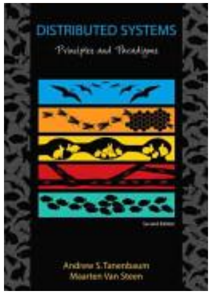
Chapter 5: NAMING

How to refer to an entity in Distributed Systems?



Thanks to the authors of the textbook [**TS**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.

Turgay Korkmaz
korkmaz@cs.utsa.edu

Chapter 5: NAMING

- NAMES, IDENTIFIERS, AND ADDRESSES
- FLAT NAMING
  - Simple Solutions
  - Home-Based Approaches
  - Distributed Hash Tables
  - Hierarchical Approaches
- STRUCTURED NAMING
  - Name Spaces
  - Name Resolution
  - The Implementation of a Name Space
  - Example: The Domain Name System
- ATTRIBUTE-BASED NAMING
  - Directory Services
  - Hierarchical Implementations: LDAP
  - Decentralized Implementations

Objectives

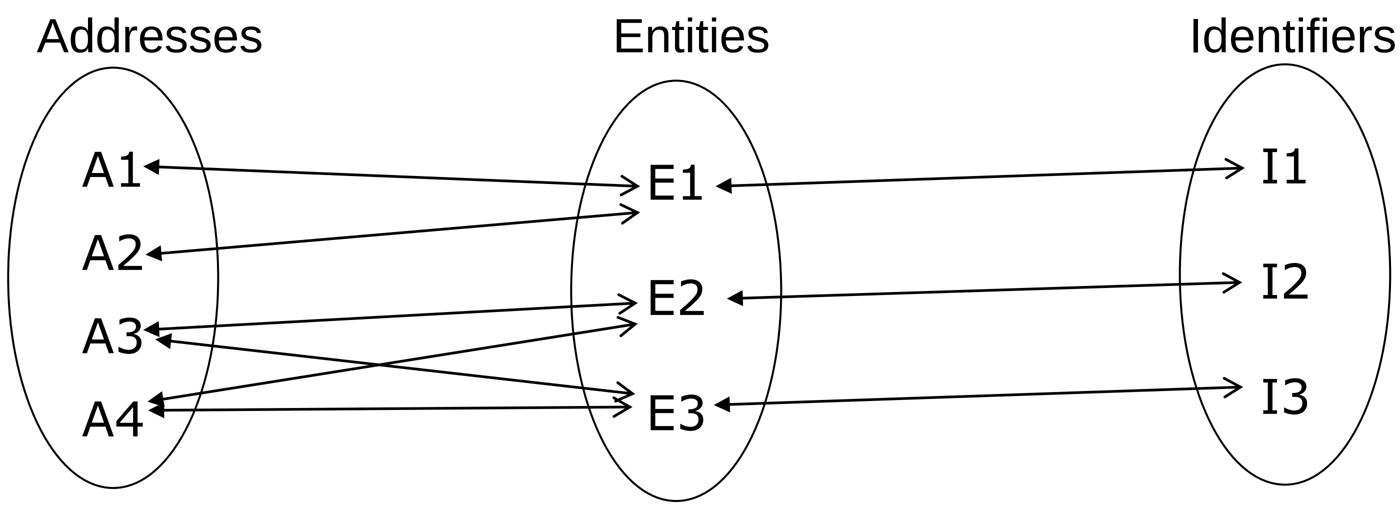
- To understand naming and related issues in DS
- To learn naming space and implementation
- To learn flat and structured names and how they are resolved
- To learn Attributed-based naming

What a Name is in DS?

- A name is a string of bits or characters that is used to refer to an entity (an entity could be anything such as host, printer, file, process, mailbox, user etc.)
- To operate on an entity, we need to access it, for which we need an access point.
- Access point is a special kind of entity and its name is called an address (address of the entity, e.g., IP, port #, phone #)
  - An entity may have more than one access point/address
  - An entity may change its access points/addresses
  - So using an address as a reference is inflexible and human unfriendly
  - A better approach is to use a name that is location independent, much easier, and flexible to use

Identifier

- A special name to uniquely identify an entity (SSN, MAC)
- A true identifier has the following three properties:
  - **P1:** Each identifier refers to at most one entity
  - **P2:** Each entity is referred to by at most one identifier
  - **P3:** An identifier always refers to same entity (no reuse)



Addresses and identifiers are important and used for different purposes, but they are often represented in machine readable format (MAC, memory address)

Human-friendly names

- File names, www.cs.utsa.edu, variable names etc. are human-friendly names given to each entity
- **Question**: how to map/resolve these names to addresses so that we can access the entities on which we want to operate?
- **Solution**: have a naming system that maintains name-to-address binding!
- The simplest form is to have a centralized table!
  - Why or why not this will work?
- We will study three different naming systems and how they maintain such a table in a distributed manner!

Naming Systems and Their Goals

- Naming Systems
  - Flat names
  - Structured names
  - Attributed-based names
- Goals
  - Scalable to arbitrary size
  - Have a long lifetime
  - Be highly available
  - Have fault isolation
  - Tolerate mistrust

FLAT NAMES

Flat Naming

- **Flat name**: random bits of string, no structure
  - E.g., SSN, MAC address
- **Resolution problem**:

  Given a flat (unstructured) name, how can we find/locate its associated access point and its address?
- **Solutions**:
  - Simple solutions (broadcasting)
  - Home-based approaches
  - Distributed Hash Tables (structured P2P)
  - Hierarchical location service

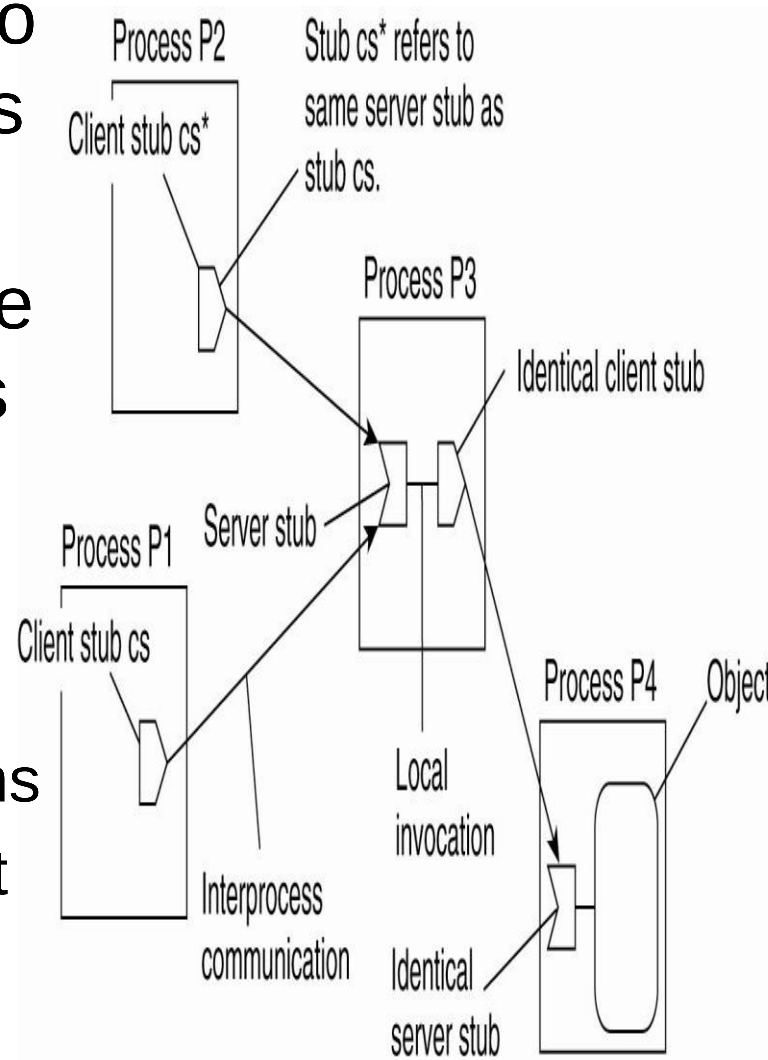
Simple Solution: Broadcasting

- Simply **broadcast** the target ID to every entity
- Each entity compares the requested ID with its own ID
- The target entity returns its current address
- Example:
  - Recall ARP in LAN
- Adv/Disadvantages
  - + simple
  - - not scale beyond LANs
  - - it requires all entities to listen to all incoming requests

Forwarding Pointers

How to locate mobile entities?

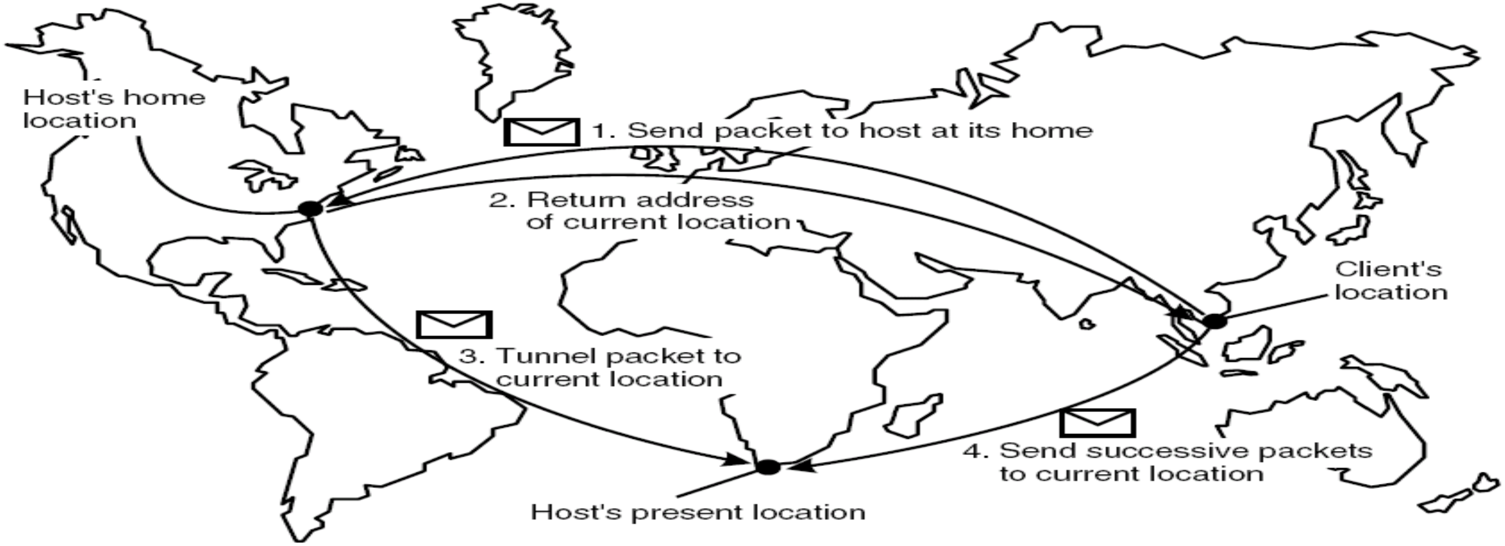
- When an entity moves from A to B, leaves a pointer to A that it is at B now…
- Dereferencing: simply follow the chain of pointers and make this entirely transparent to clients
- Adv/Disadvantages
  - + support for mobile nodes
  - - geographical scalability problems
  - - long chains are not fault tolerant
  - - increased network latency
  - Short-cuts can be introduced



Home-Based Approaches

How to deal with scalability problem when locating mobile entities?

- Let a **home** keep track of where the entity is!



- How will the clients continue to communicate?
  - Home agent gives the new location to the client so it can directly communicate
    - efficient but not transparent
  - Home agent forwards the messages to new location
    - Transparent but may not be efficient

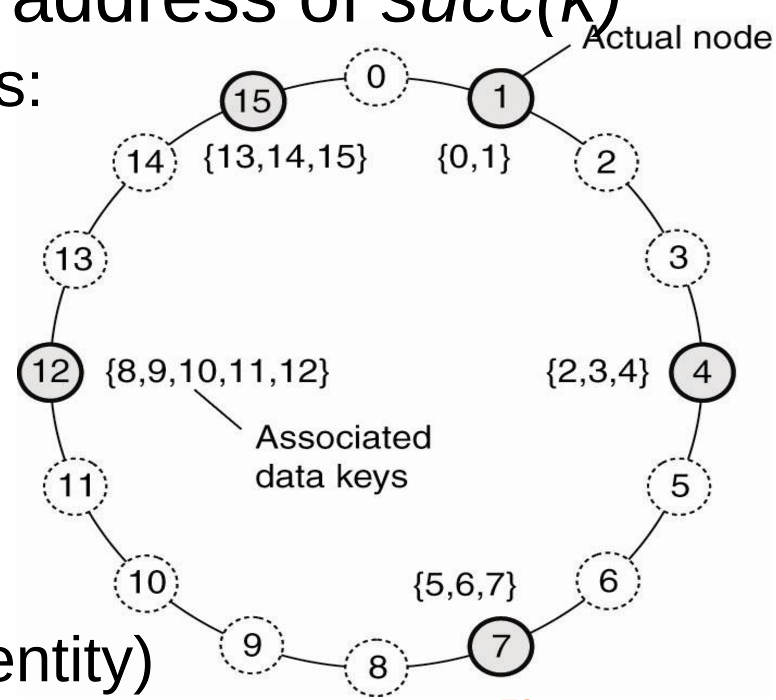
Problems with home-based approaches

- The home address has to be supported as long as the entity lives.
- The home address is fixed, which means an unnecessary burden when the entity **permanently moves** to another location
  - How can we solve the “permanent move” problem?
- Poor geographical scalability (the entity may be next to the client)

Distributed Hash Tables

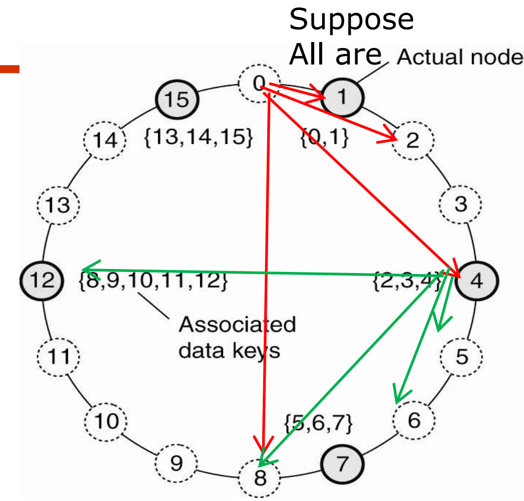
How to use DHT to resolve flat ID

- Recall Chord from Chapter 2, which organizes many nodes into a logical ring
  - Each node is assigned a random *m*-bit **identifier**.
  - Every entity is assigned a unique *m*-bit **key**.
  - Entity with key *k* falls under jurisdiction of node with smallest *id* >= *k* (called its **successor**)
- **Linearly** resolve a key *k* to the address of *succ(k)*
  - Each node p keeps two neighbors: succ(p+1) and pred(p)
  - If k > p then forward to succ(p+1)
  - if k <= pred(p) then forward k to pred(p)
  - If pred(p) < k <= p then return p's address (p holds the entity)



DHT: Finger Table

How to improve efficiency?



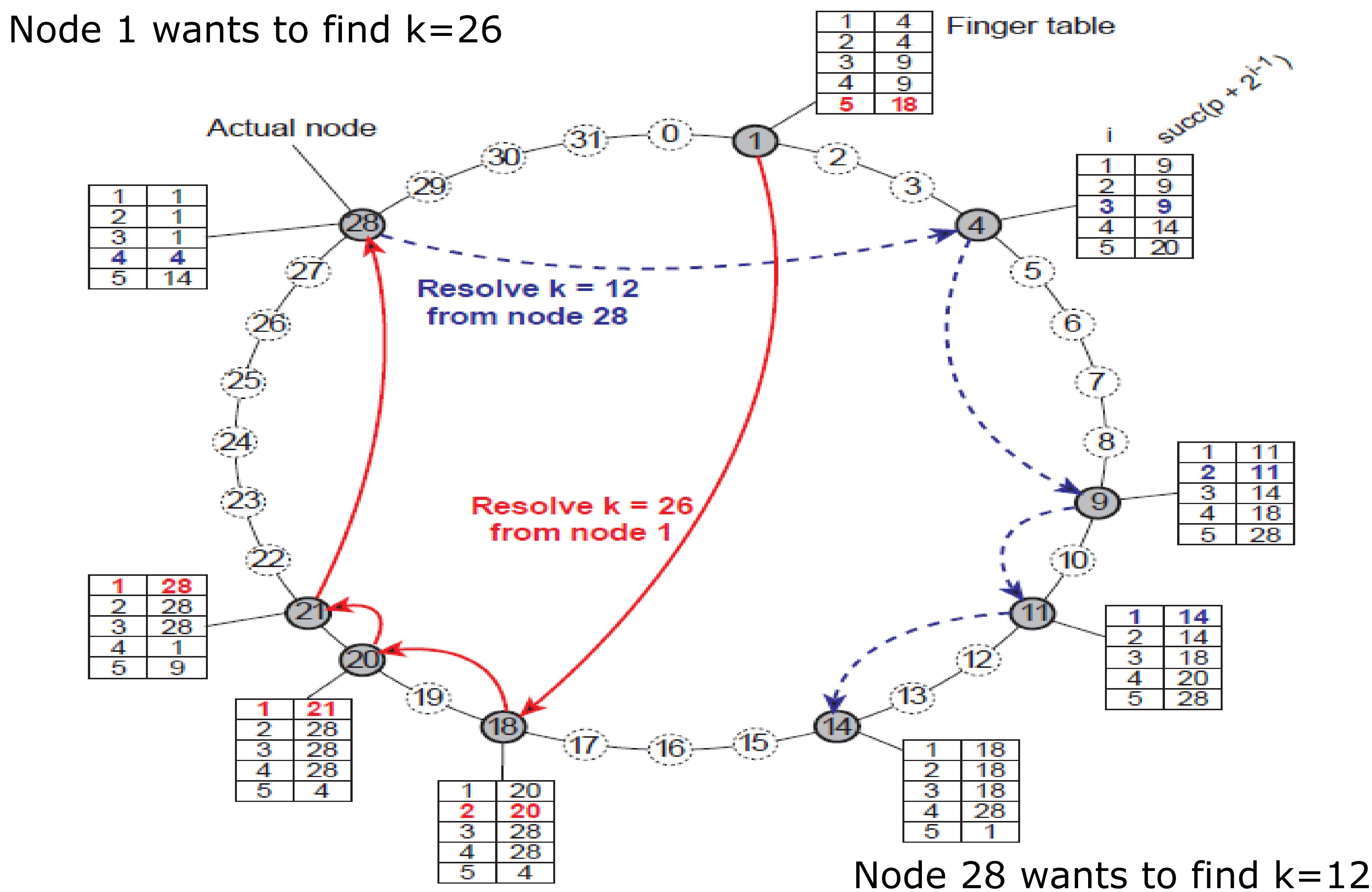
- Each node *p* maintains a **finger table**
  - at most *m* entries (short cuts) with exponentially increasing size

    $FTp[i] = succ(p + 2^{i-1})$
- *FTp[i]* points to the first node succeeding *p* by at least 2^{i-1}
- To look up a key *k*, node *p* forwards the request to node with index *j* satisfying (e.g., node 0 gets a req for k=6)

  $q = FTp[j] \leq k < FTp[j+1]$ (e.g., node 0 sends req →4 → 6)

  If $p < k < FTp[1]$, the request is also forwarded to *FTp*[1]
- Need at most O(log N) steps, where N is the number of nodes in the systems

DHT: Example


Node 1 wants to find k=26
Finger table
i
succ(p + 2^(i-1))
Actual node
Resolve k = 12
from node 28
Resolve k = 26
from node 1
Node 28 wants to find k=12
Node 1: 1 4, 2 4, 3 9, 4 9, 5 18
Node 4: 1 9, 2 9, 3 9, 4 14, 5 20
Node 9: 1 11, 2 11, 3 14, 4 18, 5 28
Node 11: 1 14, 2 14, 3 18, 4 20, 5 28
Node 14: 1 18, 2 18, 3 18, 4 28, 5 1
Node 18: 1 20, 2 20, 3 28, 4 28, 5 4
Node 20: 1 21, 2 28, 3 28, 4 28, 5 4
Node 21: 1 28, 2 28, 3 28, 4 1, 5 9
Node 28: 1 1, 2 1, 3 1, 4 4, 5 14
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

DHT: Finger Table (cont'd)

- How to handle
  - Join
  - Leave
  - Fail
- The complexity comes from keeping the finger tables up to date
- By-and-large Chord tries to keep them consistent
  - But a simple mechanism may lead to performance problems
  - To fix this we need to exploit network proximity when assigning node ID

Exploiting network proximity

Problem: The logical organization of nodes in the overlay may lead to erratic message transfers in the underlying Internet: node k and node succ(k +1) may be very far apart.

- Topology-aware node assignment:

  When assigning an ID to a node, make sure that nodes close in the ID space are also close in the network. Can be very difficult.

- Proximity routing:

  Maintain more than one possible successor, and forward to the closest.

  Example: in Chord FTp[i] points to first node in INT = [p+2i−1,p+2i −1]. Node p can also store pointers to other nodes in INT .

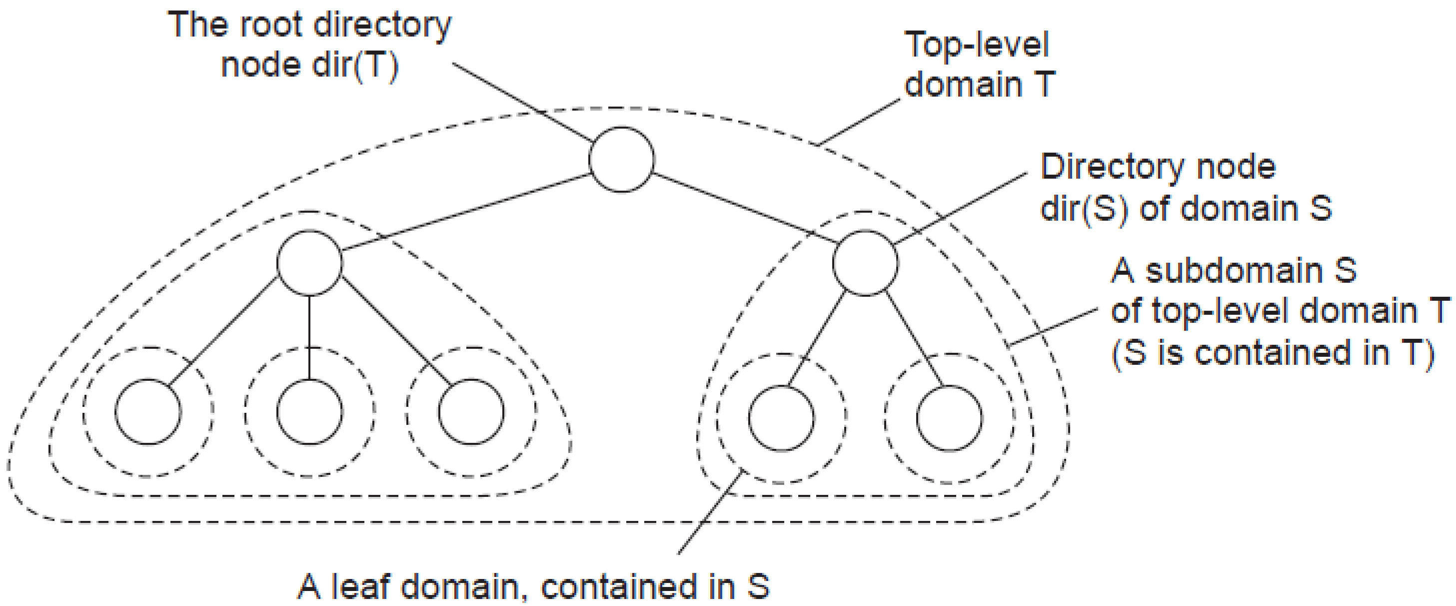
- Proximity neighbor selection:

  When there is a choice of selecting who your neighbor will be (not in Chord), pick the closest one.

Hierarchical Location Services (HLS)

to resolve flat names

- Build a large-scale search tree for which the underlying network is divided into hierarchical domains. Each domain is represented by a separate directory node.



www.cs.utsa.edu

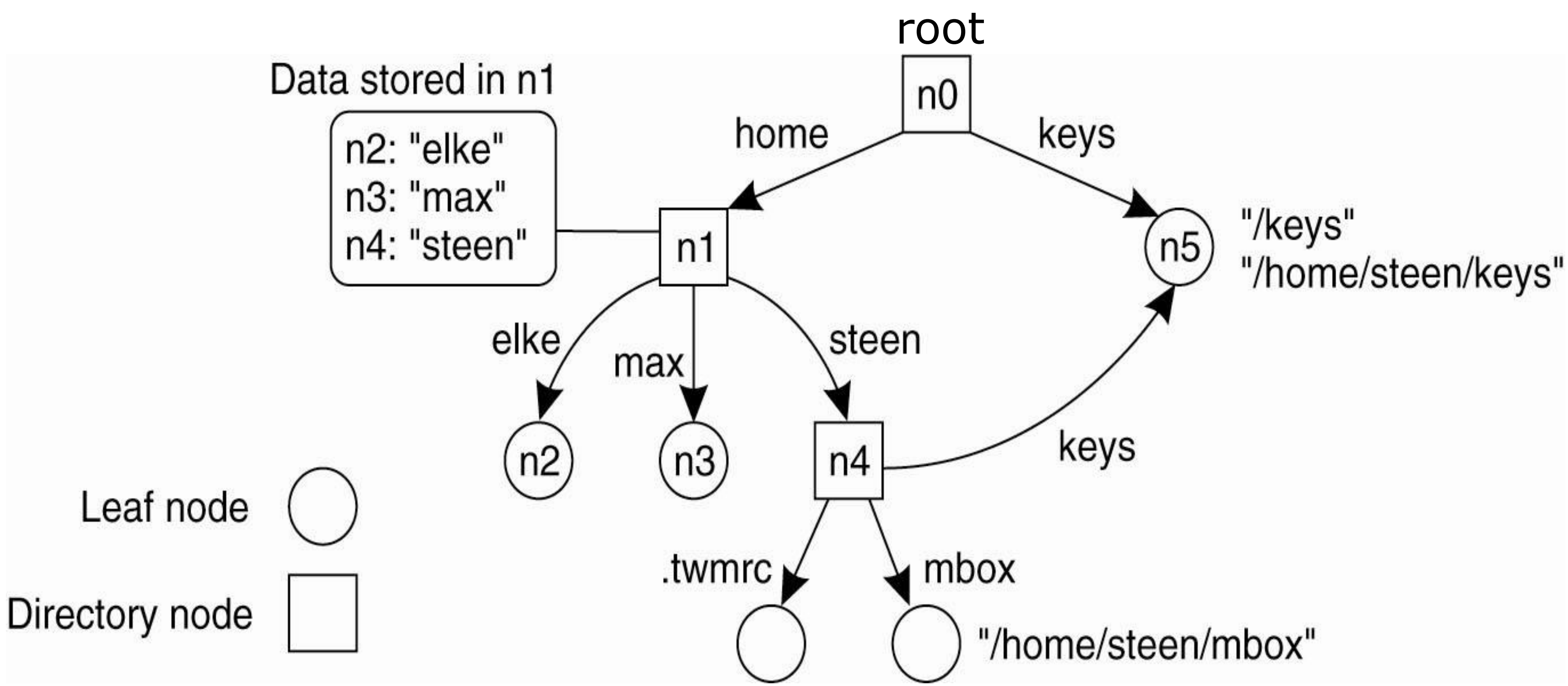
dir1/dir2/file.txt

STRUCTURED NAMING

Name Space

Collection of valid names

- A directed **graph** with two types of nodes
  - **Leaf node** represents a (named) entity, has no outgoing link, and stores information about the entity (e.g., address)
  - A **directory node** is an entity that refers to other nodes: contains a (directory) table of *(edge label, node identifier)* pairs



Name Space (cont.)

- Each node in the graph is actually considered to be another entity and we can easily store all kinds of **attributes** in a node, describing aspects of the entity the node represents:
  - Type of the entity
  - An **identifier** for that entity
  - **Address** of the entity's location
  - Nicknames
  - … …

Name Resolution

looking up a name

N: <label-1, label-2, …, label-n>

Start at directory node N

find label-1 in directory table of N

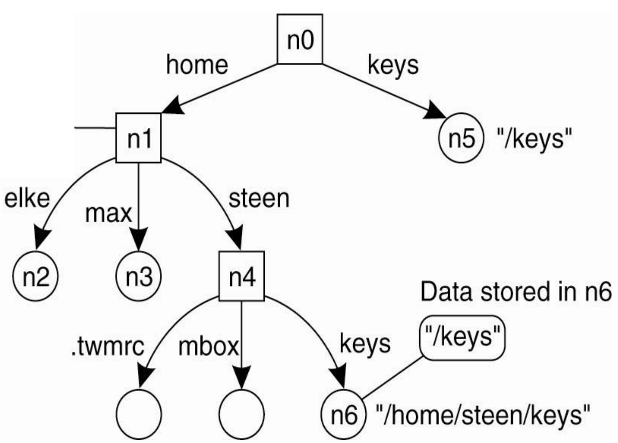
get the identifier

continue resolving at that node until reaching label-n

- **Problem:** where to start? How do we actually find that **(initial) node**?
- **Closure mechanism:** knowing how and where to start name resolution. It is always implicit. Why?
  - Inode in unix is the first block in logical disk
  - www.cs.vu.nl: start at a DNS name server
  - /home/steen/mbox: start at the local NFS file server (possible recursive search)

Name Resolution: Aliases and linking

- Alias is another name for the same entity.
- There are 2 ways of aliasing in naming graphs
  - **Hard Links**: What we have described so far as a path name: a name that is resolved by following a specific path in a naming graph from one node to another (i.e., there are more than one absolute paths to a certain node)
  - **Soft Links**: We can represent an entity by a leaf node that stores an absolute path name of another node. (like symbolic links in UNIX file system)
    - Node O contains a name of another node:
    - First resolve O's name (leading to O)
    - Read the content of O, yielding name
    - Name resolution continues with name



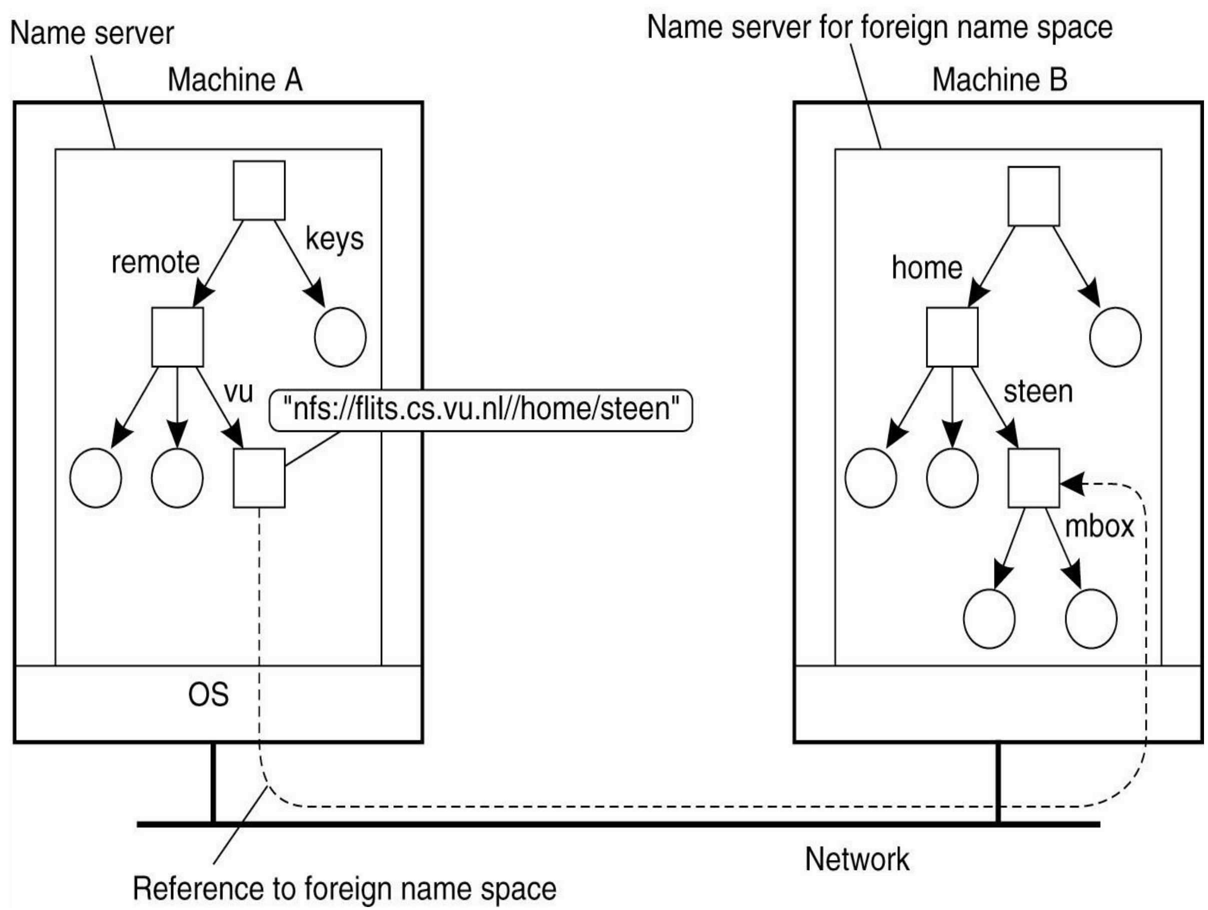
Name Resolution: linking and mounting

- Different name spaces can be merged in a transparent way using mounted file system, which corresponds to letting a directory node store the identifier of a directory node from a different namespace.
- Mounting a foreign name space requires at least the following:

  1. The name of an access protocol.

  2. The name of the server.

  3. The name of the mounting point in the foreign namespace.



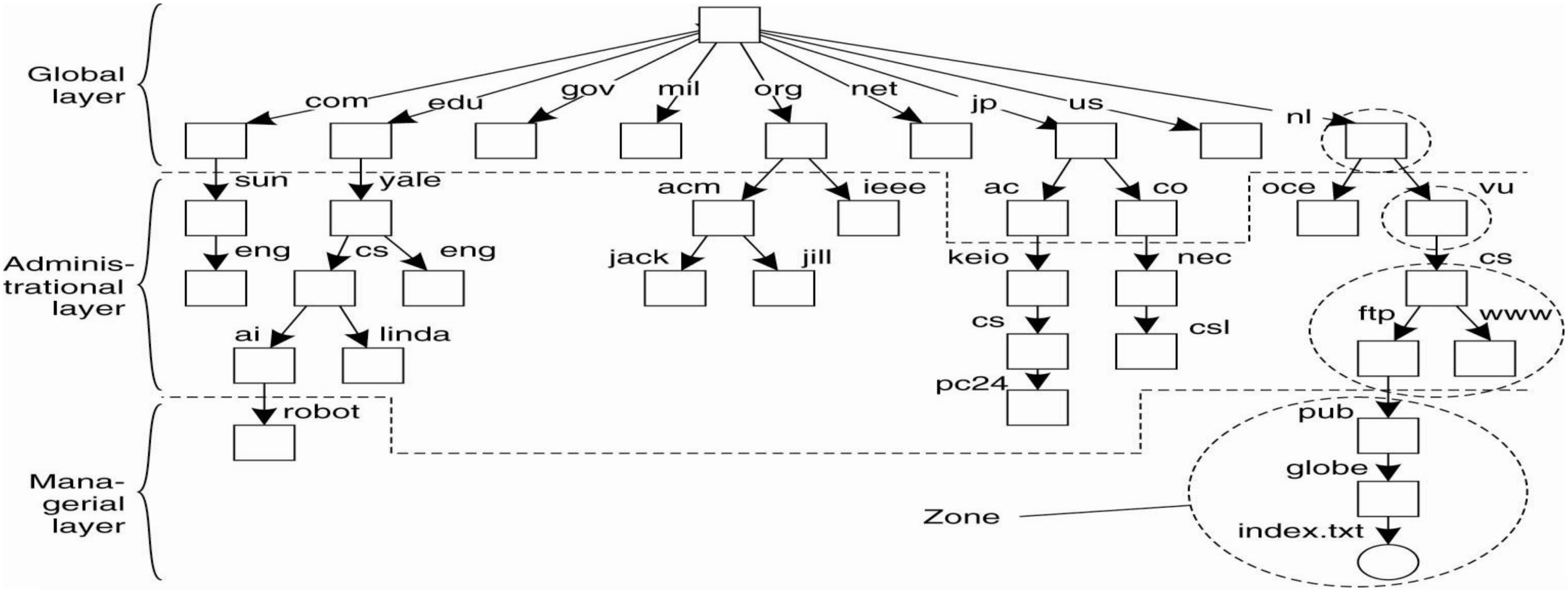
Distributed vs. centralized

NAME SPACE IMPLEMENTATION

Name Space Distribution

- **Basic issue**: Distribute the name resolution process as well as name space management across multiple machines, by distributing nodes of the naming graph
- Large name spaces are organized in a hierarchical way. There are three logical layers
  - **Global level**: Consists of the high-level directory nodes representing different organizations or groups
    - Stable (directory tables don't change often)
    - Have to be jointly managed by different administrations
  - **Administrational level**: Contains mid-level directory nodes managed within a single organization
    - Relatively stable
  - **Managerial level**: Consists of low-level directory nodes within a single administration.
    - Nodes may change often, requiring effective mapping of names
    - Managed by admins or users

Name Space Distribution (cont.)



Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

Name Space Distribution (cont.)

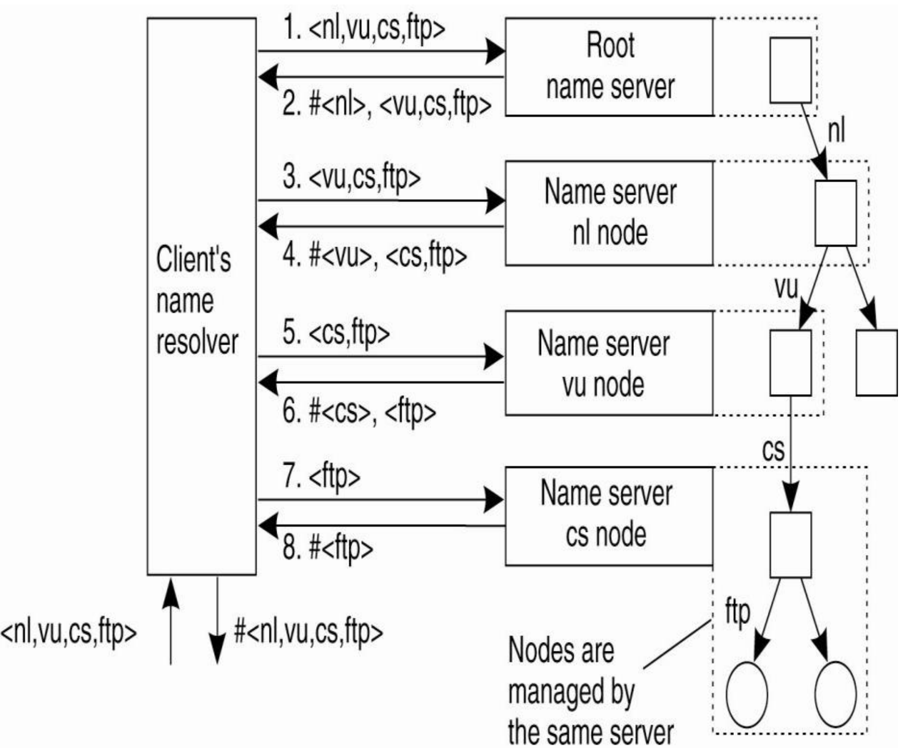
- Servers in each layer have different requirements regarding availability and performance

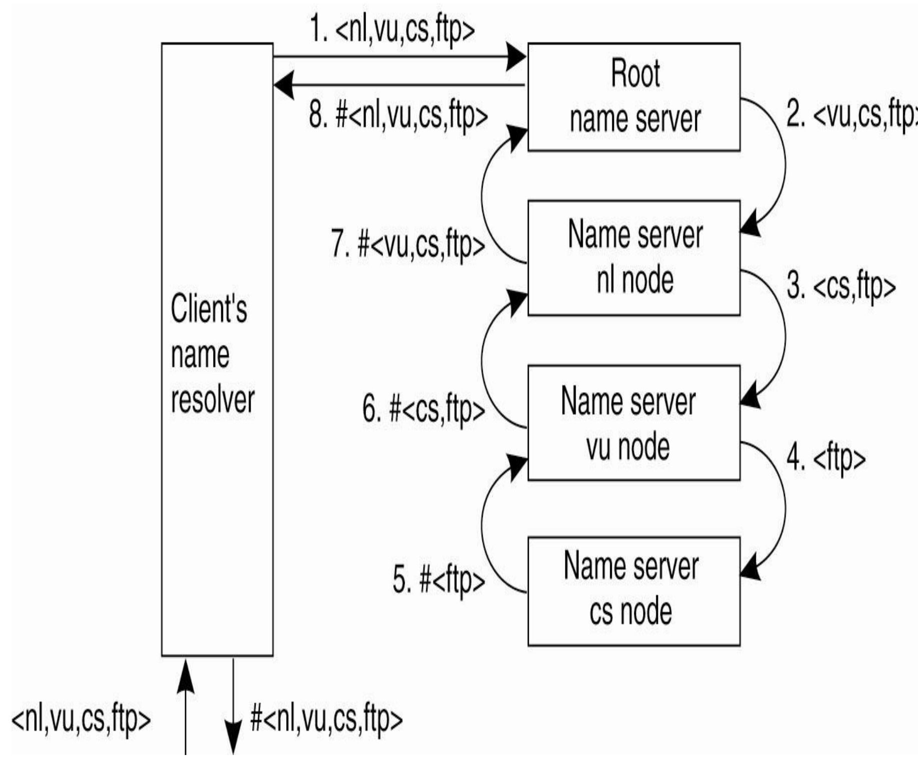
	Availability	Performance
Global	Must be very high Replication may help	Can be cached (stability) Replication may help
Administrative	Must be very high particularly for the clients in the same organization	Looks up should be fast Use high-end machines
Managerial	Less demanding One dedicated server might be enough	Performance is crucial Operations should take place immediately Caching would not be eff.

Item	**Global**	**Administrational**	**Managerial**
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

Implementation of Name Resolution

Iterative vs. Recursive





-Caching is restricted to client
-Communication cost, Delay
+ less overhead on root

+Caching can be more effective
+Communication cost might be reduced
- Too much overhead on root

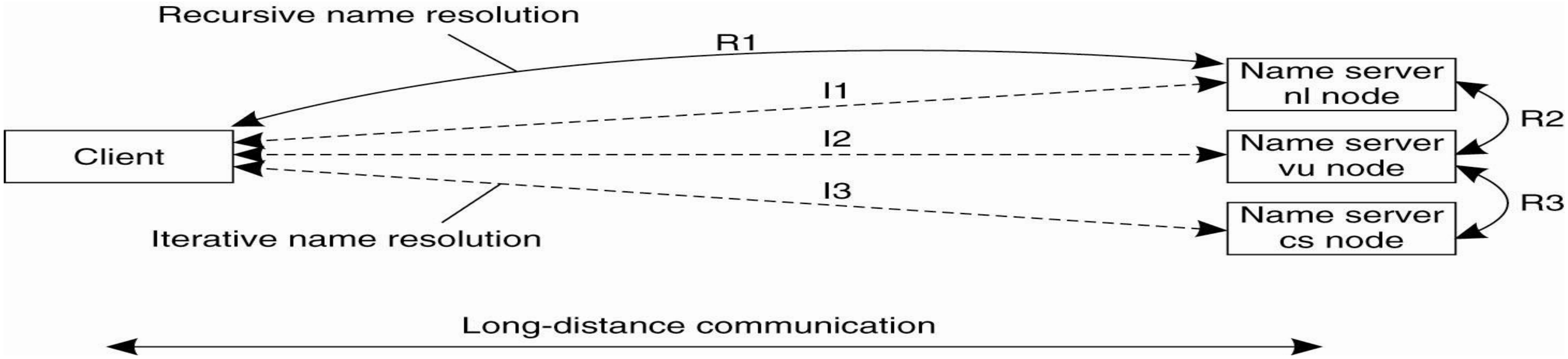
Cache in Recursive Naming Resolution

- Recursive name resolution of <nl, vu, cs, ftp>.
- Name servers cache intermediate results for subsequent lookups

Server for node	Should resolve	Looks up	Passes to child	Receives and caches	Returns to requester
cs	<ftp>	#<ftp>	—	—	#<ftp>
vu	<cs,ftp>	#<cs>	<ftp>	#<ftp>	#<cs> #<cs, ftp>
nl	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<cs> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
root	<nl,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>	#<nl> #<nl,vu> #<nl,vu,cs> #<nl,vu,cs,ftp>

Scalability Issues

- **Size scalability:** We need to ensure that servers can handle a large number of requests per time unit → high-level servers are in big trouble.
  - **Solution:** Assume (at least at global and administrational level) that content of nodes **hardly ever changes**. In that case, we can apply extensive **replication** by mapping nodes to multiple servers, and start name resolution at the nearest server.
- **Geographical scalability:** We need to ensure that the name resolution process scales across large geographical distances.



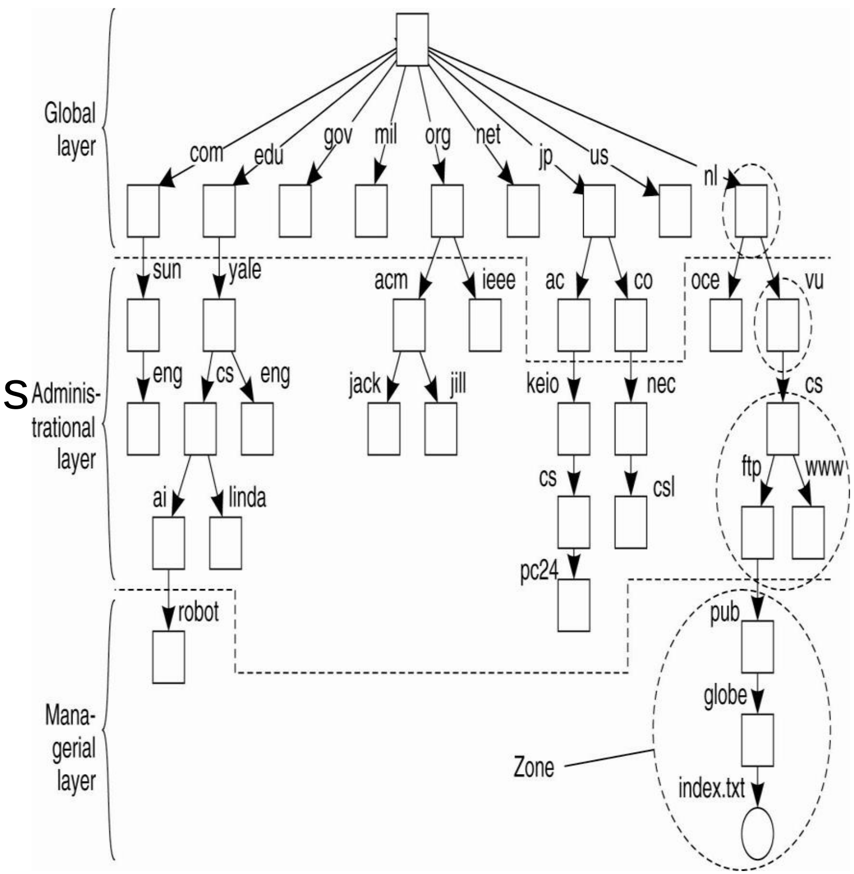
How to map

Names (www.cs.utsa.edu) to IP addresses (129.115.28.4)

CASE STUDY: DOMAIN NAME SYSTEM (DNS)

Case Study: Domain Name System (DNS)

- One of the largest distributed naming database/service
- The DNS name space is hierarchically organized as a rooted tree. Name structure reflects administrative structure of the Internet
- Rapidly resolves domain names to IP addresses
  - exploits caching heavily
  - typical query time ~100 milliseconds
- Scales to millions of computers
  - partitioned database
  - caching
- Resilient to failure of a server
  - replication



Domain names (last element of name)

- com - commercial organizations
- edu - universities and educational institutions
- gov - US government agencies
- mil - US military organizations
- net - major network support centers
- org - organizations not included in first five
- int - international organization
- country codes - (e.g., cn, us, uk, fr, etc.)

Name spaces in DNS

- Hierarchical structure - one or more components or labels separated by periods (.)
- Only absolute names - referred relative to global root
- Clients usually have a list of default domains that are appended to single-component domain names before trying global root
- Allows aliases such as www.utsa.edu → web.cs.utsa.edu

Zone partitioning of DNS name space

- **Zone** - contains attribute data for names in domain minus the sub-domains administrated by lower-level authorities:
  - Example: UTSA has a name server for utsa.edu, but cs.utsa.edu names are resolved by the CS Dept. server
- Names of the servers for the sub-domains
- **At least two name servers that provide authoritative data for the zone**
- Zone management parameters: cache, replication

Authoritative name servers

- A server may be an authoritative source for zero or more zones
- Data for a zone is entered into a **local master file**
- Master (primary) server reads the zone data directly from the master file
- Secondary authoritative servers download zone data from primary server
- Secondary servers periodically check their version number against the master server

DNS server functions and configuration

- Main function is to resolve domain names for computers, i.e. to get their IP addresses
  - caches the results of previous searches until they pass their 'time to live'
- Other functions:
  - get ***mail host*** for a domain
  - reverse resolution - get domain name from IP address
  - Host information - type of hardware and OS
  - Well-known services - a list of services offered by a host
  - Other attributes can be included (optional)

DNS resource records

Type of record	Associated entity	Description
SOA	Zone	Holds information on the represented zone
A	Host	Contains an IP address of the host this node represents
MX	Domain	Refers to a mail server to handle mail addressed to this node
SRV	Domain	Refers to a server handling a specific service
NS	Zone	Refers to a name server that implements the represented zone
CNAME	Node	Symbolic link with the primary name of the represented node
PTR	Host	Contains the canonical name of a host
HINFO	Host	Holds information on the host this node represents
TXT	Any kind	Contains any entity-specific information considered useful

DNS resource records: Example

An excerpt from the DNS database for the zone *cs.vu.nl*.

Name	Record type	Record value
cs.vu.nl.	SOA	star.cs.vu.nl. hostmaster.cs.vu.nl. 2005092900 7200 3600 2419200 3600
cs.vu.nl.	TXT	"Vrije Universiteit - Math. & Comp. Sc."
cs.vu.nl.	MX	1 mail.few.vu.nl.
cs.vu.nl.	NS	ns.vu.nl.
cs.vu.nl.	NS	top.cs.vu.nl.
cs.vu.nl.	NS	solo.cs.vu.nl.
cs.vu.nl.	NS	star.cs.vu.nl.
star.cs.vu.nl.	A	130.37.24.6
star.cs.vu.nl.	A	192.31.231.42
star.cs.vu.nl.	MX	1 star.cs.vu.nl.
star.cs.vu.nl.	MX	666 zephyr.cs.vu.nl.
star.cs.vu.nl.	HINFO	"Sun" "Unix"
zephyr.cs.vu.nl.	A	130.37.20.10
zephyr.cs.vu.nl.	MX	1 zephyr.cs.vu.nl.
zephyr.cs.vu.nl.	MX	2 tornado.cs.vu.nl.
zephyr.cs.vu.nl.	HINFO	"Sun" "Unix"
ftp.cs.vu.nl.	CNAME	soling.cs.vu.nl.
www.cs.vu.nl.	CNAME	soling.cs.vu.nl.
soling.cs.vu.nl.	A	130.37.20.20
soling.cs.vu.nl.	MX	1 soling.cs.vu.nl.
soling.cs.vu.nl.	MX	666 zephyr.cs.vu.nl.
soling.cs.vu.nl.	HINFO	"Sun" "Unix"
vucs-das1.cs.vu.nl.	PTR	0.198.37.130.in-addr.arpa.
vucs-das1.cs.vu.nl.	A	130.37.198.0
inkt.cs.vu.nl.	HINFO	"OCE" "Proprietary"
inkt.cs.vu.nl.	A	192.168.4.3
pen.cs.vu.nl.	HINFO	"OCE" "Proprietary"
pen.cs.vu.nl.	A	192.168.4.2
localhost.cs.vu.nl.	A	127.0.0.1

Caching in DNS

- *A*ny server can cache any name
- Non-authoritative servers note **time-to-live** when they cache data
- Non-authoritative servers indicate that they are such when responding to clients with cached names

DNS clients (resolvers)

- Resolvers are usually implemented as library routines (e.g., `gethostbyname`).
- The request is formatted into a DNS record.
- DNS servers use a well-known port.
- A request-reply protocol is used
  - TCP or UDP why?
- The resolver times out and resends if it doesn't receive a response in a specified time.

DNS name resolution

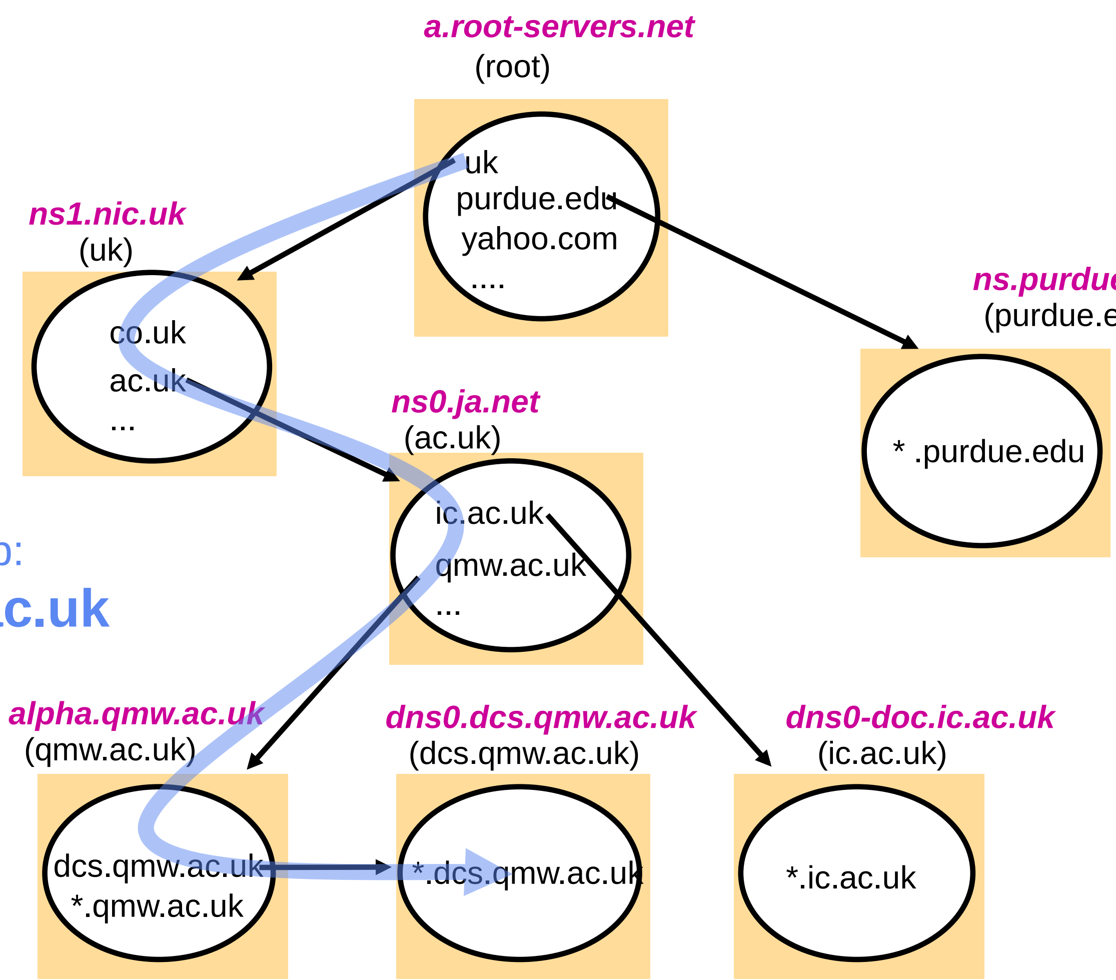
- Domain name → IP address ???
- Look for the name in the local **cache**
- Try a superior DNS server, which responds with:
  - the IP address (which may not be entirely up to date)
  - Or, another recommended DNS server (iterative)

DNS name servers

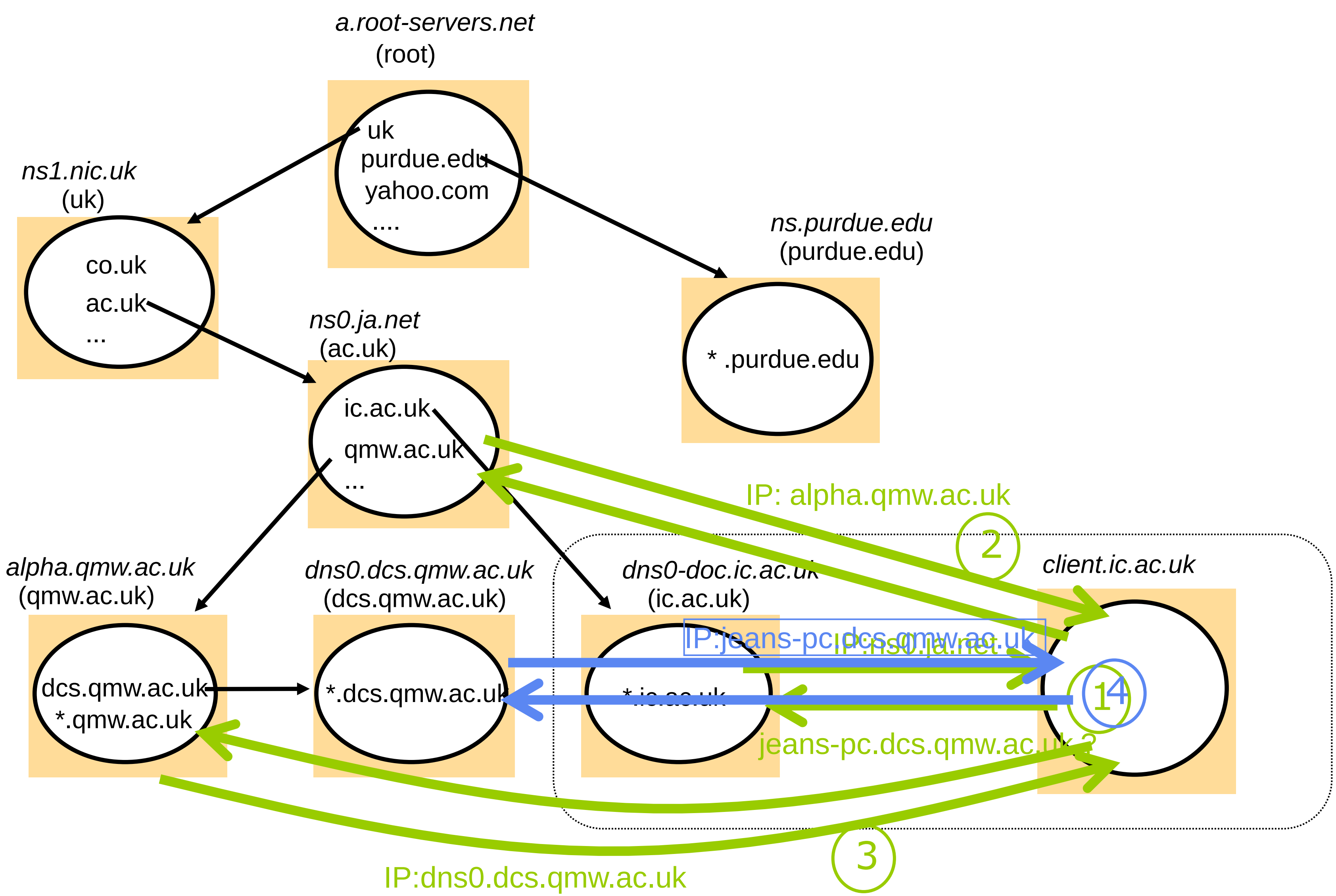
Note: Name server names are in italics, and the corresponding domains are in parentheses. Arrows denote name server entries

authoritative path to lookup:

jeans-pc.dcs.qmw.ac.uk



DNS in typical operation



DNS issues

- Name tables change infrequently, but when they do, caching can result in the delivery of stale data.
  - Clients are responsible for detecting this and recovering
- Its design makes changes to the structure of the name space difficult. For example:
  - merging previously separate domain trees under a new root
  - moving sub-trees to a different part of the structure

Example: Decentralized DNS

Basic idea: Take a full DNS name, hash into a key k, and use a DHT-based system to allow for key lookups.

- + scalability
- - we loose the structure of the original DNS name so we may not efficiently find all nodes in a subdomain (but very few people were doing this anyway).

In many cases, it is much more convenient to name, and look up entities by means of their **attributes** (e.g., look for a student who got A in OS)

ATTRIBUTE-BASED NAMING

ALSO KNOWN AS DIRECTORY SERVICES

Directory Services

- Entities have a set of attributes (e.g., email: send, recv, subject, ...)
- In most cases, attributes are determined manually
- Setting values consistently is a crucial problem ...
- Often organized in a hierarchy
  - Examples of directory services: X.500, Microsoft's Active Directory Services,
- Then, **look up** entities by means of their **attributes**
- **Problem:** Lookup operations can be extremely expensive, as they require to match requested attribute values, against actual attribute values
  - In the simplest form, **inspect all entities**.

Directory Services (cont'd)

Solutions:

- Lightweight Directory Access Protocol (LDAP):
  - Implement **basic directory service as database**, and Combine it with traditional **structured naming** system.
  - Derived from OSI's X.500 directory service, which maps a person's name to attributes (email address, etc.)
- DHT-based decentralized implementation

Hierarchical implementation: LDAP (1)

- LDAP directory service consists of a set of records
- Each directory entry (record) is made up of a set of (Attribute, Value(s)) pairs
- Collection of all directory entries is called Directory Information Base (DIB)

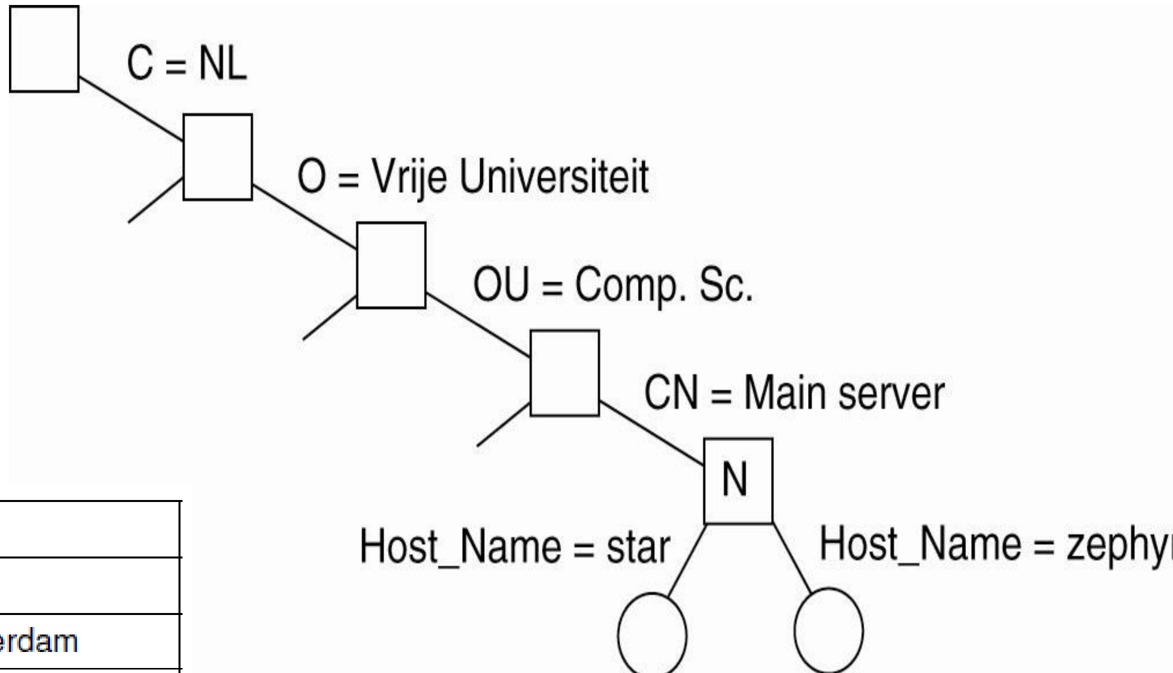
Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
OrganizationalUnit	OU	Comp. Sc.
CommonName	CN	Main server
Mail_Servers	—	137.37.20.3, 130.37.24.6, 137.37.20.10
FTP_Server	—	130.37.20.20
WWW_Server	—	130.37.20.20

- Each record is **uniquely** named by using naming attributes in the record *(e.g., first five in the above record)*
- Each naming attribute is called relative distinguished name (RDN)

Hierarchical implementation: LDAP (2)

- We can create a directory information tree (DIT) by listing RDNs in sequence

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	137.37.20.10

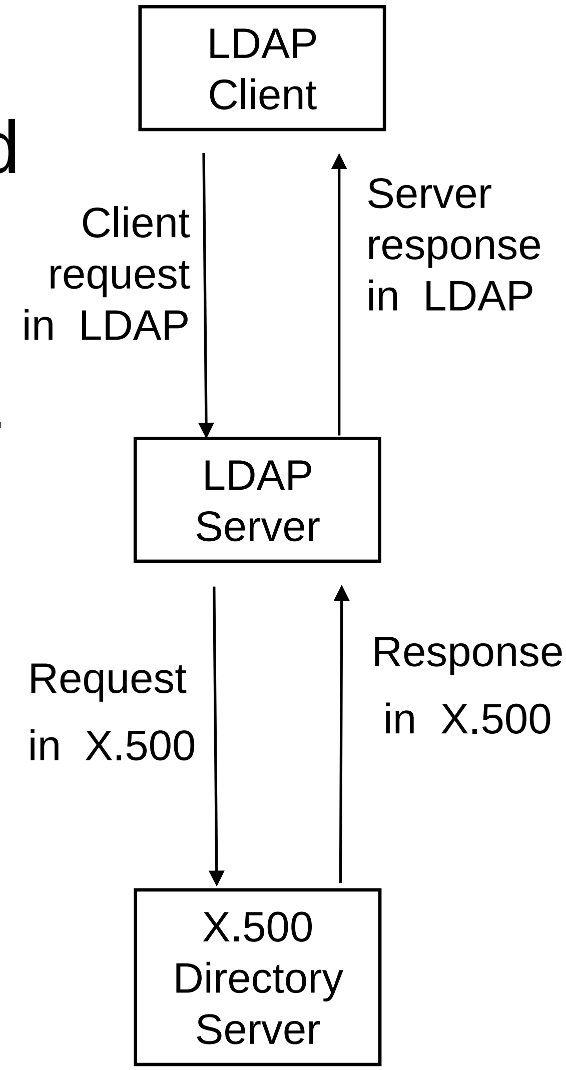


Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

answer =
search("&(C = NL) (O = Vrije Universiteit) (OU = *) (CN = Main server)")

Hierarchical implementation: LDAP (3)

- Clients called Directory User Agent (DUA), similar to name resolver and contacts the server
- LDAP server known as Directory Service Agent (DSA) maintains DIT and looks up entries based on attr.
- In case of a large scale directory, DIT is partitioned and distribute across several DSAs
- Implementation of LDAP is similar to DNS, but LDAP provides more advanced lookup operations



Hierarchical implementation: LDAP (4)

- Simple DUA interface to X.500 (see extra slides)
- LDAP runs over TCP/IP
- Uses textual encoding
- Provides secure access through authentication
- Other directory services have implemented it
- See RFC 2251 [Wahl et al. 1997]

LDAP Evolution

- University of Michigan added to LDAP servers the capability of accessing own database.
- Use of LDAP databases became widespread
- Schemes were developed for registering changes and exchanging deltas between LDAP servers
- In 1996 three engineers from U of Michigan joined Netscape. 40 companies (w/o Microsoft) announced support of LDAP as the standard for directory services
- Core specifications for LDAPv3 was published as IETF RFCs 2251-2256.

DHT-based Decentralized Implementation

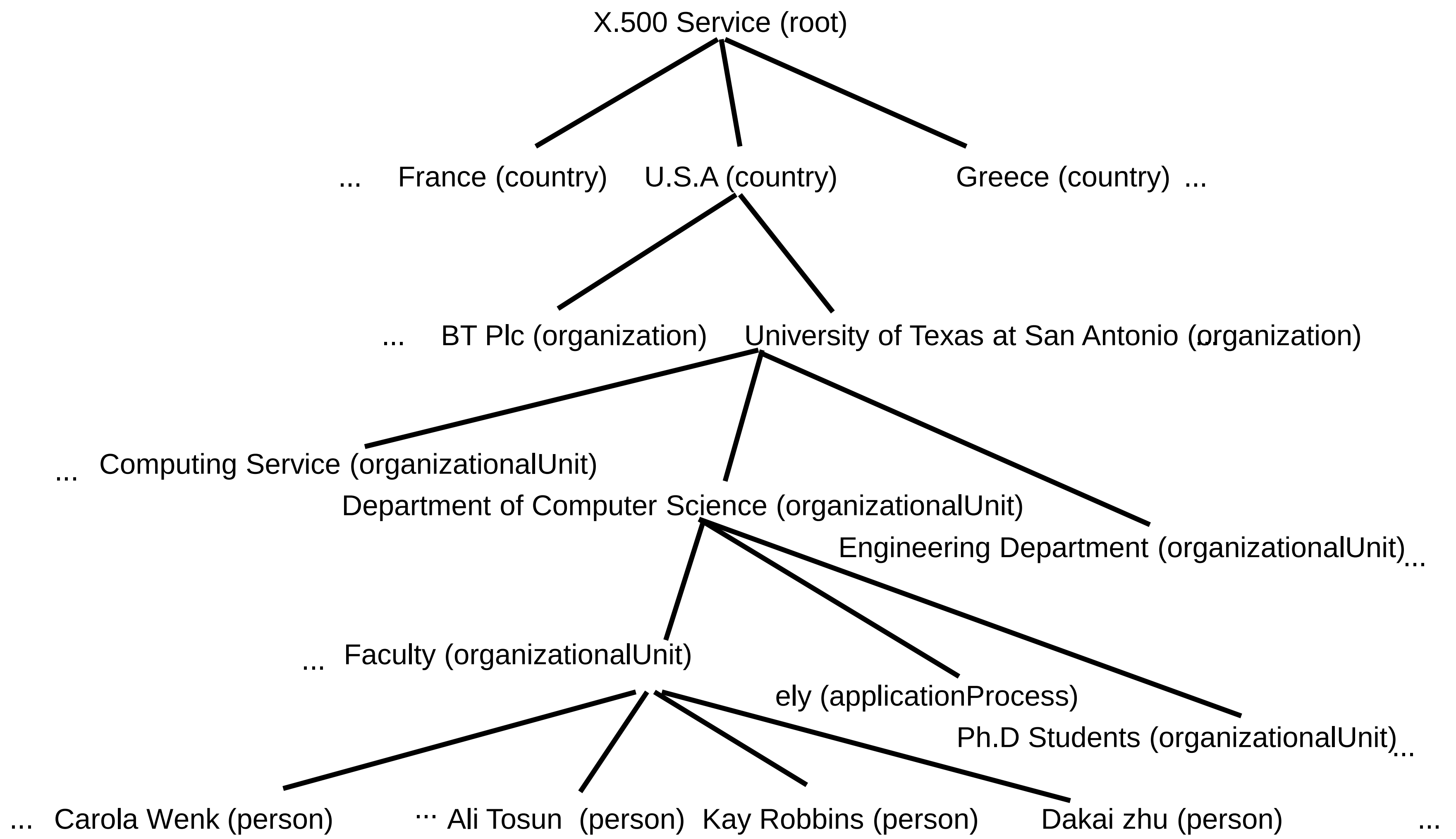
- How to map (Attribute, value(s)) pairs to nodes so that searching can be done efficiently
- Self-study

EXTRAS

Terms in X.500 directory service

- Standardized by ITU and ISO
- Specified as an application-level in OSI
- Data is organized in tree (**DIT** = directory information tree) with named nodes (**DIB**=directory information base)
- A DIB entry has a name and a set of attributes.
- The full name is the fully-qualified path
- A client (**DUA** = directory user agent) can query any server (**DSA** = directory service agent)
- Server will respond, query other services, or send back server response

Part of the X.500 Directory Information Tree



An X.500 DIB Entry

info	
Dakai Zhu, Faculty, Department of Computer Science, University of Texas at San Antonio	
commonName	*uid*
Dakai Zhu D. Zhu	dzhu
	email
	dzhu@cs.utsa.edu
surname	zhudakai@gmail.edu
Zhu	*roomNumber*
	SB 4.01.18
telephoneNumber	*userClass*
+1 210 458 7453	Assistant Professor

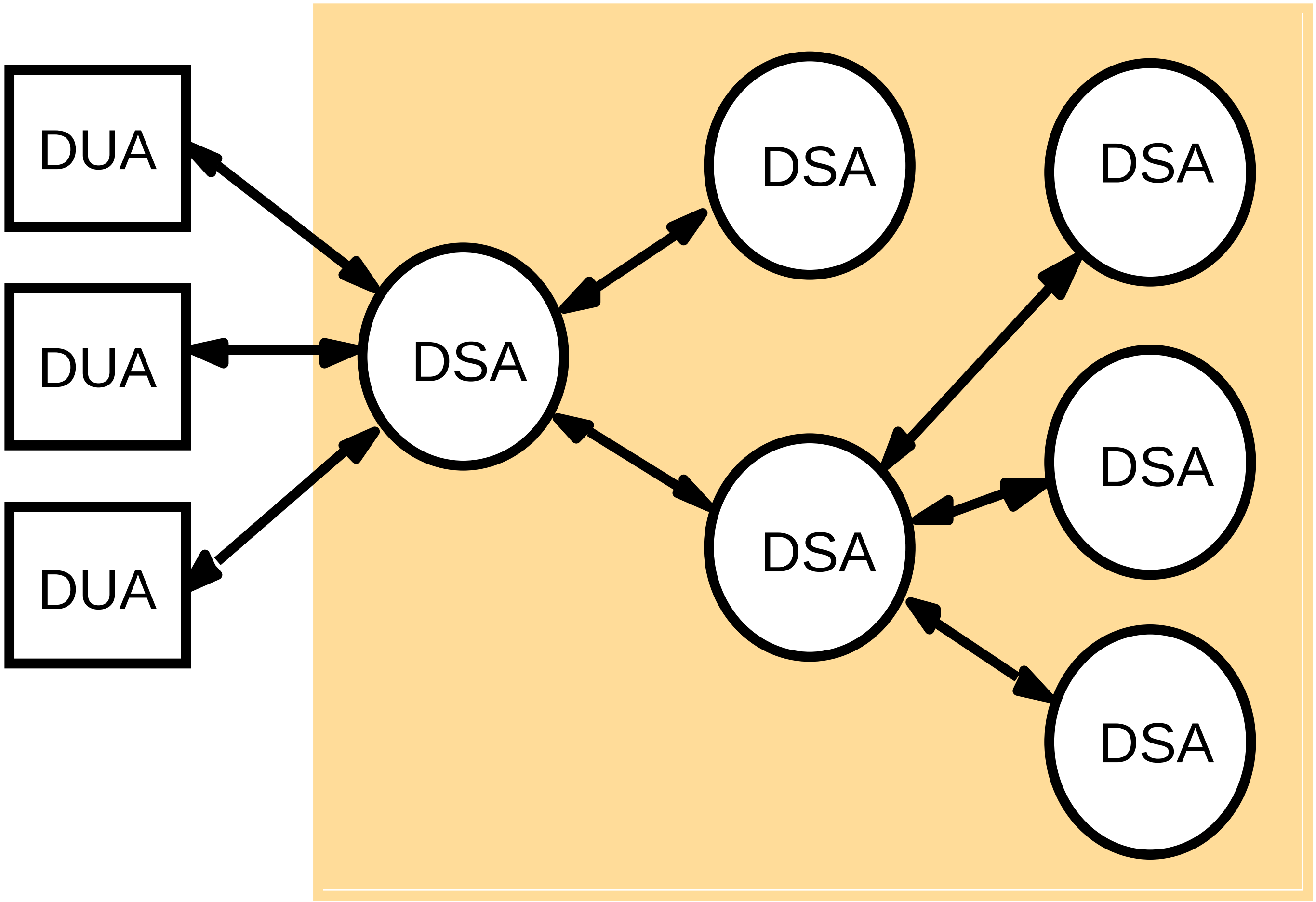
X.500 data Structures

- Attributes are **typed** (e.g. `countryName`, `commonName`)
- DIB entries are organized like OO classes.
- DIB entries have a class-name.
- The definition of a class determines which attributes are **mandatory** and which are **optional**.
- Mandatory and optional attributes are **inherited**.

X.500 Operations

- Read – locates attributes associated with given name
- Search – finds records based on attributes and filters
- DSA also has operations for adding, deleting and modifying entries

X.500 service architecture



X.500 Directory Service

- Directory Administrative Model – how global DIT is managed and split into domains
- Directory Server Protocol (DSP) – used to chain user requests between directory servers
- Directory Information Shading Protocol (DISP) – protocol for directory replication
- DOP protocol to automate connection agreements between servers between and across management domains

X.500 History and Deployment

- Specification was first released in 1988 with a significant update in 1993
- **Heavyweight** system:
  - Full OSI protocol stack that wasn't supported by MACs or PCs in the early 90's when X.500 was being deployed
  - The DUAs (directory user agents) also could not be run on PCs or MACs in the early 90's
- University of Michigan responded with the development of a DUA that understood a lightweight access protocol called **LDAP**