Naming Entities

- Names, identifiers, and addresses
- Name resolution
- Name space implementation

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.

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

Flat naming

Problem

Given an essentially unstructured name (e.g., an identifier), how can we locate its associated access point?

- Simple solutions (broadcasting)
- Home-based approaches
- Distributed Hash Tables (structured P2P)
- Hierarchical location service

Simple solutions

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

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

Simple solutions

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

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

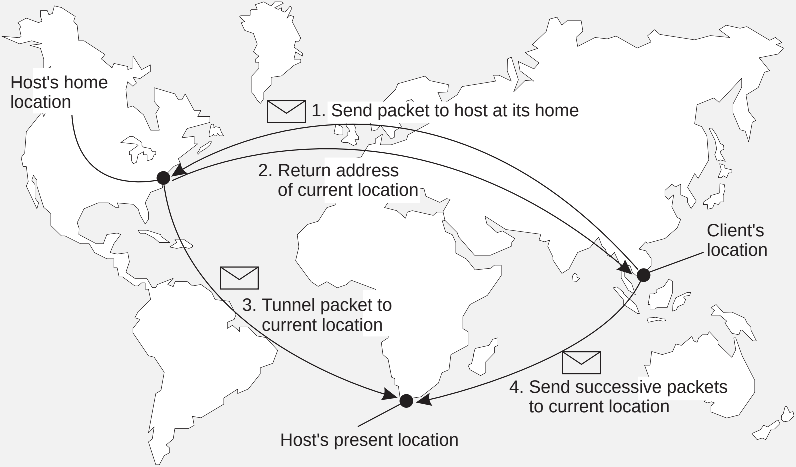
Home-based approaches

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

Home-based approaches



Home-based approaches

Two-tiered scheme

Keep track of visiting entities:

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

Problems with home-based approaches

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

Question

How can we solve the "permanent move" problem?

Home-based approaches

Two-tiered scheme

Keep track of visiting entities:

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

Problems with home-based approaches

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

Question

How can we solve the "permanent move" problem?

Home-based approaches

Two-tiered scheme

Keep track of visiting entities:

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

Problems with home-based approaches

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

Question

How can we solve the "permanent move" problem?

Distributed Hash Tables (DHT)

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).

Nonsolution

Let node *id* keep track of $succ(id)$ and start linear search along the ring.

DHTs: 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: $FT_p[i]$ 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 $FT_p[1]$

DHTs: 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: $FT_p[i]$ 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 $FT_p[1]$

DHTs: Finger tables

Principle

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

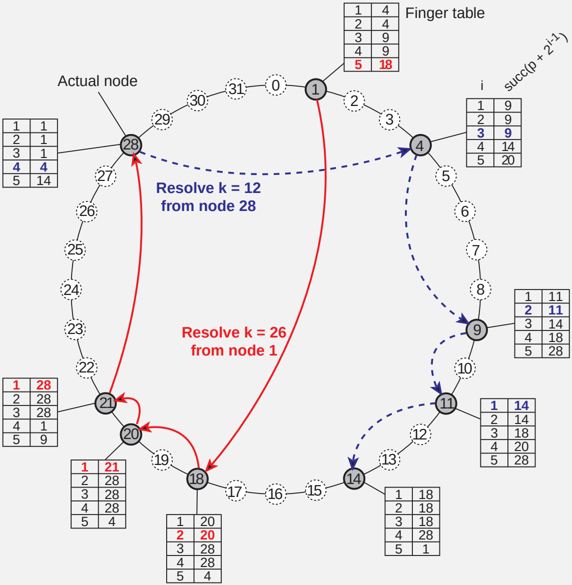
$$FT_p[i] = succ(p+2^{i-1})$$

  Note: $FT_p[i]$ 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 $FT_p[1]$

DHTs: Finger tables



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.

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 $INT = [p+2^{i-1}, p+2^i-1]$. Node p can also store pointers to other nodes in INT.

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

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.

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 $INT = [p+2^{i-1}, p+2^i-1]$. Node p can also store pointers to other nodes in INT.

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

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.

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 $INT = [p+2^{i-1}, p+2^i-1]$. Node p can also store pointers to other nodes in INT.

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

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.

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.

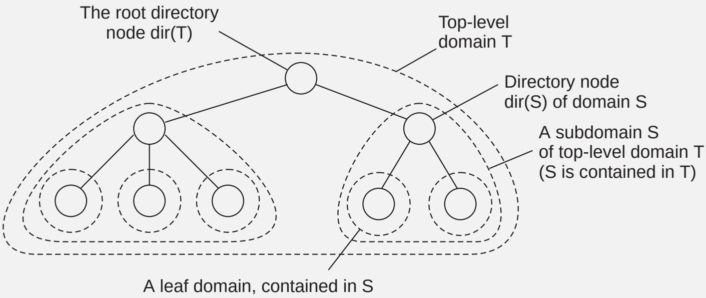
Proximity routing: Maintain more than one possible successor, and forward to the closest.
Example: in Chord $FT_p[i]$ points to first node in $INT = [p+2^{i-1}, p+2^i-1]$. Node p can also store pointers to other nodes in INT.

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

Hierarchical Location Services (HLS)

Basic idea

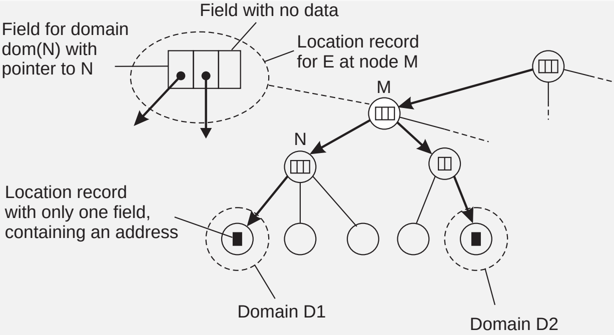
Build a large-scale search tree for which the underlying network is divided into hierarchical domains. Each domain is represented by a separate directory node.



HLS: Tree organization

Invariants

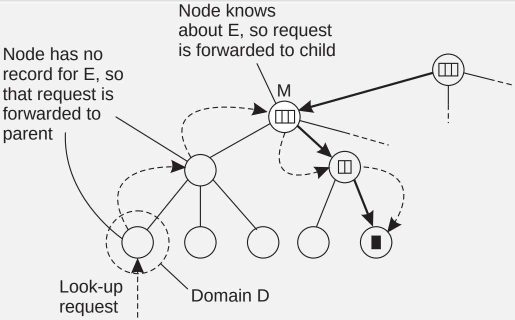
- Address of entity *E* is stored in a leaf or intermediate node
- Intermediate nodes contain a pointer to a child iff the subtree rooted at the child stores an address of the entity
- The root knows about all entities



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



HLS: Insert operation