Naming in Distributed Systems

- Concepts
- Flat Naming
- Structured Naming
- Attribute-based Naming

NAMING

- NAMES, IDENTIFIERS, AND ADDRESSES
- FLAT NAMING
  - Simple Solutions
  - Home-Based Approaches
  - Distributed Hash Tables (More in P2P)
  - 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

Objectives

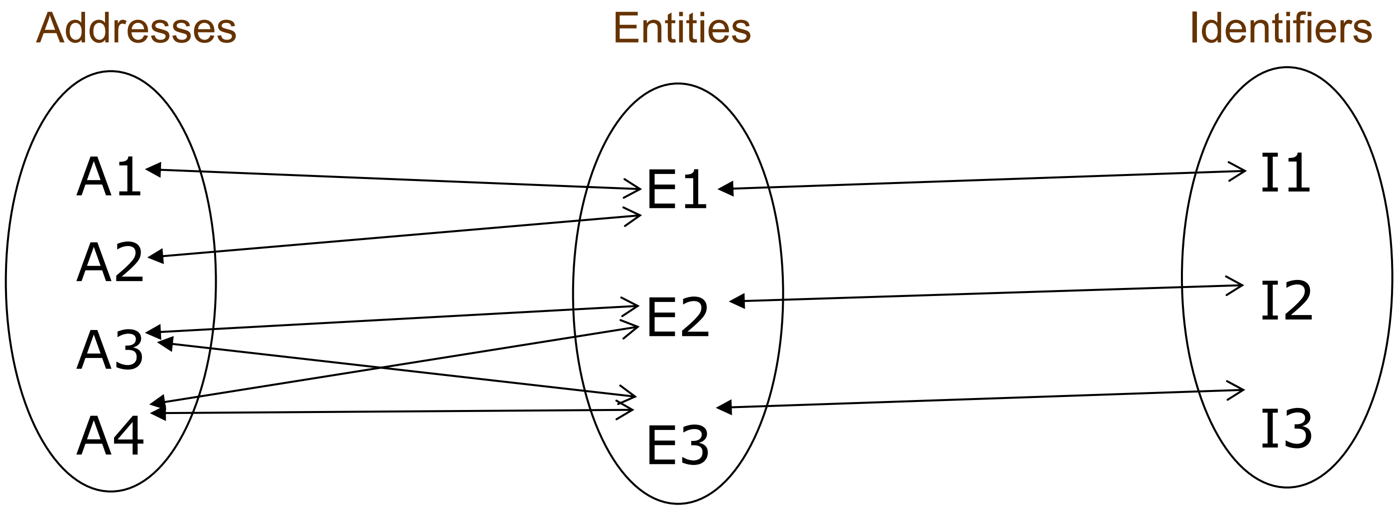
- To understand naming and related issues in DS
- To learn naming space and implementation
- To learn flat and structured names and how they are resolved
- To learn Attributed-based naming

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, 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

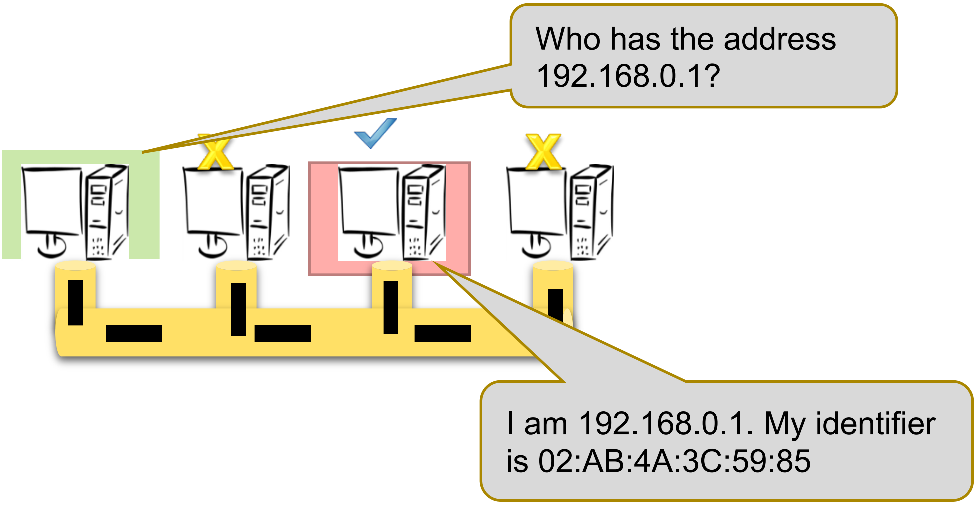
Flat Naming

- **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



Who has the address 192.168.0.1?

I am 192.168.0.1. My identifier is 02:AB:4A:3C:59:85

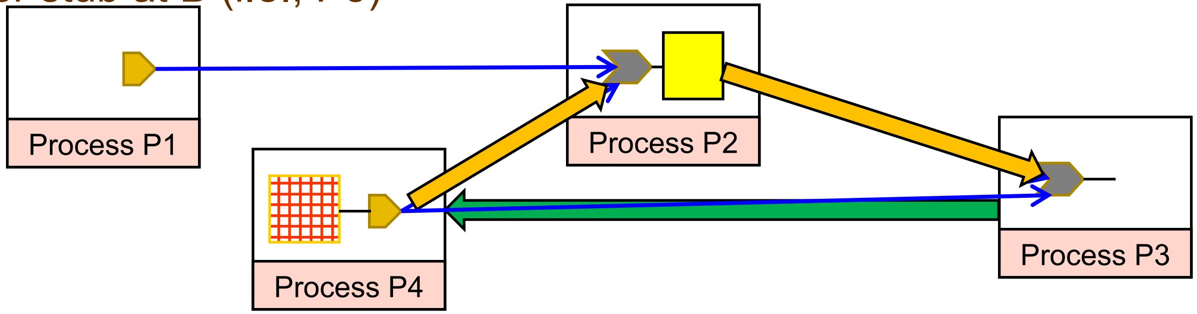
- Adv/Disadvantages
  - + simple
  - - not scale beyond LANs
  - - it requires all entities to listen to all incoming requests

Forwarding Pointers

- Stub-Scion Pair (SSP) chains implement remote invocations for mobile entities using *forwarding pointers*
  - Server stub is referred to as *Scion* in the original paper
- Each forwarding pointer is implemented as a pair:

```
(client stub, server stub)
```

  - The server stub contains a local reference to the actual object or a local reference to another client stub
- When object moves from A (e.g., P2) to B (e.g., P3),
  - It leaves a client stub at A (i.e., P2)
  - It installs a server stub at B (i.e., P3)



n = Process n; 🟨 = Remote Object; ▦ = Caller Object; ❯ = Server stub; ▷ = Client stub

Forwarding Pointers

How to locate mobile entities?

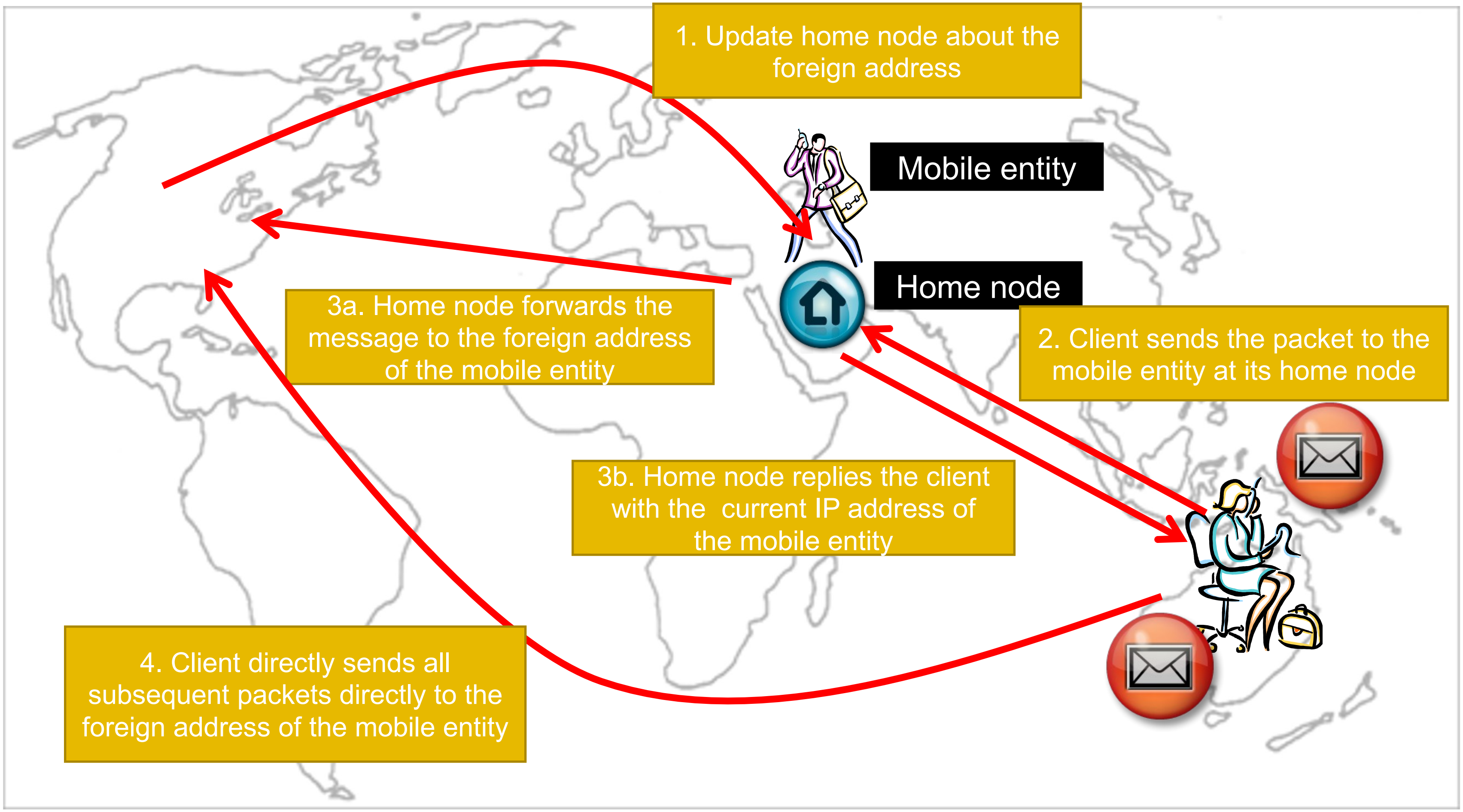
- When an entity moves from A to B, leaves a pointer to A that it is at B now…
- Dereferencing: simply follow the chain of pointers and make this entirely transparent to clients
- Adv/Disadvantages
  - + support for mobile nodes
  - - geographical scalability problems
  - - long chains are not fault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

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

Home-Based Approaches: An Example



1. Update home node about the foreign address
Mobile entity
Home node
2. Client sends the packet to the mobile entity at its home node
3a. Home node forwards the message to the foreign address of the mobile entity
3b. Home node replies the client with the current IP address of the mobile entity
4. Client directly sends all subsequent packets directly to the foreign address of the mobile entity

From CS15-440 CMU Qatar, Hammoud

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)

Next: Distributed Hash Table

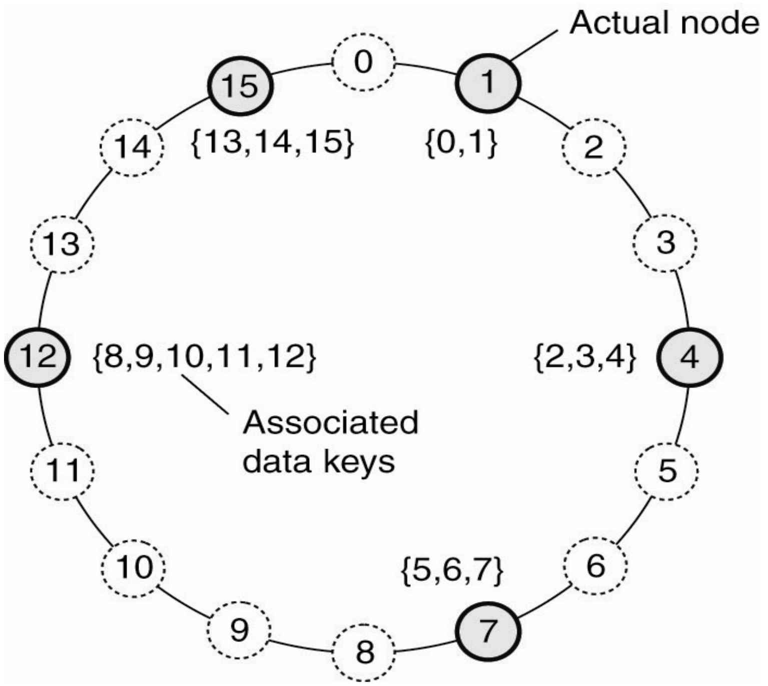
- Distributed Hash Table – In a nutshell

https://www.youtube.com/watch?v=tz-Q-eW8FbQ

Distributed Hash Tables

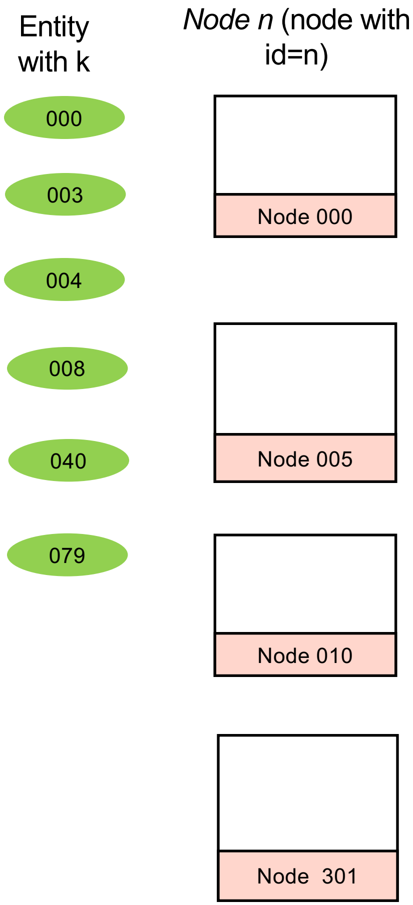
How to use DHT to resolve flat ID

- 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)



Chord

- Chord assigns an *m-bit identifier* (randomly chosen) to each node
  - A node can be contacted through its network address
- Alongside, it maps each entity to a node
  - Entities can be processes, files, etc.,
- Mapping of entities to nodes
  - Each node is responsible for a set of entities
  - An entity with key `k` falls under the jurisdiction of the node with the smallest identifier `id >= k`. This node is known as the *successor of* `k`, and is denoted by `succ(k)`



Entity with k

Node n (node with id=n)

000
003
004
008
040
079

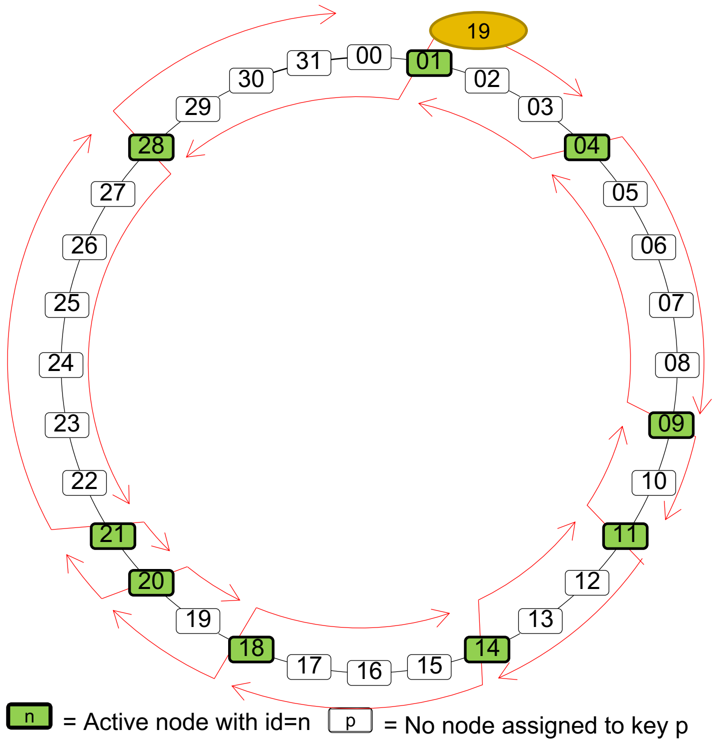
Node 000
Node 005
Node 010
Node 301

Map each entity with key `k` to node `succ(k)`

From CS15-440 CMU Qatar, Hammoud

A Naïve Key Resolution Algorithm - Linearly

- The main issue in DHT is to efficiently resolve a key *k* to the network location of `succ(k)`
  - *Given an entity with key `k`, how to find the node `succ(k)`?*



n = Active node with id=n   p = No node assigned to key p

1. All nodes are arranged in a logical ring according to their IDs
2. Each node ‘`p`’ keeps track of its immediate neighbors: `succ(p)` and `pred(p)`
3. If ‘`p`’ receives a request to resolve key ‘`k`’:
   - If `pred(p) < k <=p`, node `p` will handle it
   - Else it will forward it to `succ(n)` or `pred(n)`

Solution is not scalable:
- As the network grows, forwarding delays increase
  - Key resolution has a time complexity of `O(n)`

From CS15-440 CMU Qatar, Hammoud

DHT: Finger Table (Chord)
How to improve efficiency?

- Chord improves key resolution by reducing the time complexity to O(log n)
- All nodes are arranged in a logical ring according to their IDs
- Each node 'p' keeps a table FT_p of at-most *m* entries. This table is **Finger Table**

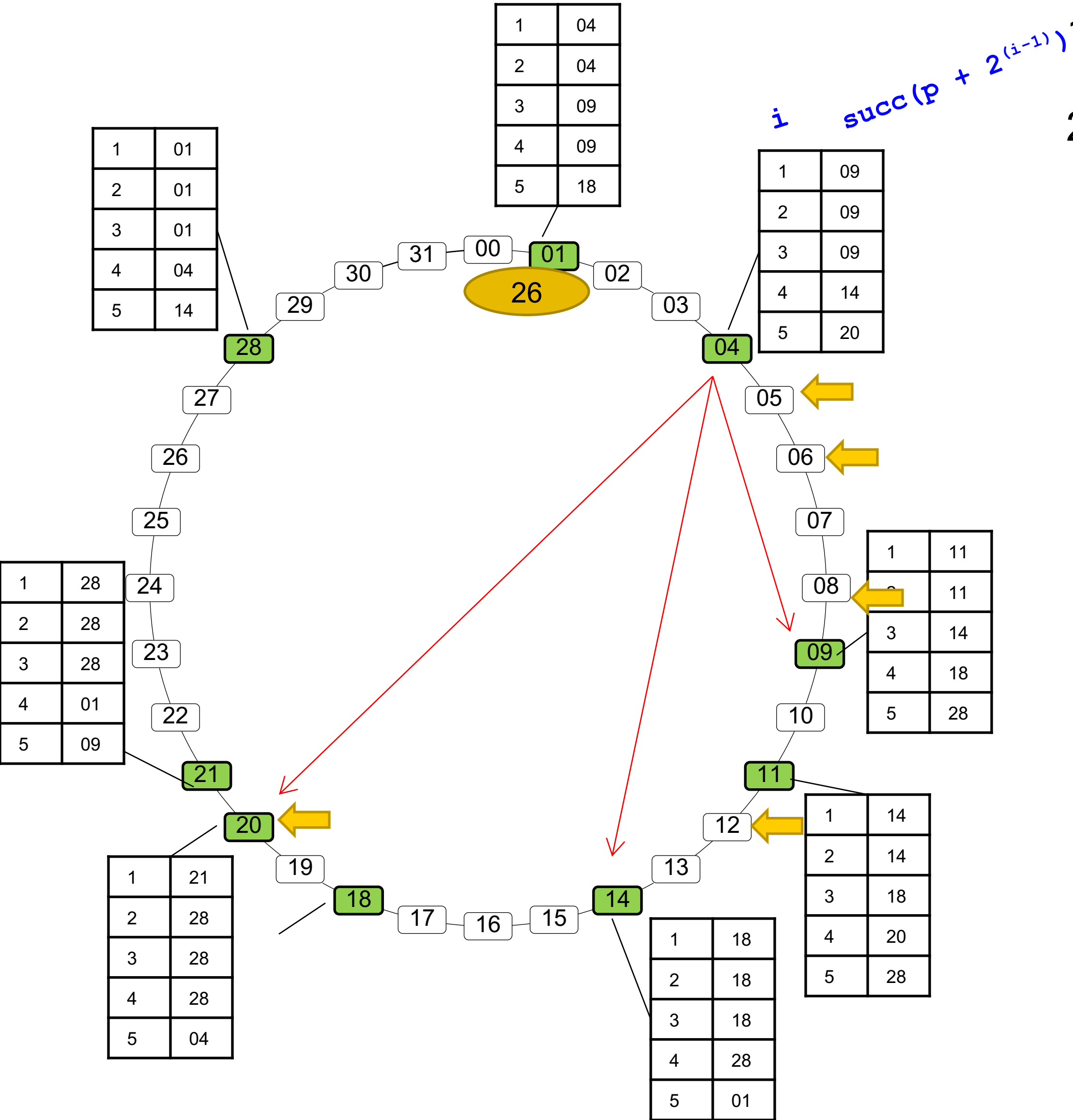
  $FT_p[i] = succ(p + 2^{(i-1)})$

  NOTE: $FT_p[i]$ increases exponentially
- If node 'p' receives a request to resolve key 'k':
  - Node p will forward it to node q with index j in F_p where

    $q = FT_p[j] <= k < FT_p[j+1]$

    - If $k > FT_p[m]$, then node p will forward it to $FT_p[m]$
    - If $k < FT_p[1]$, then node p will forward it to $FT_p[1]$

Chord DHT Example



1	04
2	04
3	09
4	09
5	18

1	01
2	01
3	01
4	04
5	14

i	succ(p + 2(i-1))
1	09
2	09
3	09
4	14
5	20

1	11
2	11
3	14
4	18
5	28

1	28
2	28
3	28
4	01
5	09

1	14
2	14
3	18
4	20
5	28

1	21
2	28
3	28
4	28
5	04

1	18
2	18
3	18
4	28
5	01

1. All nodes are arranged in a logical ring according to their IDs
2. Each node ‘p’ keeps a table FT_p of at-most *m* entries. This table is called Finger Table

   $$FT_p[i] = succ(p + 2^{(i-1)})$$

   NOTE: $FT_p[i]$ increases exponentially

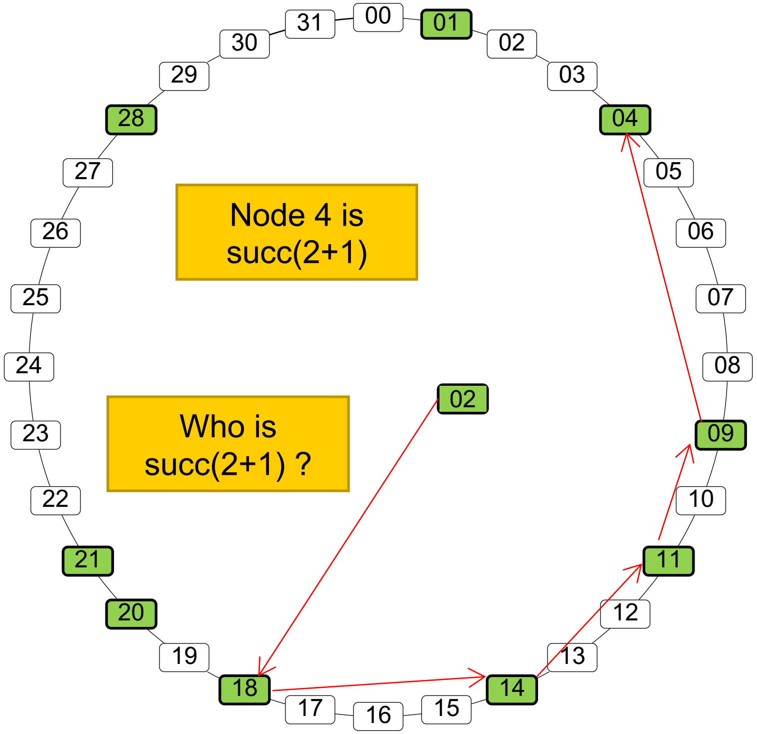
3. If node ‘p’ receives a request to resolve key ‘k’:
   - Node p will forward it to node q with index j in F_p where

     $$q = FT_p[j] <= k < FT_p[j+1]$$

     - If $k > FT_p[m]$, then node p will forward it to $FT_p[m]$
     - If $k < FT_p[1]$, then node p will forward it to $FT_p[1]$

DHT: Finger Table (cont'd)

- How to handle
  - Join
    - It contacts arbitrary node, looks up for `succ(p+1)`, and inserts itself into the ring
  - Leave
    - It contacts `pred(p)` and `succ(p+1)` and updates them
  - Fail



DHT: Finger Table (cont'd)

- 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 = $[p+2^{i-1}, p+2^{i}-1]$.

- Proximity neighbor selection:

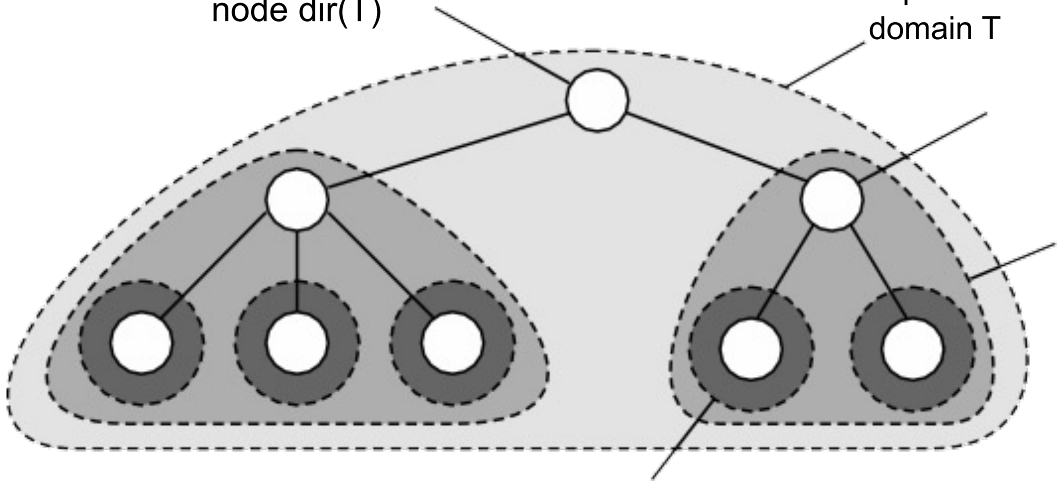
  When there is a choice of selecting who your neighbor will be (not in Chord), pick the closest one.

Hierarchical Location Services (HLS)

Basic idea

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.

Principle



The root directory node dir(T)

Top-level domain T

Directory node dir(S) of domain S

A subdomain S of top-level domain T (S is contained in T)

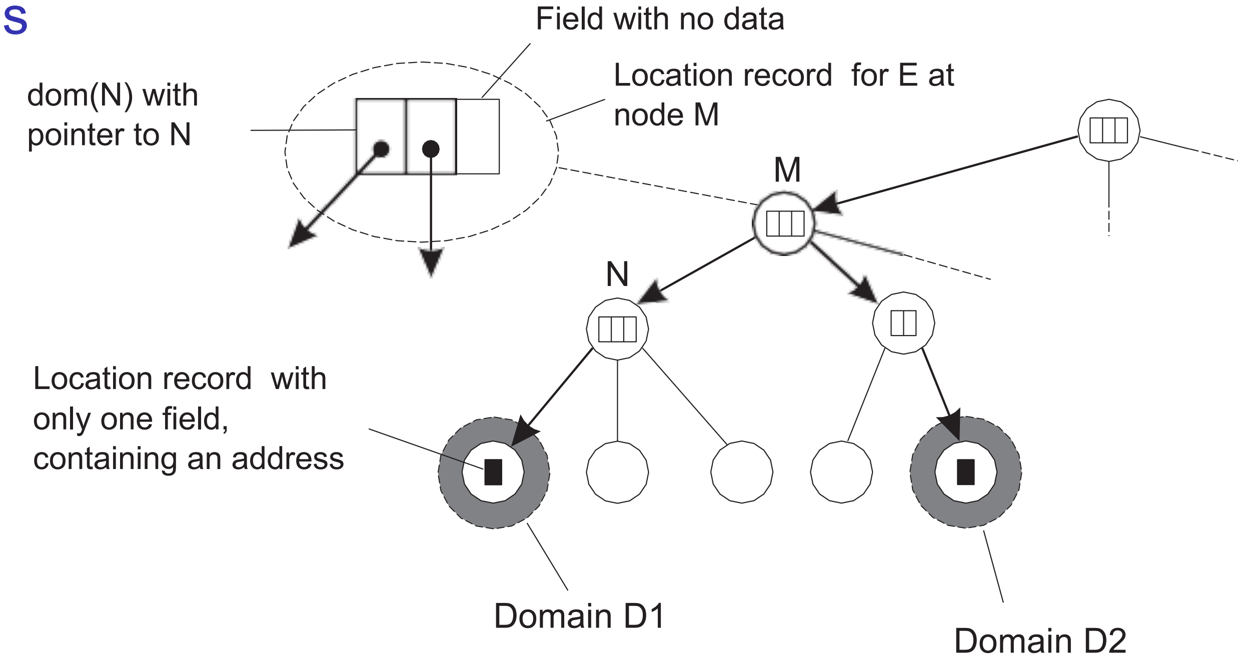
A leaf domain, contained in S

HLS: Tree organization

Invariants

- ▶Address of entity *E* is stored in a leaf or intermediate node
- ▶Intermediate nodes contain a pointer to a child if and only if the subtree rooted at the child stores an address of the entity
- ▶The root knows about all entities

Storing information of an entity having two addresses in different leaf domains



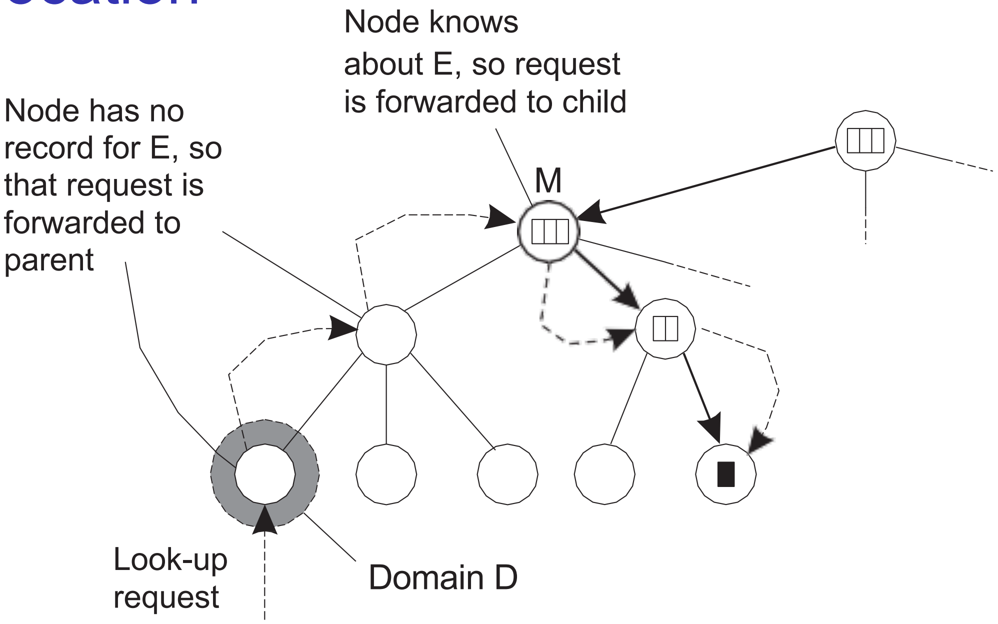
Field with no data
Location record for E at node M
dom(N) with pointer to N
M
N
Location record with only one field, containing an address
Domain D1
Domain D2

HLS: Lookup operation

Basic principles

- Start lookup at local leaf node
- Node knows about *E* ⇒ follow downward pointer, else go up
- Upward lookup always stops at root

Looking up a location

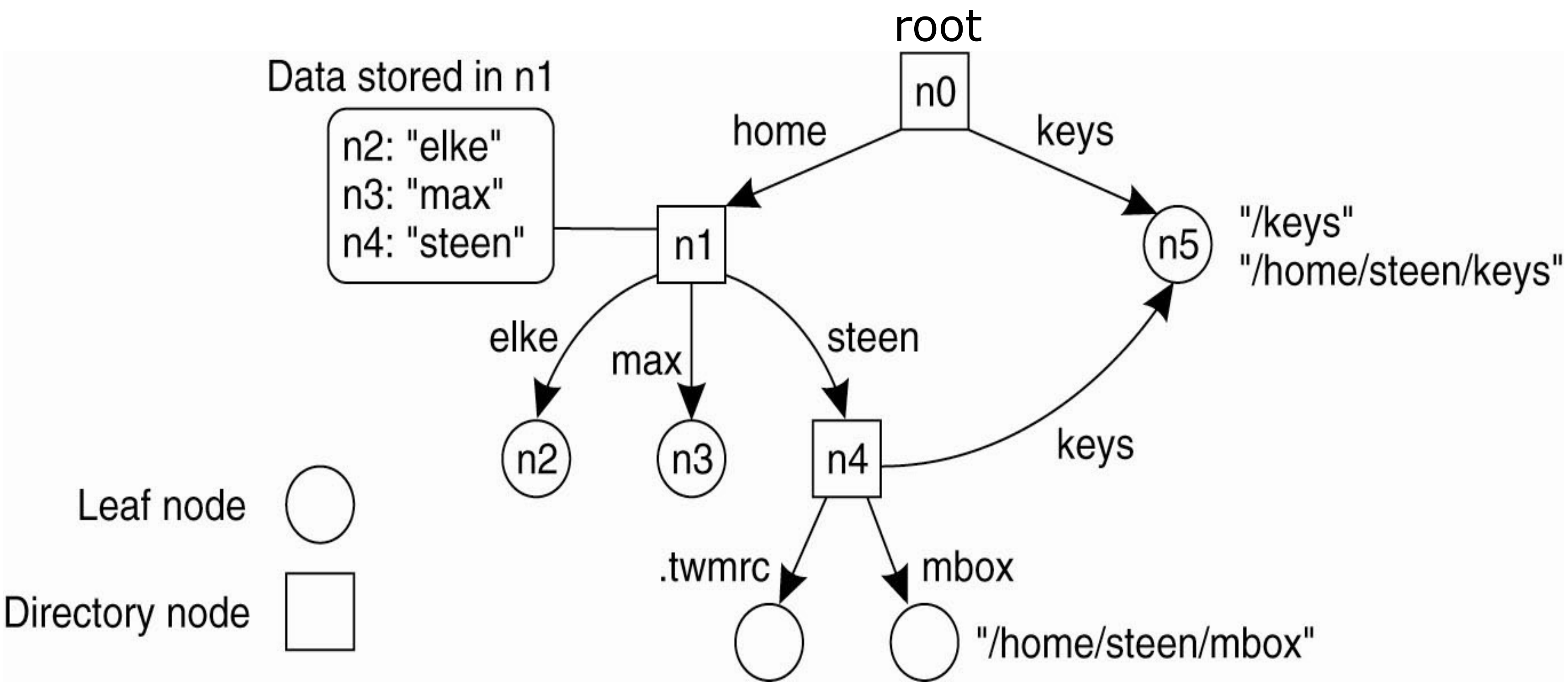

Node knows about E, so request is forwarded to child
Node has no record for E, so that request is forwarded to parent
M
Look-up request
Domain D

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

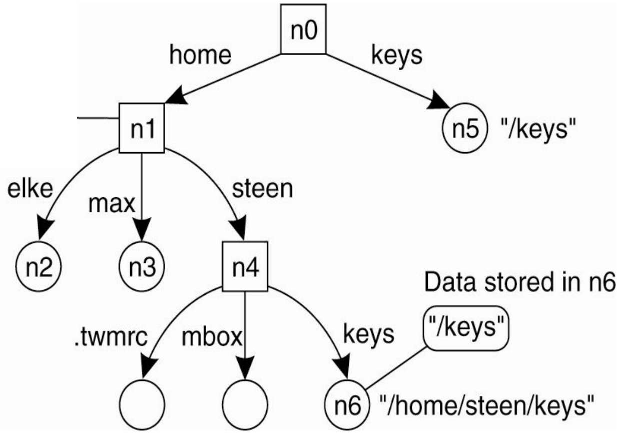
get the identifier

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 Linking

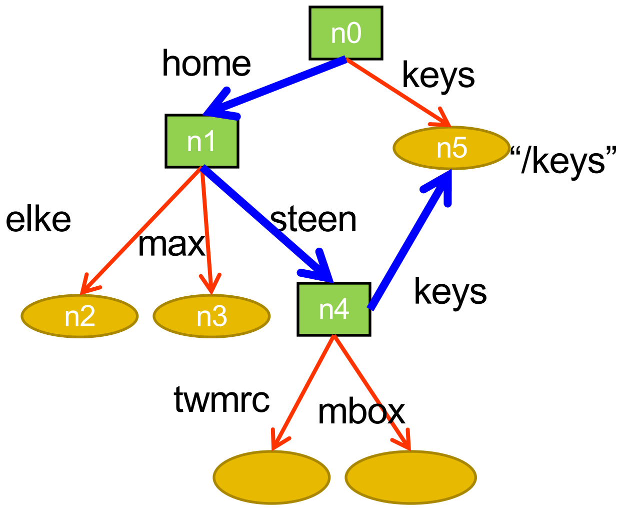
- The name space can be effectively used to link two different entities

- Two types of links can exist between the nodes:
  1. Hard Links
  2. Symbolic Links

1. Hard Links

- There is a directed link from the hard link to the actual node
- Name resolution:
  - Similar to the general name resolution
- Constraint:
  - There should be no cycles in the graph

"/home/steen/keys" is a hard link to "/keys"



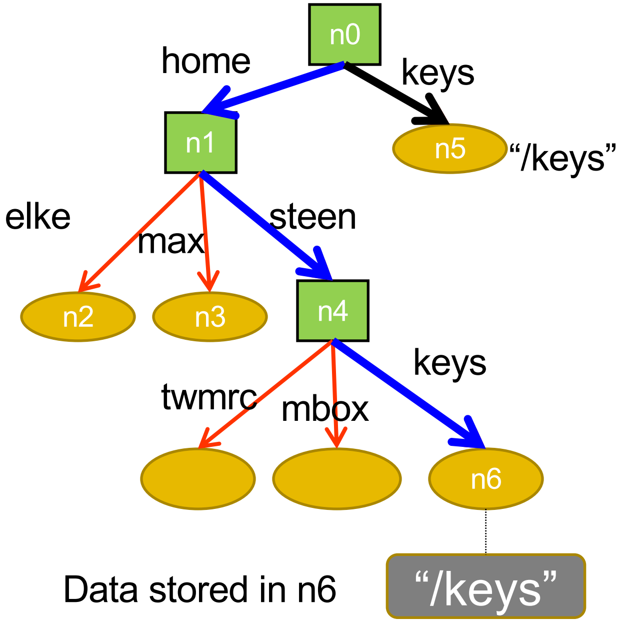
2. Symbolic Links

- Symbolic link stores the name of the original node as *data*

- Name resolution for a symbolic link SL
  - First resolve SL's name
  - Read the content of SL
  - Name resolution continues with content of SL

- Constraint:
  - No cyclic references should be present

"/home/steen/keys" is a symbolic link to "/keys"



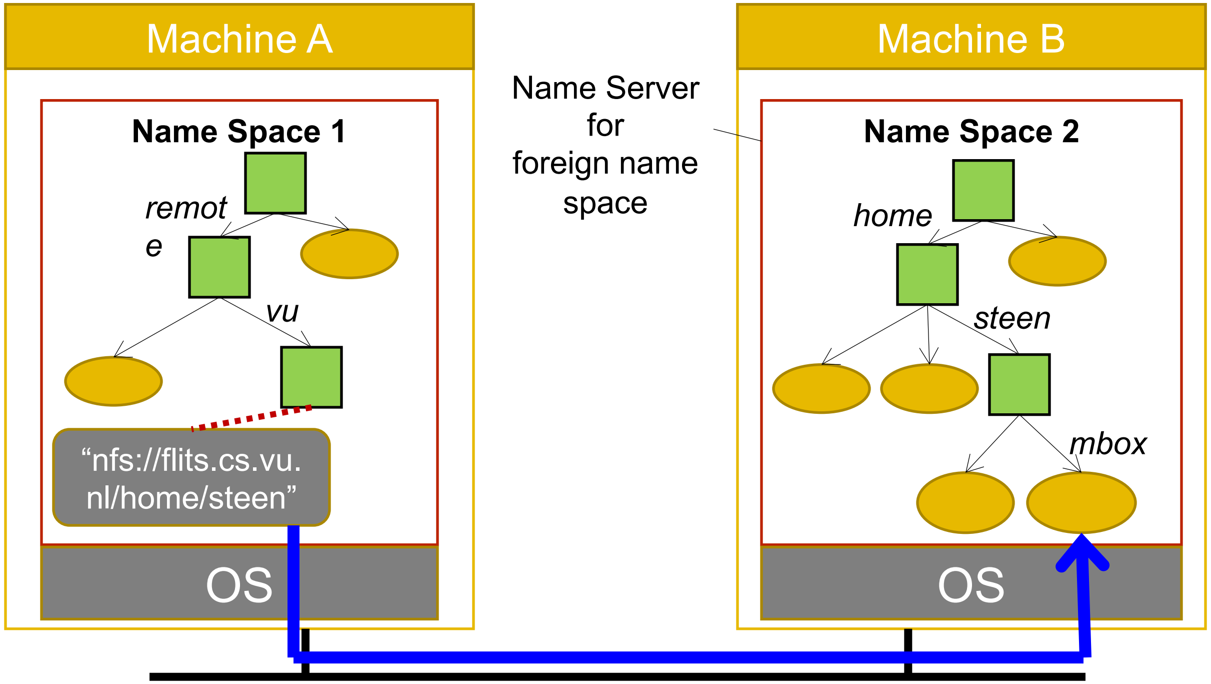
Mounting of Name Spaces

- Two or more name spaces can be merged transparently by a technique known as *mounting*

- With mounting, a directory node in one name space will store the identifier of the directory node of another name space

- Network File System (NFS) is an example where different name spaces are mounted
  - NFS enables *transparent* access to remote files

Example of Mounting Name Spaces in NFS



Name resolution for "/remote/vu/home/steen/mbox" in a distributed file system

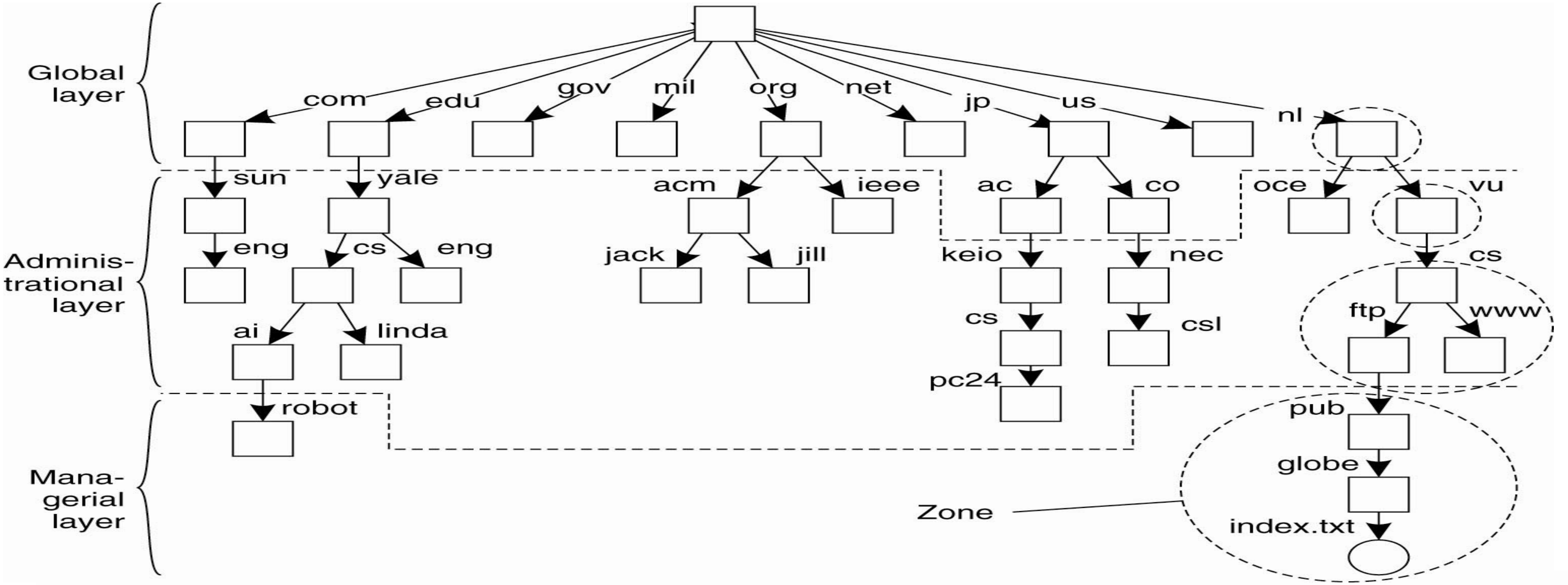
Name Space Implementation

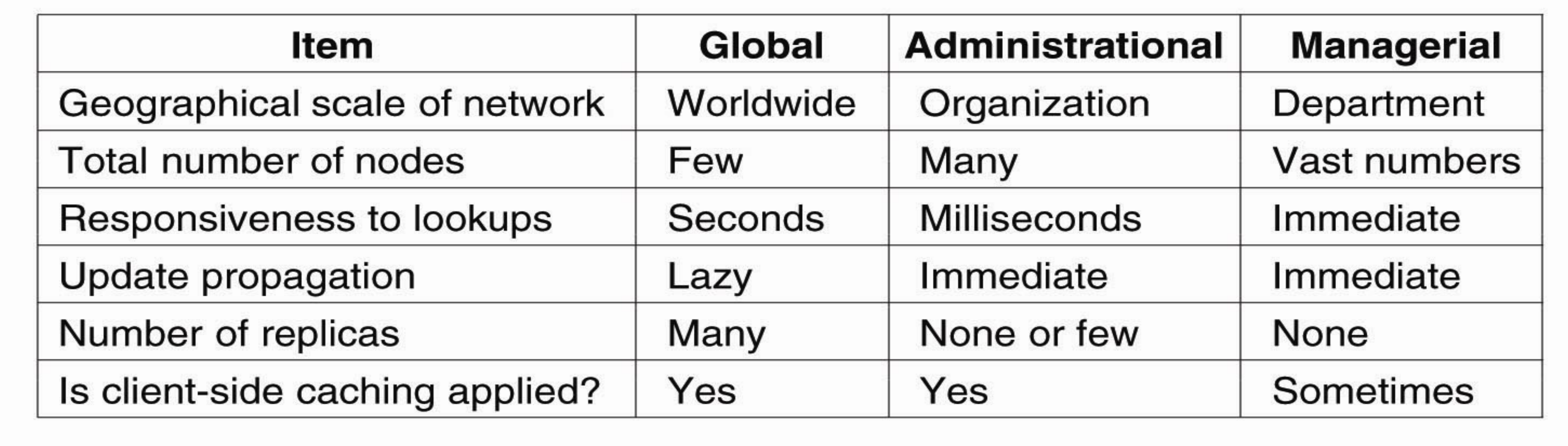
Distributed vs. centralized

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.)



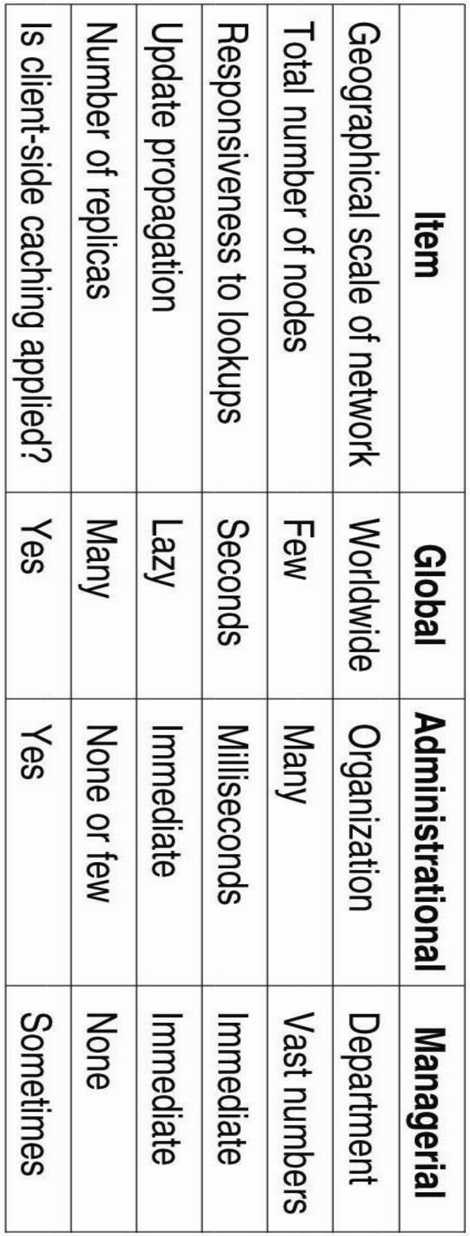


Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side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

Name Space Distribution (cont.)

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

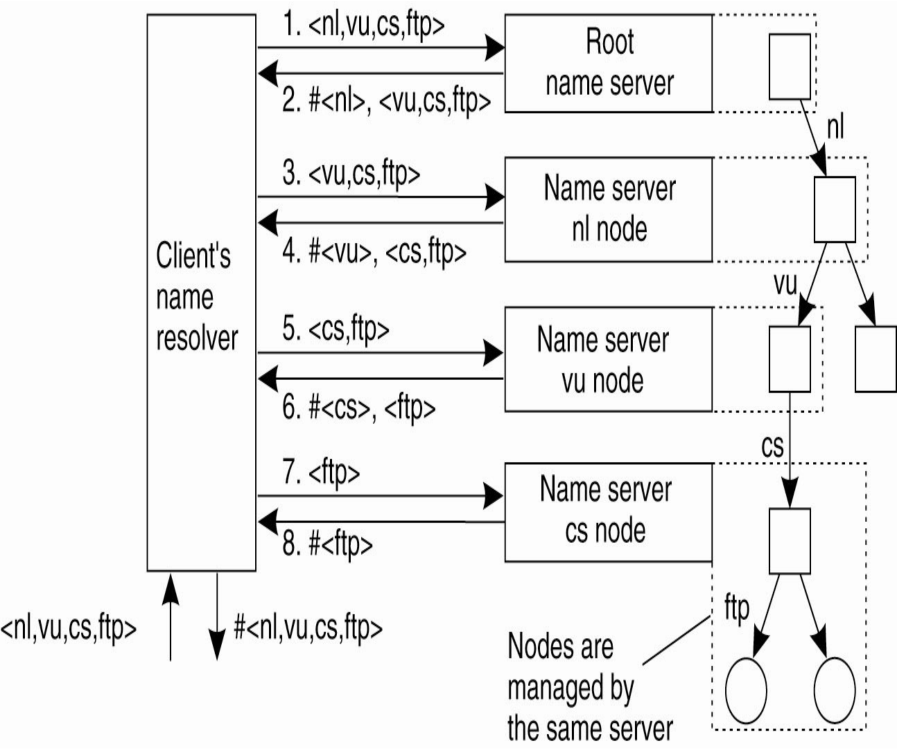
	Availability	Performance
Global	Must be very high Replication may help	Can be cached (stability) Replication may help
Administrative	Must be very high particularly for the clients in the same organization	Looks up should be fast Use high-end machines
Managerial	Less demanding One dedicated server might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

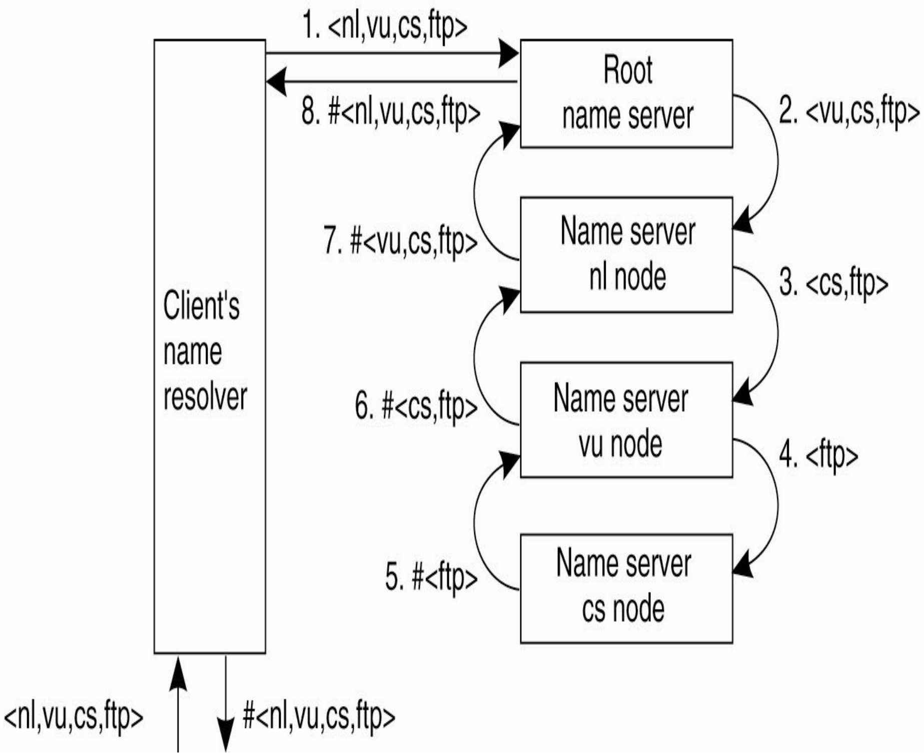


Item	**Global**	**Administrational**	**Managerial**
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None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

Implementation of Name Resolution

Iterative vs. Recursive



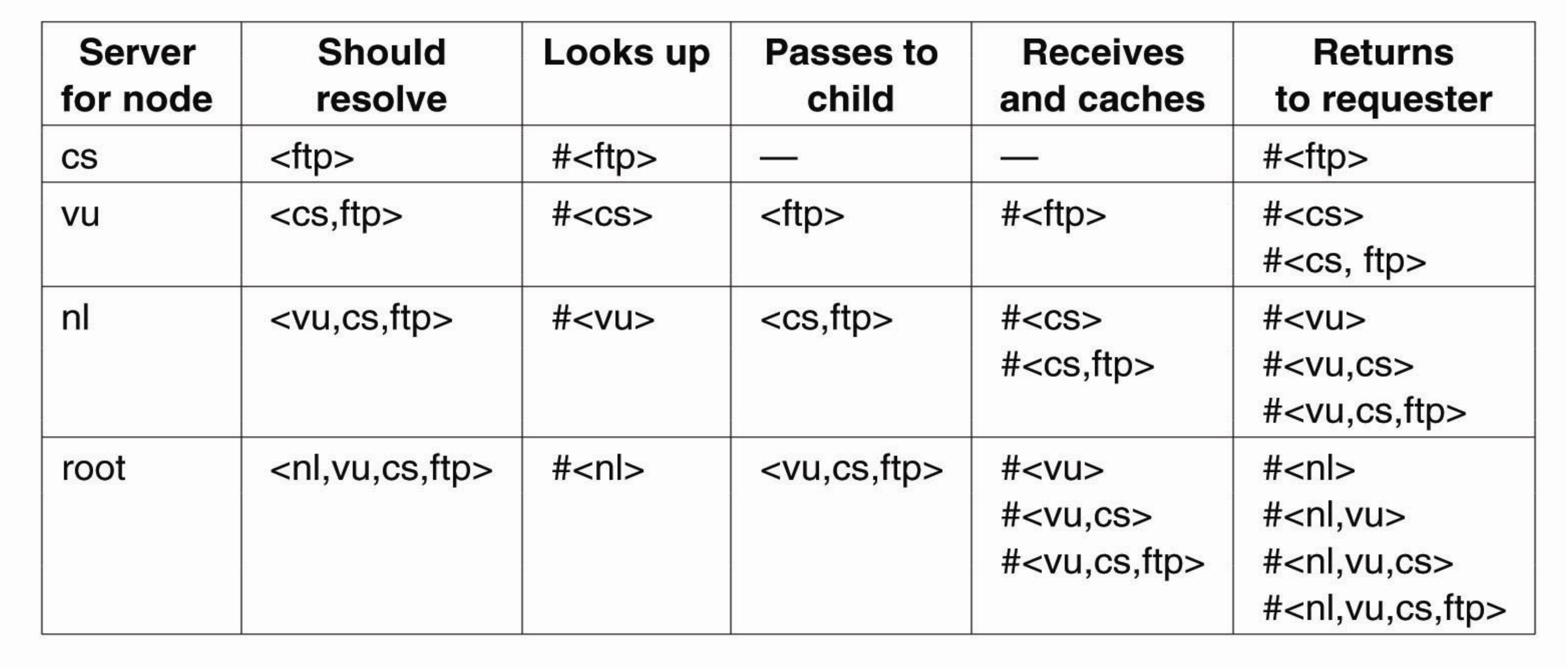


-Caching is restricted to client
-Communication cost, Delay
+ less overhead on root

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

Cache in Recursive Naming Resolution

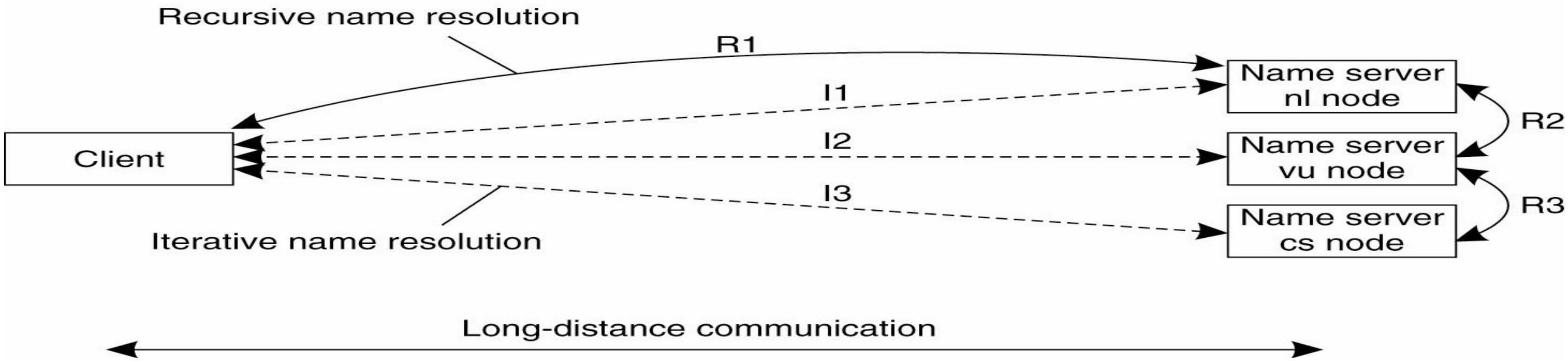
- Recursive name resolution of <nl, vu, cs, ftp>.
- Name servers cache intermediate results for subsequent lookups



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

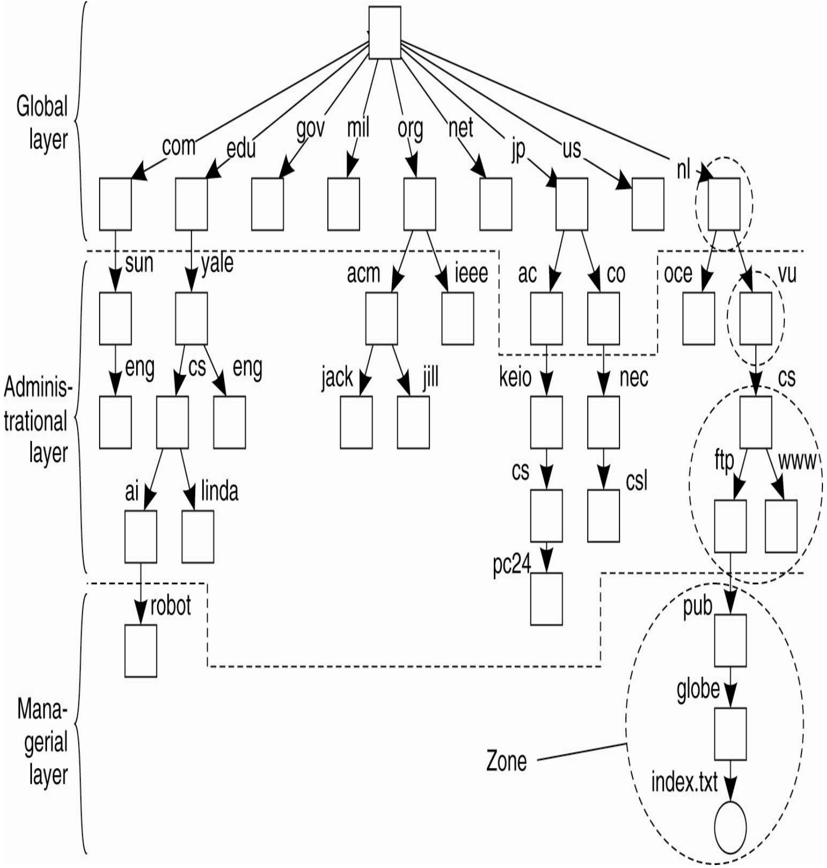
- **Size scalability:** We need to ensure that servers can handle a large number of requests per time unit → high-level servers are in big trouble.
  - **Solution:** Assume (at least at global and administrational level) that content of nodes **hardly ever changes**. In that case, we can apply extensive **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.


Recursive name resolution
R1
I1
I2
I3
Client
Name server nl node
Name server vu node
Name server cs node
R2
R3
Iterative name resolution
Long-distance communication

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

Zone partitioning of DNS name space

- **Zone** - contains attribute data for names in domain minus the sub-domains administrated by lower-level authorities:
- 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)

Caching in DNS

- *Any* 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

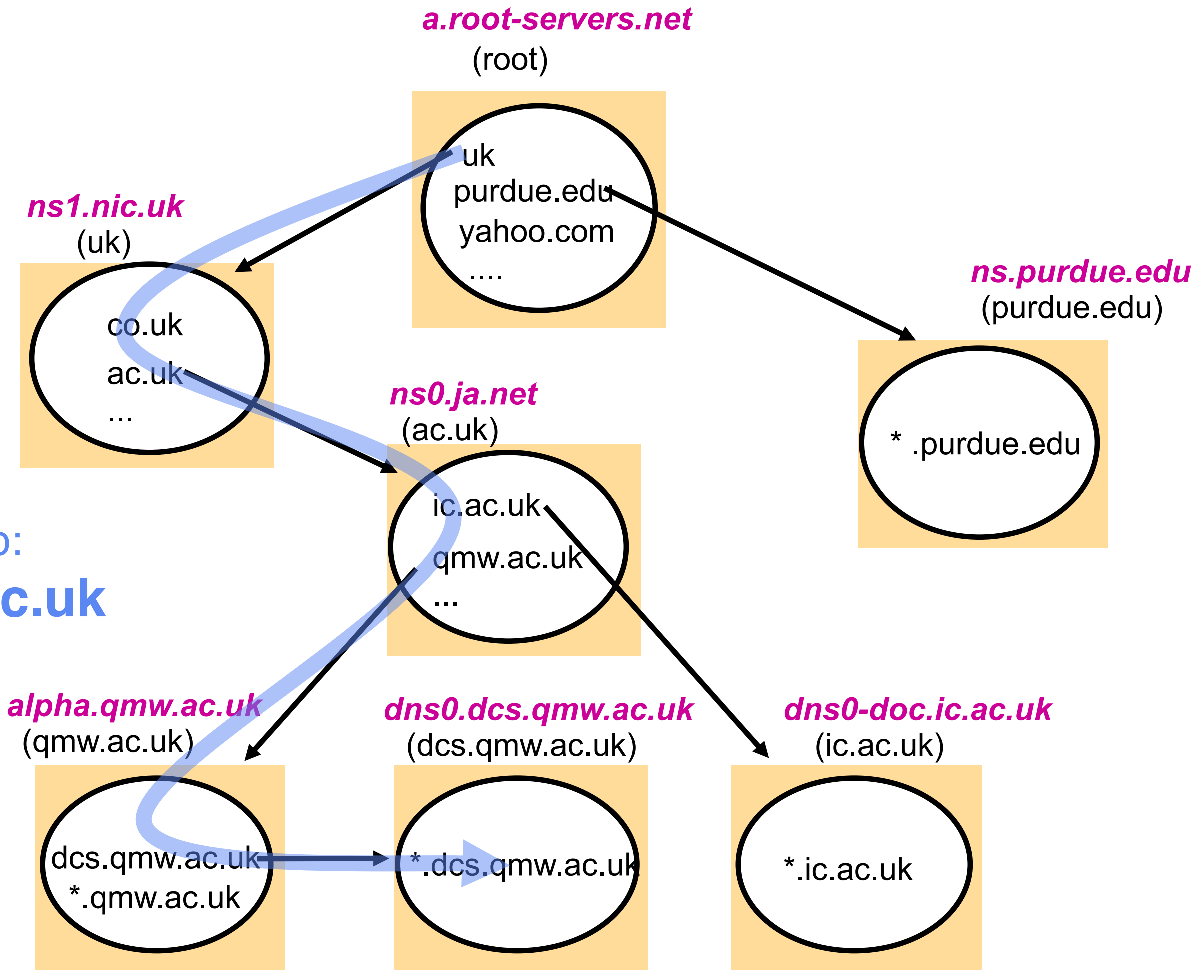
- Domain name → IP address ???
- Look for the name in the local **cache**
- Try a superior DNS server, which responds with:
  - the IP address (which may not be entirely up to date)
  - Or, another recommended DNS server (iterative)

DNS name servers

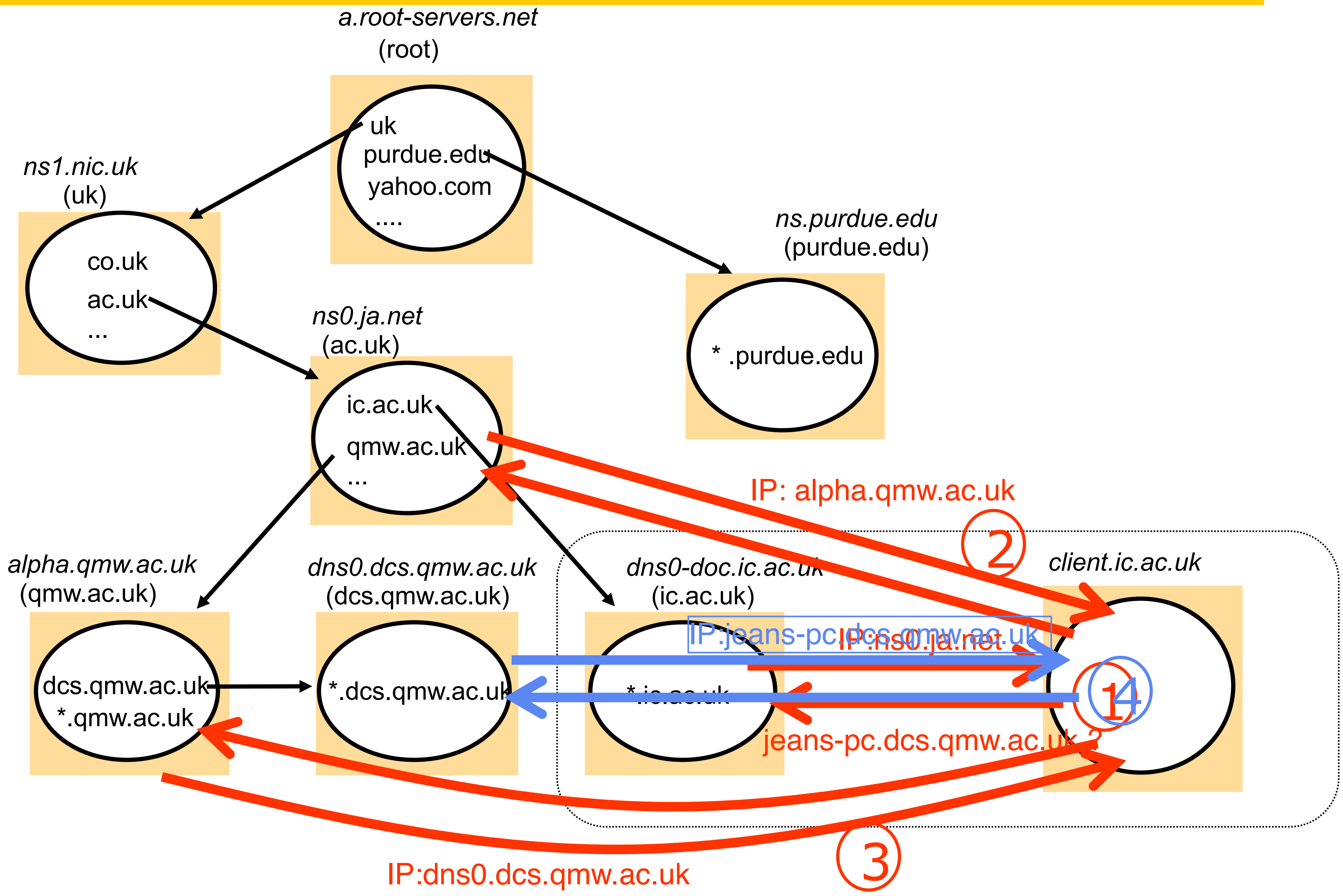
Note: Name server names are in italics, and the corresponding domains are in parentheses. Arrows denote name server entries

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

Attribute-Based Naming

Also known as Directory Services

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)

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**.

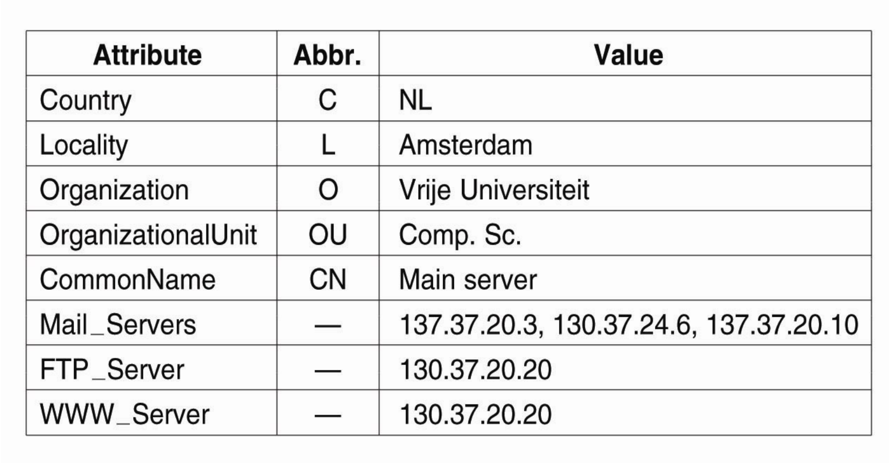
Directory Services (cont'd)

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)
  (Attribute, Value(s)) pairs



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

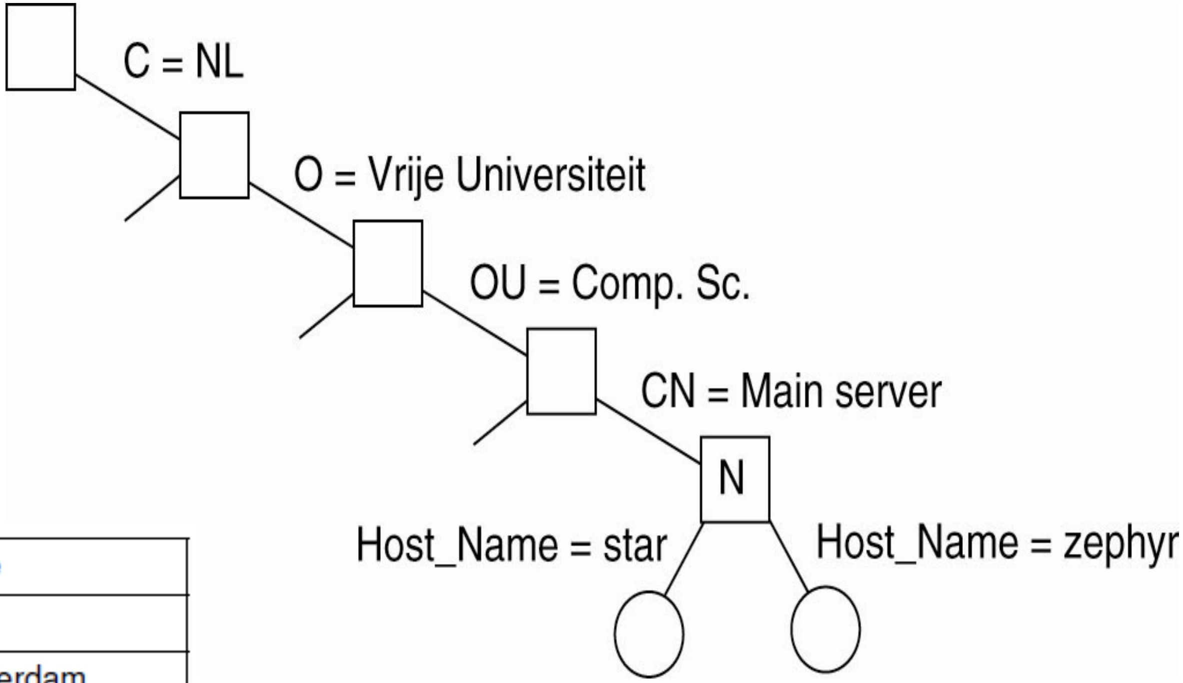
- Collection of all directory entries is called Directory Information Base (DIB)

- 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

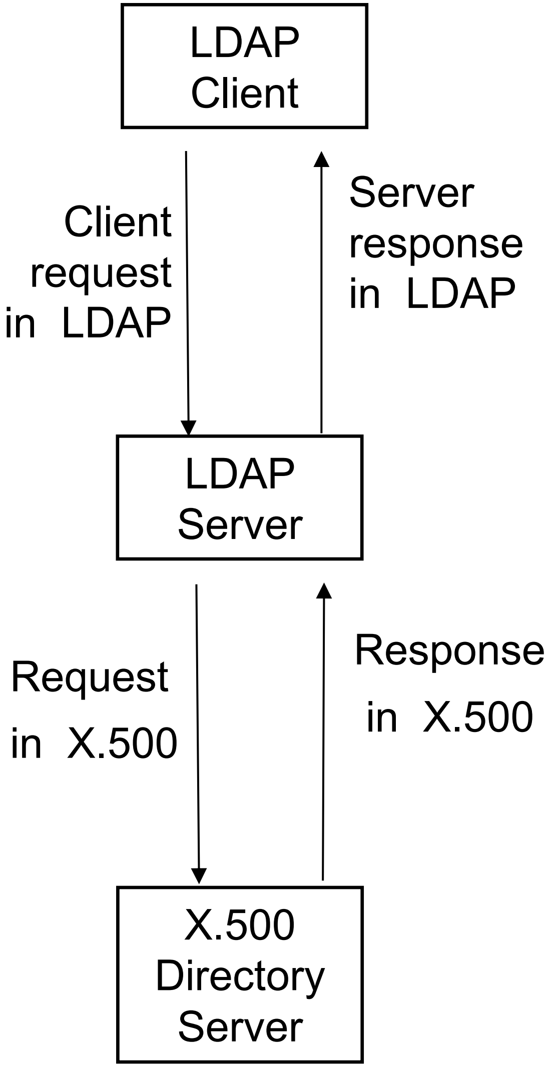


Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

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
- 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.