Lecture 9: Naming

**Basic Concepts**

- **Name:**
  - String of bits or characters
  - Refers to an entity

- **Entity:**
  - Resource, process, user, etc.
  - Operations performed on entities at access points

- **Address:**
  - Access point named by an address
  - Entity address = address of entity’s access point
  - Multiple access points per entity
  - Entity’s access points may change

- **Identifier:**
  - Name that uniquely identifies entity
  - Properties:
    1. Refers to at most one entity
    2. Entity referred to by at most one identifier
    3. Always refers to same entity (i.e. no reuse)
  - Allows easy comparison of references

**What is Naming?**

Systems manage a wide collection of entities of different kinds. They are identified by different kinds of names:

- Files (/boot/vmlinuz), Processes (1, 14293), Users (chak, ikuz, cs9243), Hosts (weill, facebook.com),...

Examples of naming in distributed systems? What’s the difficulty?
**SYSTEM-ORIENTED VS HUMAN-ORIENTED NAMES**

**System-Oriented Names:**
- Represented in machine readable form (32 or 64 bit strings)
- Structured or unstructured
- Easy to store, manipulate, compare
- Not easy to remember, hard for humans to use
- Example: inode (0x00245dad)

**Human-Oriented Names:**
- Variable length character strings
- Usually structured
- Often many human-oriented names map onto a single system-oriented name
- Easy to remember and distinguish between
- Hard for machine to process
- Example: URL (http://www.cse.unsw.edu.au/~cs9243/lectures)

**NAME SPACES**

Container for a set of related names

**Slide 6**
**Structure options:**
- Flat (only leaf nodes)
- Hierarchical (Strictly hierarchical, DAG, Multiple root nodes)
- Tag-based

**Slide 7**
**Path Names (in hierarchies):**
- Sequence of edge labels
- Absolute: if first node in path name is a root node
- Relative: otherwise

**Slide 8**
**Aliasing:**
- Alias: another name for an entity
- Hard link: two or more paths to an entity in the graph
- Soft link: leaf node stores a (absolute) path name to another node
Merging:

- Mounting
  - Directory node stores info about a directory node in other name space
  - Need: protocol, server, path name, authentication and authorisation info, keys for secure communication, etc.

Combining name spaces

- Name Spaces: Protocol, DNS, File System

### Naming Services

A naming service provides a name space

Name Server:
- Naming service implemented by name servers
- Implements naming service operations

Operations:
- Lookup: resolve a path name, or element of a path name
- Add: add a directory or leaf node
- Remove: remove a subtree or leaf node
- Modify: modify the contents of a directory or leaf node

Client:
- Invokes naming service operations

Centralised vs Distributed Naming Service

### Name Resolution

The process of looking up a name

Resolution:
- Mapping a name onto the node referred to by the name
- Interested in the data stored by the node

Path Name Resolution:
- Starts at a begin node (first element of the path name)
  - Root node for absolute name
  - Directory node for relative name
- Ends with data from (or a reference to) the last node (last element of path name)

Resolver:
- Does name resolution on behalf of client
- In client process, in client’s kernel, process on client’s machine

Iterative Resolution:

- Caching only at resolver
- Lots of communication
Recursive Resolution:

Split name space over multiple servers

Structured Partitioning:
- Split name space according to graph structure
- Name resolution can use zone hints to quickly find appropriate server

Improved lookup performance due to knowledge of structure
- Rigid structure

Fault Tolerance:
- Replication

Flexible
- Decreased lookup performance, increased load on root

Naming Service Implementation Issues

Performance and Scalability:
- Limit load on name servers
- Limit communication required
- Partitioning: split name space over multiple name servers
- Replication: copy (parts of) name space on multiple name servers

Fault Tolerance:
- Replication

Authoritative Name Server:
- Name server that stores an entity’s original attributes
**REPLICATION**

Copy name space to multiple servers

**Full Replication:**
- ➜ copy complete name space
- ✓ Fast performance
- x Size (each server must store whole name space)
- x Consistency (any change has to be performed at all replicas)
- x Administration (who has rights to make changes where?)

**Partial replication:**
- ➜ Replicate full name servers
- Replicate zones
- ✓ Improved performance, less consistency overhead
- ✓ Less administrative problems

---

**Caching:**

- ➜ Cache query results
- ✓ No administrative problems
- Types of caches:
  - Directory cache: cache directory node information
  - Prefix cache: cache path name prefixes
  - Full-name cache: cache full names

- ➜ Cache implementations:
  - Process-local cache: in address space of process
  - Kernel cache: cache kept by kernel
  - User-process cache: separate shared service

- Cache updates and consistency
  - On use checking
  - Timeout
  - Invalidation
  - Slow propagation

---

**DNS (Domain Name System)**

**Structure:**

- ➜ Hierarchical structure (tree)
- Top-level domains (TLD) (.com, .org, .net, .au, .nl, ...)
- Zone: a (group of) directory node
- Resource records: contents of a node
- Domain: a subtree of the global tree
- Domain name: an absolute path name

---

<table>
<thead>
<tr>
<th>Type of record</th>
<th>Associated entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA</td>
<td>Zone</td>
<td>Holds information on the represented zone</td>
</tr>
<tr>
<td>A</td>
<td>Host</td>
<td>Contains an IP address of the host this node represents</td>
</tr>
<tr>
<td>MX</td>
<td>Domain</td>
<td>Refers to a mail server to handle mail addressed to this node</td>
</tr>
<tr>
<td>SRV</td>
<td>Domain</td>
<td>Refers to a server handling a specific service</td>
</tr>
<tr>
<td>NS</td>
<td>Zone</td>
<td>Refers to a name server that implements this represented zone</td>
</tr>
<tr>
<td>CNAME</td>
<td>Node</td>
<td>Symbolic link with the primary name of the represented node</td>
</tr>
<tr>
<td>PTR</td>
<td>Host</td>
<td>Contains the canonical name of a host</td>
</tr>
<tr>
<td>HINFO</td>
<td>Host</td>
<td>Holds information on the host this node represents</td>
</tr>
<tr>
<td>TXT</td>
<td>Any kind</td>
<td>Contains any entity-specific information considered useful</td>
</tr>
</tbody>
</table>
Partitioning:
- Each zone implemented by a name server

Replication:
- Each zone replicated on at least two servers
- Updates performed on primary
- Contents transferred to secondary using zone transfer
- Higher levels have many more replicas (13 root servers: A-M.root-servers.net. Actually 386 replicas using anycast)

Caching:
- Servers cache results of queries
- Original entries have time-to-live field (TTL)
- Cached data is non-authoritative, provided until TTL expires

Name Resolution:
- Query sent to local server
- If cannot resolve locally then sent to root
- Resolved recursively or iteratively

Attribute-Based Naming (＆LDAP)

White Pages vs Yellow Pages:
- White Pages: Name ➤ Phone number
- Yellow Pages: Attribute ➤ Set of entities with that attribute
- Example: X.500 and LDAP

Attribute-Based Names:
- Example: /C=AU/O=UNSW/OU=*)(CN=WWW Server)
- Attributes stored in directory entry, all stored in directory

Name Space:
- Flat: no structure in directory service
- Hierarchical: structured according to a hierarchy
- Distinguished name (DN): set of attributes (distinguished attributes) that forms a canonical name for an entity
**Directory Services**

A directory service implements a directory

**Operations:**
- **Lookup:** resolve a distinguished name
- **Add:** add an entity
- **Remove:** remove an entity
- **Modify:** modify the attributes of an entity
- **Search:** search for entities that have particular attributes
- **Search can use partial knowledge**
- **Search does not have to include distinguished attributes**
- **Most important qualities:** allow browsing and allow searching

**Client:**
- Invokes directory service operations

**Partitioning:**
- Partitioned according to name space structure (e.g., hierarchy)

**Replication:**
- Replicate whole directory
- Replicate partitions
- Read/Write and read only replicas (e.g. primary-backup)
- Catalog and cache replicas
**Searching and Lookup in a Distributed Directory**

- **Approaches:**
  - Chaining (recursive)
  - Referral (iterative)
  - Multicasting (uncommon)

- **Performance of Searching:**
  - Searching whole name space: must visit each directory server
  - Bad scalability
  - Limit searches by specifying context
  - Catalog: stores copy of subset of DIB information in each server
  - Main problem: multiple attributes mean multiple possible decompositions for partitioning BUT only one decomposition can be implemented

**X.500 and LDAP**

- **X.500:**
  - ISO standard
  - Global DIT
  - Defines DIB, DIB partitioning, and DIB replication

- **LDAP (Lightweight Directory Access Protocol):**
  - X.500 access over TCP/IP
  - X.500 is defined for OSI Application layer
  - Textual X.500 name representation
  - Popular on Internet
  - Also X.500 free implementations (e.g. openldap)
  - Used in Windows for Active Directory

**Address Resolution of Unstructured Names**

- **Unstructured Names:**
  - Practically random bit strings
  - Example: random key, hash value
  - No location information whatsoever
  - How to find corresponding address of entity?
Simple Solution: Broadcasting:

- Resolver broadcasts query to every node
- Only nodes that have access point will answer

Example – ARP:

- Resolver broadcasts: Who has 129.94.242.201? Tell 129.94.242.200
- 129.94.242.201 answers to 129.94.242.200: 129.94.242.201 is at 00:15:C5:FB:AD:95

DISTRIBUTED HASH TABLES

Hash table (key value store) as overlay network:

- put(key, value), value = get(key), remove(key)

Example: look up unstructured host names:

- put(weill, 129.94.242.49)
- put(beethoven, 129.94.172.11)
- put(maestro, 129.94.242.33)

address = get(beethoven)

- How high is performance cost of lookup?

CHORD: DISTRIBUTED HASH TABLE

General Structure:

- keys and node IP addresses mapped to identifier
- consistent hashing (SHA-1 m-bits)
- key assigned to first node with id > key → successor(key)

A simple lookup:

- use successors function
- recursive RPCs until node with key is found
- O(1) cost
A scalable lookup:

- Routing table at every node: finger table
- $i$th entry is $\text{successor}(n + 2^{i-1})$
- $\text{finger}[1]$ is successor

Slide 37

- Lookup greatest node id in table < $k$
- Ask it to lookup the key
- Exponentially smaller jumps

Slide 38

Adding a node:

- Stabilize: ensure successor pointers up-to-date
- Fix fingers: ensure that finger tables updated

Slide 39

Dealing with node failure:

- Successor list: $r$ successors to handle $r - 1$ failures
- Higher level must handle loss of data relating to failure

Slide 40

Analysis:

- Finger table size: $O(\log n)$
- $O(\log n)$ nodes contacted for lookup
- $1/2\log n$ average
Homework

- How could you use a DHT to implement a directory service?
- How could you use a DHT to implement a file system?

Hacker’s edition:

- Use an existing DHT implementation to implement a simple file system.
- Implement the DHT yourself

Reading List

Domain Names - Implementation and Specification RFC 1035
DNS

The Lightweight Directory Access Protocol: X.500 Lite
LDAP

Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications
Chord