Naming: Names, identifiers, and addresses

Naming

Essence

Names are used to denote entities in a distributed system. To operate on an entity, we need to access it at an access point. Access points are entities that are named by means of an address.

Note

A location-independent name for an entity $E$, is independent from the addresses of the access points offered by $E$. 
Naming: Names, identifiers, and addresses

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 some specific properties
1. An identifier refers to at most one entity.
2. Each entity is referred to by at most one identifier.
3. An identifier always refers to the same entity (i.e., it is never reused).

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

Flat naming
What does it mean?

Broadcast the ID, requesting that entity to return its current address
- Can never scale beyond local-area networks
- Requires all processes to listen to incoming location requests

Address Resolution Protocol (ARP)
To find out which MAC address is associated with an IP address, broadcast the query “who has this IP address”?
Forwarding pointers

When an entity moves, it leaves behind a pointer to its next location

- Dereferencing can be made entirely transparent to clients by simply following the chain of pointers
- Update a client’s reference when present location is found
- Geographical scalability problems (for which separate chain reduction mechanisms are needed):
  - Long chains are not fault tolerant
  - Increased network latency at dereferencing
Illustrative: Chord

Consider the organization of many nodes into a **logical ring**

- 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** $succ(k)$).

**Nonsolution**

Let each node keep track of its neighbor and start linear search along the ring.

**Notation**

We will speak of node $p$ as the node have identifier $p$
Chord finger tables

**Principle**

- Each node $p$ maintains a **finger table** $FT_p[]$ with at most $m$ entries ($m = \log n$):
  
  $$FT_p[i] = \text{succ}(p + 2^{i-1})$$

  **Note:** the $i$-th entry points to the first node succeeding $p$ by at least $2^{i-1}$. 

---

General mechanism
Chord finger tables

Principle

- Each node $p$ maintains a finger table $FT_p[]$ with at most $m$ entries ($m = \log n$):

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

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

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

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

**Principle**

- Each node $p$ maintains a **finger table** $FT_p[]$ with at most $m$ entries ($m = \log n$):
  $$FT_p[i] = succ(p + 2^{i-1})$$

**Note**: the $i$-th entry points to the first node succeeding $p$ by at least $2^{i-1}$.

- To look up a key $k$, node $p$ forwards the request to node with index $j$ satisfying
  $$q = FT_p[j] \leq k < FT_p[j + 1]$$

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

Resolving key 26 from node 1 and key 12 from node 28

General mechanism
Hierarchical Location Services (HLS)

Basic idea

Build a large-scale search tree for which the underlying network is divided into hierarchical domains. Each domain is represented by a separate directory node.

Principle
HLS: Tree organization

Invariants

- Address of entity $E$ is stored in a leaf or 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

Diagram showing the structure of the tree with nodes for different domains and fields for location records.
HLS: Lookup operation

Basic principles

- Start lookup at local leaf node
- Node knows about $E \Rightarrow$ follow downward pointer, else go up

Looking up a location

Node knows about $E$, so request is forwarded to child

Node has no record for $E$, so that request is forwarded to parent

Look-up request
HLS: Insert operation

(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.

Node has no record for $E$, so request is forwarded to parent.

Node knows about $E$, so request is no longer forwarded.

Insert request

Domain $D$

Node creates record and stores address.

Node creates record and stores pointer.
Can an HLS scale?

Observation
A design flaw seems to be that the root node needs to keep track of all identifiers ⇒ make a distinction between a logical design and its physical implementation.

Basic idea for scaling
- Choose different physical servers for the logical name servers on a per-entity basis
  - (at root level, but also intermediate)
- Implement a mapping of entities to physical servers such that the load of storing records will be distributed
Can an HLS scale?

Solution

- For each level, the set of hosts is partitioned subsets, with each host running a location server representing exactly one of the domains.

Principle of distributing logical location servers
Name space

Naming graph

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

A general naming graph with a single root node

Data stored in n1

n2: "elke"
n3: "max"
n4: "steen"

elke
max
.steene
n2
n3
n4

home
steen
keys
n1

"/keys"
"/home/steen/keys"
"/home/steen/mbox"

Leaf node
Directory node

Note

A directory node contains a table of (node identifier, edge label) pairs.
Name space

We can easily store all kinds of **attributes** in a node

- Type of the entity
- An identifier for that entity
- Address of the entity’s location
- Nicknames
- ...
We can easily store all kinds of attributes in a node:

- Type of the entity
- An identifier for that entity
- Address of the entity’s location
- Nicknames
- ...

Note:
Directory nodes can also have attributes, besides just storing a directory table with \((\text{identifier}, \text{label})\) pairs.
Name resolution

Problem
To resolve a name we need a directory node. How do we actually find that (initial) node?
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.distributed-systems.net: start at a DNS name server
- /home/maarten/mbox: start at the local NFS file server (possible recursive search)
- 0031 20 598 7784: dial a phone number
- 77.167.55.6: route message to a specific IP address
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.distributed-systems.net: start at a DNS name server
- /home/maarten/mbox: start at the local NFS file server (possible recursive search)
- 0031 20 598 7784: dial a phone number
- 77.167.55.6: route message to a specific IP address

Note

You cannot have an explicit closure mechanism – how would you start?
Naming: Structured naming

Name resolution

Name linking

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 $N$ to contain a name of another node

- First resolve $N$’s name (leading to $N$)
- Read the content of $N$, yielding name
- Name resolution continues with name

Observations

The name resolution process determines that we read the content of a node, in particular, the name in the other node that we need to go to.

One way or the other, we know where and how to start name resolution given name.
Name linking

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 $N$ to contain a name of another node

- First resolve $N$’s name (leading to $N$)
- Read the content of $N$, yielding name
- Name resolution continues with name

Observations

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

The concept of a symbolic link explained in a naming graph

Data stored in n1
- n2: "elke"
- n3: "max"
- n4: "steen"

Data stored in n6
- n6: "/home/steen/keys"

Node *n5* has only one name
Mounting

Issue

Name resolution can also be used to merge different name spaces in a transparent way through mounting: associating a node identifier of another name space with a node in a current name space.

Terminology

- **Foreign name space**: the name space that needs to be accessed
- **Mount point**: the node in the current name space containing the node identifier of the foreign name space
- **Mounting point**: the node in the foreign name space where to continue name resolution

Mounting across a network

1. The name of an access protocol.
2. The name of the server.
3. The name of the mounting point in the foreign name space.
Mounting in distributed systems

Mounting remote name spaces through a specific access protocol

Machine A
- Name server
- remote
- keys
- vu
- "nfs://flits.cs.vu.nl/home/steen"

Machine B
- Name server for foreign name space
- home
- steen
- mbox

Reference to foreign name space
Network

Linking and mounting
### Name-space implementation

#### 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 implementation

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

### Distinguish three levels
- **Global level**: Consists of the high-level directory nodes. Main aspect is that these directory nodes have to be jointly managed by different administrations.
- **Administrational level**: Contains mid-level directory nodes that can be grouped in such a way that each group can be assigned to a separate administration.
- **Managerial level**: Consists of low-level directory nodes within a single administration. Main issue is effectively mapping directory nodes to local name servers.
Name-space implementation

Basic issue

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

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.
Name-space implementation

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

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.
# Name-space implementation

## Basic issue

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

## Distinguish three levels

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

- **Administrational level:** Contains mid-level directory nodes that can be grouped in such a way that each group can be assigned to a separate administration.

- **Managerial level:** Consists of low-level directory nodes within a single administration. Main issue is effectively mapping directory nodes to local name servers.
Name-space implementation

An example partitioning of the DNS name space, including network files

[Diagram showing a hierarchical structure of DNS names, divided into global, administrative, and managerial layers, with examples of names like oracle, yale, acm, and robot, and network files such as index.htm and globule.]
# Name-space implementation

A comparison between name servers for implementing nodes in a name space

<table>
<thead>
<tr>
<th>Item</th>
<th>Global</th>
<th>Administrative</th>
<th>Managerial</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Worldwide</td>
<td>Organization</td>
<td>Department</td>
</tr>
<tr>
<td>2</td>
<td>Few</td>
<td>Many</td>
<td>Vast numbers</td>
</tr>
<tr>
<td>3</td>
<td>Seconds</td>
<td>Milliseconds</td>
<td>Immediate</td>
</tr>
<tr>
<td>4</td>
<td>Lazy</td>
<td>Immediate</td>
<td>Immediate</td>
</tr>
<tr>
<td>5</td>
<td>Many</td>
<td>None or few</td>
<td>None</td>
</tr>
<tr>
<td>6</td>
<td>Yes</td>
<td>Yes</td>
<td>Sometimes</td>
</tr>
</tbody>
</table>

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

**Principle**

1. $\text{resolve}(\text{dir}, [\text{name}_1, \ldots, \text{name}_K])$ sent to $\text{Server}_0$ responsible for $\text{dir}$
2. $\text{Server}_0$ resolves $\text{resolve}(\text{dir}, \text{name}_1) \rightarrow \text{dir}_1$, returning the identification (address) of $\text{Server}_1$, which stores $\text{dir}_1$.
3. Client sends $\text{resolve}(\text{dir}_1, [\text{name}_2, \ldots, \text{name}_K])$ to $\text{Server}_1$, etc.
Recursive name resolution

Principle

1. \(\text{resolve}(\text{dir}, [\text{name}_1, \ldots, \text{name}_K])\) sent to \(\text{Server}_0\) responsible for \(\text{dir}\).
2. \(\text{Server}_0\) resolves \(\text{resolve}(\text{dir}, \text{name}_1) \rightarrow \text{dir}_1\), and sends \(\text{resolve}(\text{dir}_1, [\text{name}_2, \ldots, \text{name}_K])\) to \(\text{Server}_1\), which stores \(\text{dir}_1\).
3. \(\text{Server}_0\) waits for result from \(\text{Server}_1\), and returns it to client.
### Caching in recursive name resolution

#### Recursive name resolution of \([nl, vu, cs, ftp]\)

<table>
<thead>
<tr>
<th>Server for node</th>
<th>Should resolve</th>
<th>Looks up</th>
<th>Passes to child</th>
<th>Receives and caches</th>
<th>Returns to requester</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cs)</td>
<td>([ftp])</td>
<td>([ftp])</td>
<td>—</td>
<td>—</td>
<td>([ftp])</td>
</tr>
<tr>
<td>(vu)</td>
<td>([cs, ftp])</td>
<td>([cs])</td>
<td>([ftp])</td>
<td>([ftp])</td>
<td>([cs]) ([cs, ftp])</td>
</tr>
<tr>
<td>(nl)</td>
<td>([vu, cs, ftp])</td>
<td>([vu])</td>
<td>([cs, ftp])</td>
<td>([cs]) ([cs, ftp])</td>
<td>([vu]) ([vu, cs]) ([vu, cs, ftp])</td>
</tr>
<tr>
<td>(root)</td>
<td>([nl, vu, cs, ftp])</td>
<td>([nl])</td>
<td>([vu, cs, ftp])</td>
<td>([vu]) ([vu, cs]) ([vu, cs, ftp])</td>
<td>([nl]) ([nl, vu]) ([nl, vu, cs]) ([nl, vu, cs, ftp])</td>
</tr>
</tbody>
</table>
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.
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. We can then apply extensive replication by mapping nodes to multiple servers, and start name resolution at the nearest server.
Scalability issues

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

Limitation

By mapping nodes to servers that can be located anywhere, we introduce an implicit location dependency.
Attribute-based naming

Observation
In many cases, it is much more convenient to name, and look up entities by means of their attributes ⇒ traditional directory services (aka yellow pages).
Observation
In many cases, it is much more convenient to name, and look up entities by means of their attributes ⇒ traditional directory services (aka yellow pages).

Problem
Lookup operations can be extremely expensive, as they require to match requested attribute values, against actual attribute values ⇒ inspect all entities (in principle).
Implementing directory services

Solution for scalable searching

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

Lightweight Directory Access Protocol (LDAP)

Each directory entry consists of \((attribute, value)\) pairs, and is uniquely named to ease lookups.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Abbr.</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country</td>
<td>C</td>
<td>NL</td>
</tr>
<tr>
<td>Locality</td>
<td>L</td>
<td>Amsterdam</td>
</tr>
<tr>
<td>Organization</td>
<td>O</td>
<td>VU University</td>
</tr>
<tr>
<td>OrganizationalUnit</td>
<td>OU</td>
<td>Computer Science</td>
</tr>
<tr>
<td>CommonName</td>
<td>CN</td>
<td>Main server</td>
</tr>
<tr>
<td>Mail Servers</td>
<td>–</td>
<td>137.37.20.3, 130.37.24.6, 137.37.20.10</td>
</tr>
<tr>
<td>FTP Server</td>
<td>–</td>
<td>130.37.20.20</td>
</tr>
<tr>
<td>WWW Server</td>
<td>–</td>
<td>130.37.20.20</td>
</tr>
</tbody>
</table>
LDAP

**Essence**

- **Directory Information Base**: collection of all directory entries in an LDAP service.
- Each record is uniquely named as a sequence of naming attributes (called **Relative Distinguished Name**), so that it can be looked up.
- **Directory Information Tree**: the naming graph of an LDAP directory service; each node represents a directory entry.

**Part of a directory information tree**

```
C = NL
O = VU University
OU = Computer Science
CN = Main server

HostName = star

HostName = zephyr
```
Two directory entries having *HostName* as RDN

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Locality</em></td>
<td><em>Amsterdam</em></td>
<td><em>Locality</em></td>
<td><em>Amsterdam</em></td>
</tr>
<tr>
<td><em>Organization</em></td>
<td><em>VU University</em></td>
<td><em>Organization</em></td>
<td><em>VU University</em></td>
</tr>
<tr>
<td><em>OrganizationalUnit</em></td>
<td><em>Computer Science</em></td>
<td><em>OrganizationalUnit</em></td>
<td><em>Computer Science</em></td>
</tr>
<tr>
<td><em>CommonName</em></td>
<td><em>Main server</em></td>
<td><em>CommonName</em></td>
<td><em>Main server</em></td>
</tr>
<tr>
<td><em>HostName</em></td>
<td><em>star</em></td>
<td><em>HostName</em></td>
<td><em>zephyr</em></td>
</tr>
<tr>
<td><em>HostAddress</em></td>
<td>192.31.231.42</td>
<td><em>HostAddress</em></td>
<td>137.37.20.10</td>
</tr>
</tbody>
</table>

Result of `search(```(C=NL) (O=VU University) (OU=*) (CN=Main server)```)`