# Distributed Systems

(3rd Edition)

# Chapter 05: Naming

Version: February 25, 2017

## Naming

**Essence**

Names are used to denote entities in a distributed system. To operate on an entity, we need to access it at an access point. Access points are entities that are named by means of an address.

**Note**

A location-independent name for an entity $E$, is independent from the addresses of the access points offered by $E$.

## Identifiers

### Pure name

A name that has no meaning at all; it is just a random string. Pure names can be used for comparison only.

### Identifier: A name having some specific properties

1. An identifier refers to at most one entity.
2. Each entity is referred to by at most one identifier.
3. An identifier always refers to the same entity (i.e., it is never reused).

### Observation

An identifier need not necessarily be a pure name, i.e., it may have content.

## Broadcasting

**Broadcast the ID, requesting the entity to return its current address**

- Can never scale beyond local-area networks
- Requires all processes to listen to incoming location requests

**Address Resolution Protocol (ARP)**

To find out which MAC address is associated with an IP address, broadcast the query “who has this IP address”?

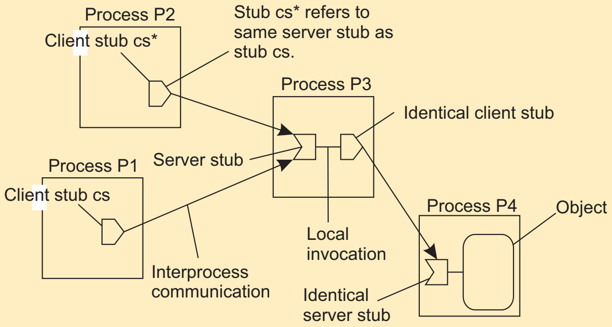
## Forwarding pointers

**When an entity moves, it leaves behind a pointer to its next location**

- Dereferencing can be made entirely transparent to clients by simply following the chain of pointers
- Update a client's reference when present location is found
- Geographical scalability problems (for which separate chain reduction mechanisms are needed):
    - Long chains are not fault tolerant
    - Increased network latency at dereferencing

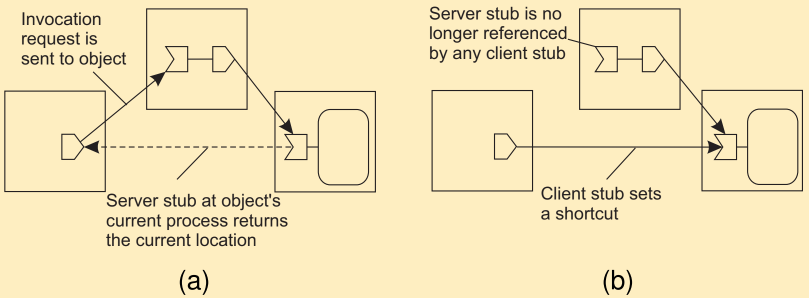
## Example: SSP chains

### The principle of forwarding pointers using *(client stub, server stub)*



## Example: SSP chains

### Redirecting a forwarding pointer by storing a shortcut in a client stub

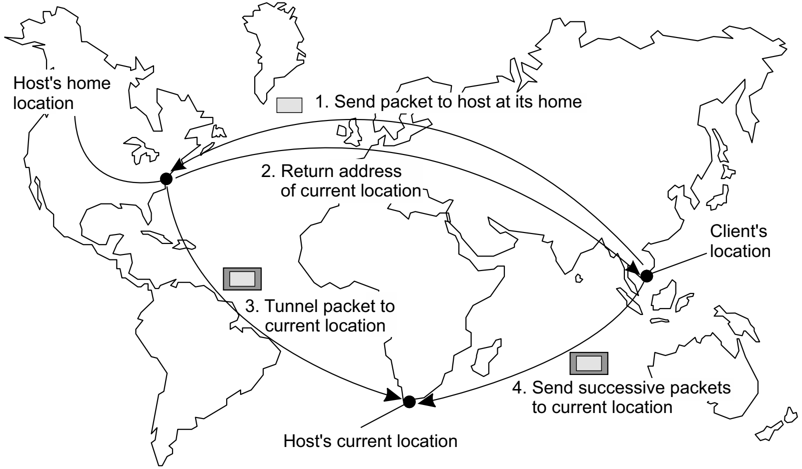

Invocation request is sent to object
Server stub at object's current process returns the current location
(a)
Server stub is no longer referenced by any client stub
Client stub sets a shortcut
(b)

## Home-based approaches

**Single-tiered scheme: Let a home keep track of where the entity is**

- Entity's home address registered at a naming service
- The home registers the foreign address of the entity
- Client contacts the home first, and then continues with foreign location

## The principle of mobile IP


Host's home location
1. Send packet to host at its home
2. Return address of current location
Client's location
3. Tunnel packet to current location
4. Send successive packets to current location
Host's current location

## Home-based approaches

**Problems with home-based approaches**

- Home address has to be supported for entity's lifetime
- Home address is fixed $\Rightarrow$ unnecessary burden when the entity permanently moves
- Poor geographical scalability (entity may be next to client)

**Note**

Permanent moves may be tackled with another level of naming (DNS)

## Illustrative: Chord

### Consider the organization of 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 $id \geq k$ (called its successor $succ(k)$).

### Nonsolution

Let each node keep track of its neighbor and start linear search along the ring.

### Notation

We will speak of node $p$ as the node have identifier $p$

## Chord finger tables

### Principle

- Each node $p$ maintains a finger table $FT_p[]$ with at most $m$ entries:

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

Note: the $i$-th entry points to the first node succeeding $p$ by at least $2^{i-1}$.

## Chord finger tables

### Principle

- Each node $p$ maintains a finger table $FT_p[]$ with at most $m$ entries:

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

  Note: the $i$-th entry 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

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

## Chord finger tables

### Principle

- Each node $p$ maintains a finger table $FT_p[]$ with at most $m$ entries:

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

  Note: the $i$-th entry points to the first node succeeding $p$ by at least $2^{i-1}$.

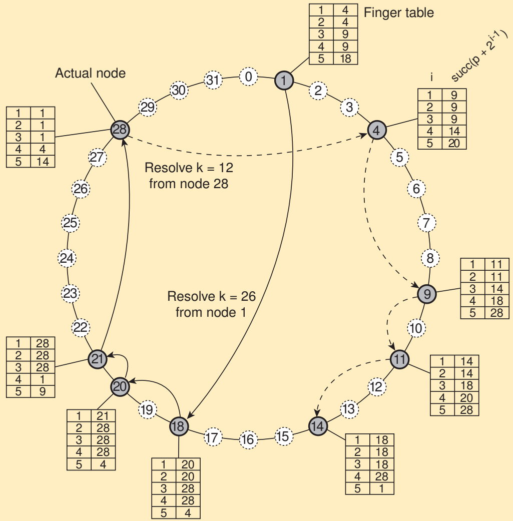
- To look up a key $k$, node $p$ forwards the request to node with index $j$ satisfying

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

- If $p < k < FT_p[1]$, the request is also forwarded to $FT_p[1]$

## Chord lookup example

### Resolving key 26 from node *1* and key *12* from node *28*



| | |
|---|---|
| 1 | 4 |
| 2 | 4 |
| 3 | 9 |
| 4 | 9 |
| 5 | 18 |

| i | $succ(p + 2^{i-1})$ |
|---|---|
| 1 | 9 |
| 2 | 9 |
| 3 | 9 |
| 4 | 14 |
| 5 | 20 |

| | |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |
| 4 | 4 |
| 5 | 14 |

| | |
|---|---|
| 1 | 11 |
| 2 | 11 |
| 3 | 14 |
| 4 | 18 |
| 5 | 28 |

| | |
|---|---|
| 1 | 28 |
| 2 | 28 |
| 3 | 28 |
| 4 | 1 |
| 5 | 9 |

| | |
|---|---|
| 1 | 14 |
| 2 | 14 |
| 3 | 18 |
| 4 | 20 |
| 5 | 28 |

| | |
|---|---|
| 1 | 21 |
| 2 | 28 |
| 3 | 28 |
| 4 | 28 |
| 5 | 4 |

| | |
|---|---|
| 1 | 18 |
| 2 | 18 |
| 3 | 18 |
| 4 | 28 |
| 5 | 1 |

| | |
|---|---|
| 1 | 20 |
| 2 | 20 |
| 3 | 28 |
| 4 | 28 |
| 5 | 4 |

## Chord in Python

```
class ChordNode:
  def finger(self, i):
    succ = (self.nodeID + pow(2, i-1)) % self.MAXPROC    # succ(p+2^(i-1))
    lwbi = self.nodeSet.index(self.nodeID)               # self in nodeset
    upbi = (lwbi + 1) % len(self.nodeSet)                # next neighbor
    for k in range(len(self.nodeSet)):                   # process segments
      if self.inbetween(succ, self.nodeSet[lwbi]+1, self.nodeSet[upbi]+1):
        return self.nodeSet[upbi]                        # found successor
      (lwbi,upbi) = (upbi, (upbi+1) % len(self.nodeSet)) # next segment

  def recomputeFingerTable(self):
    self.FT[0]  = self.nodeSet[self.nodeSet.index(self.nodeID)-1] # Pred.
    self.FT[1:] = [self.finger(i) for i in range(1,self.nBits+1)] # Succ.

  def localSuccNode(self, key):
    if self.inbetween(key, self.FT[0]+1, self.nodeID+1): # in (FT[0],self]
      return self.nodeID                                 # responsible node
    elif self.inbetween(key, self.nodeID+1, self.FT[1]): # in (self,FT[1]]
      return self.FT[1]                                  # succ. responsible
    for i in range(1, self.nBits+1):                     # rest of FT
      if self.inbetween(key, self.FT[i], self.FT[(i+1) % self.nBits]):
        return self.FT[i]                                # in [FT[i],FT[i+1])
```

## Exploiting network proximity

### Problem

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

### Solutions

## Exploiting network proximity

### Problem

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

### Solutions

- 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.

## Exploiting network proximity

### Problem

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

### Solutions

- 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 $FT_p[i]$ points to first node in $INT = [p+2^{i-1}, p+2^i-1]$. Node $p$ can also store pointers to other nodes in $INT$.

## Exploiting network proximity

### Problem

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

### Solutions

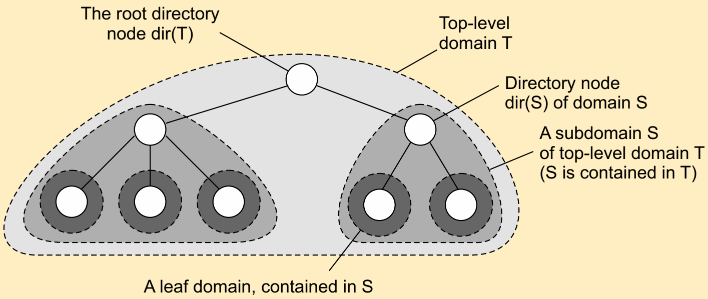
- Topology-aware node assignment: When assigning an ID to a node, make sure that nodes close in the ID space are also close in the network. Can be very difficult.
- Proximity routing: Maintain more than one possible successor, and forward to the closest.
  Example: in Chord $FT_p[i]$ points to first node in $INT = [p+2^{i-1}, p+2^i-1]$. 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)

### 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

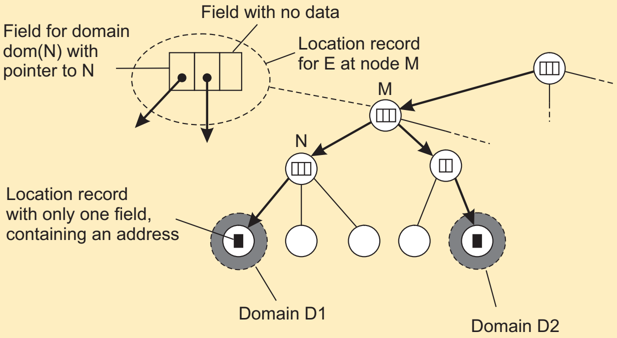

The root directory node dir(T)
Top-level domain T
Directory node dir(S) of domain S
A subdomain S of top-level domain T (S is contained in T)
A leaf domain, contained in S

## 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

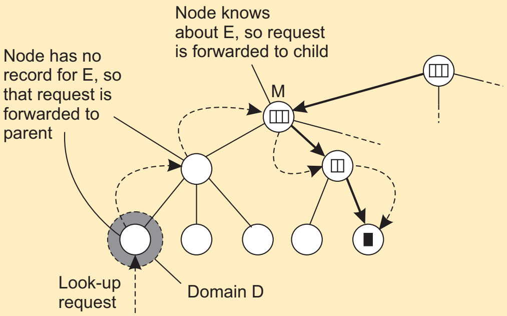

Field with no data
Field for domain dom(N) with pointer to N
Location record for E at node M
M
N
Location record with only one field, containing an address
Domain D1
Domain D2

## HLS: Lookup operation

### Basic principles

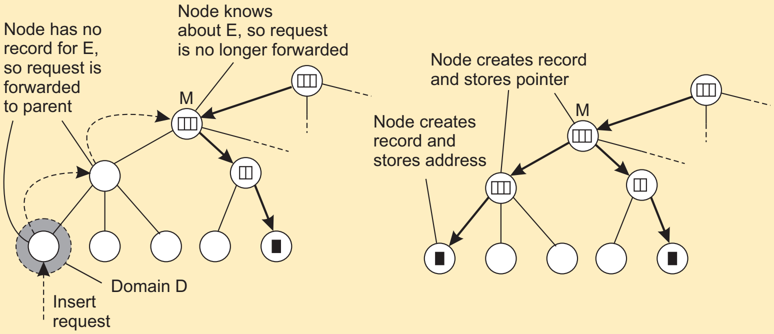
- Start lookup at local leaf node
- Node knows about $E \Rightarrow$ follow downward pointer, else go up
- Upward lookup always stops at root

### 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

## HLS: Insert operation

(a) An insert request is forwarded to the first node that knows about entity *E*.
(b) A chain of forwarding pointers to the leaf node is created


Node has no record for E, so request is forwarded to parent
Node knows about E, so request is no longer forwarded
M
Domain D
Insert request
Node creates record and stores pointer
M
Node creates record and stores address

(a)

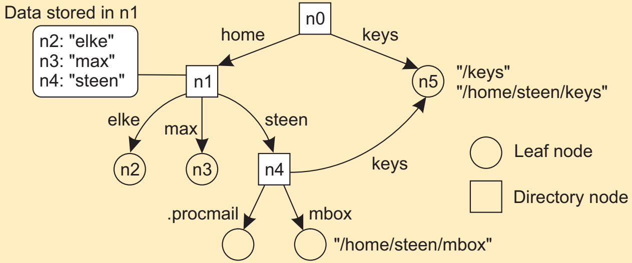
(b)

## Name space

### Naming graph

A graph in which a leaf node represents a (named) entity. A directory node is an entity that refers to other nodes.

### A general naming graph with a single root node



### Note

A directory node contains a table of *(node identifier, edge label)* pairs.

## Name space

We can easily store all kinds of attributes in a node

- Type of the entity
- An identifier for that entity
- Address of the entity's location
- Nicknames
- ...

## Name space

**We can easily store all kinds of attributes in a node**

- Type of the entity
- An identifier for that entity
- Address of the entity's location
- Nicknames
- ...

**Note**

Directory nodes can also have attributes, besides just storing a directory table with *(identifier, label)* pairs.

## Name resolution

**Problem**

To resolve a name we need a directory node. How do we actually find that (initial) node?

## Name resolution

**Problem**

To resolve a name we need a directory node. How do we actually find that (initial) node?

**Closure mechanism: The mechanism to select the implicit context from which to start name resolution**

- `www.distributed-systems.net`: start at a DNS name server
- `/home/maarten/mbox`: start at the local NFS file server (possible recursive search)
- `0031 20 598 7784`: dial a phone number
- `77.167.55.6`: route message to a specific IP address

## Name resolution

**Problem**

To resolve a name we need a directory node. How do we actually find that (initial) node?

**Closure mechanism: The mechanism to select the implicit context from which to start name resolution**

- `www.distributed-systems.net`: start at a DNS name server
- `/home/maarten/mbox`: start at the local NFS file server (possible recursive search)
- `0031 20 598 7784`: dial a phone number
- `77.167.55.6`: route message to a specific IP address

**Note**

You cannot have an explicit closure mechanism – how would you start?

## Name-space implementation

**Basic issue**

Distribute the name resolution process as well as name space management across multiple machines, by distributing nodes of the naming graph.

## Name-space implementation

**Basic issue**

Distribute the name resolution process as well as name space management across multiple machines, by distributing nodes of the naming graph.

**Distinguish three levels**

## Name-space implementation

### Basic issue

Distribute the name resolution process as well as name space management across multiple machines, by distributing nodes of the naming graph.

### Distinguish three levels

- Global level: Consists of the high-level directory nodes. Main aspect is that these directory nodes have to be jointly managed by different administrations

## Name-space implementation

### Basic issue

Distribute the name resolution process as well as name space management across multiple machines, by distributing nodes of the naming graph.

### Distinguish three levels

- Global level: Consists of the high-level directory nodes. Main aspect is that these directory nodes have to be jointly managed by different administrations
- Administrational level: Contains mid-level directory nodes that can be grouped in such a way that each group can be assigned to a separate administration.

## Name-space implementation

### Basic issue

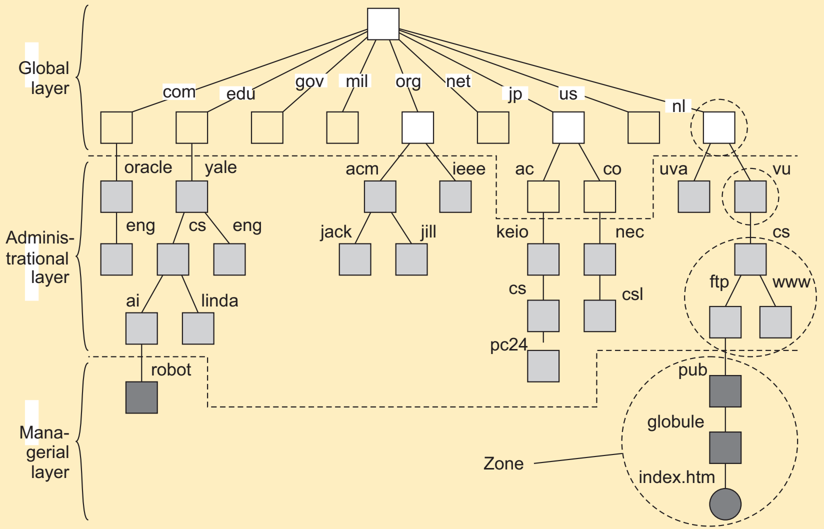
Distribute the name resolution process as well as name space management across multiple machines, by distributing nodes of the naming graph.

### Distinguish three levels

- Global level: Consists of the high-level directory nodes. Main aspect is that these directory nodes have to be jointly managed by different administrations
- Administrational level: Contains mid-level directory nodes that can be grouped in such a way that each group can be assigned to a separate administration.
- Managerial level: Consists of low-level directory nodes within a single administration. Main issue is effectively mapping directory nodes to local name servers.

## Name-space implementation

### An example partitioning of the DNS name space, including network files



## Name-space implementation

**A comparison between name servers for implementing nodes in a name space**

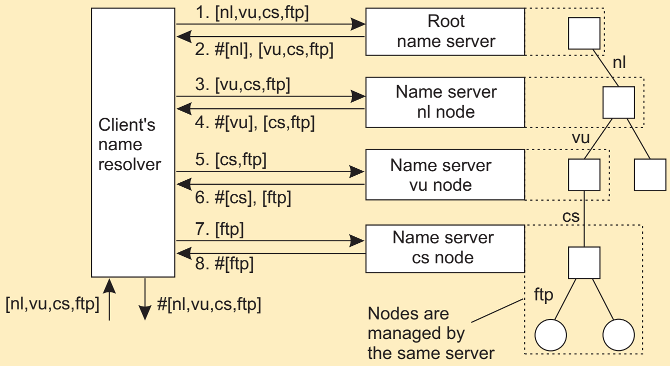
| Item | Global | Administrational | Managerial |
|---|---|---|---|
| 1 | Worldwide | Organization | Department |
| 2 | Few | Many | Vast numbers |
| 3 | Seconds | Milliseconds | Immediate |
| 4 | Lazy | Immediate | Immediate |
| 5 | Many | None or few | None |
| 6 | Yes | Yes | Sometimes |

| | |
|---|---|
| 1: Geographical scale | 4: Update propagation |
| 2: # Nodes | 5: # Replicas |
| 3: Responsiveness | 6: Client-side caching? |

## Iterative name resolution

### Principle

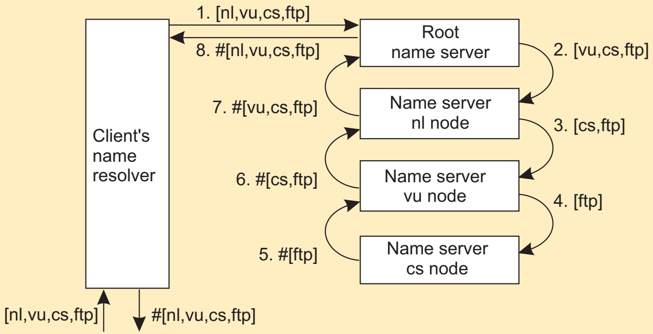
1. *resolve*$(dir, [name_1, ..., name_K])$ sent to *Server*$_0$ responsible for *dir*
2. *Server*$_0$ resolves *resolve*$(dir, name_1) \rightarrow dir_1$, returning the identification (address) of *Server*$_1$, which stores $dir_1$.
3. Client sends *resolve*$(dir_1, [name_2, ..., name_K])$ to *Server*$_1$, etc.



## Recursive name resolution

### Principle

1. $resolve(dir, [name_1, ..., name_K])$ sent to $Server_0$ responsible for $dir$
2. $Server_0$ resolves $resolve(dir, name_1) \rightarrow dir_1$, and sends $resolve(dir_1, [name_2, ..., name_K])$ to $Server_1$, which stores $dir_1$.
3. $Server_0$ waits for result from $Server_1$, and returns it to client.



## Caching in recursive name resolution

### Recursive name resolution of [*nl*, *vu*, *cs*,*ftp*]

| 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*]<br>#[*cs*, *ftp*] |
| *nl* | [*vu*, *cs*, *ftp*] | #[*vu*] | [*cs*, *ftp*] | #[*cs*]<br>#[*cs*, *ftp*] | #[*vu*]<br>#[*vu*, *cs*]<br>#[*vu*, *cs*, *ftp*] |
| *root* | [*nl*, *vu*, *cs*, *ftp*] | #[*nl*] | [*vu*, *cs*, *ftp*] | #[*vu*]<br>#[*vu*, *cs*]<br>#[*vu*, *cs*, *ftp*] | #[*nl*]<br>#[*nl*, *vu*]<br>#[*nl*, *vu*, *cs*]<br>#[*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.

## 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. We can then apply extensive replication by mapping nodes to multiple servers, and start name resolution at the nearest server.

## 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

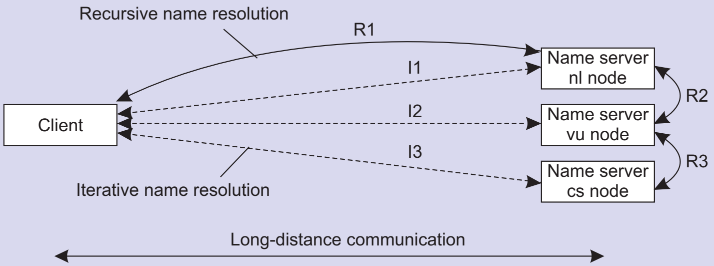
Assume (at least at global and administrational level) that content of nodes hardly ever changes. We can then apply extensive replication by mapping nodes to multiple servers, and start name resolution at the nearest server.

### Observation

An important attribute of many nodes is the address where the represented entity can be contacted. Replicating nodes makes large-scale traditional name servers unsuitable for locating mobile entities.

## Scalability issues

We need to ensure that the name resolution process scales across large geographical distances


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

### Problem

By mapping nodes to servers that can be located anywhere, we introduce an implicit location dependency.

## DNS

### Essence

- Hierarchically organized name space with each node having exactly one incoming edge $\Rightarrow$ edge label = node label.
- domain: a subtree
- domain name: a path name to a domain's root node.

### Information in a node

| **Type** | **Refers to** | **Description** |
|---|---|---|
| *SOA* | Zone | Holds info on the represented zone |
| *A* | Host | IP addr. of host this node represents |
| *MX* | Domain | Mail server to handle mail for this node |
| *SRV* | Domain | Server handling a specific service |
| *NS* | Zone | Name server for the represented zone |
| *CNAME* | Node | Symbolic link |
| *PTR* | Host | Canonical name of a host |
| *HINFO* | Host | Info on this host |
| *TXT* | Any kind | Any info considered useful |

## Attribute-based naming

**Observation**

In many cases, it is much more convenient to name, and look up entities by means of their attributes ⇒ traditional directory services (aka yellow pages).

## Attribute-based naming

**Observation**

In many cases, it is much more convenient to name, and look up entities by means of their attributes ⇒ traditional directory services (aka yellow pages).

**Problem**

Lookup operations can be extremely expensive, as they require to match requested attribute values, against actual attribute values ⇒ inspect all entities (in principle).

# Implementing directory services

## Solution for scalable searching

Implement basic directory service as database, and combine with traditional structured naming system.

## Lightweight Directory Access Protocol (LDAP)

Each directory entry consists of (*attribute, value*) pairs, and is uniquely named to ease lookups.

| **Attribute** | **Abbr.** | **Value** |
|---|---|---|
| Country | *C* | NL |
| Locality | *L* | Amsterdam |
| Organization | *O* | VU University |
| OrganizationalUnit | *OU* | Computer Science |
| 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 |