Distributed Systems
Principles and Paradigms

Chapter 05
(*version September 20, 2007*)

Maarten van Steen

Vrije Universiteit Amsterdam, Faculty of Science
Dept. Mathematics and Computer Science
Room R4.20. Tel: (020) 598 7784
E-mail:steen@cs.vu.nl, URL: www.cs.vu.nl/∼steen/

Naming Entities

- Names, identifiers, and addresses
- Name resolution
- Name space implementation

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 the following 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 the same entity (prohibits reusing an identifier)

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

Question: Can the content of an identifier ever change?

Flat Naming

Problem: Given an essentially **unstructured name** (e.g., an identifier), how can we locate its associated **access point**?

- Simple solutions (broadcasting)
- Home-based approaches
- Distributed Hash Tables (structured P2P)
- Hierarchical location service

Simple Solutions

Broadcasting: Simply broadcast the ID, requesting the entity to return its current address.

- Can never scale beyond local-area networks (think of ARP/RARP)
- Requires all processes to listen to incoming location requests

Forwarding pointers: Each time an entity moves, it leaves behind a pointer telling where it has gone to.

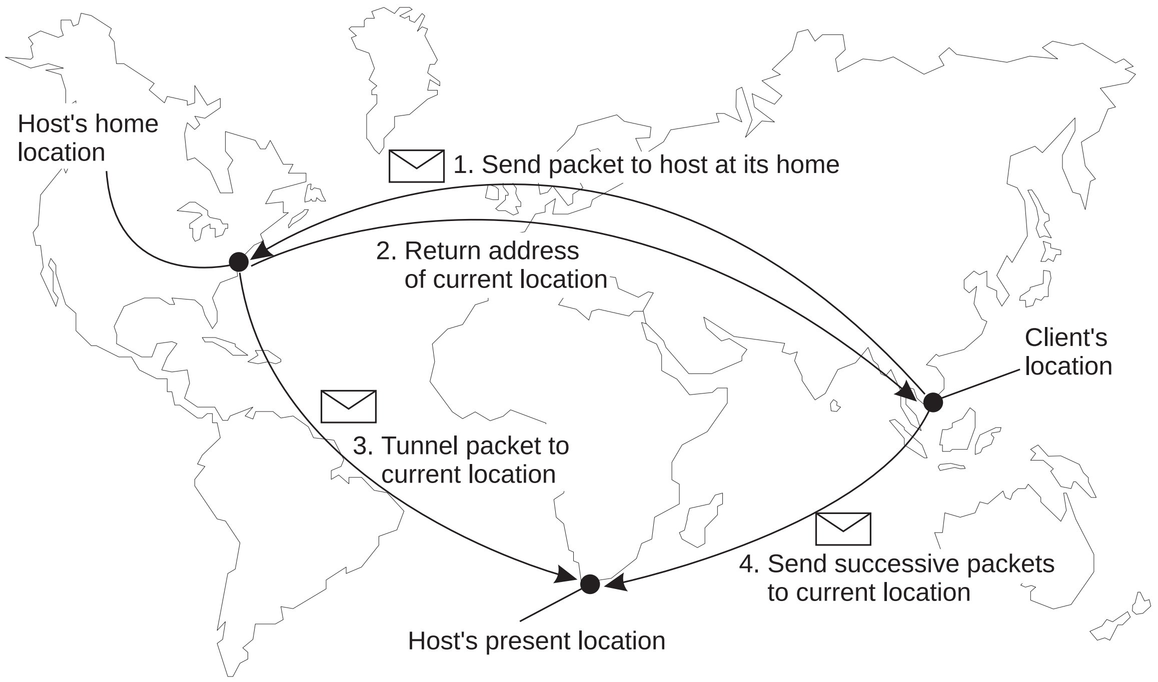
- Dereferencing can be made entirely transparent to clients by simply following the chain of pointers
- Update a client's reference as soon as present location has been found
- Geographical scalability problems:
  - Long chains are not fault tolerant
  - Increased network latency at dereferencing

  Essential to have separate chain reduction mechanisms

Home-Based Approaches (1/2)

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

- An entity's **home address** is registered at a naming service
- The home registers the **foreign address** of the entity
- Clients always contact the home first, and then continues with the foreign location


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 present location

Home-Based Approaches (2/2)

Two-tiered scheme: Keep track of **visiting** entities:

- Check local visitor register first
- Fall back to home location if local lookup fails

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
- Poor geographical scalability (the entity may be next to the client)

Question: How can we solve the “permanent move” problem?

Distributed Hash Tables

Example: Consider the organization of many nodes into a **logical ring** (**Chord**)

- 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**).

Nonsolution: Let node id keep track of $succ(id)$ and start linear search along the ring.

DHTs: Finger Tables (1/2)

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

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

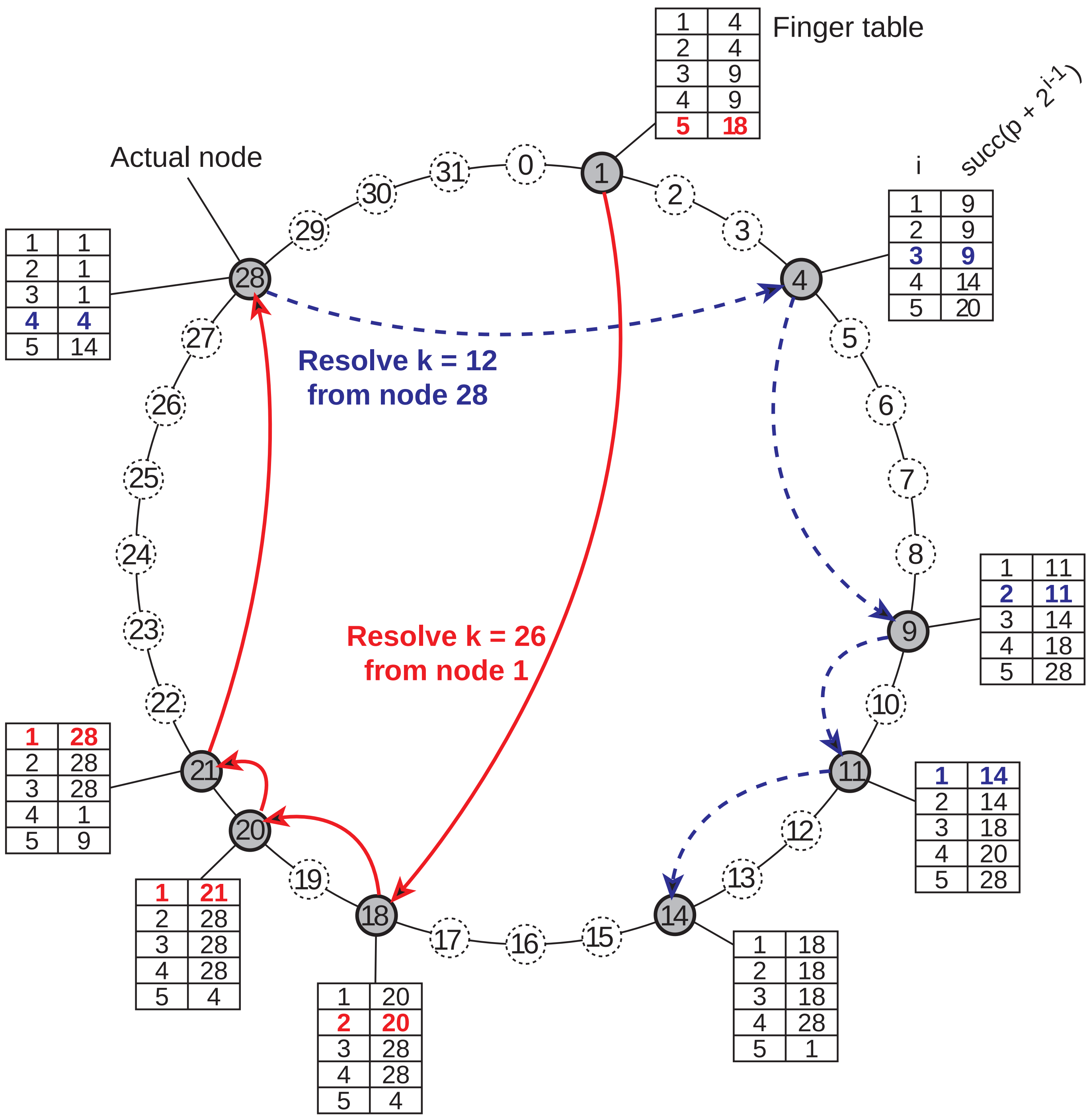
  Note: $FT_p[i]$ points to the first node succeeding p by at least 2^{i-1}.

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

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

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

DHTs: Finger Tables (2/2)



Node 1:

i	succ
1	4
2	4
3	9
4	9
5	18

Node 4:

i	succ
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

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.

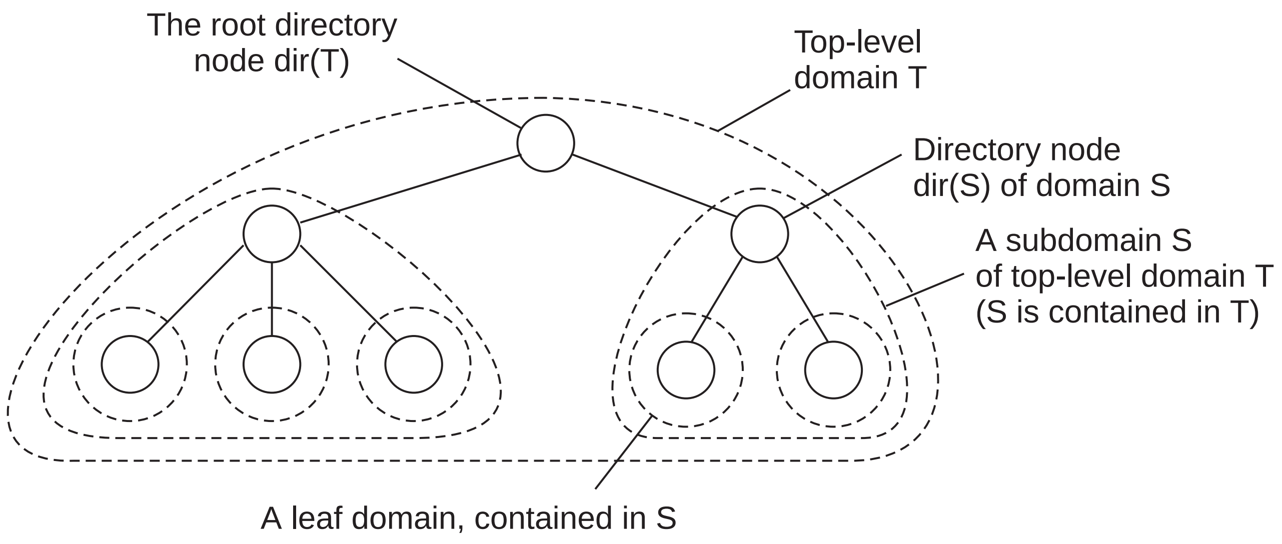
Topology-aware node assignment: When assigning an ID to a node, make sure that nodes close in the ID space are also close in the network. **Can be very difficult**.

Proximity routing: Maintain more than one possible successor, and forward to the closest.
Example: in Chord $FT_p[i]$ 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.

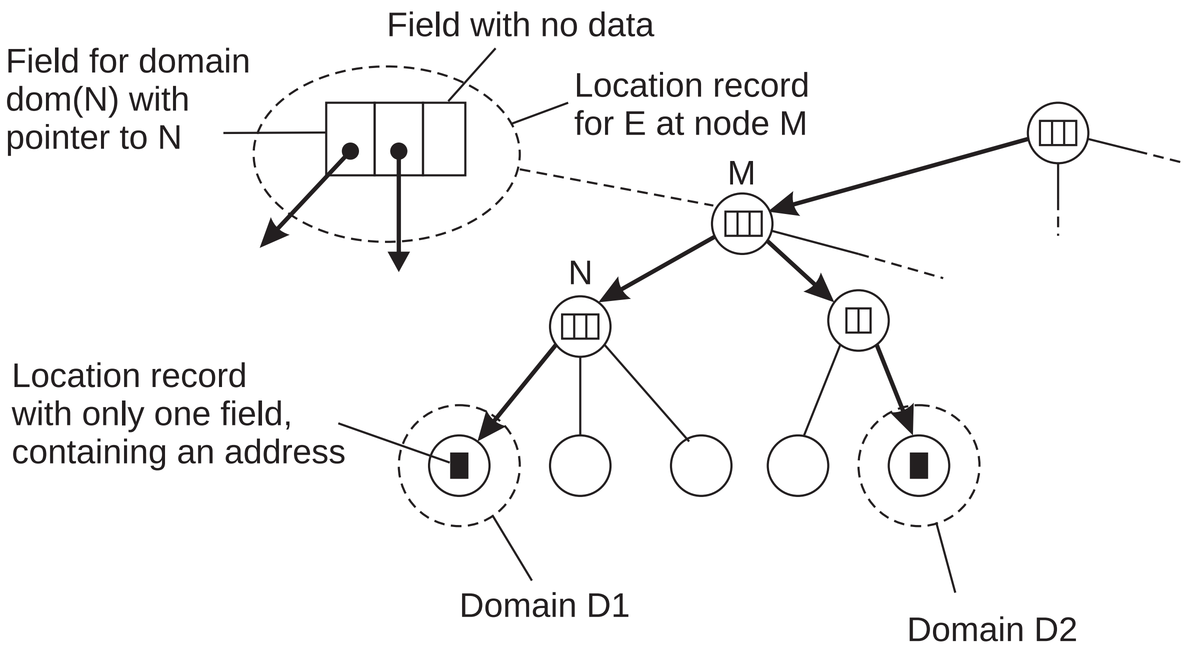
Hierarchical Location Services (HLS)

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.


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

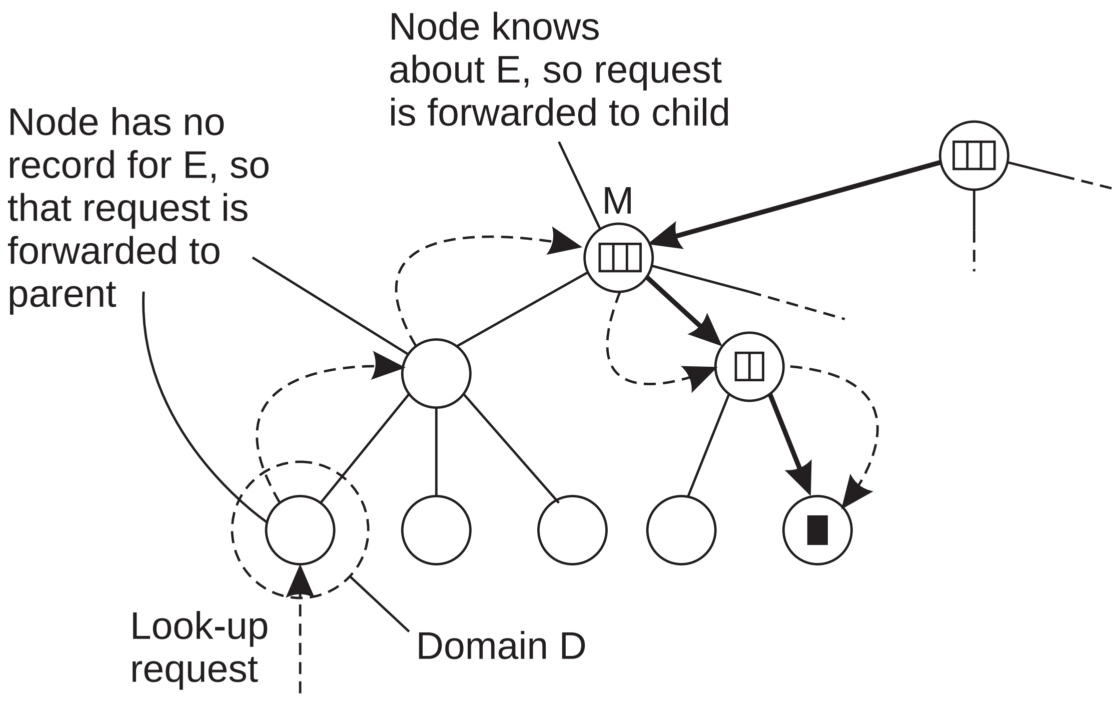
- The address of an entity is stored in a leaf node, or in an intermediate node
- Intermediate nodes contain a pointer to a child if and only if the subtree rooted at the child stores an address of the entity
- The root knows about all entities


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

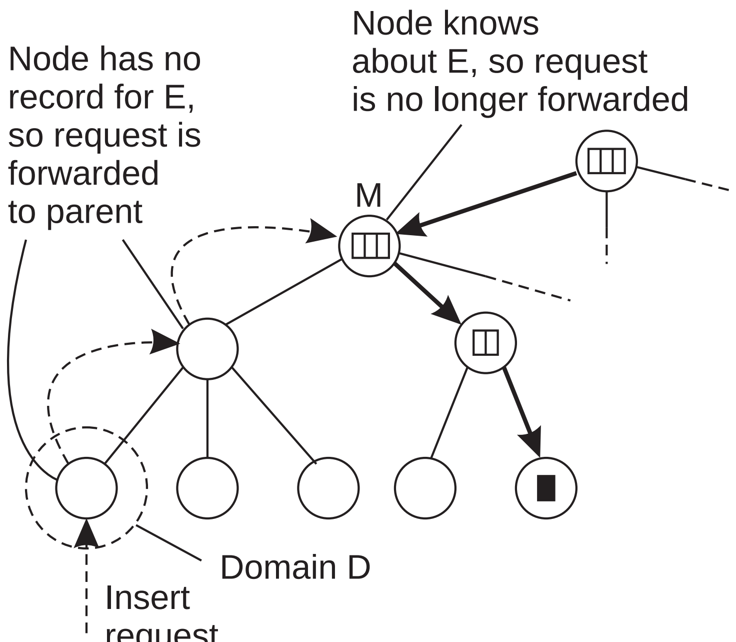
HLS: Lookup Operation

Basic principles:

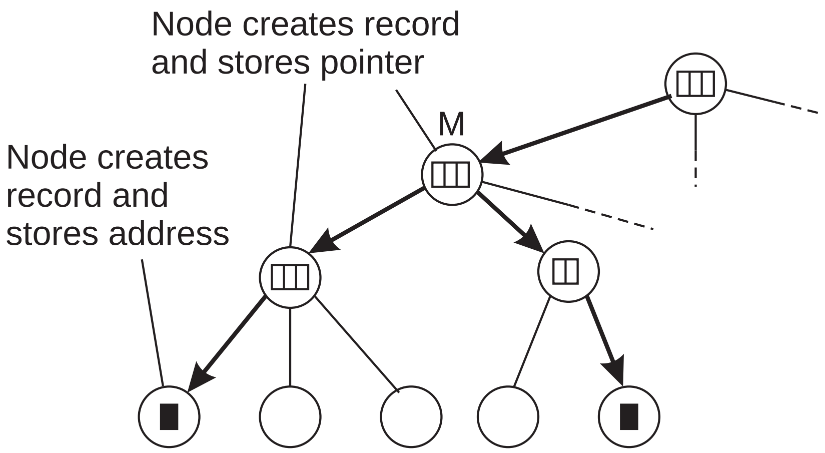
- Start lookup at local leaf node
- If node knows about the entity, follow downward pointer, otherwise go one level up
- Upward lookup always stops at root


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


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

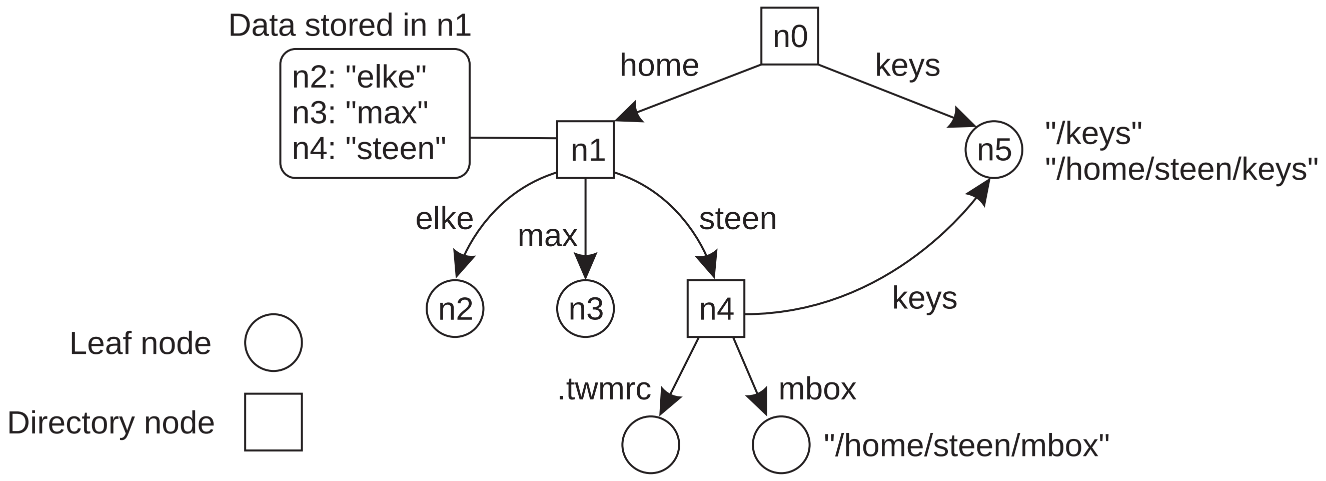
(a)


Node creates record and stores pointer
Node creates record and stores address
M

(b)

Name Space (1/2)

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



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

Name Space (2/2)

Observation: 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
- ...

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

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.cs.vu.nl`: start at a DNS name server
- `/home/steen/mbox`: start at the local NFS file server (possible recursive search)
- `0031204447784`: dial a phone number
- `130.37.24.8`: route to the VU's Web server

Question: Why are closure mechanisms always **implicit**?

Observation: A closure mechanism may also determine how name resolution should proceed

Name Linking (1/2)

Hard link: 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.

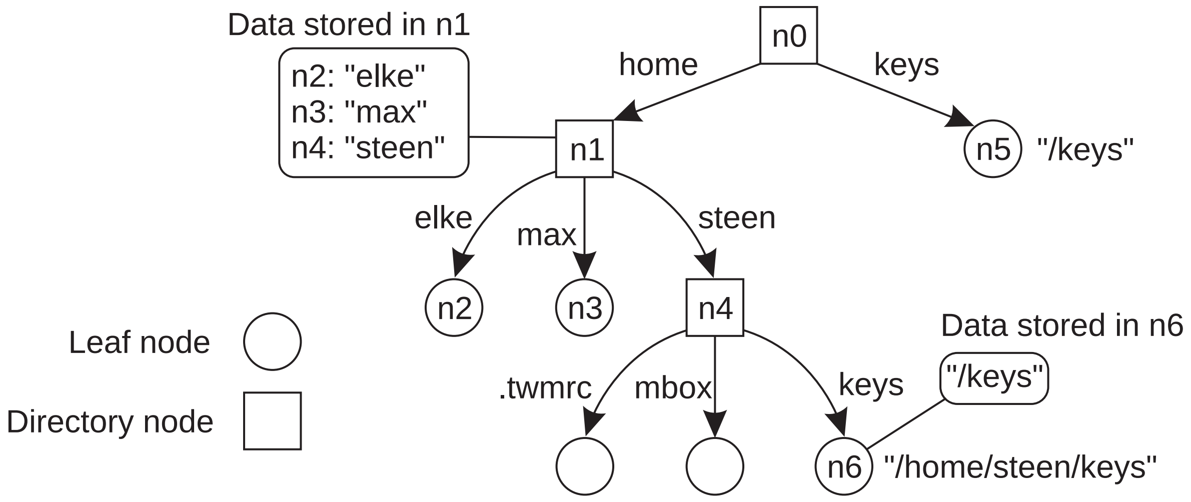
Soft link: Allow a node O to contain a name of another node:

- First resolve O's name (leading to O)
- Read the content of O, yielding `name`
- Name resolution continues with `name`

Observations:

- The name resolution process determines that we read the content of a node, in particular, the name in the other node that we need to go to.
- One way or the other, we know where and how to start name resolution given `name`

Name Linking (2/2)



Observation: Node `n5` has only one name

Name Space Implementation (1/2)

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

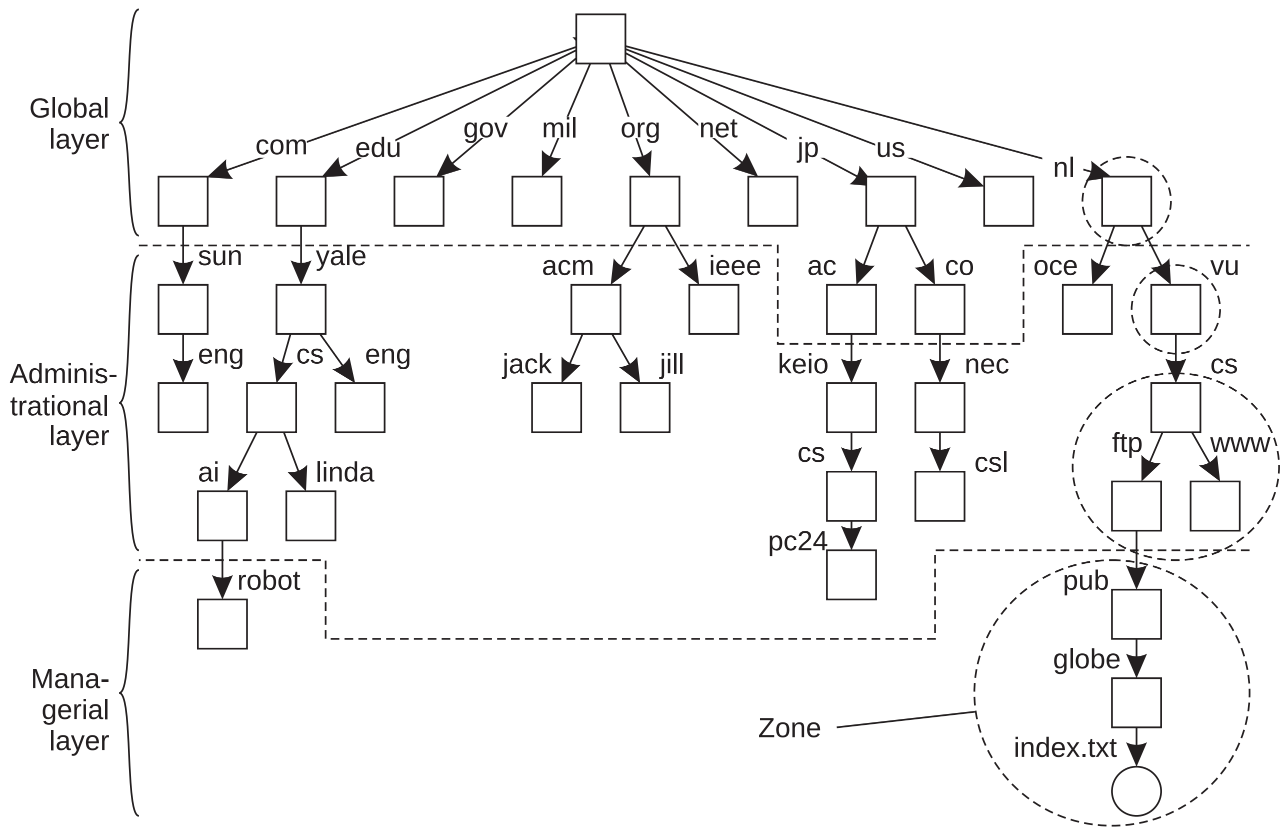
Consider a hierarchical naming graph and distinguish three levels:

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 (2/2)

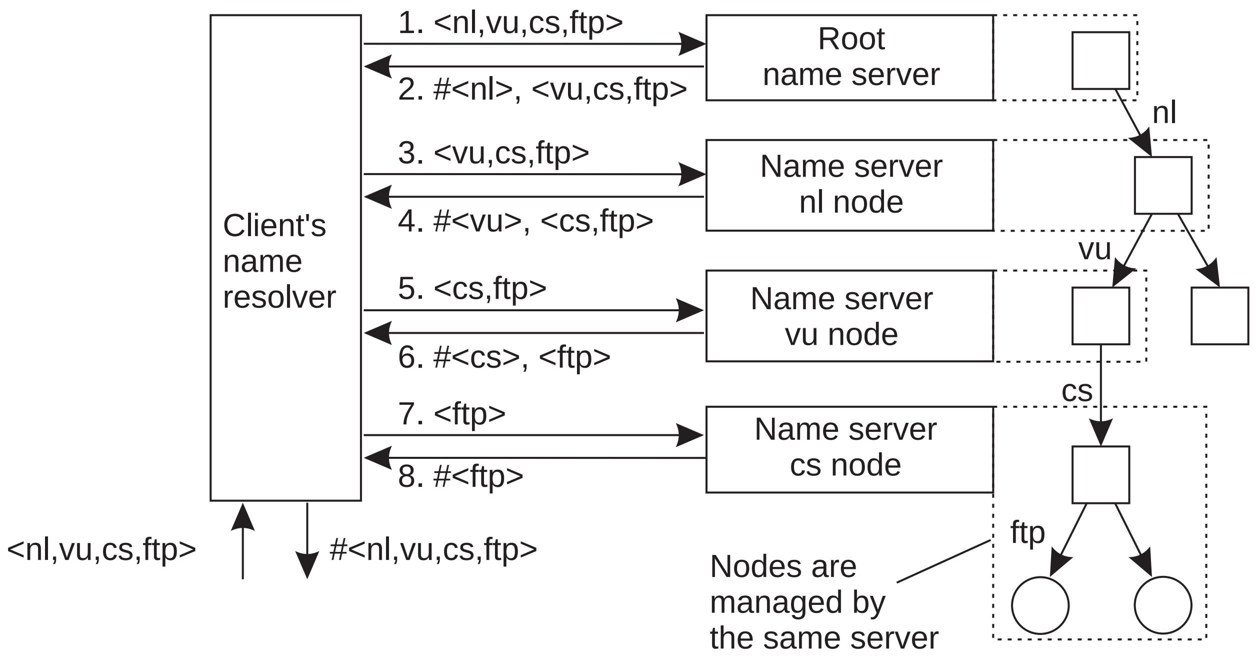


Item	Global	Administrational	Managerial
1	Worldwide	Organization	Department
2	Few	Many	Vast numbers
3	Seconds	Milliseconds	Immediate
4	Lazy	Immediate	Immediate
5	Many	None or few	None
6	Yes	Yes	Sometimes

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

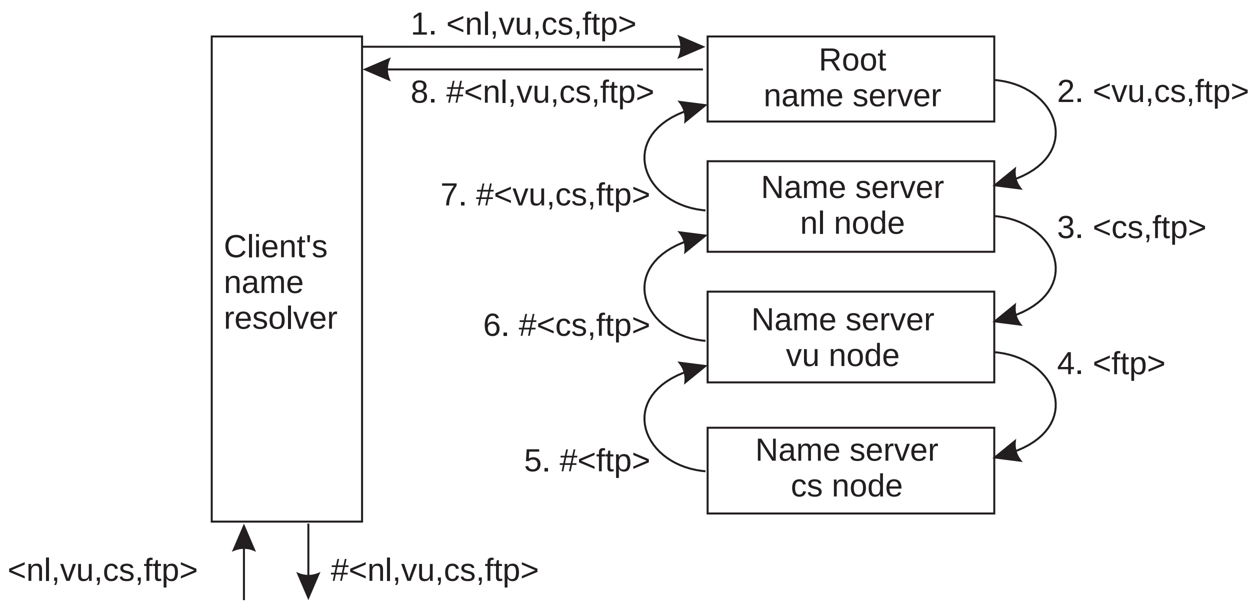
Iterative Name Resolution

- `resolve(dir,[name1,...,nameK])` is sent to `Server0` responsible for `dir`
- `Server0` resolves `resolve(dir,name1)` $\rightarrow$ `dir1`, returning the identification (address) of `Server1`, which stores `dir1`.
- Client sends `resolve(dir1,[name2,...,nameK])` to `Server1`, etc.



Recursive Name Resolution

- `resolve(dir,[name1,...,nameK])` is sent to `Server0` responsible for `dir`
- `Server0` resolves `resolve(dir,name1)` $\rightarrow$ `dir1`, and sends `resolve(dir1,[name2,...,nameK])` to `Server1`, which stores `dir1`.
- `Server0` waits for the result from `Server1`, and returns it to the client.



Caching in Recursive Name Resolution

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 (1/2)

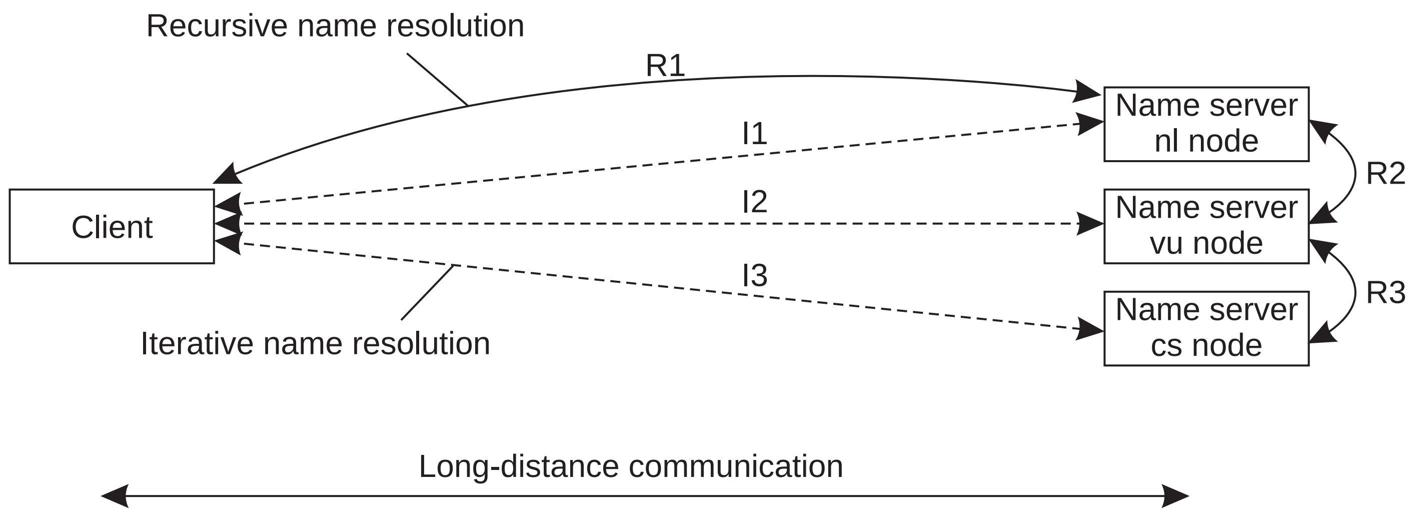
Size scalability: We need to ensure that servers can handle a large number of requests per time unit $\Rightarrow$ high-level servers are in big trouble.

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.

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 (2/2)

Geographical scalability: 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 may, in principle, be located anywhere, we introduce an implicit location dependency in our naming scheme.

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.

Main drawback: You can't ask for all nodes in a subdomain (but very few people were doing this anyway).

Information in a node: Typically what you find in a DNS record, of which there are different kinds:

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

DNS on Pastry

Pastry: DHT-based system that works with **prefixes** of keys. Consider a system in which keys come from a 4-digit number space. A node with ID 3210 keeps track of the following nodes:

n_0	a node whose identifier has prefix 0
n_1	a node whose identifier has prefix 1
n_2	a node whose identifier has prefix 2
n_{30}	a node whose identifier has prefix 30
n_{31}	a node whose identifier has prefix 31
n_{33}	a node whose identifier has prefix 33
n_{320}	a node whose identifier has prefix 320
n_{322}	a node whose identifier has prefix 322
n_{323}	a node whose identifier has prefix 323

Note: Node 3210 is responsible for handling keys with prefix 321. If it receives a request for key 3012, it will forward the request to node n_{30}.

DNS: A node responsible for key k stores DNS records of names with hash value k.

Replication of Records (1/2)

Definition: replicated at level i – record is replicated to all nodes with i matching prefixes. **Note:** # hops for looking up record at level i is generally i.

Observation: Let x_i denote the fraction of most popular DNS names of which the records should be replicated at level i, then:

$$x_i = \left[\frac{d^i(\log N - C)}{1 + d + \cdots + d^{\log N - 1}} \right]^{1/(1-\alpha)}$$

with N is the total number of nodes, $d = b^{(1-\alpha)/\alpha}$ and $\alpha \approx 1$, assuming that popularity follows a **Zipf distribution**:

The frequency of the n-th ranked item is proportional to $1/n^\alpha$

Replication of Records (2/2)

What does this mean? If you want to reach an average of $C = 1$ hops when looking up a DNS record, then with $b = 4$, $\alpha = 0.9$, $N = 10{,}000$ and $1{,}000{,}000$ records that

61	most popular records should be replicated at level 0
284	next most popular records at level 1
1323	next most popular records at level 2
6177	next most popular records at level 3
28826	next most popular records at level 4
134505	next most popular records at level 5

the rest should not be replicated

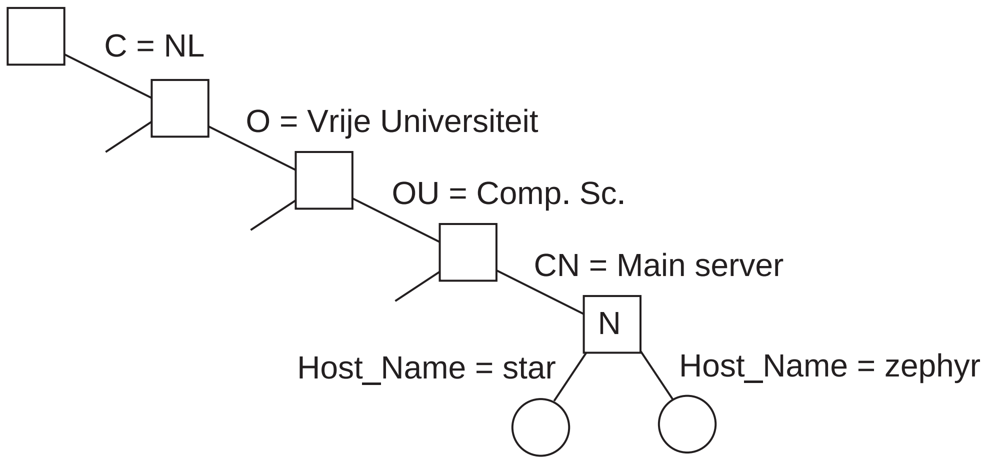
Attribute-Based Naming

Observation: In many cases, it is much more convenient to name, and look up entities by means of their **attributes** $\Rightarrow$ 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 $\Rightarrow$ inspect all entities (in principle).

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

Example: LDAP



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

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

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