Naming

Topics to be covered

Naming Entities
Locating Mobile Entities
Removing Unreferenced Entities

Naming Entities

Definitions
Name Resolution
Name Spaces
Examples (DNS, X.500)

Topics to be covered

- Names are used to uniquely identify resources/services.
- Name resolution: process to determine the actual entity that a name refers to.
- In distributed settings, the naming system is often provided by a number of sites.

Name: strings of bits of characters used to denote an entity

What is an entity in a distributed system? Resources (hosts, printers, etc) processes, users, newsgroup, web pages, network connections, etc)

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. (**address** is the name of an access point)

- An entity can offer more than one access points.
- An entity may change its access point.

- A **location-independent** name for an entity *E*, is independent from the addresses of the access points offered by *E*.
- Addresses: 32/64 bit-string (48bits for Ethernet addresses) vs. User-friendly names or **human-friendly names** (in Unix each file can have up to 255 bytes name).
- **Identifiers**: special type of name with the following properties
  - an id refers to at most one entity
  - each entity is referred to by at most one id
  - an id always refers to the same entity (i.e., it is never reused)

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.

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 Spaces

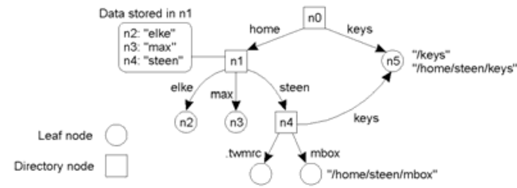
Names organized into a **name space**

A name space can be represented as a labeled directed graph (**naming graph**)

- a leaf node represents a (named) entity
- a **directory node** refers to other nodes; stores a (directory) table of *(edge label, node identifier)* pairs.

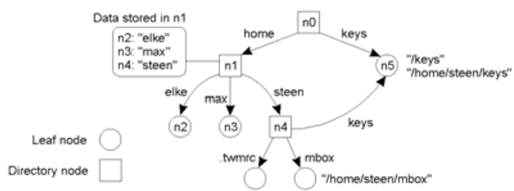
Path name: sequence of labels

- Absolute (first node in the path is the root)
- Relative (otherwise)



- Each node in a naming graph is consider as an entity, it has an associated id

Name Spaces



- Global vs local name
  Global name denotes the same entity (i.e., always interpeted with respect to the same directory name
  Local name: its interpretation depends on where the name is being used (context)
- Close to what is implemented in many file systems
- More than one paths to a node
- Naming graph: tree (hierarchical), more than one roots, DAG

Name Spaces

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

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

Name Spaces

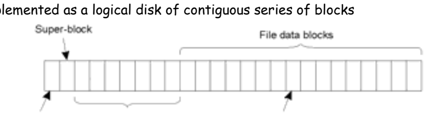
UNIX File Systems

Directory node: file directory

Leaf node: file

Root node (single): root directory

Implemented as a logical disk of contiguous series of blocks



Boot block: used to load the OS

Super-block: contains information about the whole file system

i-nodes: contains information on where the data of its associated file can be found on disk; referred to by an index number (index no 0 represents the root directory)

Directories implemented as files; Root directory contains a mapping between file names and index numbers of i-nodes

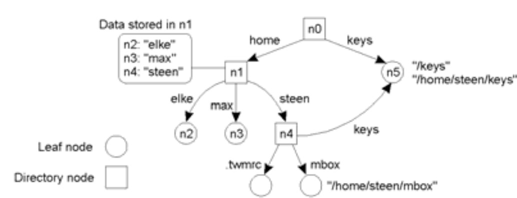
Index of an i-node: id of a node in the naming graph

Name Resolution

Name resolution: given a path name: lookup information stored in the node referred to by that name

Example: n1:<steen, mbox>

Name lookup at each node (access the directory table of the node) returns the id of a node where name resolution continues (in UNIX?)



Name Resolution: Closure Mechanisms

Problem: Where and how to start name resolution

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

How to do it:
UNIX file system (example /home/steen/mbox)
Dial a phone number

Local names
Example environmental variables, e.g., HOME (refers to the user's local directory)

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

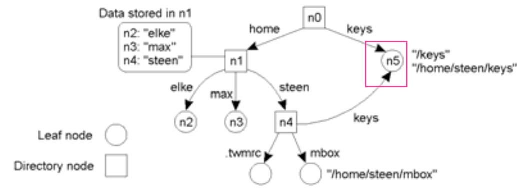
Name Linking

Alias: another name for the same entity

Solution 1

Multiple absolute paths to the same node (**hard links**)

In term of i-nodes?

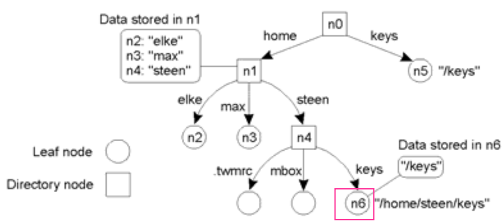


Name Linking

Solution 2

Allow a leaf node to contain an absolute path name (**symbolic link**)

In terms of i-nodes?



Merging Name Spaces

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

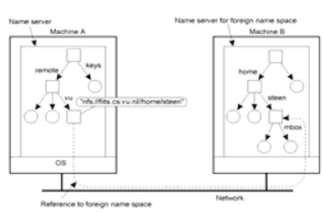
Solution 1: Mounting

A directory node called a **mount point** stores the id of (or all the necessary information for identifying and accessing) a directory node from a *foreign name space* called **mounting point**

Merging Name Spaces

In general, if two name spaces exist NS1, NS2 – to mount a foreign entity in a *Distributed System*, we require at least the following information:

1. Name the access protocol (resolved to the implementation of a communication protocol)
2. Name the server (resolved to an address where the server can be reached)
3. Name the mounting point (resolved to a node id in the foreign name space, by the foreign server)

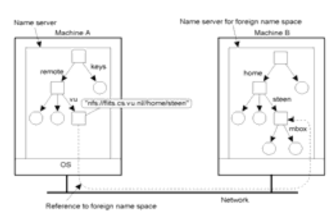


Merging Name Spaces

NFS: distributed file system that describes exactly how a client can access a file stored on a (remote) NFS file server

Through a NFS URL

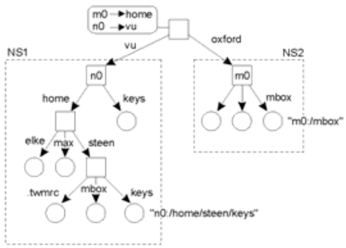
file
nfs://flits.cs.vu.nl//homes/steen
protocol NFS file server



/remote: includes mount points for foreign name spaces (in the form of NFS URLs)

remote.vu/mbox

Solution 3: Add a new root node and makes the existing root nodes its children (example DEC's Global Name Service (GNS))



Problem: existing names need to be changed

home/steen -> /vu/home/steen

Solution: New root includes a mapping (old roots, their new names). (Implicitly) include the identifier of the node (old root) from where resolution should start

/home/steen/keys->

n0:/home/steen/keys

Root node becomes a bottleneck

Merging Name Spaces

The Merging Name Spaces Problem

Solution 1: Mounting

Solution 2: Introduce a naming scheme by which pathnames of different name spaces are simply concatenated (URLs).

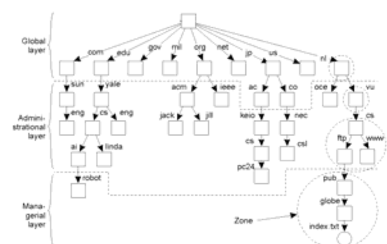
Solution 3: Add a new root node

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 (root and its children). Stability (rarely change) 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. Typically change regularly. For example, hosts in a local area network. Main issue is effectively mapping directory nodes to local name servers.

Example: DNS name space.



The name space is divided into non overlapping parts, called **zones**, each implemented by a separate name server

Name Space Implementation

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

Iterative Name Resolution

Issue: Name resolution when the name space is distributed across multiple name servers

Assume: no replication or client-side caching

Each client has access to a local *name resolver*

Example

root:<nl, vu, cs, ftp, pub, globe, index.txt>

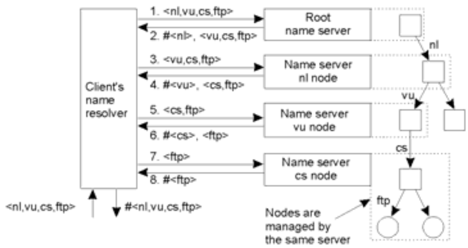
Using URL notation

ftp://ftp.cs.vu.nl/pub/globe/index.txt

Method 1: **Iterative Name Resolution**

The name resolver hands over the complete name to the root server

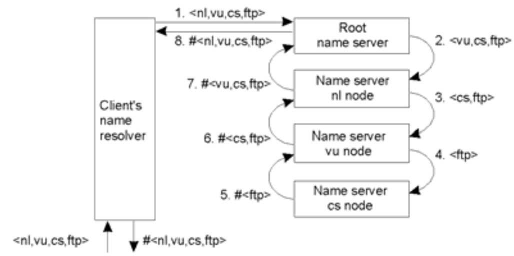
The root server resolves the path name as far as it can and returns the result addr1 (address of the associated name server) to the client, then the client passes the remaining path name server to the addr1 and so on



Note: the last resolution step (contacting the ftp server and asking it to transfer the indicated file) is generally carried out as a separate process by the client

Method 2: **Recursive Name Resolution**

Instead of returning each intermediate result back to the client's name resolver, each name server passes the result to the next name server it finds



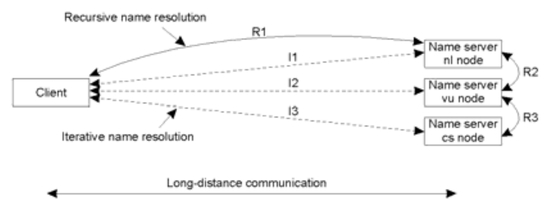
- Puts a higher performance demand on each name server
- Caching results is more effective
- Communication costs may be reduced

What can be cached:

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>
ni	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<cs> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
root	<ni,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>	#<nl> #<nl,vu> #<nl,vu,cs> #<nl,vu,cs,ftp>

Caching with iterative name resolution?

Implement a local intermediate name server shared by all clients (besides caching, only this server needs to know the root name server)



Size scalability: We need to ensure that servers can handle a large number of requests per time unit

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.

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.

The DNS Name Space

Internet Domain Name Server (DNS): the largest distributed name service in use

Primarily used to lookup host addresses and mail servers

The DNS name space is hierarchically organized as a rooted tree

Path name root:<nl, vu, cs, flits> represented as flits.cs.vu.nl. (rigtmost dot indicates the root)

Name of the node: label of the incoming edge

Domain: subtree

Domain name: path name to its root node

Contents of a node: a collection of resource records

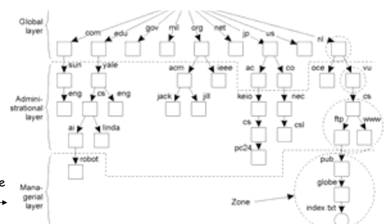
The most important types of resource records forming the contents of nodes in the DNS name space.

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

Type of record	Associated entity	Description
SOA (start of authority)	Zone	Holds information on the represented zone (such as an email address of the system administrator, host name from where data on the node can be fetched, etc)
A (address)	Host	Contains an IP address of the host this node represents
MX (mail exchange)	Domain	A symbolic link to a node representing a mail server Example: the node representing the domain cs.vu.nl has an MX record zephyr.cs.vu.nl
SRV	Domain	Contain the name of a server for a specific service The service is identified by a name + name of a protocol Example: the web server of the cs.vu.nl domain named http.tcp.cs.vu.nl refer to the actual server soling.cs.vu.nl
NS (name server)	Zone	Refers to a name server that implements the represented zone

Type of record	Associated entity	Description
CNAME (canonical names)	Node	Each host is assumed to have a canonical or primary name. An alias is implemented by means of node storing a CNAME record containing the canonical name of a host (symbolic link)
PTR (pointer)	Host	Inverse mapping of IP addresses to host names For instance, for host www.cs.vu.nl with IP 130.37.24.11 the DNS creates a node named 11.24.37.130.in-addr.arpa used to save the canonical name of the host (solings.cs.vu.ntl) in a PTR record
HINFO	Host	Holds information on the host this node represents (e.g., machine type, operating system)
TXT	Any kind	Contains any entity-specific information considered useful

Generally formed by file system; formally, not part of the DNS



The name space is divided into non overlapping parts, called **zones**, each implemented by a separate name server; almost always replicated

Updates by modifying the DNS database local to the primary server

Secondary name servers through zone transfer

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

cs.vu.nl (zone and domain)

start.cs.vu.nl (name server for the zone)

Name	Record type	Record value
cs.vu.nl	SOA	star (1999121502,7200,3600,2419200,86400)
cs.vu.nl	NS	star.cs.vu.nl
cs.vu.nl	NS	top.cs.vu.nl
cs.vu.nl	NS	solo.cs.vu.nl
cs.vu.nl	TXT	"Vrije Universiteit - Math. & Comp. Sc."
cs.vu.nl	MX	1 zephyr.cs.vu.nl
cs.vu.nl	MX	2 tornado.cs.vu.nl
cs.vu.nl	MX	3 star.cs.vu.nl
star.cs.vu.nl	HINFO	Sun Unix
star.cs.vu.nl	MX	1 star.cs.vu.nl
star.cs.vu.nl	MX	10 zephyr.cs.vu.nl
star.cs.vu.nl	A	130.37.24.6
star.cs.vu.nl	A	192.31.231.42
zephyr.cs.vu.nl	HINFO	Sun Unix
zephyr.cs.vu.nl	MX	1 zephyr.cs.vu.nl
zephyr.cs.vu.nl	MX	2 tornado.cs.vu.nl
zephyr.cs.vu.nl	A	192.31.231.66
www.cs.vu.nl	CNAME	soling.cs.vu.nl
ftp.cs.vu.nl	CNAME	soling.cs.vu.nl
soling.cs.vu.nl	HINFO	Sun Unix
soling.cs.vu.nl	MX	1 soling.cs.vu.nl
soling.cs.vu.nl	MX	10 zephyr.cs.vu.nl
soling.cs.vu.nl	A	130.37.24.11
laser.cs.vu.nl	HINFO	PC MS-DOS
laser.cs.vu.nl	A	130.37.30.32
vucs-das.cs.vu.nl	PTR	0.26.37.130.in-addr.arpa
vucs-das.cs.vu.nl	A	130.37.26.0

Name	Record type	Record value
cs.vu.nl	NS	solo.cs.vu.nl
solo.cs.vu.nl	A	130.37.21.1

Part of the description for the *vu.nl* domain which contains the *cs.vu.nl* domain.

Directory Service: special form of a naming service, lookup an entity based on a description of properties (rather than name)

Similar to Yellow-Page look up.

X.500 consists of a number of records (directory entries) <*attribute, value*>
Each attribute has a type, multiple-valued attributes

Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	L	Vrije Universiteit
OrganizationalUnit	OU	Math. & Comp. Sc.
CommonName	CN	Main server
Mail_Servers	--	130.37.24.6, 192.31.231,192.31.231.66
FTP_Server	--	130.37.21.11
WWW_Server	--	130.37.21.11

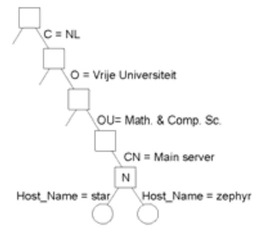
A simple example of a X.500 directory entry using X.500 naming conventions.

The collection of all entries makes up the **Directory Information Base (DIB)**
Each record is uniquely named and can be looked up
A globally unique names: a sequence of naming attributes
Each naming attribute called RDN (**Relative Distinguished Name**).

Example: /C=NL/O=Vrije Universiteit/OU=Math & Comp. Sc./CN=Main server

Directory Information Tree (DIT)

A node is both a directory and an X.500 record



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

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Math. & Comp. Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	192.31.231.66

Two directory entries having *Host_Name* as RDN.

A node is both a directory and an X.500 record

Two different lookup operations

- read: returns a single record
- list: returns the names of all outgoing edges

Example:

list(/C=NL/O=Vrije Universiteit/OU=Math & Comp. Sc./CN=Main server)

read(/C=NL/O=Vrije Universiteit/OU=Math & Comp. Sc./CN=Main server)

- DIT is "partitioned" across several servers (termed Directory Service Agents (DSA)- similar to zones in DNS)
- Clients are represented by Directory User Agents (DUA): similar to a name resolver
- What is different between X.500-DNS?
  - Provides facilities for querying a DIB, example
    - answer = search("&(C=NL)(O=Vrije Universiteit)(OU=*)(CN=Main Server))
    - Find all the "main servers" but not in a particular organizational unit
  - An operation may be "expensive" – the above will have to search all entries for all departments (access many leaf nodes) and combine the results..
- LDAP (Lightweight Directory Access Protocol) a simplified protocol used to accommodate X.500 directory services in the Internet
  - Application-level protocol on top of TCP

Locating Mobile Entities

Difference from Naming
Simple Solutions
Home-Based Approach
Hierarchical Approach

Naming versus Locating Entities

Problem: what happens and how are names are resolved when entities are mobile?

Example: move ftp.cs.uoi.gr to ftp.cs.unisa.edu.au

How is the change addressed? Create aliases!

1. Record the address of the new machine in the DNS database of cs.uoi.gr (two addresses pointing to the same node) hard link)

   Lookup ok, but what if the server moves again?

2. Record the name of the new machine in the DNS database of cs.uoi.gr (symbolic link)

   Lookup less efficient, updates ok

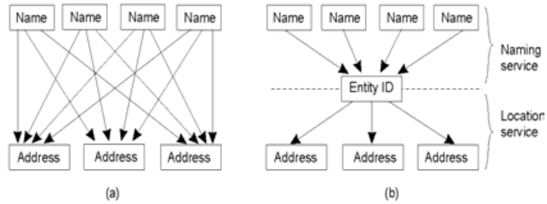
For highly mobile entities both solutions are problematic (especially if there are multiple phases in the name resolution/address determination).

The name is not allowed to change.

Naming versus Locating Entities

In (a) there is a single level mapping between names and addresses. Each time an entity changes location, the mapping needs to change!

Decouple naming from locating entities



Separate naming from locating by introducing **identifiers**.
An identifier does not have a human-friendly representation (optimized for machine processing only).
An entity's name is now completely independent from its location.

Naming versus Locating Entities

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).

Simple Solutions: Broadcasting

Broadcasting

Simply broadcast the id to each machine, each machine is requested to check whether it has that entity, and is so, send a reply message containing the address of the access point

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

Simple Solutions: Forwarding Pointers

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

Forwarding Pointers

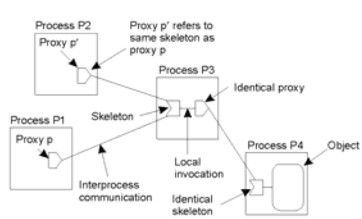
SSP chains

Forwarding pointers for distributed objects

Each forwarding pointer is implemented as a (proxy, skeleton) pair

A skeleton (i.e., server-side stub) contains a local reference to the actual object or a local reference to a proxy (i.e., client-side stub) for the object

Skeleton (entry items for remote references) Proxies (exit items)

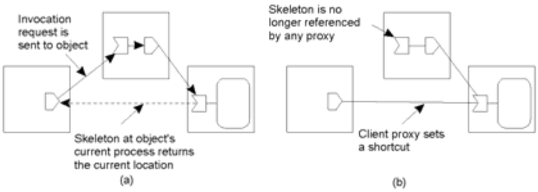


When an object moves from A to B, it leaves behind a *proxy* in its place *in A* and installs a *skeleton* that refers to it in *B*.

Transparent to the client

Forwarding Pointers

A chain (proxy, skeleton) can be short cut.
- The current location is piggybacked with the response of the distributed object.
- Send the response directly or along the reverse path?
- When no skeleton references a proxy, the skeleton can be removed.



Redirecting a forwarding pointer, by storing a shortcut in a proxy.

Home-Based Approaches

Let the home location keep track of the entity's current address

(usually where the entity was created)

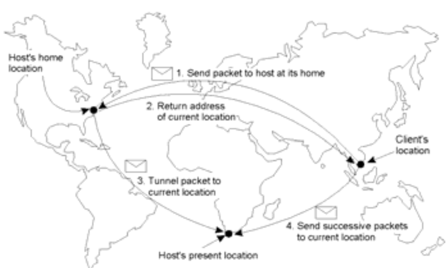
An entity's home address is registered at a naming service.

The home registers the foreign address of the entity

Clients always contact the home first, and then continue with the foreign location

Home-Based Approaches

The principle of Mobile IP



Home-Based Approaches

Two-tiered scheme

Keep track of visiting entities:
Check local visitor register first
Fall back to home location if local lookup fails

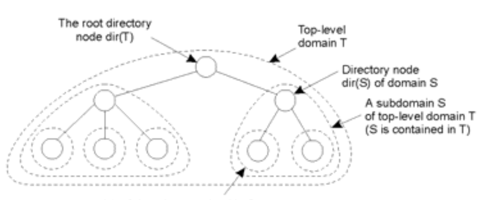
Problems with home-based approaches:

- 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 Approaches

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 dir(d)
Leaf domains typically correspond to a local-area network or a cell
The root (directory) node knows all the entities
Each entity currently in a domain D is represented by a location record in the directory node dir(D) which is the entity's current address or a pointer



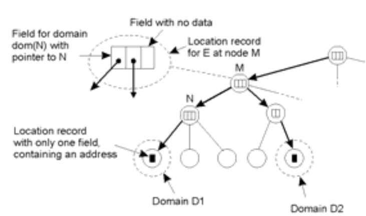
Hierarchical Approaches

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

An entity may have multiple addresses (e.g., if it is replicated

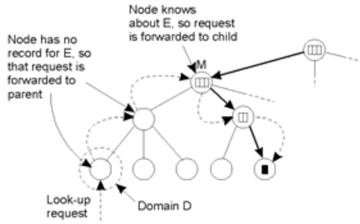


Hierarchical Approaches: Lookup

Basic principles:

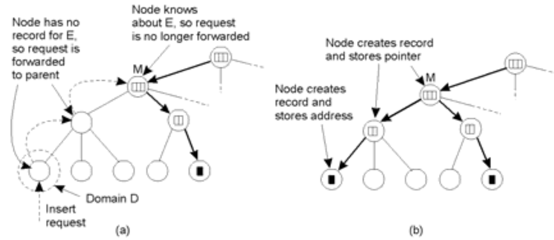
Start lookup at local leaf node
If node knows about the entity, follow downward pointer, otherwise go one level up

Upward lookup always stops at root



Exploits locality

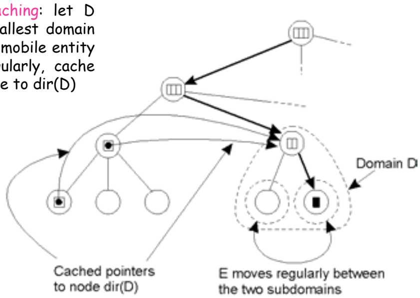
Hierarchical Approaches: Update



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.

Hierarchical Approaches: Pointer Caches

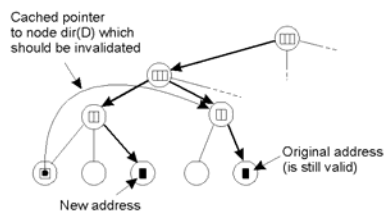
Pointer caching: let D be the smallest domain in which a mobile entity mover regularly, cache a reference to dir(D)



Caching a reference to a directory node of the lowest-level domain in which an entity will reside most of the time.

Hierarchical Architectures: Pointer Caches

- Let dir(D) not store a pointer to the subdomain where E resides but the actual address of E
- Cache invalidation:



A cache entry that needs to be invalidated because it returns a nonlocal address, while such an address is available.

Hierarchical Approaches: Scalability Issues

Size scalability

The 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: Unfortunately, subnode placement is not that easy, and only a few tentative solutions are known

Hierarchical Approaches: Scalability Issues

Root is a bottleneck. Divide the root. The scalability issues related to uniformly placing subnodes of a partitioned root node across the network covered by a location service.

Removing Unreferenced Entities

The Problem
Reference Counting
Reference Listing
Tracing

Removing Unreferenced Objects

Distributed Garbage Collection:

Remove unreferenced entities

The Problem of Unreferenced Objects

Reference through a (proxy, skeleton) pair
client-side proxy associated with a server-side skeleton

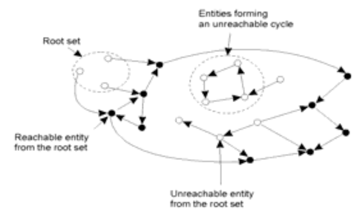
Assumption: Objects may exist only if it is known that they can be contacted (may be accessed only if there is a remote reference to it)

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?

(proxy, skeleton) pair takes care of garbage collection (transparent to objects/client)

The Problem of Unreferenced Objects

Represented by a **reference graph**, each node represents an object



Garbage collection harder in distributed systems, because of network communication (scalability/efficiency and failures)

Reference Counting

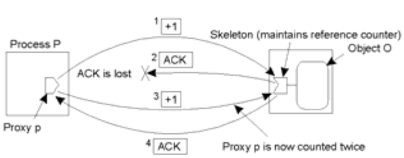
Each time a client creates (removes) a reference to an object *O*, a reference counter local to *O* (e.g., at the skeleton) is incremented (decremented)

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

P creates a reference to remote object O, installs a proxy p for O

p sends an incr message to s

s sends ACK to p



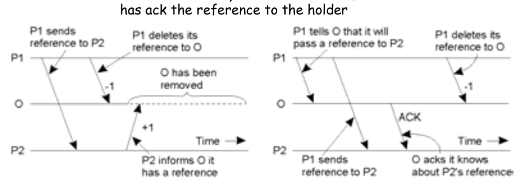
- An increment is lost so that the object may be prematurely removed
- An increment is send twice
- A decrement is lost so that the object is never removed
- An ACK is lost, so that the increment/decrement is resent.

Reference Counting

Problem 2: (copying a remote reference) process P1 passes a reference to process P2 of object O. (Skeleton s of) O does not know this, until P2 communicates with O

P2 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 *P2* talks to *O*, the object is removed prematurely Problem in (a) below

Solution 1:
- P1 tells O, it will pass a reference to P2
- O contacts P2 immediately
- A reference may never be removed, before O has ack the reference to the holder



Passing a reference requires 3 messages

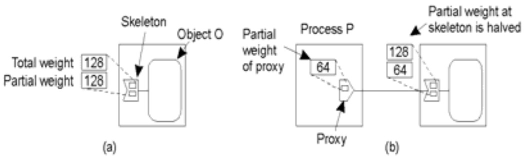
Advanced Referencing Counting

Solution 2: Avoid increment messages

Weighted reference counting

Associate a fixed **total weight** with each object O

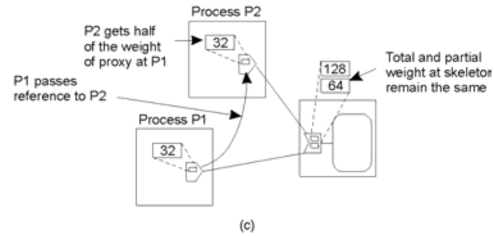
- Upon creation of O, each skeleton s of O: total weight, **partial weight** (=total weight, initially
- When a new remote reference is created, the new proxy is assigned half of the partial weight. Remaining half at the skeleton.



(a) The initial assignment of weights in weighted reference counting (b) Weight assignment when creating a new reference.

Advanced Referencing Counting

- When P1 passes O to P2, half of the partial weight of P1's proxy assigned to the copied proxy of P2
- When a reference is destroyed, dec message (including the associated partial weight) send to the skeleton



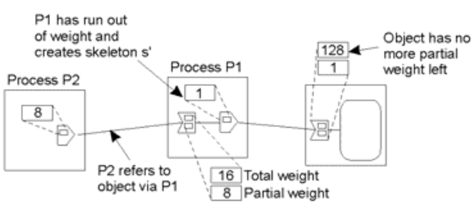
(c) Weight assignment when copying a reference.

Advanced Referencing Counting

Problem: maximum number of references

Solution: Indirection

When the partial weight of P1 reaches 1, P1 cerates a skeleton s' in its address space with an appropriate total weight



As with forwarding pointers: long chains degrade performance, more susceptible to failures

Advanced Referencing Counting

Solution: Generation Reference Counting

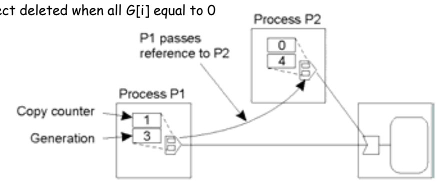
Associate with each proxy a generation number + a copy counter (number of times it has been copied)

The skeleton s maintains a table G, G[i] number of generation i copies

When p is deletes, sends message to s with generation number, k, and number of copies, n

s decrements G[k] by one, increments G[k+1] by n

Object deleted when all G[i] equal to 0



Creating and copying a remote reference in generation reference counting.

Referencing Listing

The skeleton maintains an explicit list of all proxies pointing to it

Idempodent operation: an operation that can be repeated without affecting the end result

Message to add/delete a proxy to a list as apposed to increment/decrement operations

Used in Java RMI

Leases (timeout)

Tracing

Entities that hold references to each other but none can be reached from the root

Tracing-based garbage collection: check which methods can be reached from the root and remove all others

Mark-and sweep collectors

Mark phase: follow chains of entities originated from entities in the root set and mark them

Sweep phase: exhaustively examine memory to locate entities that have not been marked

Distributed Version of Mark-and sweep collectors

A local garbage collector is started at each process, with all the collectors running in parallel

Color proxies, skeletons and the actual objects

Three colors: white, grey, black

Initially, all white

An object in P, reachable form the root in P, marked grey

When an object is marked grey, all proxies contained in that object are marked grey

When a proxy is marked grey, a message is sent to the associate skeleton to mark itself grey

The object becomes grey, when the skeleton becomes grey

When all proxies grey, the object and skeleton is marked black, and then the proxy

Mark phase ends when no greys

Remove all white objects

"Stop-the-world" assumption

- Processes (which contain objects) are hierarchically organized in groups)

- Phase 1: Initial marking, in which only skeletons are marked
- Phase 2: Intra-process propagation of marks from skeleton to proxies
- Phase 3: Inter-process propagation of marks from proxies to skeletons
- Phase 4: Stabilization by repetition of the previous two steps
- Phase 5: Garbage reclamation

For details, read the book