CS865 – Distributed Software Development
Lecture 5

Tannenbaum and Van Steen – Chapter 5

Naming

Names, Identifiers, and Addresses

Naming is about mapping between names, addresses, identifiers and the referred entities
- Names (a bit-or character-string referring to an entity)
  - e.g. John Smith or ftp-server
- Can be *human-friendly*(or not) and *location dependent*(or not)

Addresses (define access points)
- Entities can be operated through an *access point*
- The name of an access point is an address
  - e.g. phone number, or IP-address + port for a service

Identifiers (*unique* identifiers)
- A (true) identifier is a name with the following properties
  1. Each identifier refers to at most 1 entity and
  2. Each entity is referred to by at most 1 identifier
  3. An identifier always refers to the same entity (never reused)
     - e.g. John Smith + social security number, or MAC address

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 idenifier refers to at most one entity
P2 Each entity is referred to by at most one identifier
P3 An idenifier 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.

Identifiers
- Special type of (usually, computer readable) name with the following properties:
  - An id *refers* to at most one entity
  - Each entity is referred by at most one id
  - An id always refers to the same entity (never reused)
- Identifier includes or can be transformed to an address for an object
  - e.g. NFS file handle, Java RMI remote object reference, etc.

Properties of a true identifier: (Wieringa and de Jonge, 1995):
An identifier refers to at most one entity.
Each entity is referred to by at most one identifier.
An identifier always refers to the same entity

Names
- A **name** is human-readable value (usually a string) that can be **resolved** to an identifier or address
  - Internet domain name, file pathname, process number
    - e.g. /etc/passwd, http://www.csc.liv.ac.uk/
- For many purposes, names are preferable to identifiers
  - because the binding of the named resource to a physical location is deferred and can be changed
  - because they are more meaningful to users
- Resource names are **resolved** by name services
  - to give identifiers and other useful attributes

Examples

Resource	*Name*	*Identifiers*
File	Pathname	File within a given file system
Process	Process ID	Process on a given computer
Port	Port number	IP port on a given computer

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.

Flat Naming

- Identifiers are simply random bit strings referred to as unstructured or flat names.
- Property: it does not contain any information on how to locate the access point of its associated entity.

Simple Solutions

Locating an entity:

Broadcasting

- A message containing the identifier of the entity is broadcast to each machine and each machine is requested to check whether it has that entity.
- Only the machines that can offer an access point for the entity send a reply message containing the address of that access point.
  - e.g. used in the Internet Address Resolution Protocol (ARP) to find the data-link address of a machine when given only an IP address.
    - A machine broadcasts a packet on the local network asking who is the owner of a given IP address.
    - When the message arrives at a machine, the receiver checks whether it should listen to the requested IP address.
    - If so, it sends a reply packet containing, for example, its Ethernet address.

Multicasting

- Internet supports network-level multicasting by allowing hosts to join a specific multicast group.
- These groups are identified by a multicast address.
- When a host sends a message to a multicast address, the network layer provides a best-effort service to deliver that message to all group members. See Deering and Cheriton (1990) and Deering et al. (1996).

Forwarding Pointers

- When an entity moves from A to B, it leaves behind in A a reference to its new location at B.

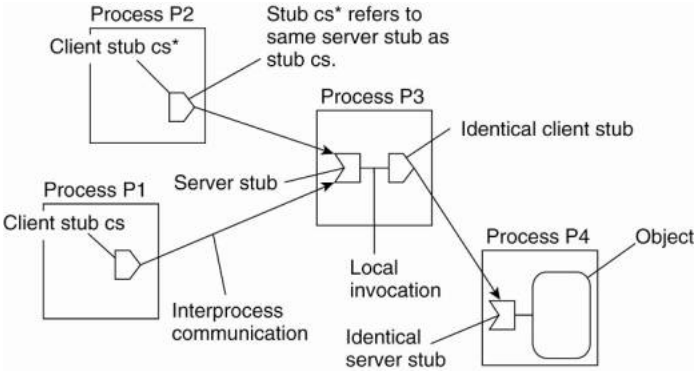
Advantages:

- Simple

Disadvantages

- No special measures are taken, a chain for a highly mobile entity can become so long that locating that entity is prohibitively expensive.
- All intermediate locations in a chain will have to maintain their part of the chain of forwarding pointers as long as needed.
- A drawback is the vulnerability to broken links.

The principle of forwarding pointers using (client stub, server stub) pairs.



- When an object moves from address space A to B, it leaves behind a client stub in its place in A and installs a server stub that refers to it in B.
- Migration is completely transparent to a client.
- The only thing the client sees of an object is a client stub.
- A client's request is forwarded along the chain to the actual object.

Home-Based Approaches

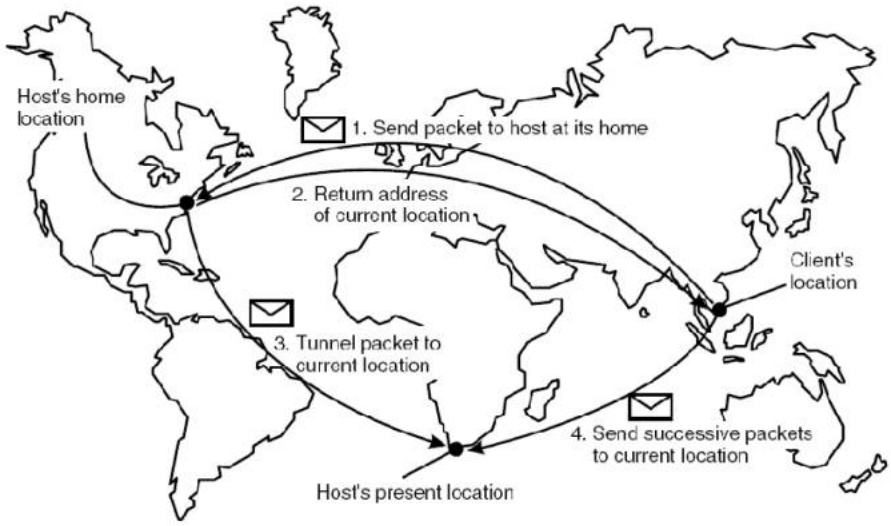
Single-tiered scheme: Let a **home** keep track of where the entity is:

- 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

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

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

The principle of Mobile IP.



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)

Distributed Hash Tables

Various DHT-based systems exist:

- Overview is given in Balakrishnan et al. (2003).
- The Chord system (Stoica et al., 2003) is a representative system.

Example: Consider the organization of many nodes into a **logical ring** (e.g. **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

(Finger is a protocol used to retrieve information about a system's users)

- 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 lookup a key *k*, node *p* forwards the request to node with index *j* satisfying

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

Sample lookup: Resolve k = 26 from node 1 in figure below.

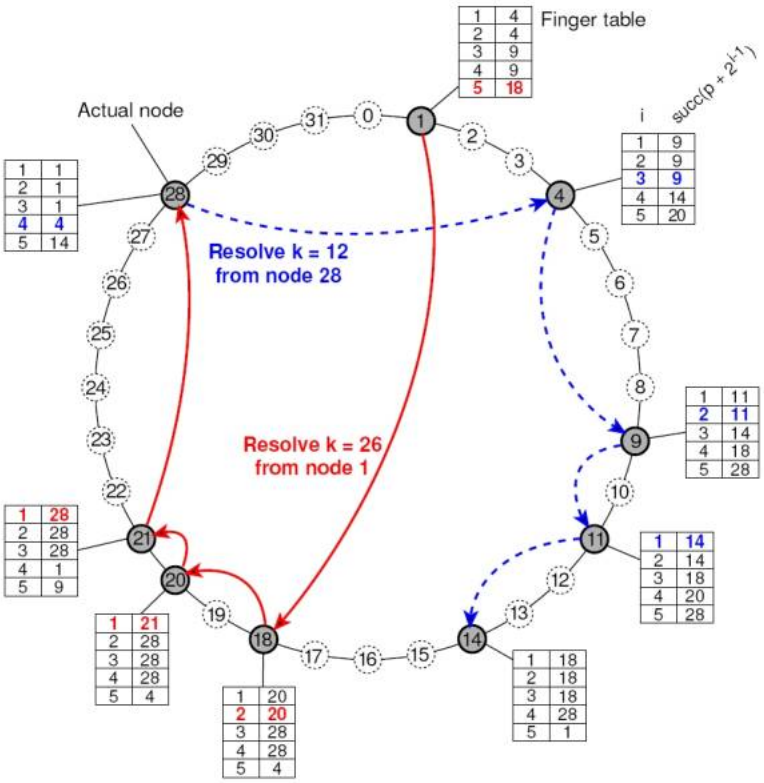
1. Node 1 will look up k = 26 in its finger table to discover that this value is larger than FT 1[5], meaning that the request will be forwarded to node 18 = FT1[5].
2. Node 18 will select node 20, as FT18 [2] < k FT18 [3].
3. The request is forwarded from node 20 to node 21 and from there to node 28, which is responsible for k = 26.
4. At that point, the address of node 28 is returned to node 1 and the key has been resolved.

Sample lookup: Resolve k = 12 from node 28

5. Request will be routed as.

Efficiency: A lookup will generally require O(log (N)) steps, with N being the number of nodes in the system.

Resolving key 26 from node 1 and key 12 from node 28 in a Chord system.



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 (e.g. geographically).

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 *FTp*[*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 you neighbor will be (not in Chord), pick the closest one.

Hierarchical Approaches

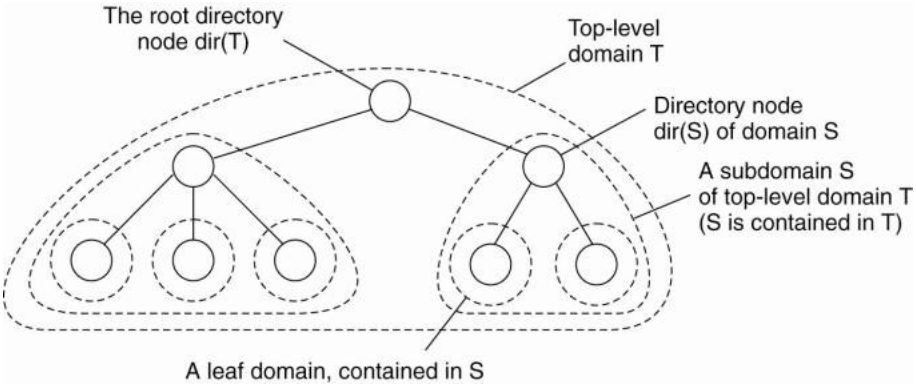
Background Resources:

- An overview can be found in van Steen et al. (1998).
- Personal Communication Systems general overview can be found in Pitoura and Samaras (2001).

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.

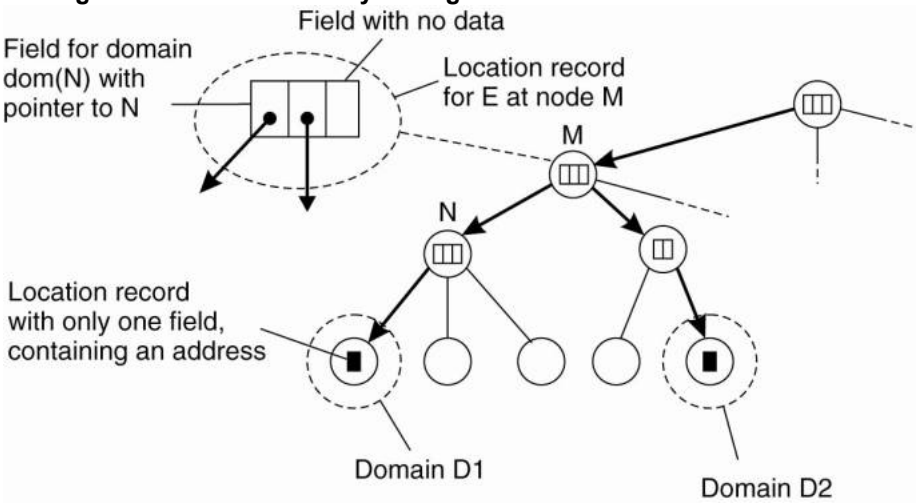
Hierarchical organization of a location service into domains, each having an associated directory node.



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

Storing information of an entity having two addresses in different leaf domains.

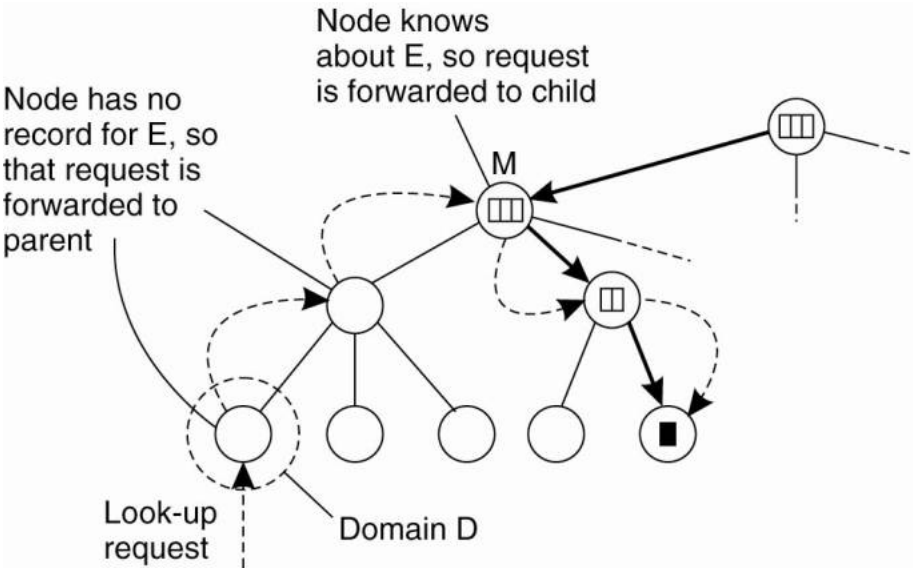


HLS: Lookup Operation

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

Looking up a location in a hierarchically organized location service.



Observation: If an entity *E* moves regularly between leaf domains D1 and D2, it may be more efficient to store *E's* contact record at the least common ancestor LCA of dir(D1) and dir(D2)
• Lookup operations from either D1 or D2 are on average cheaper
• Update operations (i.e., changing the current address) can be dome directly at LCA
• Note: Assuming that *E* generally stays in dom(LCA), it does make sense to cache a **pointer** to LCA

HLS: Scalability Issues

Size scalability: Again, we have a problem of over-loading higher-level nodes:
• Only solution is to partition a node into a number of subnodes and evenly assign entities to subnodes
• Naïve partitioning may introduce a node management problem, as a subnode may have to know how its parent and children are partitioned.

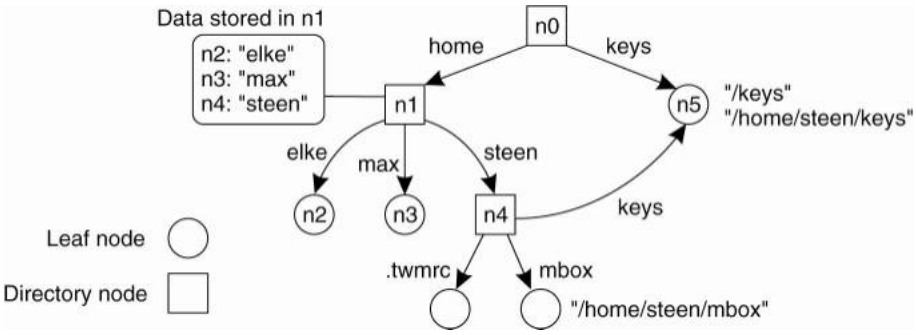
Geographical scalability: We have to ensure that lookup operations generally proceed monotonically in the direction of where we'll find an address:
• If entity *E* generally resides in California, we should not let a root subnode located in France store *E's* contact record.
• Unfortunately, subnode placement is not that easy, and only a few tentative solutions are known.

Structured Naming

Name Spaces

- Names are commonly organized into what is called a name space.
- Name spaces for structured names can be represented as a labeled, directed graph with two types of nodes.
  - A leaf node represents a named entity and has the property that it has no outgoing edges.
    - generally stores attribute information on the entity it is representing—for example, its address—so that a client can access it.
    - could store the state of that entity, such as in the case of file systems in which a leaf node actually contains the complete file it is representing. We return to the contents of nodes below.
  - Directory node has a number of outgoing edges, each labeled with a name of other nodes
    - A directory node stores a directory table in which an outgoing edge is represented as a pair (edge label, node identifier).
    - Directory nodes can also have attributes, besides just storing a directory table with *(edge label, node identifier)* pairs.

A general naming graph with a single root node.



- The naming graph has one node, namely n0, which has only outgoing and no incoming edges called the root (node) of the naming graph.
- Each path in a naming graph can be referred to by the sequence of labels corresponding to the edges in that path, such as

  N:<label-1, label-2, ..., label-n> where N refers to the first node in the path.

- Such a sequence is called a path name.
- If the first node in a path name is the root of the naming graph, it is called an absolute path name ELSE, it is called a relative path name.

NOTE: path naming similar to file systems.

- BUT path names in file systems are represented as a single string in which the labels are separated by a special separator character, such as a slash ("/").
- This character is also used to indicate whether a path name is absolute.
  - E.g, instead of using n0:<home, steen, mbox>, the actual path name common practice to use its string representation /home/steen/mbox.
- If several paths that lead to the same node, it can be represented by different path names.
  - e.g, node n5 can be referred to by /home/steen/keys or /keys. The string representation of path names can be equally well applied to naming graphs other than those used for only file systems.
- More general example of naming is Plan 9 operating system
  - all resources, such as processes, hosts, I/O devices, and network interfaces, are named in the same fashion as traditional files.
  - Approach is analogous to implementing a single naming graph for all resources in a distributed system.

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

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

Linking and Mounting

Name Linking

Linking: allow multiple absolute path names to refer to the same node

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.

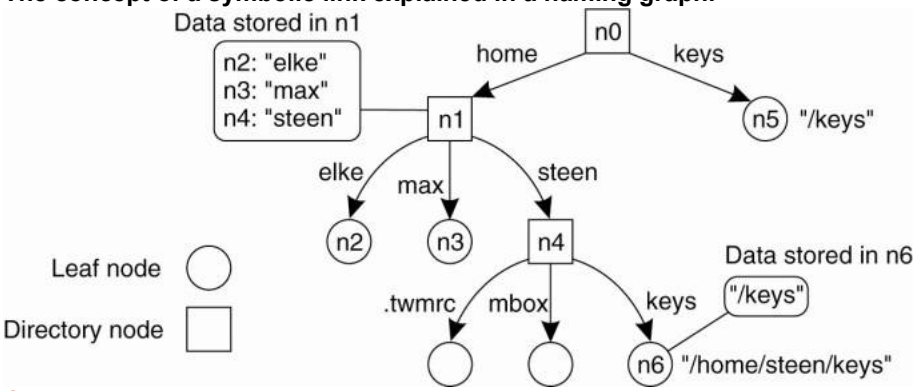
Soft link: Allow a node O to contain a *name* of another node:

- 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

The concept of a symbolic link explained in a naming graph.



Observation: Node n5 has only one name

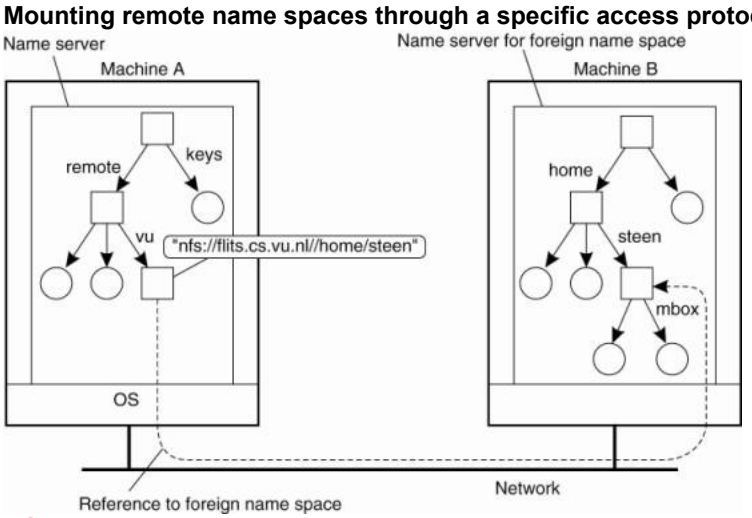
Mounting

Mounting: allow a node to refer to a node from a different name space

Mounting remote name spaces ("remote symbolic link") needs

- A specific protocol (e.g. NFS)
- At a certain *mount point* of a given server
- A name, containing access protocol, remote server, foreign mounting point (e.g. URL)

Mounting remote name spaces through a specific access protocol.



NOTE:

- The root directory has a number of user-defined entries, including a subdirectory called /remote.

- This subdirectory includes mount points for foreign name spaces such as the user's home directory at the Vrije Universiteit.
- A directory node named /remote/vu is used to store the URL nfs://flits.cs.vu.nl//home/steen.

Consider the name /remote/vu/mbox.

- This name is resolved by starting in the root directory on the client's machine and continues until the node /remote/vu is reached.
- The process of name resolution then continues by returning the URL nfs://flits.cs.vu.nl//home/steen, leading the client machine to contact the file server flits.cs.vu.nl by means of the NFS protocol, and to subsequently access directory /home/steen.
- Name resolution can then be continued by reading the file named mbox in that directory, after which the resolution process stops.

The Implementation of a Name Space

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 Distribution

- Consider a hierarchical naming graph and distinguish three levels:

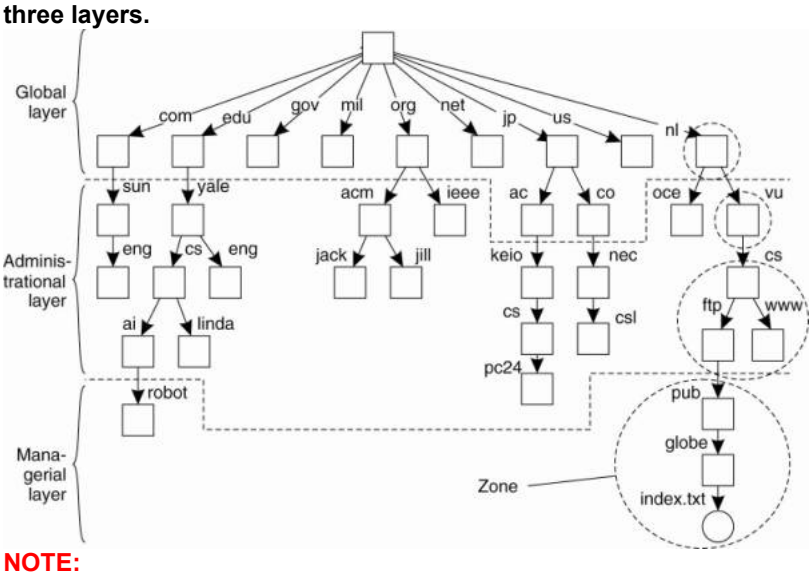
  Global level: Consists of the high-level directory nodes. Main aspect is that these directory nodes have to be jointly managed by different administrations

  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.

To make matters more concrete, Fig. 5-13 shows an example of the partitioning of part of the DNS name space, including the names of files within an organization that can be accessed through the Internet, for example, Web pages and transferable files. The name space is divided into nonoverlapping parts, called zones in DNS (Mockapetris, 1987). A zone is a part of the name space that is implemented by a separate name server.

An example partitioning of the DNS(Domain Name System) name space, including Internet-accessible files, into three layers.



NOTE:

- Name space is divided into nonoverlapping parts, called zones in DNS (Mockapetris, 1987).

- A zone is a part of the name space that is implemented by a separate name server.

- Comparison between name servers for implementing nodes from a large-scale name space partitioned into a global layer, an administrational layer, and a managerial layer.

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

Implementation of Name Resolution

Problem: Want to resolve absolute pathname:

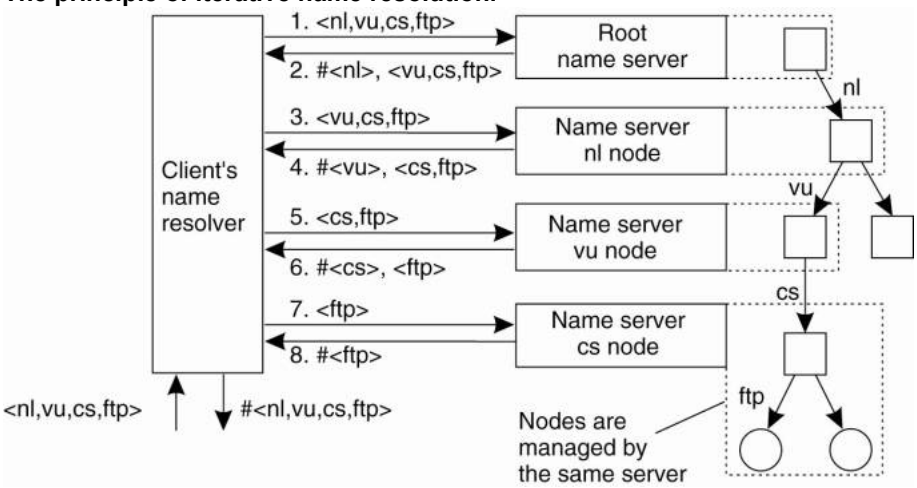
`root:<nl, vu, cs, ftp, pub, globe, index.html >`

Note: could alternatively use URL notation : ftp://ftp.cs.vu.nl/pub/globe/index.html

Iterative Name Resolution

1. The address of the root server must be well known
2. A name resolver hands over the complete name to the root name server
3. The root server will resolve the path name as far as it can, and return the result to the client.
   - the root server can resolve only the label nl, for which it will return the address of the associated name server.
4. The client passes the remaining path name (i.e., **`nl:<vu, cs, ftp, pub, globe, index.html >`**) to that name server.
5. This server can resolve only the label vu, and returns the address of the associated name server, along with the remaining path name vu:<cs, ftp, pub, globe, index.html >.
6. The client's name resolver will then contact this next name server, which responds by resolving the label cs, and also ftp, returning the address of the FTP server along with the path name ftp:<pub, globe, index.html >.
7. The client then contacts the FTP server, requesting it to resolve the last part of the original path name.
8. The FTP server will resolve the labels pub, globe, and index.html, and transfer the requested file (in this case using FTP).
   - (The notation #<cs> is used to indicate the address of the server responsible for handling the node referred to by <cs>.)

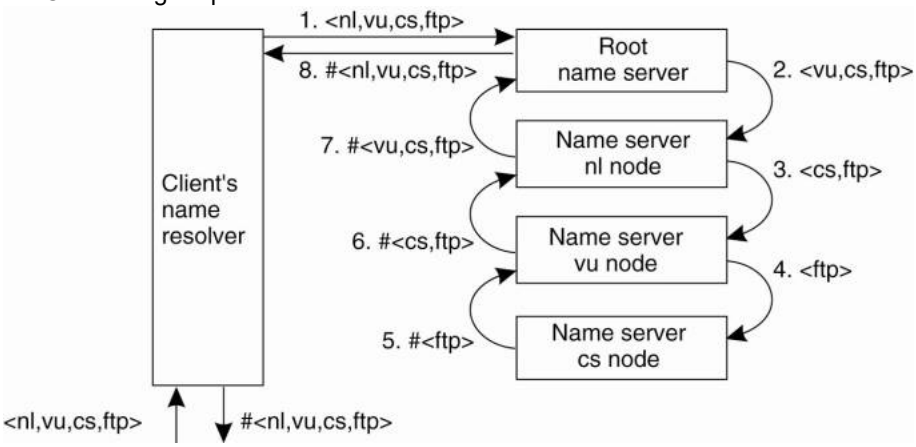
The principle of iterative name resolution.



The principle of recursive name resolution

+Enables efficient caching and reduces communication

-Causes higher performance demand on the name servers



Recursive name resolution of <nl, vu, cs, ftp>. Name servers cache intermediate results for subsequent lookups.

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

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.

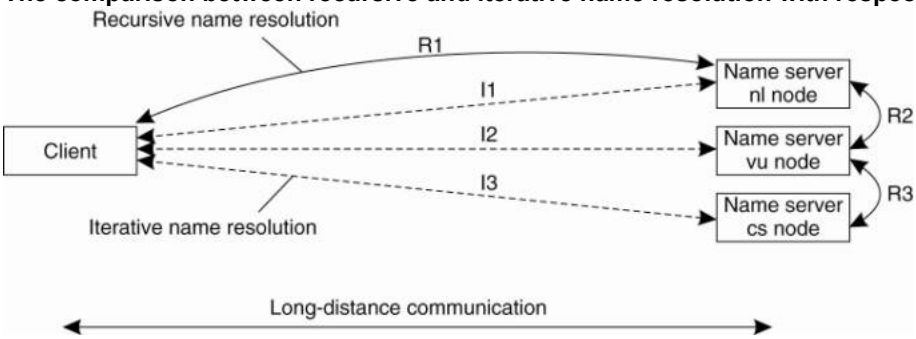
Solution: Assume (at least at global and administrational level) that content of nodes hardly ever changes.

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.

Geographical scalability: We need to ensure that the name resolution process scales across large geographical distances.

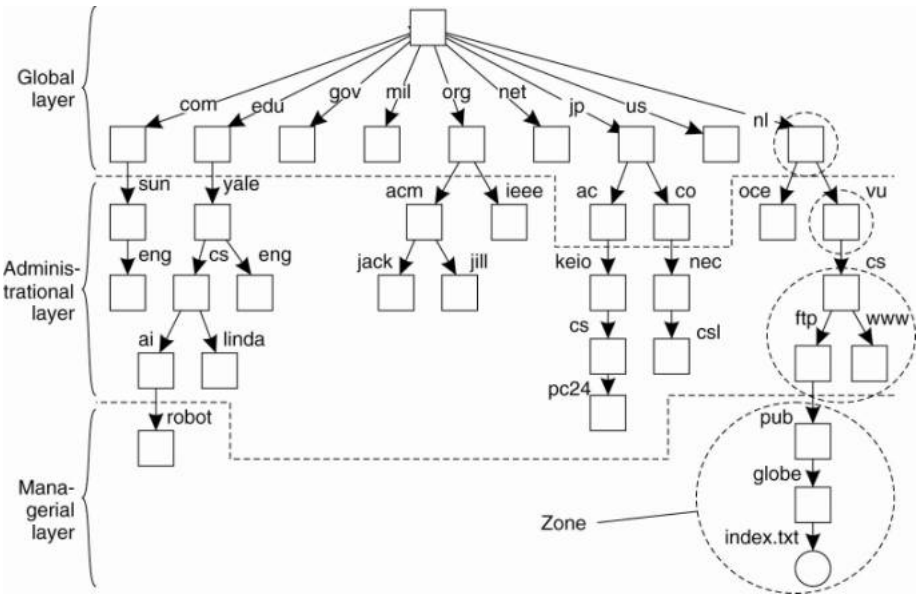
The comparison between recursive and iterative name resolution with respect to communication costs.



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

Example: The Domain Name System

- The DNS name space is hierarchically organized as a rooted tree.



- A label is a case-insensitive string made up of alphanumeric characters.
- A label has a maximum length of 63 characters; the length of a complete path name is restricted to 255 characters.
- The string representation of a path name consists of listing its labels, starting with the rightmost one, and separating the labels by a dot (".").
- The root is represented by a dot.
  - e.g, the path name root:<nl, vu, cs, flits>, is represented by the string flits.cs.vu.nl. , which includes the rightmost dot to indicate the root node - omitted for readability.
- Because each node in the DNS name space has exactly one incoming edge (with the exception of the root node, which has no incoming edges), the label attached to a node's incoming edge is also used as the name for that node.
- A subtree is called a domain; a path name to its root node is called a domain name.
  - Note - just like a path name, a domain name can be either absolute or relative.

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 tohandle 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

SOA (start of authority) resource record - contains information such as an e-mail address of the system administrator responsible for the represented zone, the name of the host where data on the zone can be fetched, and so on.

A (address) record - contains an IP address for the Internet host to allow communication.
If a host has several IP addresses, as is the case with multi-homed machines, the node will contain an A record for each address.

MX (mail exchange) record - a symbolic link to a node representing a mail server.

SRV records - contain the name of a server for a specific service. The service itself is identified by means of a name along with the name of a protocol. SRV records are defined in Gulbrandsen (2000).

NS (name server) records - an NS record contains the name of a name server that implements the zone represented by the node.

CNAME record - contains the canonical name of a host . DNS distinguishes aliases from what are called canonical names. Each host is assumed to have a canonical, or primary name. The name of the node storing such a record is thus the same as a symbolic link

PTR (pointer) record - DNS maintains an inverse mapping of IP addresses to host names by means of PTR (pointer) records. To accommodate the lookups of host names when given only an IP address, DNS maintains a domain named in-addr.arpa, which contains nodes that represent Internet hosts and which are named by the IP address of the represented host.

HINFO (host info) record - used to store additional information on a host such as its machine type and operating system.

TXT records - used for any other kind of data that a user finds useful to store about the entity represented by the node.

DNS Implementation

- Each zone is implemented by a name server, which is virtually always replicated for availability.
- A DNS database is implemented as a (small) collection of files -the most important contain all the resource records for all the nodes in a particular zone.
- Nodes are identified by means of their domain name, by which the notion of a node identifier reduces to an (implicit) index into a file.

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

Name	Record type	Record value
cs.vu.nl.	SOA	star.cs.vu.nl. hostmaster.cs.vu.nl. 2005092900 7200 3600 2419200 3600
cs.vu.nl.	TXT	"Vrije Universiteit - Math. & Comp. Sc."
cs.vu.nl.	MX	1 mail.few.vu.nl.
cs.vu.nl.	NS	ns.vu.nl.
cs.vu.nl.	NS	top.cs.vu.nl.
cs.vu.nl.	NS	solo.cs.vu.nl.
cs.vu.nl.	NS	star.cs.vu.nl.
star.cs.vu.nl.	A	130.37.24.6
star.cs.vu.nl.	A	192.31.231.42
star.cs.vu.nl.	MX	1 star.cs.vu.nl.
star.cs.vu.nl.	MX	666 zephyr.cs.vu.nl.
star.cs.vu.nl.	HINFO	"Sun" "Unix"
zephyr.cs.vu.nl.	A	130.37.20.10
zephyr.cs.vu.nl.	MX	1 zephyr.cs.vu.nl.
zephyr.cs.vu.nl.	MX	2 tornado.cs.vu.nl.
zephyr.cs.vu.nl.	HINFO	"Sun" "Unix"
ftp.cs.vu.nl.	CNAME	soling.cs.vu.nl.
www.cs.vu.nl.	CNAME	soling.cs.vu.nl.
soling.cs.vu.nl.	A	130.37.20.20
soling.cs.vu.nl.	MX	1 soling.cs.vu.nl.
soling.cs.vu.nl.	MX	666 zephyr.cs.vu.nl.

soling.cs.vu.nl.	HINFO	"Sun" "Unix"
vucs-das1.cs.vu.nl. vucs-das1.cs.vu.nl.	PTR A	0.198.37.130.in-addr.arpa. 130.37.198.0
inkt.cs.vu.nl. inkt.cs.vu.nl.	HINFO A	"OCE" "Proprietary" 192.168.4.3
pen.cs.vu.nl. pen.cs.vu.nl.	HINFO A	"OCE" "Proprietary" 192.168.4.2
localhost.cs.vu.nl.	A	127.0.0.1

Decentralized DNS Implementations

DNS on Pastry

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:

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:

```
n0 a node whose idenifier has prefix 0
n1 a node whose idenifier has prefix 1
n2 a node whose idenifier has prefix 2
n30 a node whose idenifier has prefix 30
n31 a node whose idenifier has prefix 31
n33 a node whose idenifier has prefix 33
n320 a node whose idenifier has prefix 320
n322 a node whose idenifier has prefix 322
n323 a node whose idenifier 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 $n30$.

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

Attribute-Based Naming

Directory Services

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 are required 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.

Hierarchical Implementations: LDAP (Lightweight Directory Access Protocol)

The X.500 Name Space

More than a naming service

- A directory service with *search*
- Items can be found based on *properties*(not only full names)

- X.500 defines attribute-value pairs, such as:

Attribute	*Abbr.*	*Value*
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
OrganizationalUnit	OU	Comp. Sc.
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

The collection of directory entries

- Form the Directory Information Base (DIB)
- Naming attributes (Country etc.) are called Relative Distinguished Names (RDN)
- Each record is uniquely named, by a sequence of RDNs

Directory Information Tree (DIT)

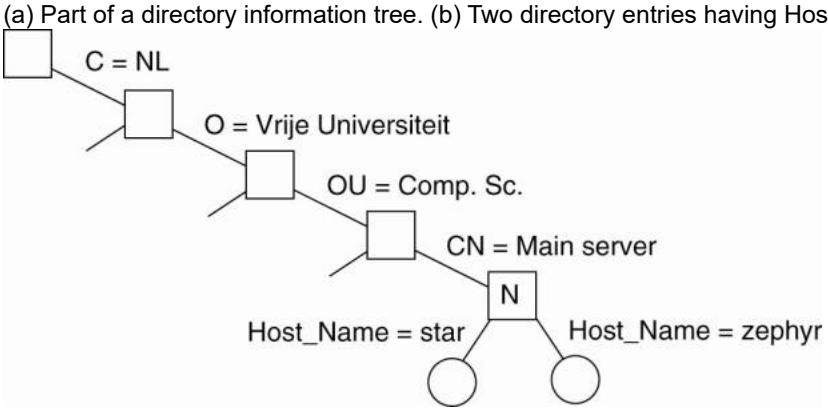
- Naming graph
- Each node represents a directory entry and an X.500 record
- With *read*we can read a certain record
- With list we can read all outgoing edges of the node

A simplified version of X.500 is generally used

- Known as Lightweight Directory Access Protocol (LDAP) - (see OpenLDAP implementation)

Example: Part of the directory information tree:
The X.500 name `/C=NL/O=VrijeUniversiteit/OU=Math. & Comp. Sc.` is analog to the DNS name nl.vu.cs

(a) Part of a directory information tree. (b) Two directory entries having Host_Name as RDN.



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

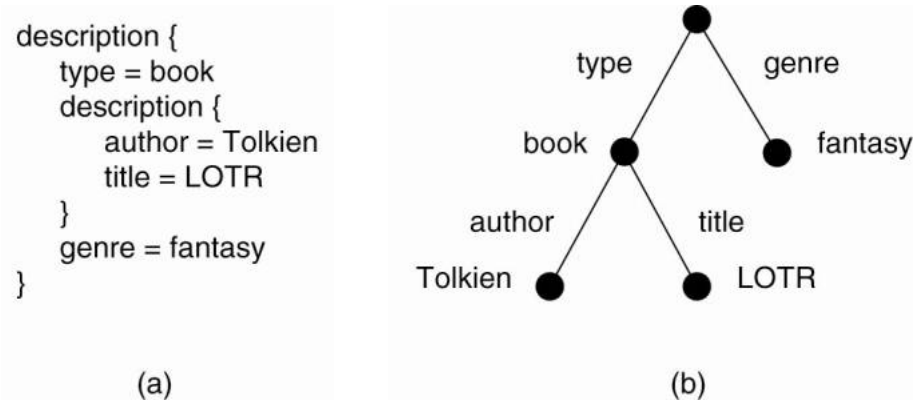
(b)

Decentralized Implementations

Mapping to Distributed Hash Tables

Example: case where (attribute, value) pairs are supported by a DHT-based system.

- Assume that queries consist of a conjunction of pairs - a user specifies a list of attributes, along with the unique value he wants to see for every attribute.
- This is a single-valued query
- Single-valued queries are supported in the INS/Twine system (Balazinska et al., 2002).
- Each entity (referred to as a resource) is assumed to be described by means of possibly hierarchically organized attributes
- Each description is translated into an attribute-value tree (AVTree) which is then used as the basis for an encoding that maps onto a DHT-based system.
- (a) general description of a resource. (b) Its representation as an AVTree.



Issue: transform the AVTrees into a collection of keys that can be looked up in a DHT system.

- every path originating in the root is assigned a unique hash value, where a path description starts with a link (representing an attribute), and ends either in a node (value), or another link.
- Figure example becomes:

```
h1: hash(type-book)
h2: hash(type-book-author)
h3: hash(type-book-author-Tolkien)
h4: hash(type-book-title)
h5: hash(type-book-title-LOTR)
h6: hash(genre-fantasy)
```

Example: case where query contains a range of specifications for attribute values.

- e.g. someone looking for a house will generally want to specify that the price must fall within a specific range.

Solution: the SWORD resource discovery system (Oppenheimer et al., 2005).

- In SWORD, (attribute, value) pairs as provided by a resource description are first transformed into a key for a DHT. Note: these pairs always contain a single value – ONLY queries may contain value ranges for attributes.
- The name of the attribute and its value are kept separate in the hash table.
- The key will contain a number of random bits to guarantee uniqueness among all keys that need to be generated.

- The space of attributes is partitioned: if n bits are reserved to code attribute names, 2^n different server groups can be used, one group for each attribute name.
- By using m bits to encode values, a further partitioning per server group can be applied to store specific (attribute, value) pairs.
- DHTs are used only for distributing attribute names.

- For each attribute name, the possible range of its value is partitioned into subranges and a single server is assigned to each subrange.
- e.g. a resource description with
  - two attributes:
    - a1 taking values in the range [1..10]
    - a2 taking values in the range [101...200].
  - two servers for a1:
    - s12 takes care of recording values of a1 in [1..5]
    - s12 for values in [6..10].
  - two servers for a2:
    - server s21 records values for a2 in range [101..150]
    - server s22 for values in [151..200].
- When the resource gets values (a1 = 7,a2 = 175), server s12 and server s22 will have to be informed.

Advantage:
- range queries can be easily supported.

Disadvantage:
- updates need to be sent to multiple servers.
- Load balancing: if certain range queries turn out to be very popular, specific servers will receive a high fraction of all queries. See Bharambe et al. (2004) for discussion within context of their multiplayer game system Mercury.

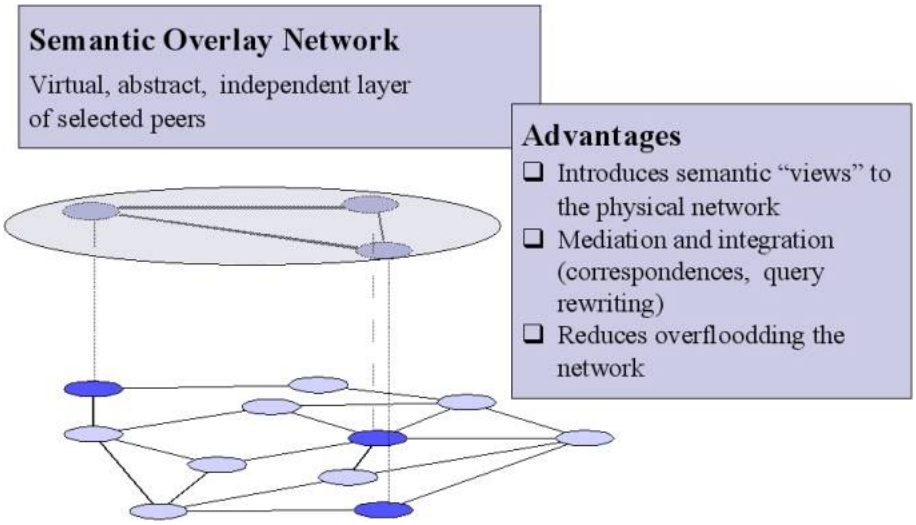
Semantic Overlay Networks

- **Semantic Overlay Network (**SON) - nodes with semantically similar content are clustered together **for reducing search overhead**

Following discussion based on (Crespo and Garcia-Molina, 2003)

Motivation
- DHT-based queries scale, but allow
  - just only exact match
  - no approximate queries
  - no range queries
    eg. query for songs from 80's to 90's
  - no text (keyword)queries
    eg. query for songs whose name contains "love"
- Broadcasting all queries to all information sources obviously doesn't scale efficiently
- Clue of solution by intuitions:
  - It's better to route queries **only** to peers that are **more likely** to have answers
  - Shared content often has **pronounced ontological structure** (music, movies, scientific papers etc.)

Conceptual idea
- Peers are clustered
- Clusters can overlap
- Query is distributed to relevant cluster(s) **only**
- Query is routed within each relevant cluster **only**
- Therefore,
  - Irrelevant clusters are not bothered with the query



Formal definition of SON

Semantic Overlay Network (SON) is a set of triples (links):

$\{(n_i, n_j, L)\}$

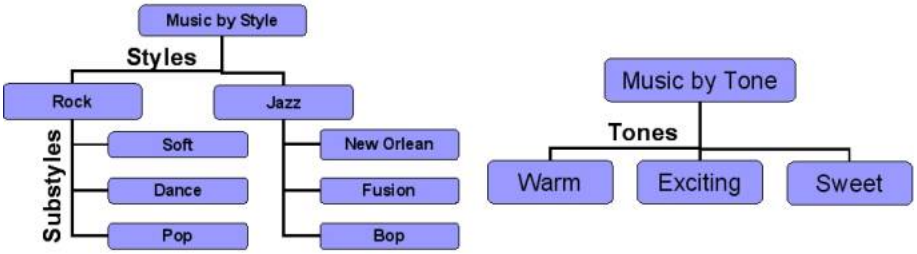
n_i, n_j - *linked peers*
L - *string (name of category)*

Each SON_L implements functions:

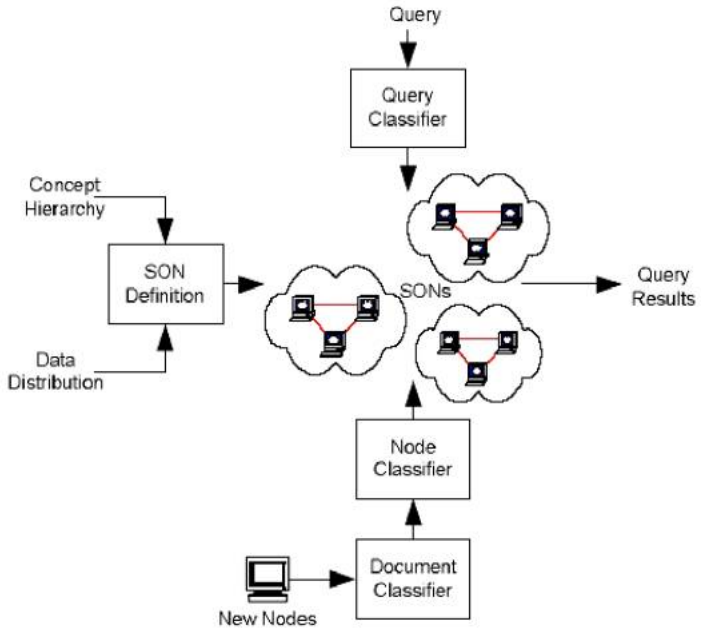
```
Join (n_i)
Search (q)
Leave (n_i)
```

SON: Hierarchical definiton

- SON is an overlay network, associated with a concept of classification hierarchy
- For example, we have 9 SONs for classification of music by style or 4 SONs for classification of music by tone
- Documents of the peer must be assigned to concepts, so that this peer could be assigned to corresponding SONs



Semantic Overlay Networks in action



- The process of building and using SONs is depicted in this figure.

Preparation step:

- **evaluate and determine some classification** to be used through out all the system.
- This hierarchy must be stored by all of the peers in the system.

Peer joining steps:

1. A peer joining the system, first floods the network with requests for this **hierarchy**.
2. The peer runs a **document classifier** based on the obtained hierarchy **on all its documents**.
3. A **peer classifier** assigns the peer to specific SON**s**.
4. The peer **joins** each SON by **finding peers** that belong to those SONs.
5. This can be done again eitherby flooding the network until peers in that SON are found or by using a central directory.

Query answering steps:

1. peer issues a query (user does it);
2. peer classifies query and build the list of SONs, to which query must be distributed;
3. peer searches for appropriate SONs in a similar fashion as when the peer connected to its SON;
4. peer sends query to the appropriate SONs.
5. After the query is sent to the appropriate SONs, peers within the SON try to find matches by using some propagation mechanism (for example, flooding)

Criteria of good classification hierarchy

- Classification hierarchy is good, if
  - Documents in each category belong to a small number of peers (high granularity + equal popularity)
  - Peers have documents in a small number of categories (sensible, moderate granilarity)
  - Classification algorithm is fast and errorless

Sources of errors

- Format of the files may not follow the expected standard
- Classification ontology may be incompatible with files
- Users make misspellings in the names of files

- So, 25% of music files were classified incorrectly
- But peer can still be correctly classified even if some of its documents are misclassified!
- So, only 4% of peers were classifed incorrectly

Peer assignment strategies

- **Conservative strategy**: place peer in SON_c,if it has any document classified in concept *c*

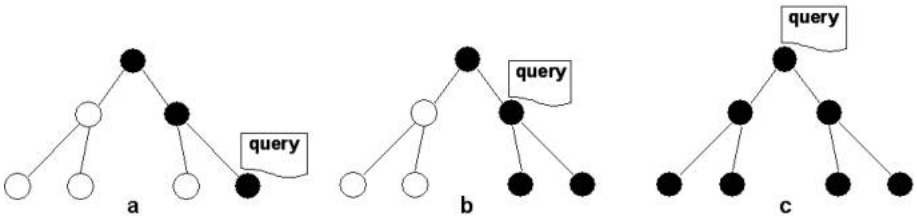
produces too many links

- **Less conservative strategy**: place peer in SON_c,if it has “significant” number of documents, classified in concept *c*

prevents from finding all documents
Final solution: use Layered SONs

Layered SONs: Searching

- Query can be assigned to:
  - Leaf concept, i.e. precisely classified (figure **a**)
  - Non-leaf concept, i.e. imprecisely classified (figures **b, c**)
- Imprecise classification leads to additional overhead



Searching Process in Layered SONs are done by:

1. First classifyi the query. (**figure a**)
2. Then, sending query to the appropriate SONs, according to the query classification.

sendi the query **progressively higher up in the hierarchy** until enough results are found.

Because, upper levels still can contain relevant results, just because nodes can join non-leaf SONs, because of lack of neccessary documents

Review of searching in all kinds of peer-to-peer systems is discussed in Risson and Moors (2006).