Today
- What’s in a name
- Flat naming
- Structured naming
- Attribute-based naming
Names, identifiers and addresses

- Names are used to denote entities in a system
  - Hosts, printers, files, processes, users ….

- To operate on an entity, e.g. print a file, we need to access it at an access point
  - An entity can offer 1+ access points (think of phone #s)
Access points are entities as well; the name of an access point is an address
  – Address of an access point of an entity = address of an entity (person – telephone - telephone #)

Location-independent name for entity $E$ – a name independent from the addresses of $E$’s access points
  – Easier and more flexible to use
  – $E$ can have multiple addresses and change them
Names, identifiers and addresses

- Pure names are un-interpreted bit patterns
  - Have no meaning, they are just random strings
  - Can be used for comparison only
  - No-pure names contain information about the object that they name (e.g., location)

- (True) Identifiers are names interpreted by programs
  - Each identifier refers to at most one entity (no reusing)
  - Each entity is referred to by at most one identifier
  - E.g. phone is not, passport number maybe

- An identifier need not necessarily be a pure name, i.e., it may have content
Names, identifiers and addresses

- Name resolution – a name is resolved when it is translated into data about the named object
- The association between a name and an object is called a *binding*
  - In general names are bound to attributes of an objects rather than the object itself; e.g., address
- Alias – a name defined to denote the same info as another name
  - Simple allow multiple absolute paths names refer to the same node – hard link
  - Allow a node $S$ to contain a name of another node – soft link
Name space

• Name space – the collection of all valid names recognized by a particular service

• Hierarchical names spaces
  – Have an internal structure that represent their position in a hierarchic name space

• Flat names spaces
  – Names chosen from a flat set of numeric or symbolic identifiers
Hierarchical name spaces

• Advantages of hierarchies
  – Makes large name spaces more manageable
  – They are potentially infinite (rather than limited by the length/base of the id space)

• Hierarchical name space
  – A graph in which a leaf node represents a (named) entity; a directory node is an entity that refers to other nodes
  – A directory node contains a (directory) table of \((\text{edge label, node identifier})\) pairs
  – We can easily store all kinds of attributes in a node, describing aspects of the entity the node represents
Name services

- Names service – stores a collection of textual names, in the form of bindings between names and attributes
  - Collection often subdivided in 1+ contexts managed as a unit
- Key operation supported – resolution
- Why is name management a separate service?
  - Unification – resources managed by different services using the same naming scheme
  - Integration - resources created in different administrative domain may eventually be shared (and so must have common naming)
Name services

- Original name services were simple, serving a limited management domain (LAN)
- Global Name Service’s goals (Lampson ‘86)
  - To handle an essentially arbitrary number of names and serve an arbitrary number of administrative organizations
  - Have a long lifetime
  - Provide high availability, fault isolation and tolerance to mistrust
Name resolution

- On the client’s behalf, a local name resolver
- To resolve a name we need a directory node. How do we actually find that (initial) node?
- Closure mechanism – needed to select the implicit context from which to start name resolution
  - www.eecs.northwestern.edu: start at a DNS name server

Name resolver can follow
- Interactive – client drives the resolution
  - Caching by clients
  - Potentially costly communication
- Recursive – the server does
  - Higher performance demand on servers
  - More effective caching
  - Reduced communication costs
Name resolution and flat names

- Given an essentially unstructured name, how to locate its associated access point?
- Key point – name does not contain any information on how to locate the access point
  - Simple solutions
    - Broadcasting
    - Forwarding pointers
  - Home-based approaches
  - Distributed Hash Tables
  - Hierarchical location service
Simple solutions - broadcasting

- Broadcasting – simply broadcast the ID, requesting the entity to return its current address
  - Think Address Resolution Protocol (ARP)
  - Can never scale beyond local-area networks
  - Requires all processes to listen to incoming location requests

- Multicasting can help
  - Join a multicast address and wait for your call
  - Can (sort-of) help when selecting nearest replica
Simple solutions – forwarding pointers

- Forwarding pointers – each time an entity moves, it leaves behind a pointer telling where it has gone to
  - Dereferencing can be made entirely transparent to clients by simply following the chain of pointers
  - Update a client’s reference as soon as present location has been found
- Geographical scalability problems:
  - Long chains are not fault tolerant
  - Increased network latency at dereferencing

Essential to have separate chain reduction mechanisms
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 home first, and then continues with foreign location
  - E.g. Mobile IP

[Diagram showing the sequence of steps for a packet: 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

- **Two-tiered scheme – keep track of visiting entities:**
  - Check local visitor register first
  - Fall back to home location if local lookup fails

- **Some problems**
  - Home address has to be supported as long as the entity lives
  - Home address is fixed, which means an unnecessary burden when the entity permanently moves to another location
  - Poor geographical scalability (entity may be next to the client)
Distributed Hash Tables (DHT)

- Chord, an easy to explain DHT
- Nodes arranged 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)
- Main issue – efficiently resolve a key $k$ to the address of $succ(k)$
- Non-solution: Let each node $p$ keep track of $succ(p+1)$ (and predecessor $pred(p)$) and do a linear search along the ring
Distributed Hash Tables (DHT)

- Finger tables – each node $p$ maintains a finger table $FT_p[.]$ with at most $m$ entries:
  - $FT_p[i] = \text{succ}(p + 2^{i-1})$
  - i.e. $i$ entry points to first node succeeding $p$ by at least $2^{i-1}$
  - Basically shortcuts to nodes in the identifier space
  - Shortcut distance grows exponentially as the index increases

- To look up key $k$,
  - $p$ forwards the request to node $q$ 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

Resolving key 26 from node 1

FT\(_1[5]\) = 18 ≤ 26

FT\(_{18}[2]\) < 26 ≤ FT\(_{18}[3]\)
Hierarchical location system

- Build a large-scale search tree – the underlying network is divided into hierarchical domains. Each domain is represented by a separate directory node.
- Description based on van Steen et al.’s Globe.
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 sub-tree rooted at the child stores an address of the entity.
- The root knows about all entities.

If entity has an address in leaf domain $D_1$ and $D_2$, the directory node of smallest domain (M) containing both $D_1$ and $D_2$ will have two pointers one for each subdomain.
HLS lookups and inserts

- Start lookup at local leaf node
- If node knows it, follow downward pointer, otherwise go one up
- Upward lookup always stops at root

- Insertion started in leaf domain D – node dir(D)
- This forwards to parent, … until it reaches directory node M, first node that knows about E
- *Request is push down with each node creating a location record*
Hierarchical – The Domain Name System

- Naming for the Internet before DNS (1983) each computer retrieved HOST.TXT from a computer at SRI
  - Legacy – a host file still exist in most modern OS
- DNS First design and implementation by Paul Mockapetris (1983)
- Berkeley Internet Name Domain Server (BIND) by Doug Terry, Mark Painter, David Riggle and Songnian Zhou (1984)
- Kevin Dunlap rewrote it in 1985 and Paul Vixie has been maintaining it since
DNS

• Domain names
  – Name space consist of a tree of domain names; each node has 0+ resource records
    • Resource record, e.g., SOA, A, CNAME, NS, MX …
  – Names are written with the highest-level domain on the right
  – Tree sub-divides into zones beginning at the root
    • Each zone could be 1+ domains and sub-domains

• Zone files – the txt file that describes a zone
  – Includes name and address for 2+ authoritative servers and for delegated subdomains
  – Management parameters (e.g. caching) and RR
  – The information contained in each zone is implemented in 2+ name servers (2 for redundancy)
DNS resolution and other uses

- DNS client, a.k.a. resolver, normally a library
- Follow simple request/reply over UDP, possible with multiple queries/responses per request/reply
- DNS supports recursive (optional) and iterative resolution
- Other uses
  - DNS, network positioning and CDNs
  - Extension mechanism for DNS (EDNS) introduced by Vixie in 1999 (an extension to messages, not in any zone file)
Directory services

- May be more convenient to name, and look up entities by attributes \{(attribute, value)\}
  - Simply agreeing on attributes and their values is a challenge

- Directory service
  - Stores collection of bindings between names and attributes
  - Allows attribute-based queries (so a.k.a. attribute-based name services or yellow-age services)
  - Expensive lookup – basically a exhaustive search!
Directory services

- Implement basic directory service as database, and combine with traditional structured naming system
  - LDAP, Active Directory

- Entities’ descriptions are translated into attribute-value trees which are encoded into a set of unique hash ids for a DHT
  - INS/Twine, SWORD, Mercury

```plaintext
description {
  type = book
  description {
    author = Tolkien
    title = LOTR
  }
  genre = fantasy
}

description {
  type = book
  author = Tolkien
  title = LOTR
  genre = fantasy
}

Query: Books written by Tolkien?
hash

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)
```
Summary

- Naming is central to computer systems in general and distributed systems in particular.
- How do you name things?
- How do you find what you are looking for?
- What if that’s a moving target?
- How do you implement name/directory services in an scalable manner?