# 04.Naming

# Basics

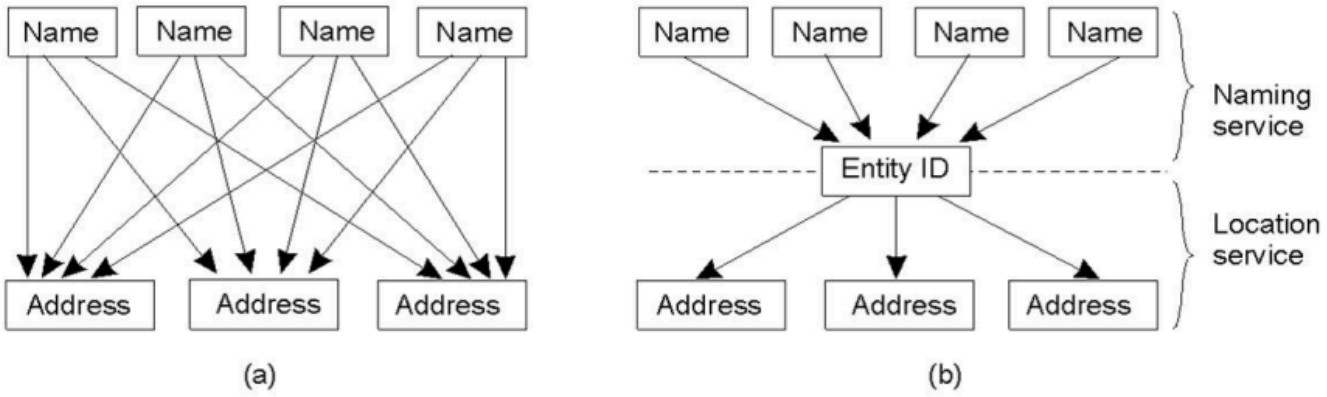
Names are used to refer to entities(Hosts, users, files, services,...). Entities can be accessed by an access point, that is a special entity characterized by an address(special case of a name).
It is not convenient to use the address of its access point as a name for an entity, it is better using location-independent names. The same entity can be accessed through several access points at the same time and it can change its access points during its lifetime.
We want to access entities through their logical names.
Global vs. local names

- A global names denotes the same entity, no matter where the name is used
- A local name is the name representing an entity in the context it is used
- Interpretation of a local name depends on where it is being used
  Human-friendly names vs. machine-friendly names
- 00:15:2A:2E:E1:97 vs “My Mobile”

## Identifiers

Resolving a name directly into an address does not work with mobility. You can have multiple addresses for the name of an entity. We use identifiers to link addresses to the name.
Identifiers are names such that:

- They never change (during the lifetime of the entity)
- Each entity has exactly one identifier
- An identifier for an entity is never assigned to another entity
  Using identifiers enables to split the problem of mapping a name to an entity and the problem of locating the entity




## Name resolution

It is the process of obtaining the address of a valid access point of an entity having its name
The way name resolution is performed depends on the nature of the naming schema employed

- Flat naming
- Structured naming
- Attribute-based naming

## Flat naming

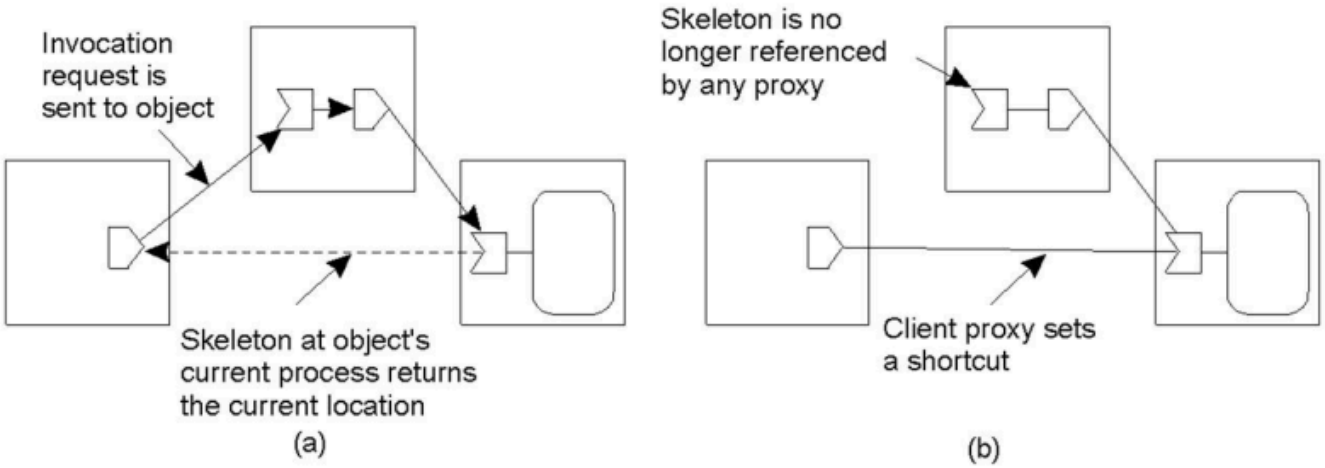
In a flat naming schema names are flat, you cannot distinguish part in it or the part are meaningless to distinguish something.
Various ways to resolve it:

- Simple Solutions:
  - Designed for small-scale (e.g., LAN) environments
  - Broadcast: Similar to ARP, send “find” message on broadcast channel: Only the interested host replies. Drawback: All hosts must process all “find” messages
  - Multicast: Same as broadcast, but send to a multicast address to reduce the scope of the search
  - Forwarding pointers (for mobile nodes)

## Forwarding pointers and distributed objects

Proxies forward requests to the real object instance

- This scheme works fine also w.r.t. parameter passing
  Keep chain short by adjusting proxies
- Send adjustment along path or direct to original proxy?
- Skeletons know when they are no longer referenced by a proxy and can be deleted




When the object moves leave a proxy that connect to the new skeleton. This could conclude the invocation or the system bring back the original address and so the message goes back to the invoker directly taking along the information of the new address.

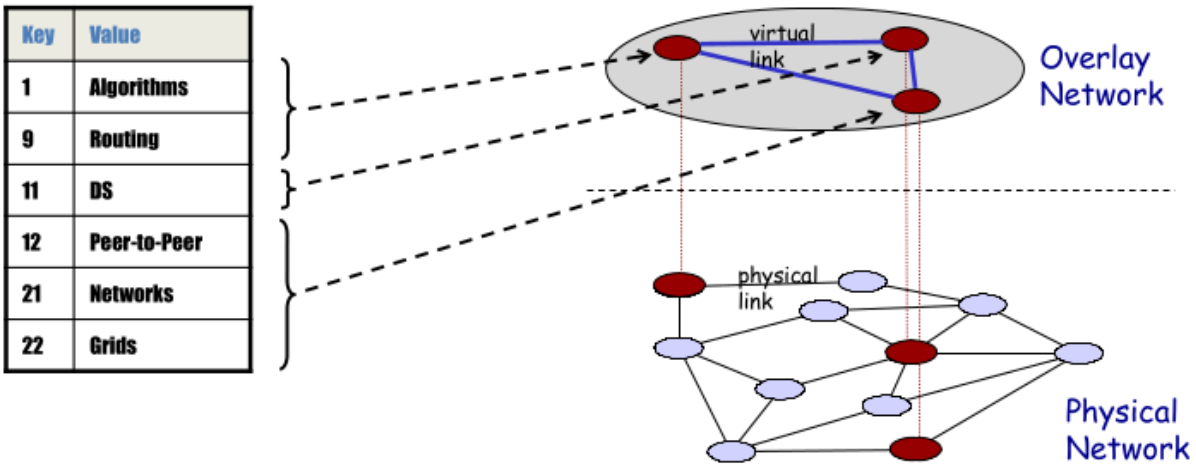
## Home-based approaches

We can solve the problem of mobile nodes relying on fixed places that knows at all time where the mobile node is and so connect to it the requests.
The problems that they take along are:

- The extra step towards the home increases latency. The setup need more time
- 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)

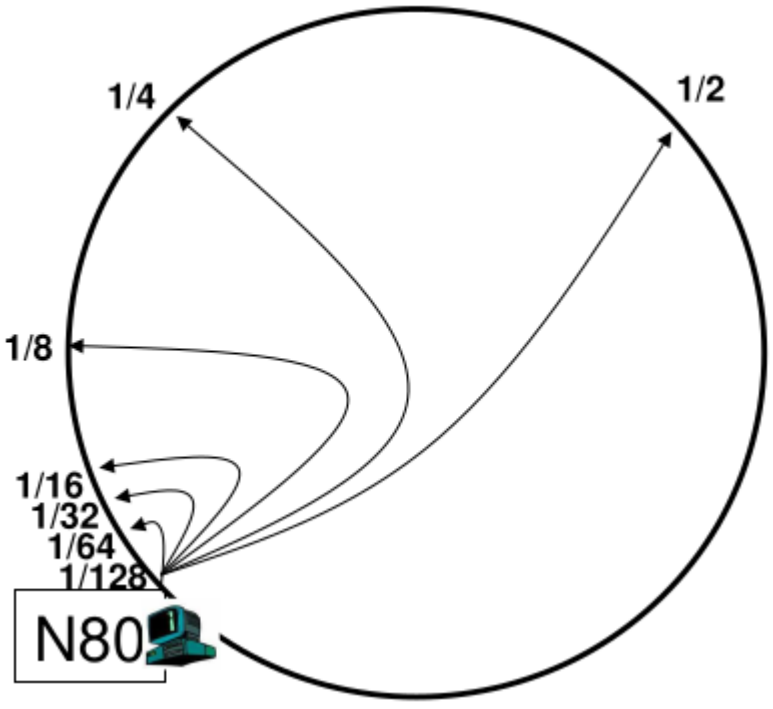
## DHT

Resolve flat names also in huge systems. It uses an hash table.




An example is Chord:
Nodes and keys are organized in a logical ring

- Each node is assigned a unique m-bit identifier
  Usually the hash of the IP address
- Every item is assigned a unique m-bit key
  Usually the hash of the item
- The item with key k is managed (e.g., stored) by the node with the smallest id $\geq$ k (the successor)
  Each node keeps track of its successor
  Search is performed “linearly”



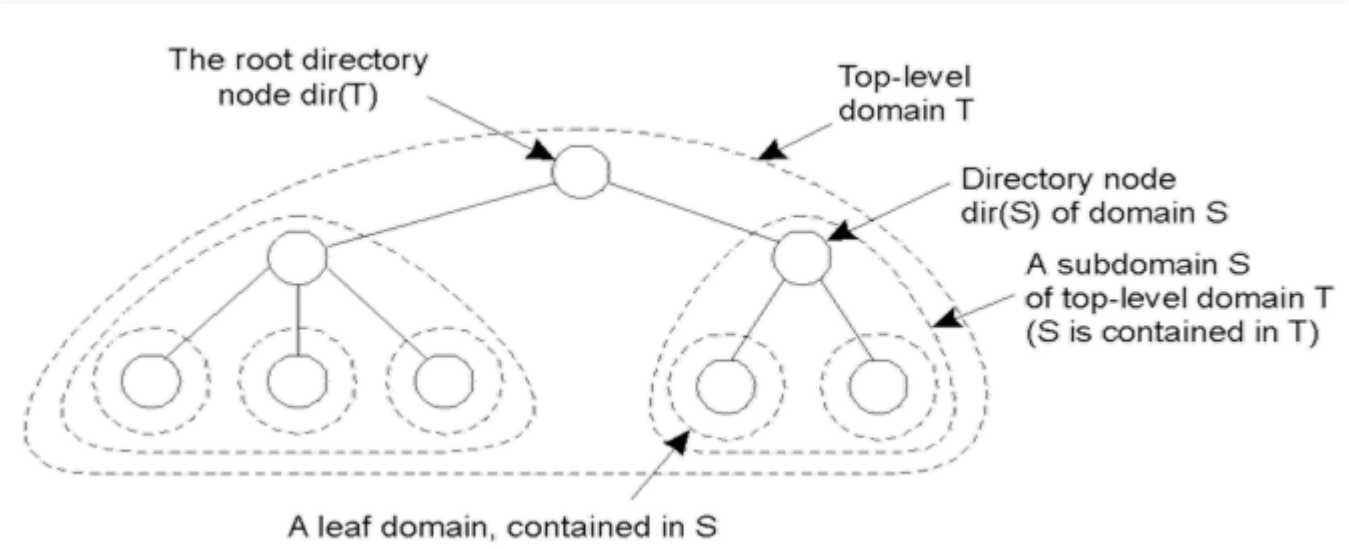

Upon receiving a query for an item with key k, a node:

1. Checks whether it stores the item locally

2. If not, forwards the query to the largest node in the $2^{nd}$ column of its successor table that does not exceed k

## Hierarchical approaches

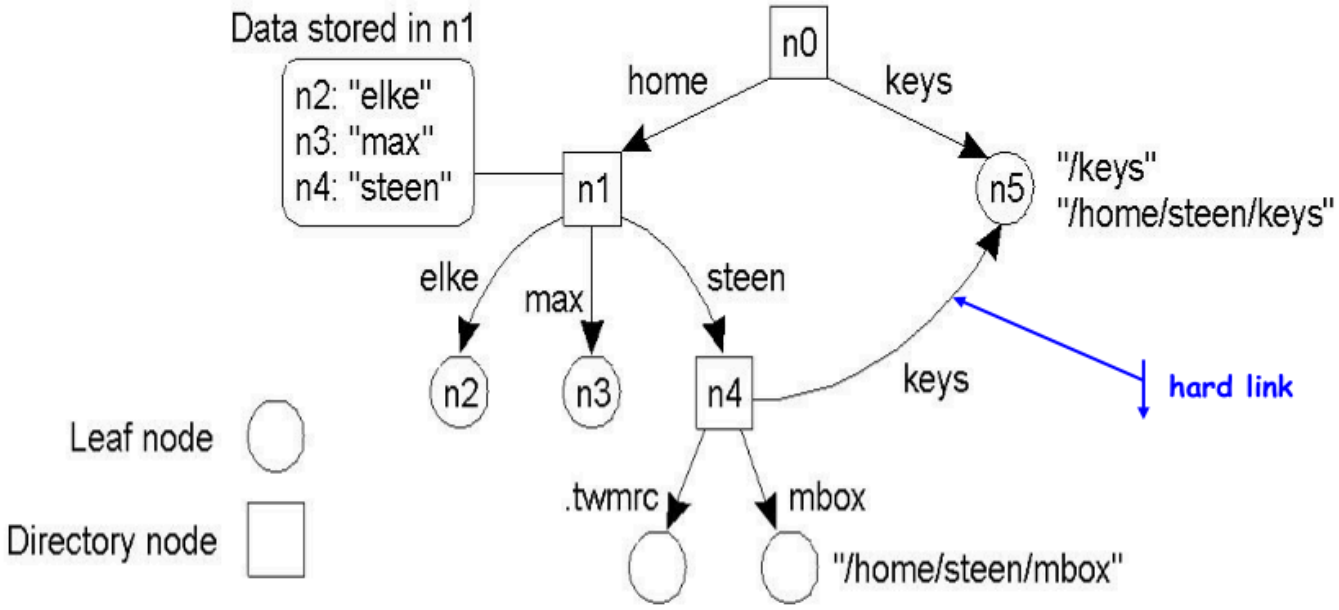
You can use multiple servers organized in a tree and when you store a name you contact the node near you that will forward the information to the other nodes going up. When you want to resolve names you go to the local name server and ask if it knows how to reach the name: if it knows it respond otherwise it ask to the node near it(we will reach the root only if we are at opposite side of the network, the more you go up the more the query slow down). This system has good locality.




To speed up the look up you can use caching, but you need to take in count that the caches are small in size.

# Structured naming

We abandon the idea of flat names. In a structured naming system names are organized in a name space. A name space is a labeled graph composed of leaf nodes and directory nodes.
A leaf node represents a named entity: it stores information about the entity it refers to. They include at least its identifiers/address. A directory node has a number of labeled outgoing edges, each pointing to a different node. Resources are referred through path names. Path names can be absolute or relative. Multiple path names may refer to the same entity (hard linking) or leaf nodes may store absolute path names of the entity they refer to instead of their identifier/address (symbolic linking). The graph doesn't have any relation to the physical structure of the system.

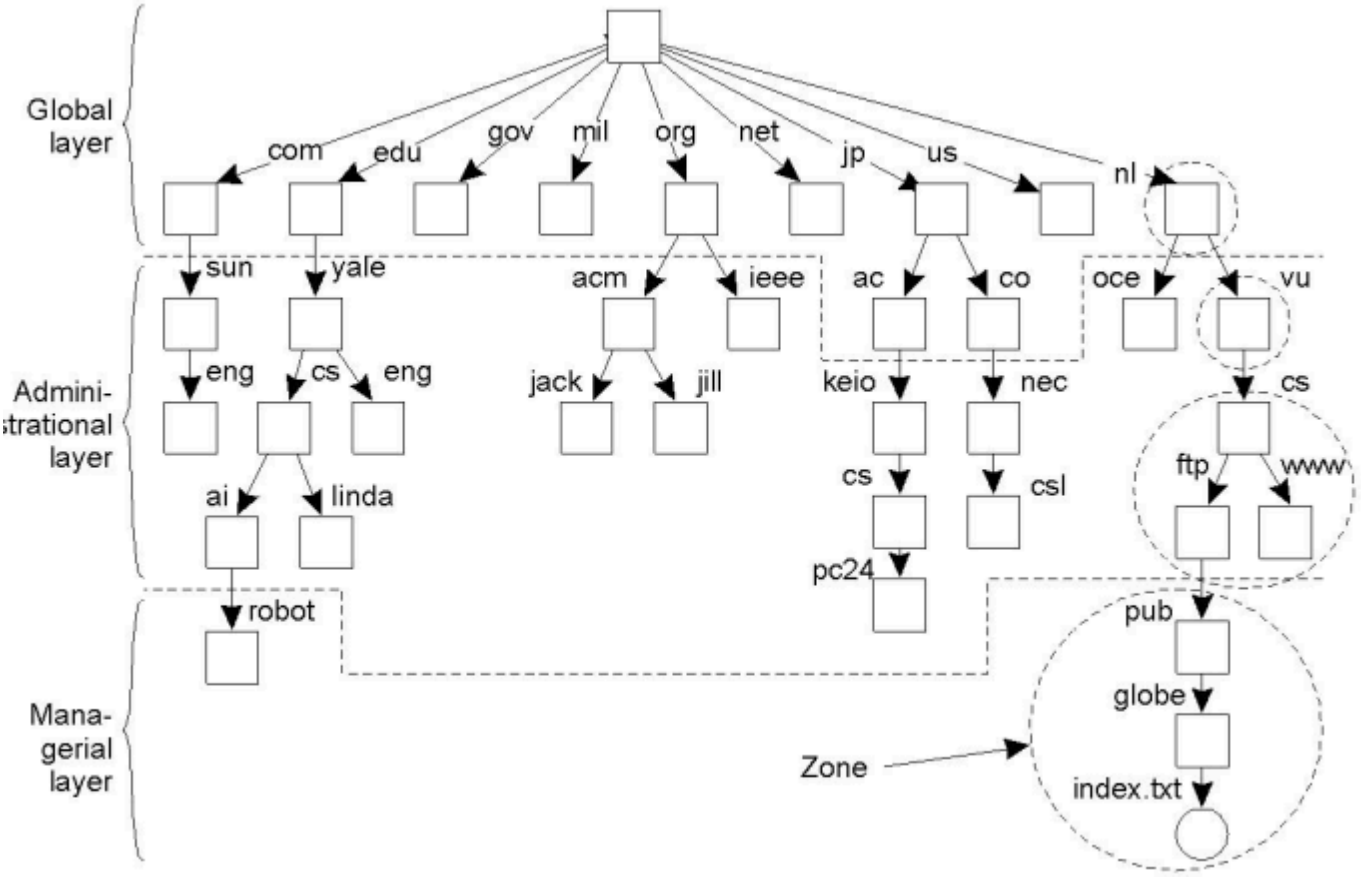



Soft link: global names that only link to nodes that gives information on where it is what you are searching and not the real information.
Hard link: link the node directly to the information you are searching

## Name space distribution

We can take the graph of names and organize this name space in different layers:

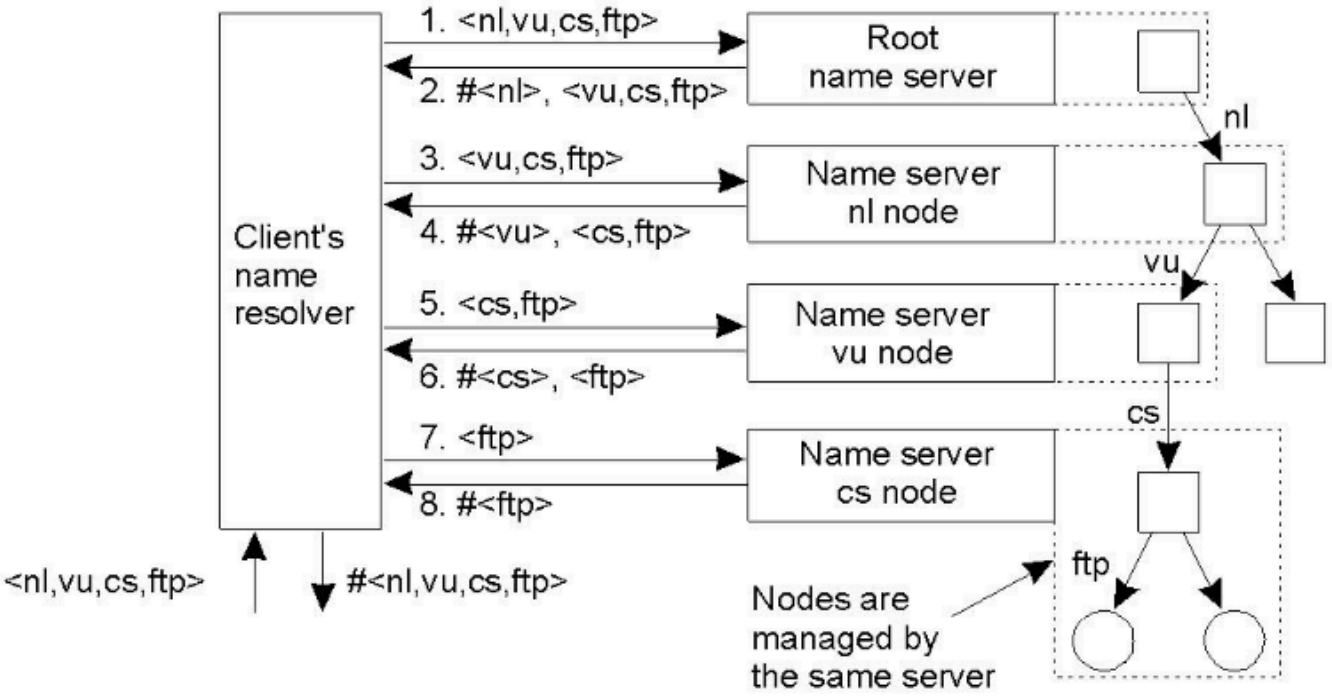
- Global level: Consists of the high-level directory nodes. Main aspect is that these directory nodes have to be jointly managed by different administrations. Names remain constantly. Usually this is done in parts of the system where you need to be stable(root).
- Administrational level: Contains mid-level 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. Names changes continuously(leafs).
  For names I also intend a portion of the name.
  DNS maps logical names of hosts in IP addresses(Ex: www.deib.polimi.it will have it in the root and the other parts of the name in the other nodes of the system, it ->polimi->deib->www).




| 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 |

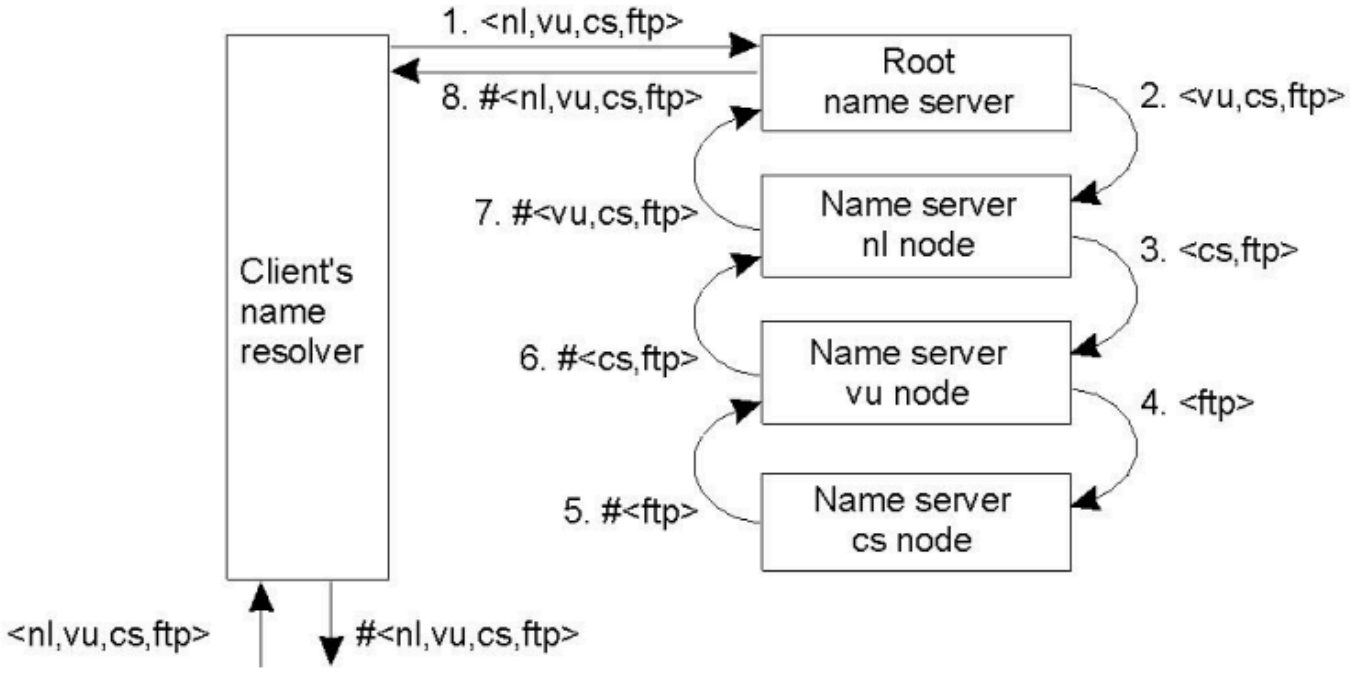
We can use caching for resolving the Global layer as it will have stable names and a lookup takes seconds so we will only do it once/few times. We can have an high number of replicas for Global layer even if the update is lazy because you it is more useful to have multiple copies of it than the cost to update the replicas every time.

## Iterative Name Resolution




- Overloading: each node is replicated and as to ask to the other nodes
- Caching can work for the top part of the name

## Recursive Name Resolution




Usually not node by Global layer, the more you go down the more you do recursive name resolution.

## Trade-offs

Pros of recursive vs. iterative resolution

- Communication costs may be reduced
- Caching is more effective because it can be performed along the whole resolution chain, leading to faster lookups
  Cons:
- Higher demand (suspended threads) on each name server
  Flat names cannot apply caching effectively BUT they can resolve immediately names in the same subnet having the root less crowded, instead the Structured naming need to ask to the root every time(root is overcrowded) BUT it can apply caching efficiently.
  Ca use Anycast: assign the same IP address to multiple machines and gives back the machine closest(networking/traffic distance) to the requester address.

## DNS resolution in practice

Clients can request the resolution mode, but servers are not obliged to comply
In practice, a mixture of the two is used: Global name servers typically support only iterative resolution
Global servers are mirrored, and IP anycast is used to route queries among them
Caching and replication are massively used:

- Secondary DNS servers are periodically (e.g., twice per day) brought up-to-date by the primary ones
- A TTL attribute is associated to information, determining its persistence in the cache
  Transient (days at the global level) inconsistencies are allowed
  Only host information is stored, but in principle other information could be stored

## DNS and mobile entities

If you want to resolve the last part of the name it has to be resolved by the root node.
If you have continuity mobility is handled by the network as the address of the machine continuously update it but this kind of situations aren't supported by DNS as it cannot update constantly its tables.

# Attribute based naming

Problem: As more information is made available, it becomes important to effectively search for items
Solution: Refer to entities not with their name but with a set of attributes, which code their properties

Each entity has a set of associated attributes
The name system can be queried by searching for entities given the values of (some) of their attributes

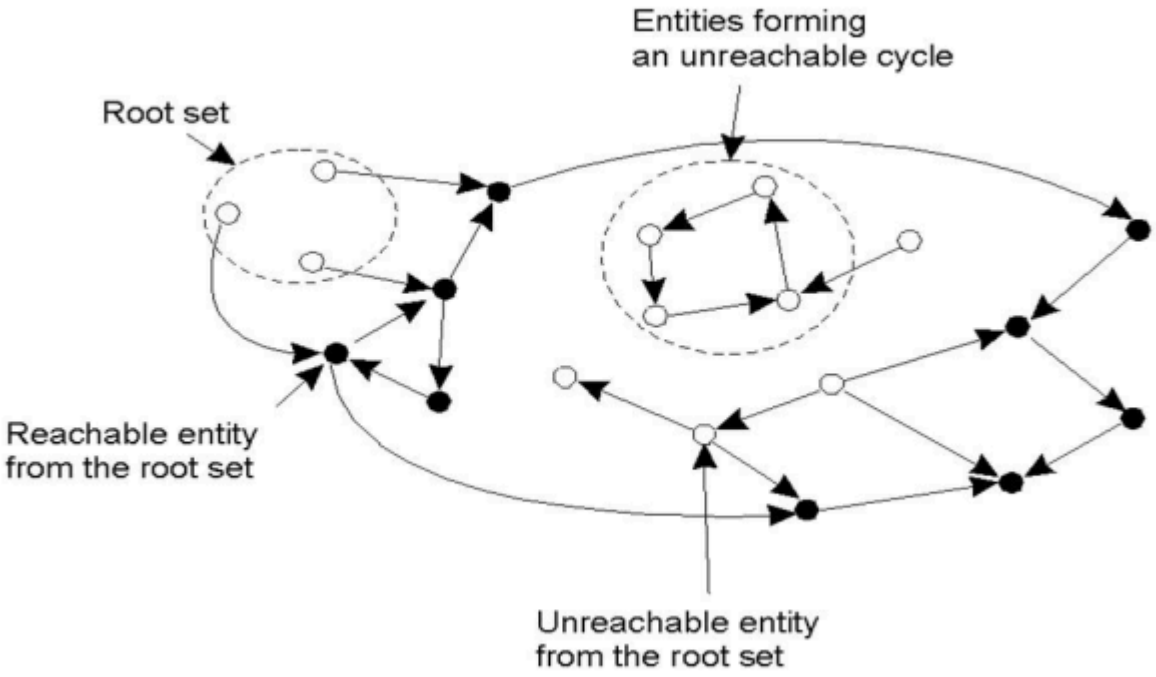
- More entities can be returned
  Attribute based naming systems are usually called directory services
  They are usually implemented by using DBMS technology

## Structured, attribute based naming:The LDAP case

A common approach to implement distributed directory services is to combine structured with attribute based naming. The LDAP (lightweight directory access protocol) directory service is becoming the de-facto standard in this field

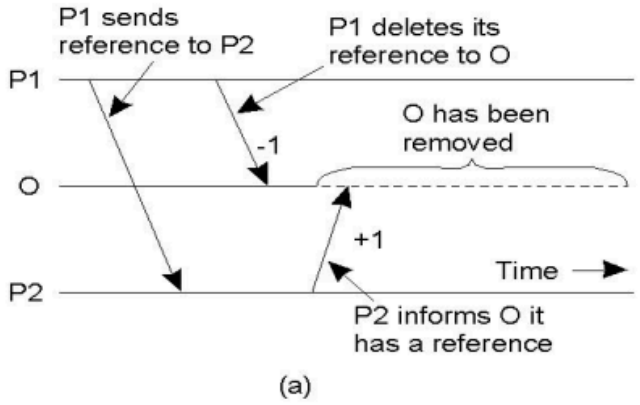
- Derived from the OSI X.500 directory service
  An LDAP directory consist of a number of records (directory entries)
- Each is made as a collection of <attribute, value> pairs
- Each attribute has a type
- Both single-valued and multiple-valued attributes exist
  Some attributes are part of the standard The collection of all records in a LDAP directory service is called Directory Information Base – DIB
  Each record has a unique name:
- Defined as a sequence of naming attributes (aka relative distinguished name – RDN)
- E.g., /C=NL/O=Vrije Universiteit/OU=Comp. Sc./CN=Main server
  This approach leads to build a directory information tree - DIT
- A node in a LDAP naming graph can thus simultaneously represent a directory in a traditional sense (i.e., in a hierarchical name space)
  LDAP is created for search and so is limited to small data bases.

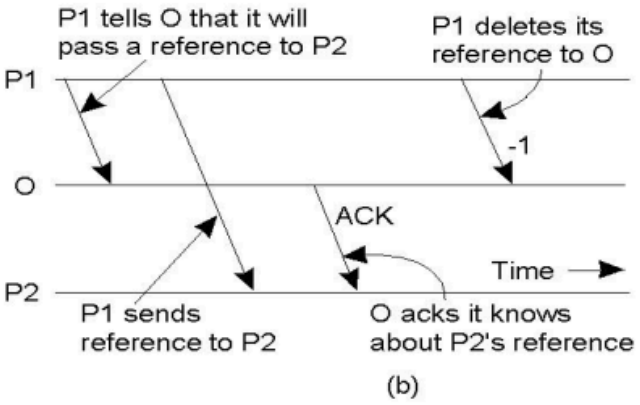
# REmoving unreferenced entities



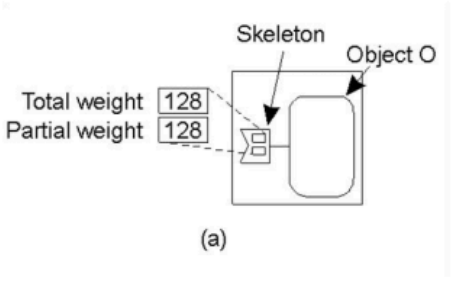

Can be situations where objects aren't reachable from the root and can be garbage collected.
Garbage collecting works in this way(single system):

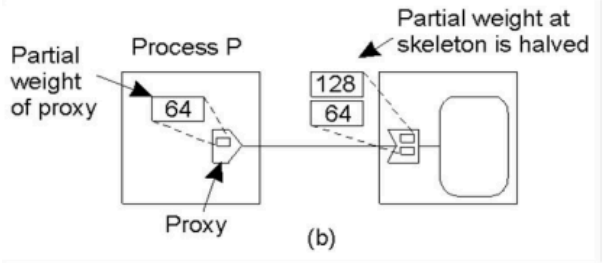
- stop the system as we don't want new reference
- start from the root and mark the object as grey when reached from another node.
- Mark a node black when the reachable object from the node are finished
- If a node remain white the node isn't reachable from the root and can be destroyed
  Garbage collecting works in this way(distributed system):
- Reference counting: the object contains in its skeleton the number of proxies that reference to it. When a proxy is eliminated you decrease the number. A problem arise when you pass a proxy through the nodes and the skeleton need to be informed of this passage(problem: the object need to be delivered **exactly** once, difficult to obtain in a network. Another issue is the Race Condition(a) that can be resolved by informing the nodes that you are sending copies of the proxies to another nodes so the node in question can update its skeleton(b)).



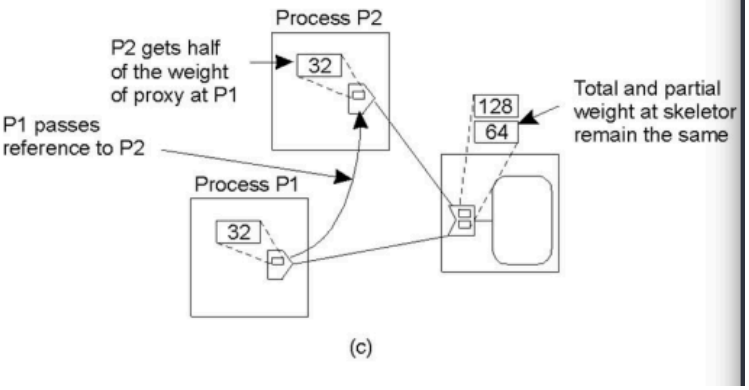



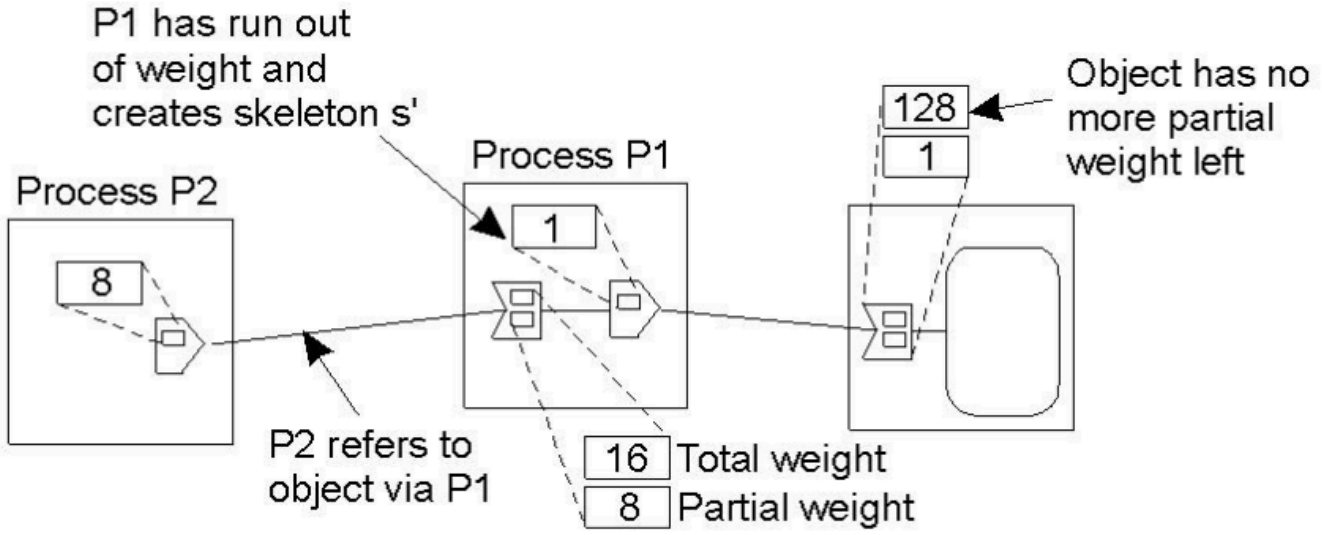

- Weighted reference counting: the idea is that the skeleton holds two numbers: total and partial weight. These numbers initially are equal to a big number(usually 128). When a proxy is created it takes half the size of the partial weight of the node. When the proxy is passed half of the number is passed. When you remove a proxy you inform the original object and return the weight of the nodes that is eliminated. So when the total and partial weights are equal the node can be erased. You don't solve the problem of object connected to each other and the problem of exactly one delivery. The race condition is resolved as there is only one type of message. It introduce the problem that you can finish the number as when you have a partial weight of 1 you cannot create another proxy. To resolve this you can create a fake skeleton and pass it to the object.




- Tries to circumvent the race condition by communicating only counter decrements
- Requires an additional counter
- Removing a reference subtracts the proxy partial counter from the total counter of the skeleton: when the total and partial weights become equal, the object can be removed
- Problem: only a fixed number of references can be created






- Reference listing: the skeleton it knows exactly the pointers to the nodes that have a connection to it. The adding/removing are idempotent: you need at least one delivery. Don't solve race conditions but since we can check exactly. The additional problem is that we need to remember a list of pointer and not a number and you still have the problem of objects reference to each other in a circle.
  We cannot use centralised protocol in a distributed system because we need a distributed mark and sweep as you need to have all the system blocked(can be implemented BUT is too costly).