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.

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

Note
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

Broadcast the ID, requesting the 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
Example: SSP chains

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

- Process P1
  - Client stub cs
  - Interprocess communication

- Process P2
  - Client stub cs*

- Server stub

- Process P3
  - Identical client stub
  - Local invocation
  - Identical server stub

- Process P4
  - Object

Stub cs* refers to the same server stub as stub cs.
Example: SSP chains

Redirecting a forwarding pointer by storing a shortcut in a client stub

(a) Invocation request is sent to object.

Server stub at object's current process returns the current location.

(b) Server stub is no longer referenced by any client stub.

Client stub sets a shortcut.
Home-based approaches

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

- Entity’s **home address** registered at a naming service
- The home registers the **foreign address** of the entity
- Client contacts the home first, and then continues with foreign location
The principle of mobile IP

1. Send packet to host at its home
2. Return address of current location
3. Tunnel packet to current location
4. Send successive packets to current location
Home-based approaches

Problems with home-based approaches

- Home address has to be supported for entity’s lifetime
- Home address is fixed ⇒ unnecessary burden when the entity permanently moves
- Poor geographical scalability (entity may be next to client)

Note

Permanent moves may be tackled with another level of naming (DNS)
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 $p$
Principle

- Each node $p$ maintains a finger table $FT_p[]$ with at most $m$ entries:

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

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

Chord finger tables
Chord finger tables

**Principle**

- Each node $p$ maintains a finger table $FT_p[]$ with at most $m$ entries:

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

  **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

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

Principle

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

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

- 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
class ChordNode:
    def finger(self, i):
        succ = (self.nodeID + \texttt{pow}(2, i-1)) \mod \texttt{self.MAXPROC} \quad \# \text{succ(p+2^{(i-1)})}

        lwbi = \texttt{self.nodeSet.index}(self.nodeID) \quad \# \text{self in nodeset}
        upbi = (lwbi + 1) \mod \texttt{len(self.nodeSet)} \quad \# \text{next neighbor}

        for k in \texttt{range(len(self.nodeSet))}: \quad \# \text{process segments}
            if self.inbetween(succ, self.nodeSet[lwbi]+1, self.nodeSet[upbi]+1):
                return self.nodeSet[upbi] \quad \# \text{found successor}

                (lwbi, upbi) = (upbi, (upbi+1) \mod \texttt{len(self.nodeSet)}) \quad \# \text{next segment}

    def recomputeFingerTable(self):
        self.FT[0] = self.nodeSet[\texttt{self.nodeSet.index}(self.nodeID)-1] \quad \# \text{Pred.}
        self.FT[1:] = [self.finger(i) \texttt{for i in range}(1,\texttt{self.nBits+1})] \quad \# \text{Succ.}

    def localSuccNode(self, key):
        if self.inbetween(key, self.FT[0]+1, self.nodeID+1): \quad \# \text{in (FT[0],self]}
            return self.nodeID \quad \# \text{responsible node}

        elif self.inbetween(key, self.nodeID+1, self.FT[1]): \quad \# \text{in (self,FT[1])}
            return self.FT[1] \quad \# \text{succ. responsible}

        for i in \texttt{range}(1,\texttt{self.nBits+1}):
            if self.inbetween(key, self.FT[i], self.FT[(i+1) \mod \texttt{self.nBits}]): \quad \# \text{in [FT[i],FT[i+1])}
                return self.FT[i]
Exploiting network proximity

Problem
The logical organization of nodes in the overlay may lead to erratic message transfers in the underlying Internet: node $p$ and node $\text{succ}(p + 1)$ may be very far apart.

Solutions
Exploiting network proximity

Problem
The logical organization of nodes in the overlay may lead to erratic message transfers in the underlying Internet: node $p$ and node $\text{succ}(p+1)$ may be very far apart.

Solutions
- **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.
Exploiting network proximity

**Problem**

The logical organization of nodes in the overlay may lead to **erratic message transfers** in the underlying Internet: node $p$ and node $\text{succ}(p + 1)$ may be very far apart.

**Solutions**

- **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 $FT_p[i]$ points to first node in $\text{INT} = [p + 2^{i-1}, p + 2^i - 1]$. Node $p$ can also store pointers to other nodes in $\text{INT}$. 
Exploiting network proximity

Problem
The logical organization of nodes in the overlay may lead to erratic message transfers in the underlying Internet: node $p$ and node $\text{succ}(p + 1)$ may be very far apart.

Solutions

- **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 $FT_p[i]$ points to first node in $\text{INT} = [p + 2^{i-1}, p + 2^i - 1]$. Node $p$ can also store pointers to other nodes in $\text{INT}$.

- **Proximity neighbor selection**: When there is a choice of selecting who your neighbor will be (not in Chord), pick the closest one.
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**

- **Top-level domain** $T$
- **Directory node** $\text{dir}(S)$ of domain $S$
- **A subdomain** $S$ of top-level domain $T$ ($S$ is contained in $T$)
- **A leaf domain**, contained in $S$

The root directory node $\text{dir}(T)$
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
HLS: Lookup operation

Basic principles

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

Looking up a location
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
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.

Notation

- Assume there are a total of $N$ physical hosts $\{H_1, H_2, \ldots, H_N\}$. Each host is capable of running one or more location servers.
- $D_k(A)$ denotes the domain at level $k$ that contains address $A$; $k = 0$ denotes the root domain.
- $LS_k(E, A)$ denotes the unique location server in $D_k(A)$ responsible for keeping track of entity $E$. 
Can an HLS scale?

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

- \( D_k = \{D_{k,1}, D_{k,2}, \ldots, D_{k,N_k}\} \) denotes the \( N_k \) domains at level \( k \)
- **Note:** \( N_0 = |D_0| = 1 \).
- For each level \( k \), the set of hosts is partitioned into \( N_k \) subsets, with each host running a location server representing exactly one of the domains \( D_{k,i} \) from \( D_k \).

Principle of distributing logical location servers

![Diagram showing the principle of distributing logical location servers](image-url)
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

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
- ...

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
- ...

Note

Directory nodes can also have attributes, besides just storing a directory table with \( (identifier, label) \) 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?
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.
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
The concept of a symbolic link explained in a naming graph

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

Node n5 has only one name

Observation

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

- Name server
- Machine A
- Remote keys
- "nfs://flits.cs.vu.nl/home/steen"
- Name server for foreign name space
- Machine B
- Home
- Steen
- Mbox

Reference to foreign name space
Network
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
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
# 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, ftp]</td>
</tr>
<tr>
<td>(nl)</td>
<td>([vu, cs, ftp])</td>
<td>#[vu]</td>
<td>([cs, ftp])</td>
<td>#[cs, ftp]</td>
<td>#[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, cs, ftp]</td>
<td>#[nl, vu, cs, ftp]</td>
</tr>
</tbody>
</table>

Implementation of name resolution
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

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.

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

Problem
By mapping nodes to servers that can be located anywhere, we introduce an implicit location dependency.
DNS

Essence

- Hierarchically organized name space with each node having exactly one incoming edge \( \Rightarrow \) edge label = node label.
- **domain**: a subtree
- **domain name**: a path name to a domain’s root node.

Information in a node

<table>
<thead>
<tr>
<th>Type</th>
<th>Refers to</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA</td>
<td>Zone</td>
<td>Holds info on the represented zone</td>
</tr>
<tr>
<td>A</td>
<td>Host</td>
<td>IP addr. of host this node represents</td>
</tr>
<tr>
<td>MX</td>
<td>Domain</td>
<td>Mail server to handle mail for this node</td>
</tr>
<tr>
<td>SRV</td>
<td>Domain</td>
<td>Server handling a specific service</td>
</tr>
<tr>
<td>NS</td>
<td>Zone</td>
<td>Name server for the represented zone</td>
</tr>
<tr>
<td>CNAME</td>
<td>Node</td>
<td>Symbolic link</td>
</tr>
<tr>
<td>PTR</td>
<td>Host</td>
<td>Canonical name of a host</td>
</tr>
<tr>
<td>HINFO</td>
<td>Host</td>
<td>Info on this host</td>
</tr>
<tr>
<td>TXT</td>
<td>Any kind</td>
<td>Any info considered useful</td>
</tr>
</tbody>
</table>
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).
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).

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 \( (\text{attribute}, \text{value}) \) 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>Locality</td>
<td>Amsterdam</td>
<td>Locality</td>
<td>Amsterdam</td>
</tr>
<tr>
<td>Organization</td>
<td>VU University</td>
<td>Organization</td>
<td>VU University</td>
</tr>
<tr>
<td>OrganizationalUnit</td>
<td>Computer Science</td>
<td>OrganizationalUnit</td>
<td>Computer Science</td>
</tr>
<tr>
<td>CommonName</td>
<td>Main server</td>
<td>CommonName</td>
<td>Main server</td>
</tr>
<tr>
<td>HostName</td>
<td>star</td>
<td>HostName</td>
<td>zephyr</td>
</tr>
<tr>
<td>HostAddress</td>
<td>192.31.231.42</td>
<td>HostAddress</td>
<td>137.37.20.10</td>
</tr>
</tbody>
</table>

Result of `search(''(C=NL)(O=VU University)(OU=*)(CN=Main server)'')`
Naming: Attribute-based naming

Distributed index

Basic idea

- Assume a set of attributes \( \{a^1, \ldots, a^N\} \)
- Each attribute \( a^k \) takes values from a set \( R^k \)
- For each attribute \( a^k \) associate a set \( S^k = \{S^k_1, \ldots, S^k_{n_k}\} \) of \( n_k \) servers
- Global mapping \( F: F(a^k, v) = S^k_j \) with \( S^k_j \in S^k \) and \( v \in R^k \)

Observation

If \( L(a^k, v) \) is set of keys returned by \( F(a^k, v) \), then a query can be formulated as a logical expression, e.g.,

\[
(F(a^1, v^1) \land F(a^2, v^2)) \lor F(a^3, v^3)
\]

which can be processed by the client by constructing the set

\[
(L(a^1, v^1) \cap L(a^2, v^2)) \cup L(a^3, v^3)
\]
Drawbacks of distributed index

Quite a few

- A query involving \( k \) attributes requires contacting \( k \) servers.
- Imagine looking up "\( lastName = Smith \land firstName = Pheriby \)": the client may need to process many files as there are so many people named "Smith."
- No (easy) support for range queries, such as "\( price = [1000 - 2500] \)."
Alternative: map all attributes to 1 dimension and then index

Space-filling curves: principle

1. Map the $N$-dimensional space covered by the $N$ attributes $\{a^1, \ldots, a^N\}$ into a single dimension
2. Hashing values in order to distribute the 1-dimensional space among index servers.

Hilbert space-filling curve of (a) order 1, and (b) order 4
Once the curve has been drawn

Consider the two-dimensional case

- A Hilbert curve of order $k$ connects $2^{2k}$ subsquares $\Rightarrow$ has $2^{2k}$ indices.
- A range query corresponds to a rectangle $R$ in the 2-dimensional case.
- $R$ intersects with a number of subsquares, each one corresponding to an index $\Rightarrow$ we now have a series of indices associated with $R$.

Getting to the entities

Each index is to be mapped to a server, who keeps a reference to the associated entity. One possible solution: use a DHT.