
  - Developed by International Organization for Standardization (OSI) in 1984
  - **Seven** layers
- Internet Protocol (IP)
  - Only **five** layers
  - The functionalities of the missing layers (i.e., Presentation and Session) are provided by the Application layer



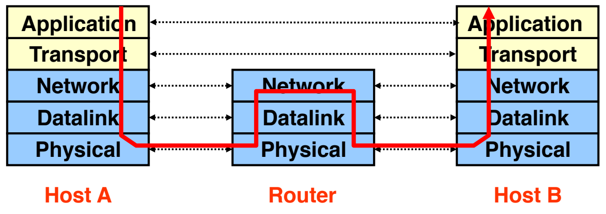
## Five Layers Summary

- Lower three layers implemented everywhere
- Top two layers implemented only at hosts
- Logically, layers interacts with peer's corresponding layer

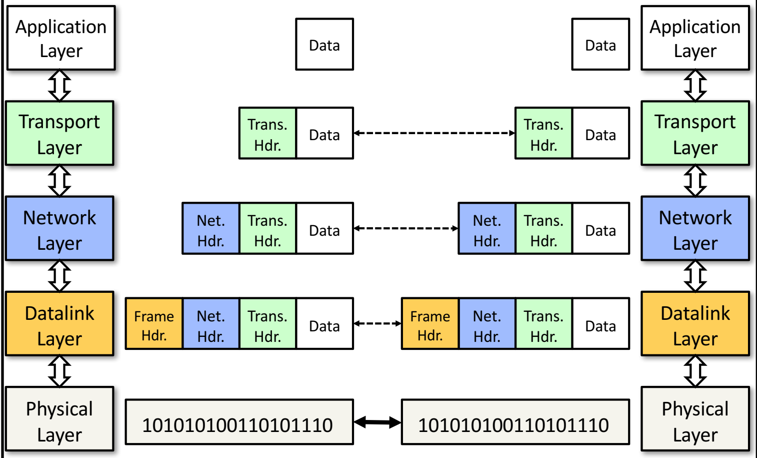


## Physical Communication

- Communication goes down to physical network
- Then from network peer to peer
- Then up to relevant layer



## Layer model and headers



## Drawbacks of Layering

- Layer N may duplicate layer N-1 functionality
  - E.g., error recovery to retransmit lost data
- Layers may need same information
  - E.g., timestamps, maximum transmission unit size
- Layering can hurt performance
  - E.g., hiding details about what is really going on
- Some layers are not always cleanly separated
  - Inter-layer dependencies for performance reasons
  - Some dependencies in standards (header checksums)
- Headers start to get really big
  - Sometimes header bytes >> actual content

## Placing Network Functionality

- Hugely influential paper: "End-to-End Arguments in System Design" by Saltzer, Reed, and Clark ('84)
- "Sacred Text" of the Internet
  - Endless disputes about what it means
  - Everyone cites it as supporting their position

## – Paper Discussion –

27

## Intro to fault tolerant + consistency

28

## What is fault tolerance?

- Building reliable systems from unreliable components

- Three basic steps
  1. **Detecting errors**: discovering presence of an error in a data value or control signal
  2. **Containing errors**: limiting how far errors propagate
  3. **Masking errors**: designing mechanisms to ensure system operates correctly despite error (+ possibly correct error)

## Why is fault tolerance hard?

**Failures Propagate**

- Say one bit in a DRAM fails…
- …it flips a bit in a memory address the kernel is writing to…
- …causes big memory error elsewhere, or a kernel panic…
- …program is running one of many distributed file system storage servers…
- …a client can't read from FS, so it hangs

## So what to do?

1. **Do nothing**: silently return the failure
2. **Fail fast**: detect the failure and report at interface
   - Ethernet station jams medium on detecting collision
3. **Fail safe**: transform incorrect behavior or values into acceptable ones
   - Failed traffic light controller switches to blinking-red
4. **Mask the failure**: operate despite failure
   - Retry op for transient errors, use error-correcting code for bit flips, replicate data in multiple places

## Masking failures

- We mask failures on one server via
  - Atomic operations
  - Logging and recovery

- In a distributed system with multiple servers, we might replicate some or all servers

- But if you give a mouse some replicated servers
  - She's going to need to figure out how to keep the state of the servers consistent (immediately? eventually?)

## Safety and liveness

## Reasoning about fault tolerance

- This is hard!
  - How do we design fault-tolerant systems?
  - How do we know if we're successful?
- Often use "properties" that hold true for every possible execution
- We focus on **safety** and **liveness** properties

## Safety

- "Bad things" don't happen
  - No stopped or deadlocked states
  - No error states
- Examples
  - **Mutual exclusion:** two processes can't be in a critical section at the same time
  - **Bounded overtaking:** if process 1 wants to enter a critical section, process 2 can enter at most once before process 1

## Liveness

- "Good things" happen
  - ...eventually
- Examples
  - **Starvation freedom:** process 1 can eventually enter a critical section as long as process 2 terminates
  - **Eventual consistency:** if a value in an application doesn't change, two servers will eventually agree on its value

## Often a tradeoff

- "Good" and "bad" are application-specific
- Safety is very important in banking transactions
- Liveness is very important in social networking sites

37

## Eventual Consistency

38

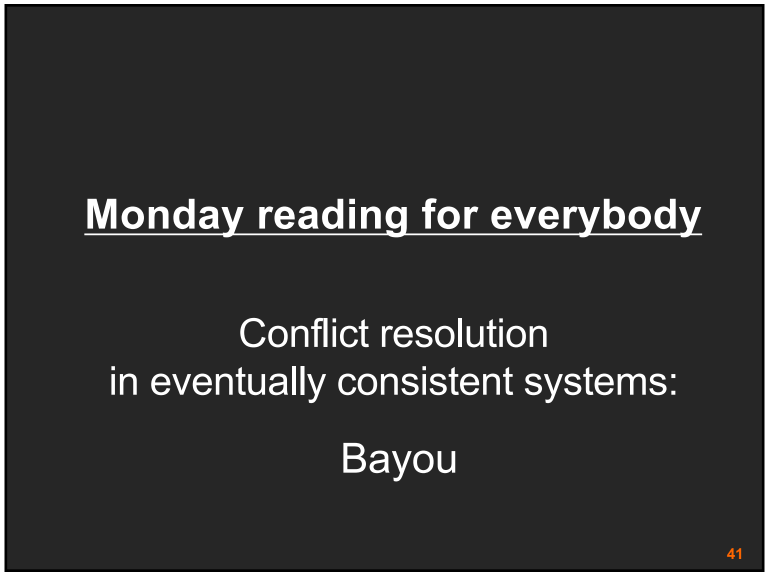
## Eventual consistency

- Def'n: If no new updates to the object, eventually all accesses will return the last updated value
- Common: git, iPhone sync, Dropbox, Amazon Dynamo
- Why do people like eventual consistency?
  - Fast read/write of local copy (no primary, no Paxos)
  - Disconnected operation
- Challenges
  - How do you discover other writes?
  - How do you resolve conflicting writes?

39

## Two prevailing styles of discovery

- Gossip pull ("anti-entropy")
  - A asks B for something it is trying to "find"
  - Commonly used for management replicated data
    - Resolve differences between DBs by comparing digests
- Gossip push ("rumor mongering"):
  - A tells B something B doesn't know
  - Gossip for multicasting
    - Keep sending for bounded period of time: *O (log n)*
  - Also used to compute aggregates
    - Max, min, avg easy. Sum and count more difficult.
- Push-pull gossip
  - Combines both : O(n log log n) msgs to spread in O(log n) time



## Monday reading for everybody

Conflict resolution
in eventually consistent systems:

Bayou

41