Distributed Systems
Principles and Paradigms

Chapter 04
(version 21st September 2001)

Maarten van Steen
Vrije Universiteit Amsterdam, Faculty of Science
Dept. Mathematics and Computer Science
Room R4.20. Tel: (020) 444 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:

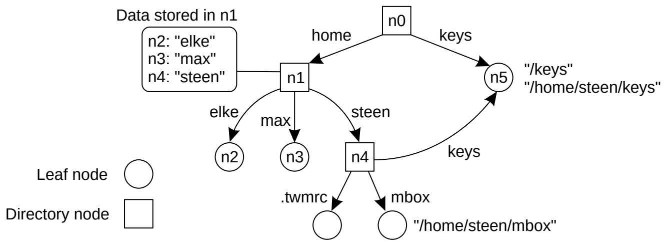
P1 Each identifier refers to at most one entity
P2 Each entity is referred to by at most one identifier
P3 An identifier always refers to the same entity (prohibits reusing an identifier)

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?

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.

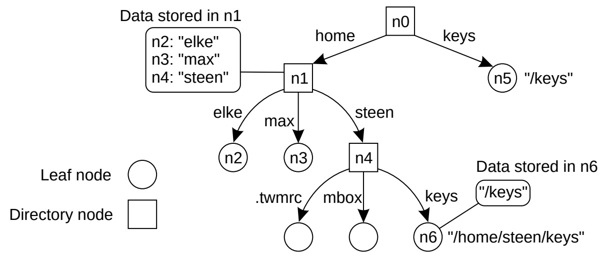
Soft link: Allow a node O to contain a *name* of another node:

- 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

Merging Name Spaces (1/3)

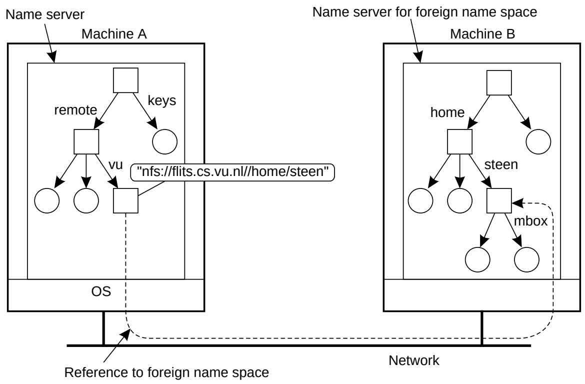
Problem: We have different name spaces that we wish to access from any given name space.

Solution 1: Introduce a naming scheme by which pathnames of different name spaces are simply concatenated (URLs).

ftp://ftp.cs.vu.nl/pub/steen/	
ftp	Name of protocol used to talk with server
://	Name space delimiter
ftp.cs.vu.nl	Name of a node representing an FTP server, and containing the IP address of that server
/	Name space delimiter
pub/steen/	Name of a (context) node in the name space rooted at the context node mapped to the FTP server

Merging Name Spaces (2/3)

Solution 2: Introduce nodes that contain the name of a node in a "foreign" name space, along with the information how to select the initial context in that foreign name space (Jade).

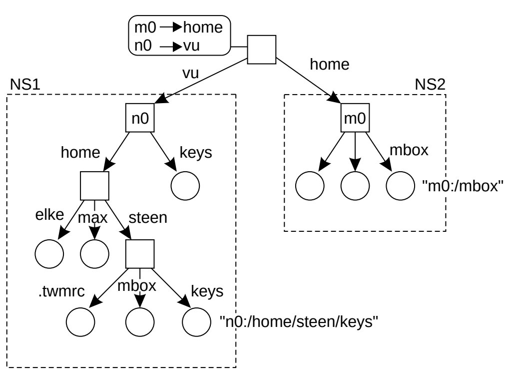


Mount point: (Directory) node in naming graph that refers to other naming graph

Mounting point: (Directory) node in other naming graph that is referred to.

Merging Name Spaces (3/3)

Solution 3: Use only *full pathnames*, in which the starting context is explicitly identified, and merge by adding a new root node (DEC's Global Name Space).



Note: In principle, you *always* have to start in the new root

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.

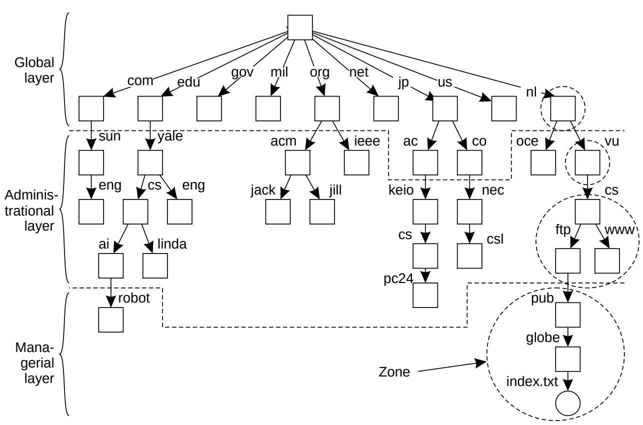
Consider a hierarchical naming graph and distinguish three levels:

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)

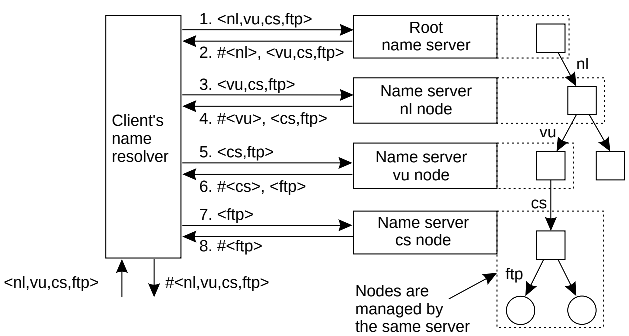


Item	Global	Administrational	Managerial
1	Worldwide	Organization	Department
2	Few	Many	Vast numbers
3	Seconds	Milliseconds	Immediate
4	Lazy	Immediate	Immediate
5	Many	None or few	None
6	Yes	Yes	Sometimes

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

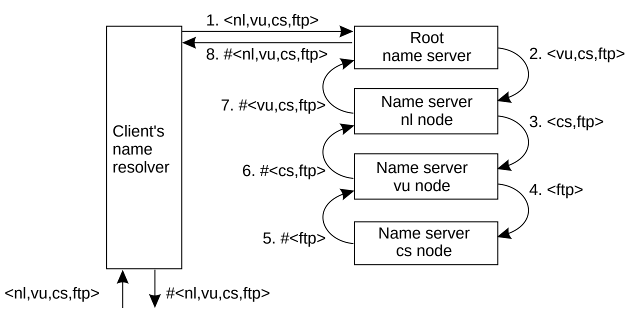
Iterative Name Resolution

- 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 ⇒ 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.

Problem: By mapping nodes to servers that may, in principle, be located anywhere, we introduce an implicit location dependency in our naming scheme.

Solution: No general one available yet.

Locating Mobile Entities

- Naming versus locating objects
- Simple solutions
- Home-based approaches
- Hierarchical approaches

Naming & Locating Objects (1/2)

Location service: Solely aimed at providing the addresses of the *current* locations of entities.

Assumption: Entities are mobile, so that their current address may change frequently.

Naming service: Aimed at providing the content of nodes in a name space, given a (compound) name. Content consists of different (attribute,value) pairs.

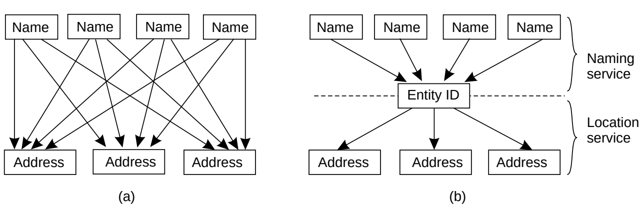
Assumption: Node contents at global and administrational level is relatively stable for scalability reasons.

Observation: If a traditional naming service is used to locate entities, we also have to assume that node contents at the managerial level is stable, as we can use only names as identifiers (think of Web pages).

Naming & Locating Objects (2/2)

Problem: It is not realistic to assume stable node contents down to the local naming level

Solution: Decouple naming from locating entities



Name: Any name in a traditional naming space

Entity ID: A true identifier

Address: Provides *all* information necessary to contact an entity

Observation: An entity's name is now completely independent from its location.

Question: What may be a typical address?

Simple Solutions for Locating Entities

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.

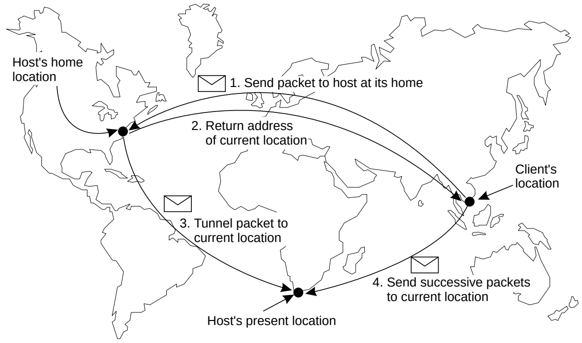
- Dereferencing can be made entirely transparent to clients by simply following the chain of pointers
- Update a client's reference as soon as present location has been found
- Geographical scalability problems:
  - Long chains are not fault tolerant
  - Increased network latency at dereferencing

  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



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

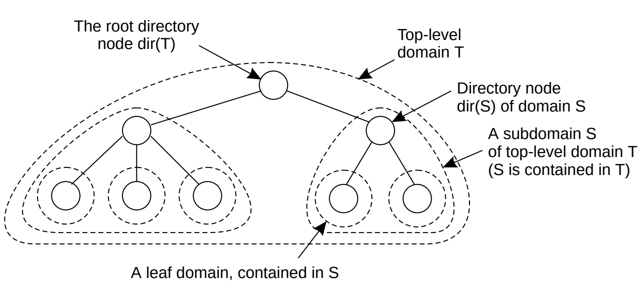
Problems with home-based approaches:

- 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?

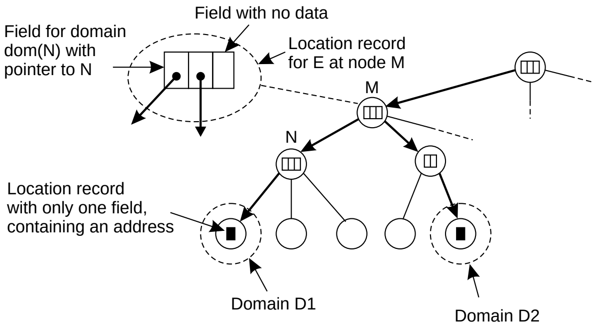
Hierarchical Location Services (HLS)

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.



HLS: Tree Organization

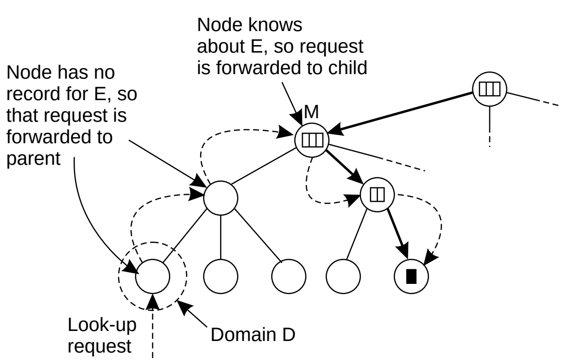
- The address of an entity is stored in a leaf node, or in an intermediate node
- Intermediate nodes contain a pointer to a child if and only if the subtree rooted at the child stores an address of the entity
- The root knows about all entities



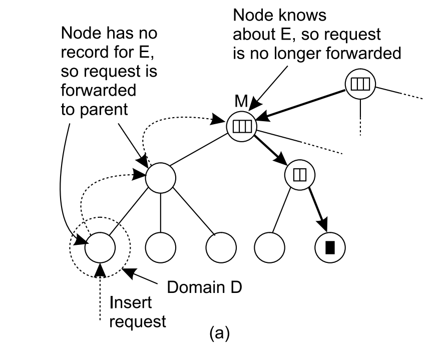
HLS: Lookup Operation

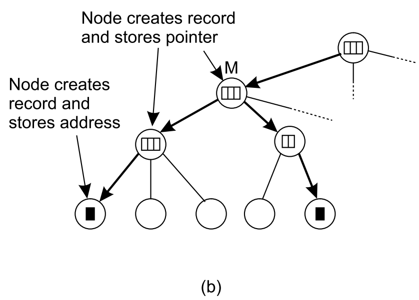
Basic principles:

- 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



HLS: Insert Operation

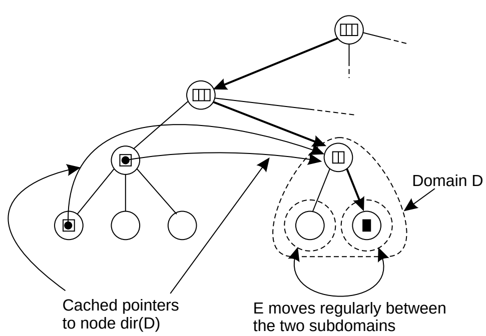




HLS: Record Placement

Observation: If an entity E moves regularly between leaf domains D_1 and D_2, it may be more efficient to store E's contact record at the least common ancestor LCA of $\mathsf{dir}(D_1)$ and $\mathsf{dir}(D_2)$.

- Lookup operations from either D_1 or D_2 are on average cheaper
- Update operations (i.e., changing the current address) can be done directly at LCA
- Note: assuming that E generally stays in $\mathsf{dom}(\mathsf{LCA})$, it does make sense to cache a **pointer** to LCA



HLS: Scalability Issues

Size scalability: Again, we have a problem of overloading higher-level nodes:

- Only solution is to partition a node into a number of subnodes and evenly assign entities to subnodes
- Naive partitioning may introduce a node management problem, as a subnode may have to know how its parent and children are partitioned.

Geographical scalability: We have to ensure that lookup operations generally proceed monotonically in the direction of where we'll find an address:

- If entity E generally resides in California, we should not let a root subnode located in France store E's contact record.
- Unfortunately, subnode placement is not that easy, and only a few tentative solutions are known

Reclaiming References

- Reference counting
- Reference listing
- Scalability issues

Unreferenced Objects: Problem

Assumption: Objects may exist only if it is known that they can be contacted:

- Each object should be named
- Each object can be located
- A reference can be resolved to client–object communication

Problem: Removing unreferenced objects:

- How do we know when an object is no longer referenced (think of cyclic references)?
- Who is responsible for (deciding on) removing an object?

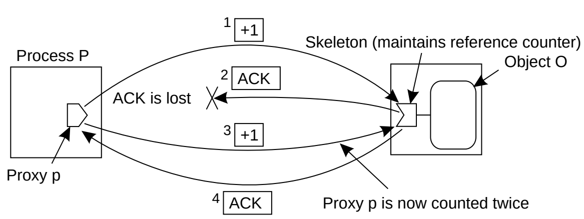
Reference Counting (1/2)

Principle: Each time a client creates (removes) a reference to an object O, a reference counter local to O is incremented (decremented)

Problem 1: Dealing with lost (and duplicated) messages:

- An `increment` is lost so that the object may be prematurely removed
- A `decrement` is lost so that the object is never removed
- An ACK is lost, so that the increment/decrement is resent.

Solution: Keep track of duplicate requests.



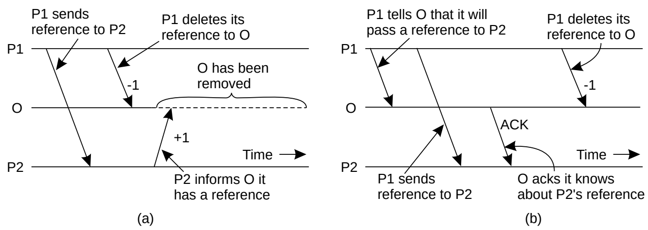
Reference Counting (2/2)

Problem 2: Dealing with duplicated references – client P_1 tells client P_2 about object O:

- Client P_2 creates a reference to O, but dereferencing (communicating with O) may take a long time
- If the last reference known to O is removed before P_2 talks to O, the object is removed prematurely

Solution 1: Ensure that P_2 talks to O on time:

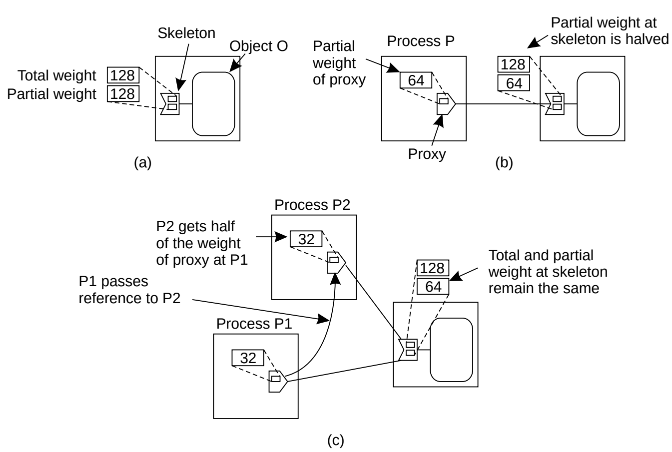
- Let P_1 tell O it will pass a reference to P_2
- Let O contact P_2 immediately
- A reference may never be removed before O has acked that reference to the holder



Weighted Reference Counting

Solution 2: Avoid increment and decrement messages:

- Let O allow a maximum M of references
- Client P_1 creates reference $\Rightarrow$ grant it $M/2$ credit
- Client P_1 tells P_2 about O, it passes half of its credit grant to P_2
- Pass *current* credit grant back to O upon reference deletion



Reference Listing

Observation: We can avoid many problems if we can tolerate message loss and duplication

Reference listing: Let an object keep a **list** of its clients:

- Increment operation is replaced by an (*idempotent*) `insert`
- Decrement operation is replaced by an (*idempotent*) `remove`

There are still some problems to be solved:

Passing references: client B has to be listed at O before last reference at O is removed (or keep a chain of references)

Client crashes: we need to remove outdated registrations (e.g., by combining reference listing with leases)

Leases

Observation: If we cannot be exact in the presence of communication failures, we will have to tolerate some mistakes

Essential issue: We need to avoid that object references are never reclaimed

Solution: Hand out a lease on each new reference:

- The object promises not to decrement the reference count for a specified time
- Leases need to be *refreshed* (by object or client)

Observations:

- Refreshing may fail in the face of message loss
- Refreshing can tolerate message duplication
- Does not solve problems related to cyclic references