9

Names and Discovery

Names are one of the first things you think about when the subject of identity comes up. Of course, identity is about more than names, but we name almost every object around us and so names become one of the most common attributes stored with an identity.

Once you've named a bunch of objects, you want to be able to find them. This is called *discovery*. Directories are one of the fundamental building blocks of discovery. As a result, information technology systems are full of directories. There are directories for files, directories for email addresses, directories for domain names, and even directories for running processes. The simplest directories associate a name with something else, such as a file, address, IP number, or process.

This chapter will discuss names, discovery, directories and the role that these play in digital identity.

Utah.gov: Naming and Directories

When I was CIO of Utah, directory issues seemed to take up a lot of our time and effort. When I became CIO, the state had been using the domain name state.ut.us. This domain name was not particularly easy to remember and when you tacked on one or two subdomains to identify a department or agency, the affect was almost comical. For example, my email address was pwindley@gov.state.ut.us. The Governor remarked that he could almost feel people start to dance to the rhythm when he told them his email address.

In addition to the official domain name, agencies in state government had gotten into the habit of registering domain names in the .org TLD (top-level domain) for every publicly facing Web site they started and Utah managed over 100 domain names outside the official one. This created a huge problem in building brand awareness around the State's Web site and meant that it was impossible to know when you were on an official State Web site and when you were not.

Shortly after I came on board, I discovered that we owned the domain name utah.gov—much shorter, much easier to remember, and more authoritative. By fiat and with the Governor's support, I declared that Utah was moving to utah.gov. Now, this is not a strategy I'd recommend as a way to endear yourself to people, but it did accomplish the goal: within a month, we were using utah.gov as the domain name for our primary Web server and contemplating how to migrate the rest of the organization.

There were two primary issues.

- Utah.gov represented a namespace that had been delegated to the State of Utah and within which we could manage things like server names and email addresses.
- The State had never had an enterprise strategy for naming and each department and agency ran its own directory service for email and passwords—some ran many with each division controlling their own directories.

The first problem called for the creation of a registration process and the appointment of a registrar though whom organizations within the State could reserve subdomains within utah.gov. In essence the job of the registrar was to create namespaces within utah.gov and ensure that the names were unique, meaningful and correctly recorded.

The second problem was more difficult. The first step was to create a voluntary program though which people who wanted an email address at utah.gov could reserve a name. A simple program forwarded email sent to that name on to their real mailbox. That step was only temporary while we went through the difficult process of creating a naming procedure for assigning unique names (which would become email addresses) to each employee. We finally settled on first initial/last name scheme with a series of fallback schemes for duplicates. The policy specifically prohibited names not associated with a person's real name to prevent people having email addresses like dumbo@utah.gov (unless their real name happened to be Doug Umbo, of course).

We also set up a metadirectory (more on this later) so that the directories in the agencies could cooperate to form a single large logical directory. This wasn't as easy as it should have been since software running many of the directories hadn't been updated for years and didn't support metadirectory linking. Thus, creating a single directory included upgrade projects for a number of directories around the State. Further, creating this logical directory from already existing directories meant that the names in those directories had to first be normalized according to the naming scheme we'd come up with earlier.

The use of multiple distributed directories had advantages in performance and local control, but caused some difficulties with integration to other enterprise systems like the HR system. The ultimate goal is to provision entries in the directory and even access control rights based on the employee's status within the HR system.

The technical problems faced in creating an enterprise directory pale in comparison to the political challenges. To begin with, you're asking many people to change their email address—some of which had been in use for many years. This has personal and organizational costs. Second, there are some people who are more equal than the rest and cannot be asked to change their email address. They get first pick of email addresses if there's a conflict. One executive director even insisted on having every possible permutation of her name and initials assigned to her to prevent anyone from accidentally sending mail intended for her to someone else with a similar name.

Ultimately we were successful in establishing a single namespace within utah.gov for all email and logins. We even converted the State's many Web servers to subdomains within the utah.gov domain. The effort took almost two years to complete, but once done, enhanced our ability to brand the State's Web services and gave people email addresses that they could give to people without having to break out the bongo drums.

Naming

Names are used to refer to things. Without names, we'd constantly be describing people, places, and things to each other whenever we wanted to talk about them. You do that now when you can't remember someone's name: "You, know, the guy who was in the green shirt, with the beard, walking a dog?" Any given entity can have multiple names that all refer to the same thing. I'm Phil, Phillip, Phil Windley, Dad, and so on depending on the circumstance.

Naming is one of the fundamental abstractions for dealing with complexity. Names provide convenient handles for large, complex things—allowing them to be manipulated and referenced by some short, easy to remember string, instead of a longer, unwieldy representation. File names, for example, let us pick a meaningful handle for what ultimately is a collection of bits located on a particular set of sectors on a particular set of tracks on a particular set of disks.

In computing, we use names for similar reasons. We want to easily refer to things like memory locations (variables), inodes (file names), IP addressed (domain names), and so on. Names usually possess several important properties, including:

- Names should be unique within some specific namespace
- Names should be memorable
- Names should be short enough to type into computing devices by humans

As Crosbie Fitch points out in his excellent treatise on identity[1], names don't need to be globally unique, just unique enough. Names are identifiers we put on things that already have an identity. Names aren't the same thing as identity.

In computers, we often use the term *identifier* to refer to a name that is unique since the term *name* often refers to things that are not. For example, there are 55,000 John Smiths in the United States, when we build systems that have a profile for a person, the person's name will be there real name whereas their identifier, or ID, will be something that is unique within the system.

Names also disassociate the reference from the thing itself, so that the underlying representation can change without changing the name. Perhaps the most familiar example of this is domain names. The domain name windley.com points to some IP address. If I decide to change the machine hosting the services at windley.com, to another one with a

[1] Fitch, Crosbie, **Ideating Identity**, http://digitalproductions.co.uk/index.php?id=69, referenced Dec 30, 2014.

different IP address, its easily done and everyone referring to the name will still end up with the services that they are looking for.

Namespaces

A namespace is the universe within which a name is guaranteed to be unique and defines where the name has meaning. For this reason namespaces are sometimes called “domains.” A family name (usually) acts as a name space wherein given names are unique and meaningful. In an email address, the name (the part before the @ symbol) is guaranteed to be unique within the namespace (the part after the @ symbol). Filenames are unique within the namespace of the directory in which they reside.

Namespaces can be flat or hierarchical. The user names on a standalone computer are an example of a flat namespace. A file system is the most familiar example of a hierarchical namespace. Domain names are another familiar example of a hierarchical namespace. Figure 9-1 shows how hierarchical namespaces work in domain names and file systems.

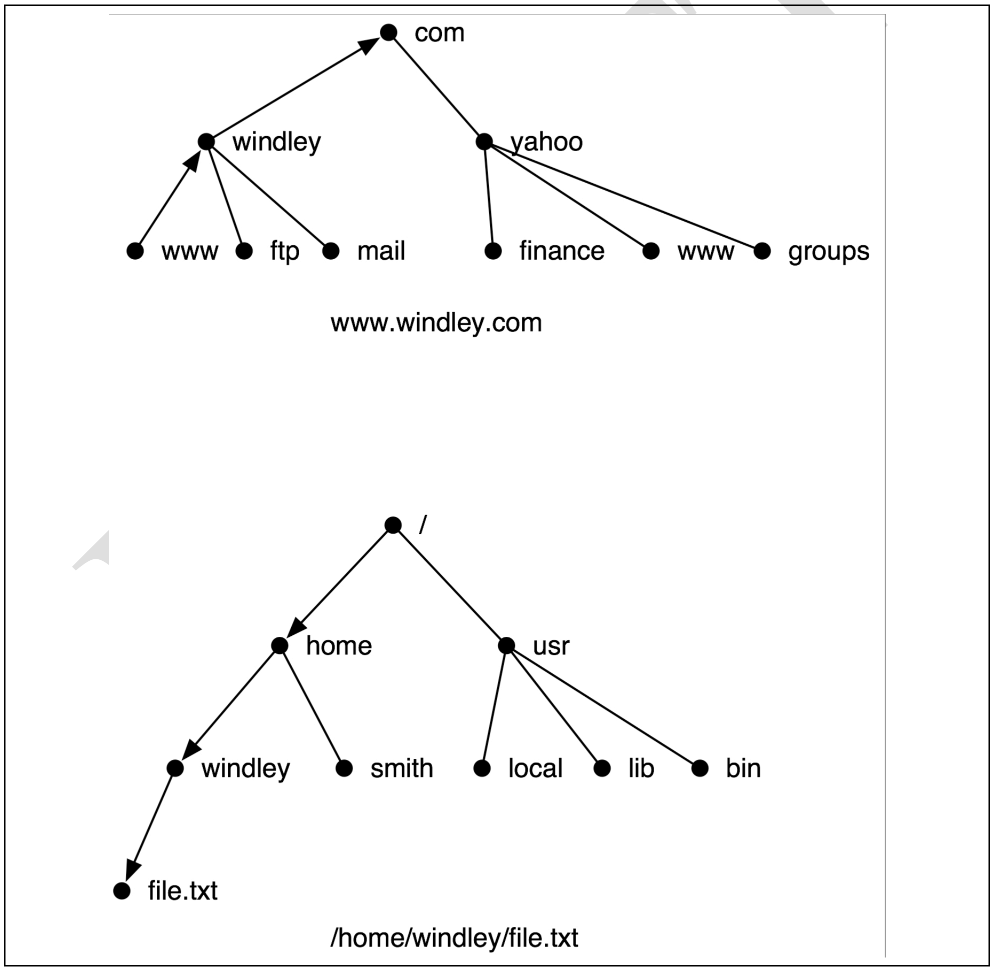

Figure 9-1 Hierarchical namespaces for domain names and file systems

Hierarchical namespaces have some interesting properties:

- A path inside the hierarchy between the root node and a leaf node can be used to specify any entry in a hierarchical namespace.
- Some paths are referenced and written from root to leaf (e.g. file systems) and some are referenced and written from leaf to root (e.g. domain names).
- In some hierarchical namespaces, like domain names, names can be both nodes and leaves. For example, I can reference both windley.com and www.windley.com with windley serving as a node in one case and a leaf in the other.
- In other hierarchical namespaces, like file systems, leaves and nodes are strictly differentiated.

In many hierarchical namespaces, the hierarchy reflects some actual hierarchy in the physical world. Usually, however the hierarchy in the namespace and the organization of the objects represented by the hierarchy do not have a one to one correspondence. For example, a file system is a hierarchy that exists entirely independent of the location of the bits on the disk and is strictly for the convenience of the user. With domain names, the hierarchy sometimes mirrors the physical world, but not always. For example, there really is an organization called Yahoo! that owns yahoo.com. On the other hand, ftp.windley.com, www.windley.com, and mail.windley.com are all the same machine.

Uniform Resource Indicators: A Universal Namespace

We've all ordered something online that gets delivered via UPS or Fedex. One of the items in the confirmation email is usually tracking number for the shipment. The tracking number is part of a URL to the package-tracking page at FedEx. Have you ever thought of this package-tracking page as the homepage for that package on the Internet? Every package shipped via FedEx, UPS, and most other companies has a homepage that is named by the URI (uniform resource indicator) that is used to reference it. The URI identifies a unique location on the Web and that URI can be linked in another document or bookmarked for later reference—just like any other Web page. The package "homepage" is no different that any other homepage on the Internet in that regard.

URIs are more general versions of URLs, or Uniform Resource Locators, the "Web page address" that you type in the address box on your browser. Whereas URLs represent locations and, as such, typically correspond to real resources on the Internet, URIs can be used to name things within a single, global namespace even when there's no Web location associated with the name. The structure is the same, however and so many URIs also function as URLs.

URIs are one of the most important features of the Web. Without URIs much of what we take for granted on the Web wouldn't work. As a simple example, having a universal namespace created using URIs allows any document, anywhere on the Web, to refer to any other document, anywhere on the Web, without the authors of the two documents having to agree on the same software package, or server beyond what's inherent in the Web itself. In fact, Paul Prescod has said: "If there is one thing that distinguishes the Web as a hypertext system from the systems that preceded it, it is the Web's adoption of a single, global, unified namespace."[2]

Apart from their use to identify resources on the web, however, URIs are finding their way into many other contexts since the URI system represents a universal namespace. Giving off-Web resources, such as database records, a URI makes them part of this same universal namespace and ensures that they can be uniquely distinguished from other resources.

URLs and URIs have three major components:

1. A protocol identifier followed by a colon (e.g. http:).
2. A domain name indicating a unique computing domain on the Internet (e.g. www.windley.com)
3. A path component indicating what specific resource in that domain is to be identified (e.g. /llp?ln=windley&lang=en).

Taken together, these components are written in the familiar fashion:

http://www.windley.com/llp?ln=windley&lang=en

There can be other components as well, including authentication information, port numbers, etc. but these three are the most common.

Cool URI's Don't Change

The URI is the public interface to a resource and, consequently, deserves great thought. One of the key factors that should be kept in mind when designing URIs is that they should not change—ever. This is not such a radical idea if you stop to consider that the URI is the name of the resource. In general, it's a bad idea to change the name of something since we cannot possibly know all the places where the name is being used and, consequently, cannot let them know when the name (the URI) changes. Thus, the URI should be chosen so that it is meaningful and unlikely to change. As the system is updated and maintained, the non-volatility of the URIs should be preserved. Numerous tools and techniques exist to make this possible. URL rewriting is one of the most powerful, allowing servers to resolve URI references to almost any resource.

Designing the URIs for your information system should be one of the most important tasks of the design phase. It may seem unusual to think of designing URIs. After all, don't we just let the network folks tell us our domain name and let the path fall out however it may? Not in a well designed system. The last section talked about the three components that are typically part of a URL. All three are usually under our control and should be carefully chosen.

With the rise of RESTful APIs, the issue of URL design has become a topic of intense discussion and there are many resources available that discuss principles of good API design, which in some part is analogous to good URL design. API URLs represent the

[2] Prescod, Paul, **Roots of the REST/SOAP Debate** (http://www.prescod.net/rest/rest_vs_soap_overview/)

names of resource collections or items. Issues like versioning, pagination, and the proper use of query strings are all part of an API, and hence URL design.

Don't construe this principle to mean that all resources need to be permanent. Just because URI's don't change, doesn't mean that the resource has to be always available. There are some resources that are transitory and some that go out of existence. Even so, we shouldn't change their name.

Directories

Directories are everywhere in information systems. Most systems have multiple directories for address books, password files, the list of authorized users for particular applications and so on. Even the much-maligned Windows registry is a directory. IT departments maintain large, enterprise-wide directories. In fact, the average IT organization maintains dozens of different directories of all types.

A directory service is a network-aware directory that allows a directory to be centrally managed and at the same time supply directory information to distributed applications. While we typically think of directories associating information with people, directories are useful for a wide range of IT and business needs. As such, directories are a critical part of the identity management infrastructure in most organizations.

A directory service contains a structured repository of information, often with complex interrelationships. The structure is defined in a schema—the metadata that defines the overall relationship of each piece of data stored in each entry in the directory to the others. The schema defines a structure within which the data is stored.

The schema specifies what properties can be associated with an entry, the allowed format, or type of the property and whether it is optional or mandatory. Each entry is defined as an object in the directory and a given object contains the properties associated with that entry. Attributes can be thought of as name-value pairs since it is customary to ask the value of a property with a given name for some specific entry.

As we've seen, it is not unusual for a namespace to be hierarchical and so too with directories. The hierarchical structure of the directory is stored in the directory tree. The directory objects at the nodes of the tree are called container objects, which can contain other container objects, or leaves or terminal objects. For example, there might be a container object that represents the organization that holds other container objects for the major organizational departments, and so on until you get to the people, printers, offices, and other resources at the leaves.

A directory service provides methods for querying the directory and managing the entries. These methods may be accessed by client programs designed for human interaction, or by other programs that need access to the information contained in the directory.

In practice directories can be physically distributed and their actions coordinated to produce a single, logical directory. Directories are also often replicated for both reliability and scalability reasons.

Directories Are Not Databases

From the description just given, it may be hard to distinguish a directory service from a standard database. Indeed, directories can be built inside databases and many of the directories in use are just that. Still, enterprise-class directories are usually different from databases in some significant ways:

- Directories are usually hierarchical whereas databases are usually relational. A directory can tell you all the people whose manager is Mary Jones and all the people who work in Salt Lake, but it can't easily tell you all the people who's manager is located in Salt Lake—a trivial task with a relational database.
- Retrieval is more important than updating in a directory and consequently, directories are optimized for retrieval. Typically, about 90% of the accesses in a directory are retrievals or queries with the remainder being additions or updates.
- Directories are optimized for storing and managing many millions of small, relatively simple objects.
- Directories do not usually support transactions, meaning that operations on the directory cannot be coordinated with the operations of other applications so that they are atomic (i.e. all happen or none happen).
- Directories offer extensive support for matching, partial matching, filtering, and other query operations.
- Most directories have preconfigured schemas so that they are immediately usable for common purposes and are customizable only in very specific ways. Databases typically require considerable schema work before any data can be stored.
- Directories are intended for external, wide-area use by default.
- Directories are simpler to manage than relational databases. Trained database administrators, who specialize in creating and managing schemas, optimizing queries and so forth, typically manage databases.
- Because queries and retrievals predominate, directories are easier to replicate for redundancy and performance reasons.

Even with these differences, and the advantages directories have in storing identity information, many organizations persist in using full-scale databases for relatively simple directory work.

An Example Directory

The Utah.gov directory I mentioned at the first of this chapter was more than a simple association of email addresses and authentication information. The Utah.gov directory contained as complete a collection of contact information for employees, contractors and others as we could manage.

The Utah.gov directory contains names, phone numbers, job titles, email addresses and office locations. The schema of this kind of directory is fairly simple, merely defining the fields stored in the directory and which are mandatory. Mandatory fields might include email address and phone number, or at least one or the other.

Such a directory is useful for more than just letting employees search for other employees. The directory can serve as an authoritative record of the employees and drive other enterprise information systems including the email system, the HR system, and the finance and payroll systems. Also the directory serves as the authentication and authorization repository for systems and applications.

In a perfect world, the online directory is fed directly from authoritative sources. For example, employee phone numbers should be fed from the phone system’s records so that they are always as accurate as possible. Alternately, the phone system could be fed from the directory.

In creating an authoritative directory, it is important to understand the relationships between various enterprise information systems, know who updates what information, specify what information is authoritative, or canonical, and to create a schema that can service these many needs. As I mentioned in the story about the Utah.gov directory, the politics are often the hardest part of a directory project. We’ll discuss governance in Chapter 14.

Enterprise Directory Services

There are hundreds of different directory implementations in use. There are a few however, that are pervasive, or that serve as good examples of a type. This section discusses a few of these.

RMIRegistry

RMI is the Java Remote Method Invocation facility that provides the groundwork for creating client-server architectures in Java over a network. Consequently, most enterprises have one or more RMIRegistries running. RMIRegistry is the RMI directory. I refer to RMIRegistry only as an example of a class of directories that are largely unseen, but important to the enterprise. RMIRegistry and its cousins provide a directory for named references to remote objects in a programming environment.

Figure 9-3 shows RMIRegistry in action. When a particular server starts, it registers with RMIRegistry, giving it a name and a reference to the method on the server. Later when the client accesses the method, it does not use a specific reference to the method, or even the machine and port the service is running on, but rather asks the RMIRegistry to return the reference to a service with a particular name. Once the RMIRegistry has returned the reference, the client can use it to contact the server and invoke the method.

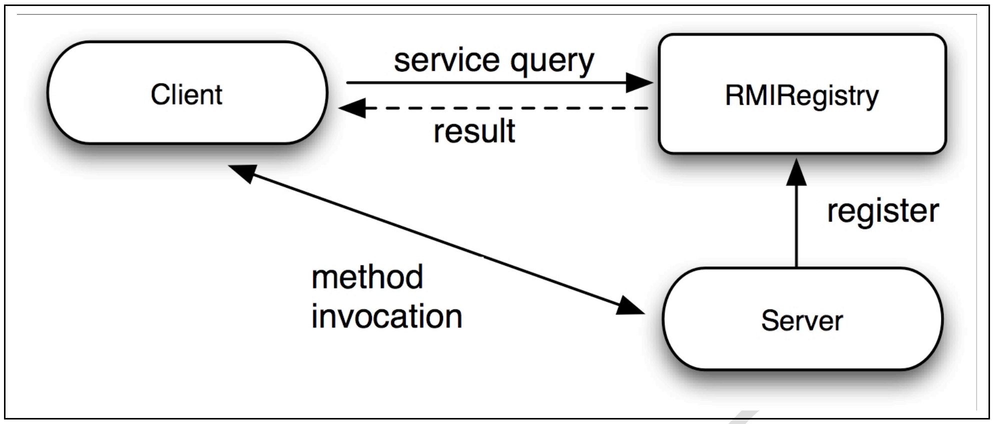

Figure 9-2. RMIRegistry is used to refer clients to services with named services

There are several advantages to using a named directory of remote references:

- The client does not need to be aware of implementation and deployment-specific information. Because it is insulated from this detail, the service is free to change these details without interrupting service or the clients having to be reconfigured.
- The indirection created by the name can be used to scale the service since multiple copies of the server could be servicing clients. Similarly, the clients could be directed to the server that is most lightly loaded or one that is closer (in a network sense) to the client.

X.500: Heavyweight Directory Services

X.500 is sometimes referred to as the granddaddy of directory services because its definition is the basis for many of the directories used in enterprises. X.500 is a comprehensive specification of directory services that was originally conceived as a distributed, network-independent directory service for a messaging service whose specification was called X.400. X.500 specified an online whitepages for the messaging system.

X.500 is actually a family of specifications developed by the International Standards Organization (ISO) and the International Telecommunications Union (ITU). Perhaps the most familiar of this family is the X.509 authentication framework specification that we discussed in Chapter 6. As we saw, X.509 specifies the public key infrastructure that is the basis for the use of digital certificates in many applications.

X.500 defines a distributed directory service that is hierarchical in nature and operates on a common namespace. The service is designed to be vastly scalable and extensible. The X.500 standards are a canonical reference for directory service functionality and as such are frequently used by industry in defining the functionality, architecture, and terminology for other directory services.

LDAP

LDAP, the lightweight directory access protocol, was created to provide simplified access to some of the functionality of X.500. Building clients to work with X.500 is complex because of all the options built into the specification. X.500 also was built and modeled on an OSI network protocol stack rather than TCP/IP, the standard protocol of the Internet. LDAP was originally conceived as a protocol that could be implemented by a gateway to an X.500 directory service. Using the lightweight protocol, clients could more easily access the X.500.

Moreover, LDAP specifies an API for clients, something that X.500 lacks. Having an API allows standard software development kits (SDKs) to be created that contain much of the code necessary to use the directory service.

LDAP has evolved to be a complete directory service, instead of just a gateway to X.500 services and a number of native LDAP directory servers exist on the market. Like X.500, LDAP specifies a network-based server with a hierarchical namespace. LDAP also specifies a method for doing referrals to other directories so that multiple LDAP servers can cooperate to create a single virtual namespace from the namespaces of the individual servers. While most LDAP servers provide a means of replication, there is, as yet, no replication standard and so LDAP servers from different vendors do not always interoperate.

Many commercial directories, including eDirectory from Novell and Active Directory from Microsoft, are LDAP compliant, meaning they have support the LDAP client API. Consequently applications that understand the LDAP API can use them with relative ease. For example, Brigham Young University's staff and student directory has an LDAP interface and I was able to couple Apple's Address Book application to it in just a few minutes, allowing me to search the directory using the same applications I store all my contacts in. If you're interested in exploring LDAP and how it works, I recommend OpenLDAP, an open-source LDAP server that is available at http://www.openldap.org.

Aggregating Directory Information

Single sign-on (SSO) has become something of a Holy Grail in many institutions to the point where many think that that is all identity management is. Anyone who's had to remember multiple user ID's and passwords just to use the email systems and file servers at work understands the pain that comes from having to manage multiple identity credentials.

Beyond causing pain for users, scattered identity data stores cause problems for the business as well. Integrating IT systems is important to businesses because of the added context that develops about a business activity when the data in multiple data stores can be linked. For example, linking the customer billing systems and the customer service systems gives employees processing invoices as well as employees providing customer service additional context about each customer.

For these reasons, and more, aggregating identity information and finding the relationship between identity records is important. To aggregate identity data, organizations have four choices:

1. Build a single central identity data store.
2. Create a metadirectory that synchronizes data from other identity data stores in the enterprise
3. Create a virtual directory that provides a single integrated view of the identity data stores in the enterprise

4. Federate directories by tethering identity data stores together.

The first solution is included for completeness, but it’s easy to see that creating a single data store of identity data is not feasible for anything but small organizations. The goal of the latter three solutions is to present identity data as if the centralized identity data store of option (1) existed, even when identity data is distributed across the organization. We’ll discuss metadirectories and virtual directories here, but save the discussion of federated identity for Chapter 12.

Metadirectories

Metadirectories are collections of directory information from various, diverse directory sources. The information from those directory sources is aggregated to provide a single view of the data. As we’ve pointed out, the modern enterprise maintains information in dozens of directories of all sorts. Not all of it is information that should be aggregated, but much of it can be without having to re-implement all of the directories and the applications that depend on them.

In the Utah.gov story I told at the first of the chapter, each agency maintained its own directory of its employees and their contact information. Because most of them were using Novell’s eDirectory and Novell had a good metadirectory product, we chose that route to creating an aggregated identity store.

Metadirectories allow the enterprise to collect information from existing directories into a single identity store that can be searched and queried as if all the information were stored in a single directory. Some of the benefits of metadirectory technology include:

- A single point of reference provides an abstraction boundary between applications and the actual implementation so that as the organization changes directory vendors, modifies system implementations or reorganizes data, the applications still query a single source.
- A single point of administration reduces the burden of accessing multiple directories with multiple interfaces to maintain the data.
- Redundant directory information can be eliminated reducing the administrative load in managing duplicate data.

There are some significant challenges in building a metadirectory. These fall into two primary categories: governance and technical. Governance includes issues such as information ownership, interorganizational administrative responsibilities, namespace choices, data formats and schemas, legal requirements, and information security. The implementation issues, while by no means trivial, are relatively straightforward and include the architecture of the metadirectory, namespace normalization, protocols, and procedures for data synchronization.

Metadirectories work through software agents whose job is to gather defined subsets of directory information from the group of directories in the metadirectory’s purview. The job may not be as easy as aggregating unconnected records. In fact, the most interesting uses of metadirectories involve the aggregation of attributes about a single subject from multiple directories to form a super record.

As an example of this, Figure 9-4 shows an example of how a metadirectory might be used to aggregate records from two directories. The metadirectory gathers data about the employee's jobs and status from the HR directory and information about the employee's payroll record from the Payroll department's directory and aggregates the data into a record that gives a single view of the employee. Of course, the metadirectory may ignore some attributes from the HR and Payroll directories because they aren't relevant to the objective of setting up the metadirectory.

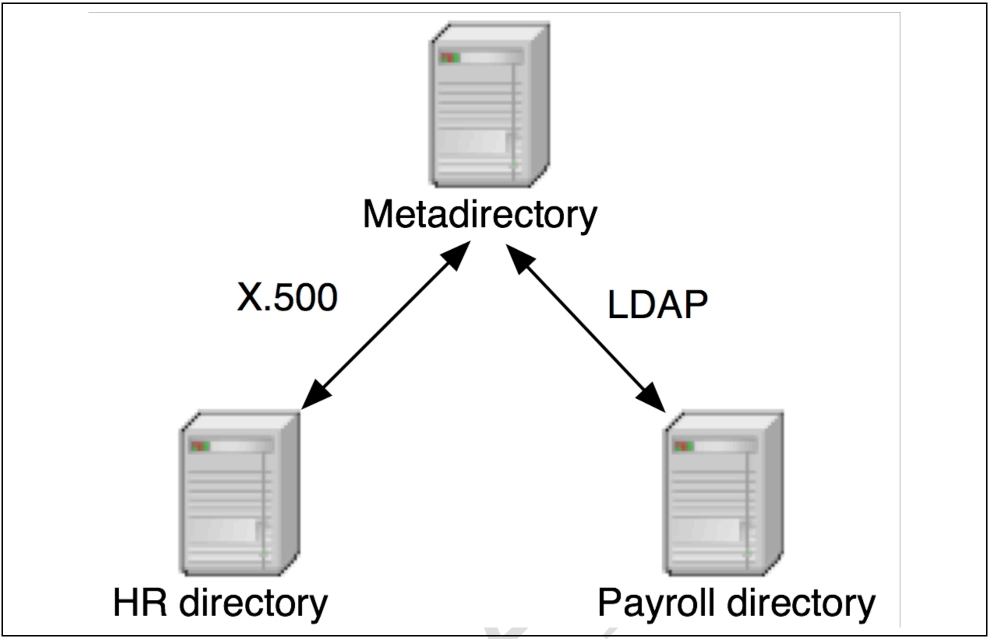

Figure 9-3. Metadirectory aggregating data from HR and Payroll directories using different standards

In this example we aggregated data from only two sources—but that needn't be the case. We could create a super record for the employee from as many data sources as we have available. An obvious extension to the example in Figure 9-4 would be to include phone information from the phone system and email address from the email system.

Creating such a super record presents the problem of knowing which records in the various directories to combine together. Metadirectories copy the information from the underlying directories into a local store, effectively doing a *join* of the data in the various directory records. The join is done using a pre-identified mapping between unique identifiers in the underlying stores, called the *join point*. Each directory aggregated by the metadirectory is connected to the metadirectory though a channel called a *namespace connector* that manages the link. Each connector can be individually configured to filter attributes as desired.

One of the most difficult tasks in creating a metadirectory is eliminating namespace conflicts between the various directories being aggregated and providing a mapping between the metadirectory namespace and the underlying namespaces. More abstract but just as important is the problem of mapping schemas from the underlying directories to the metadirectory so that attributes are meaningfully integrated.

Another significant problem is one of data synchronization. When implementing the metadirectory, choices must be made about where data can be modified. The data could be writable only at the directory level, only at the metadirectory level, or at both levels. Clearly which strategy is chosen depends on the application, but when data can be changed at the directory and metadirectory levels, synchronization becomes significantly more complex.

When data can be modified at the metadirectory level, complex authorization schemes may be needed to ensure that information owners can only modify their own data. In the example in Figure 9-3, only designated HR department representatives would be able to update those attributes that flow from the HR directory and similarly for Payroll representatives.

Because metadirectories can synchronize data bi-directionally, they can be used to populate one directory with attributes from another directory. They can even perform a data cleansing function, allowing attributes to be validated and then pushed back down to the underlying identity store.

Virtual Directories

Virtual directories are similar in concept to metadirectories in that they create a single directory view from multiple independent directories. They differ in the means used to accomplish this goal. Whereas metadirectories typically use software agents to replicate and synchronize data from various directories in what might be thought of as batch processes, virtual directories create a single view of multiple directories using real-time queries based on mappings from fields in the virtual schema to fields in the physical schemas of the real directories.

Another way of looking at the difference is that metadirectories have an associated data store where the directory information from the other directories is kept; virtual directories have no separate data store. They work by turning a single query to the virtual directory in to multiple queries to the physical directories and then aggregating the returned results in real time to create the result given to the user. Eliminating the identity store means that virtual directories also eliminate the problems related to data synchronization and replication.

In this way, virtual directories present a real-time interface to underlying identity data. Queries and updates happen in real time and are reflected to other applications using the data, either from the virtual directory or the underlying stores. Typically the interface to the virtual directory is LDAP. Like a metadirectory, the connections from the virtual directory to the underlying stores could use any of a number of protocols. As a result, the virtual directory creates a standard view of the data using a standard API.

Virtual directories are typically used in cases where real-time access to frequently changing identity data is important.

Domain Name Service

The issue of discovery online is more complicated that inside an enterprise. Whereas enterprise directories are under the control of a single organization that has the power to dictate policy and enforce naming, online systems must work cooperatively with other organizations not under their control.

The Domain Name Service is the best-known example of this and forms the basis for most online name systems. We've already seen, for example, how domain names for the basis of URLs.

When I first started using Unix in the mid-80's, DNS was not widely used. Instead we FTP'd a `hosts` files from a computer at Berkeley, merged it with a local `hosts` file, and installed it in the `/etc` directory. Mail addresses had ! in them to specify explicit internal routing from a well-known host to the local machine. We had machine names, but no global system for dereferencing them.

Domain name service, or DNS, changed all that by providing a decentralized naming service that based lookup on a hierarchy starting with a set of well-known machines servicing a top-level domain (TLD), like `.com`. Nothing was more important than a great domain name with a `.com` at the end during the 90's. As we've seen URLs are based on DNS, so having a short, memorable domain name was, and still is, an asset.

Of course the good domain names were quickly all gone. I was lucky enough to own a few good ones over the years: `superbowl.com`, `skiutah.com`, `shoppingcart.com`, `imall.com`, and `stuff.com`. I was also early enough to get my name, `windley.com`. If you're just coming to this party, however, your name is long gone unless you want to use a TLD that no one has heard of and won't recognize. Anyone in the `.pe` namespace?

DNS is the universal directory that maps domain names to IP addresses. DNS is built around the hierarchical domain namespace that we discussed earlier. DNS is a distributed directory and serves as the enabling infrastructure for a single, global directory of domain names. This directory is built from thousands of servers owned by thousands of organizations around the world. The architecture of DNS allows those machines to efficiently and cooperatively answer queries regarding domain name mappings and, at the same time, provide for delegated control over the mappings for any given namespace.

When a machine needs to resolve a domain name into an IP address, it queries a DNS server, looking to one of a few, usually local, servers for the answer. The local server may know the answer because the query regards a local machine or because it regards a machine that the DNS server has recently looked up. If it does not, the hierarchical structure of the name is used to arrive at an answer.

Suppose that a distant machine is looking for www.windley.com. (See Figure 9-2) The machine contacts its local server but that server does not know the answer. The local server contacts what is known as the root server for the top-level domain (TLD), in this case .com. The root server does not know the mapping for www.windley.com, but it does know the addresses of every DNS server for the domains in its TLD. The root server refers the local server to the DNS server handling windley.com and the local server contacts the windley.com DNS server. Since that server knows the address for www.windley.com, the address is returned to the local server, cached and sent to the original requestor. Of course, domain namespaces can be more than three deep and so can the associated servers. This process just goes on longer in those cases, but an answer is eventually returned as long as the mapping exists and the servers are properly configured and registered.

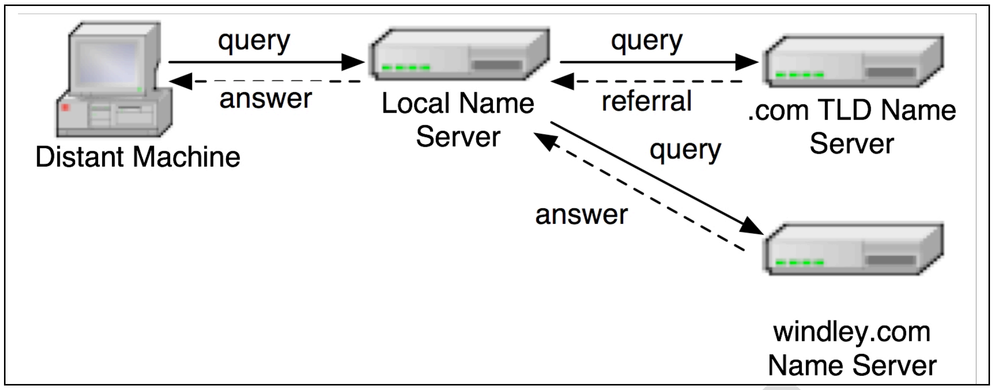

Figure 9-4. DNS query, referral, and answer pattern for www.windley.com

WebFinger

The Unix `finger` command allowed users of time-shared Unix systems to find out information about users. One of the interesting architectural features of `finger` is that it was a hybrid directory that used information the system knew, as well as information the user provided to create the response. Users were responsible for creating files in their home directory that were used to give the person using `finger` information about the user.

WebFinger is a similarly architected system for the Web. The official specification says that "WebFinger is used to discover information about people or other entities on the Internet that are identified by a URI using standard Hypertext Transfer Protocol (HTTP) methods over a secure transport."

A WebFinger query returns a JSON object called a JSON Resource Descriptor (JRD). For people the JRD might contain things like their email address, telephone number, public key, and so on. For other kinds of entities, the data can include whatever information is relevant. For example, the JRD may contain the location of a printer, the capabilities of a server, the author of a blog post, and so on.

Although the Unix `finger` command and WebFinger are very different protocols, they both allow users to attach meta-data to an identifier.

RFC 7033 describes the WebFinger protocol. Entities in WebFinger are identified using a URI. For people (or more accurately accounts), a new URL scheme, `acct:`, is used to create a URI. The format of the account URI is

```
acct:<identifier>@<domain>
```

This may look like an email address, but note that this is a URI, there's no requirement that it work as an email address.

WebFinger queries are made against a "well-known URI" for an identifier in the domain that the server represents. Well-known URIs are defined in RFC 5785 as "a URI whose path component begins with the characters `/.well-known/`, and whose scheme is `http`, `https`, or another scheme that has explicitly been specified to use well-known URIs." WebFinger has registered the well-known URI `/.well-known/webfinger`.

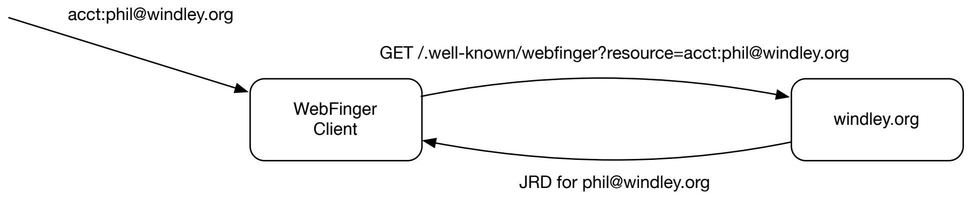

Figure 9-5. WebFinger query for phil@windley.org

Queries for the JRD of a particular URI are made using WebFinger's well-known URI and a query string containing a key-value pair with the with the key resource identifying the subject URI as shown in Figure 9-5.[3]

The returned JRD might look something like this:

```
{
 "subject" : "acct:phil@windley.org",
 "aliases" :
   [
    "https://phil.windley.org",
    "acct:phil@windley.com"
   ],
"links" :
   [
    {
     "rel" : "http://openid.net/specs/connect/1.0/issuer",
     "href" : "https://openid.windley.com"
    },
    {"rel": "http://kynetx.org/rel/well-known-eci",
     "property":
                 {
                  "http://kynetx.org/rel/eci",
                 "76161abcd-18d4-00163-abcd12"
                }
    },
    {
     "rel" : "http://webfinger.example/rel/profile-page",
     "href" : "https://phil.windley.org"
    },
    {
     "rel" : "http://webfinger.example/rel/businesscard",
     "href" : "https://phil.windley.org/phil.vcf"
    }
   ]
```

[3] Note in Figure 9-5 and elsewhere in this section, I am not URL encoding the query string for clarity even though this would be done in practice.

```
}
```

A JRD response contains name-value pairs:

- Subject—a name-value pair whose value is the subject of the JRD. This value might be different from the value in the query if the subject has moved or there is a canonical identifier.
- Aliases—a name-value pair whose value is an array of strings giving other URIs that also identify this entity. This name-value pair is optional.
- Properties—a name-value pair whose value is an object with name-value pairs that identify a property (using a property URI) and a value. This name-value pair is optional.
- Links—a name-value pair whose value is an array of objects with name-value pairs that give a relation type and (optionally) an href, set of properties, types, and titles. This name-value pair is optional.

Of course, you might not be interested in all this information when you make a query. The WebFinger specification allows the query to be restricted to certain relationship types of the subject resource in the query string. For example the following query:

```
GET /.well-known/webfinger?
    resource=acct:phil@windley.com&
    rel=http://webfinger.example/rel/profile-page
```

would return the JRD:

```
{
 "subject" : "acct:phil@windley.org",
 "aliases" :
   [
    "https://phil.windley.org",
    "acct:phil@windley.com"
   ],
"links" :
   [
    {
     "rel" : "http://webfinger.example/rel/profile-page",
     "href" : "https://phil.windley.org"
    }
   ]
}
```

The returned JRD contains only the relationship we asked for. Since there could be hundreds of relationship types for a complex entity, this reduces the amount of work the client must to find relevant resource links.

Because WebFinger is based on HTTP, you can use redirects to service accounts in multiple domains from a single WebFinger server. For example, in the previous example `acct:phil@windley.com` is an alias for `acct:phil@windley.org`. The same WebFinger server could service deliver the appropriate JRD for both `windley.com` and `windley.org` using a redirect from those servers.

WebFinger does present some privacy and security concerns. In particular, the JRD could contain information that would allow for more sophisticated phishing attacks. The WebFinger specification states that "[s]ystems or services that expose personal data via WebFinger MUST provide an interface by which users can select which data elements are exposed through the WebFinger interface" and "WebFinger MUST NOT be used to provide any personal data unless publishing that data via WebFinger by the relevant service was explicitly authorized by the person whose information is being shared."

There is nothing that prevents WebFinger servers from only allowing access to information based on an authorization scheme or limiting what data is returned to different clients based on authenticated role.

WebFinger provides a good way to provide metadata and additional information about a resource. Used in conjunction with the `acct` schema, WebFinger allows a single, easy-to-remember identifier to be used to find additional information that might be identified only using a longer, more complicated identifier. For example, using `acct:phil@windley.org` might allow you to retrieve my complete VCard record or portions of my public calendar which both likely have much more complex URIs.

Discovery

You might have disagreed when I said URLs are names. Earlier we discussed three properties of good names. While URLs are globally unique, they aren't memorable and most people hate typing them into things. If I'm looking for IBM, I'm happy to type `ibm.com` into my browser. But what if I'm looking for a technical report by IBM from 2006? Even if I know the URL, I'm not likely to type it into the browser. Instead, I'll just search for it using a few key words. Most of the time that works so well that we're surprised when it doesn't.

We can distinguish between names, identifiers, and addresses. URIs, as we've discussed, are universal identifiers. A URL, a form of URI, also gives the location of something. In other words, it's an address. Neither identifiers nor addresses are the same thing as a name, but these three things are often conflated and confused.

When we don't have good names, or where names are impractical, discovery provides an alternative. Typing keywords into a search engine is a good example of discovery. Web pages don't have names, at least not that are globally unique. And even if they did, who'd be able to remember them all? But search engines allow us to discover the address of a Web page using attributes of the page.

The World Wide Web solved several important problems but discovery wasn't one of them. As a result, Aliweb, Yahoo!, and a host of other companies or projects sprung up to solve the discovery problem. Ultimately Google won the search engine wars of the late 90's, although new search engines continue to come and go.

Discovery is a hard problem because whether we're talking about searching for Web pages on Google, looking for an old high-school friend on Facebook, or finding an API for a particular purpose, finding matches requires some level of understanding what the search target is about. That is, this is a semantic problem involving meaning, not merely a syntactic problem like looking a name up in a directory.

Determining relevance is the ultimate goal of discovery. Discovery determines relevance using algorithms to find the meaning of metadata. One of the reasons search engines like Google succeeded is because they solved two problems. First they automated the generation of metadata about Web pages (e.g. Google Web crawler). Second they developed simple, fast algorithms for determining which addresses mattered most for any given set of metadata attributes (e.g. Google's PageRank algorithm).

Heterarchical Directories

You might have noticed that DNS and WebFinger have very similar architectures. Both allow a client to retrieve properties of a given identifier from decentralized sources. There is no centralized database of domain names or WebFinger URIs. The systems both allow for the records associated with an identifier to be maintained by the entity responsible for that information. This has advantages in both scale and accuracy.

Note however that while DNS and WebFinger are decentralized, they are still hierarchical (WebFinger's hierarchy is inherited from it's dependence on DNS). Hierarchies have a significant limitation in that they introduce a single-point of failure even in a decentralized system. In DNS for example, there are a limited set of servers for any given TLD. How well the TLD's registry is architected can have a significant impact on it's availability in the face of network failures or attacks.

The hierarchical structure of DNS has a further drawback in that it enables censorship. A despotic regime can limit access to certain TLDs or even proxy them and return different results for any given query. One of the key provisions of the Stop Online Piracy Act (fortunately dead for now), would have used DNS to censor Web sites deemed to be infringing.

The alternative to hierarchy is heterarchy. Wikipedia defines heterarchy as "a system of organization where the elements of the organization are unranked (non-hierarchical) or where they possess the potential to be ranked a number of different ways."

Personal Directories and Introductions

Discovery isn't the only way to get around a lack of names. To see how, think about your house address. It's a long unwieldy string of digits and letters. Resolving a person's name to their address has no global solution. That is, there's no global directory (except maybe at Acxiom or the NSA) that maps names to addresses. Even the Post Office in over 200 years of existence hasn't thought "Hey! We need to create a global directory of names and addresses!" Or if they have, it didn't succeed.

So how do people find places? We exchange addresses with people and keep our own directories. We avoid security issues by exchanging or verifying addresses out of band. For the most part, this is good enough.

Personal directories are largely how people exchange bitcoins and other cryptocurrencies. I give you my bitcoin address in a separate channel (e.g. email, my web site, etc.). You store it in a personal directory on your own system. When you want to send me money, you put my bitcoin address in your wallet. To make it even more interesting, since bitcoin addresses are just public-private key pairs, I can generate a new one for every person creating what amount to personal, peer-to-peer channels for exchanging money.

This "introduction pattern" is a powerful way to create distributed, heterarchical directories. So long as there is some trusted way to communicate with the party you're connecting to, long addresses aren't as big a problem as you might think. We only need to resort to names and discovery when we don't have a trusted channel.

Personal directories are heterarchical since they are unranked. There's no formal ranking that cause us to prefer one address book to another. The problem with personal directories is that they make global lookup difficult. Unless I have some pre-existing relationship with you or a friend who'll do an introduction, a personal directory does me little good. One way to solve this problem is with systems that work like DNS, but are heterarchical.

Heterachical Naming Systems

The blockchain algorithm that is part of bitcoin provides a basis for creating truly heterarchical directories. Whatever you may think of bitcoin as a currency, there is little doubt that bitcoin presents a working example of a global distributed consensus system.

Distributed consensus is the foundational feature of a heterarchical naming system. To understand why, think about DNS. DNS distributes the responsibility of assigning names, but it avoids the problem of consensus (agreeing on what names stand for what IP addresses) by creating single copy of the mapping. But, as we've discussed, this single copy presents a single point of failure and a convenient means of censoring or even changing portions of the map.

If we want to distribute the copy of the mapping and make everyone responsible for maintaining their own mapping between names and addresses, we need a distributed consensus system. Bitcoin provides exactly such a system in the form of a block chain, a cryptographic data structure with a functional means of validating updates.

Onename.io and Namecoin are examples of systems than use the blockchain to map names to addresses in a heterarchical fashion[4]. I have registered `windley.bit` using Namecoin. If you type it in your browser it won't resolve since your operating system only knows how to resolve names via DNS, but that's not a fundamental limitation, you can patch your OS to resolve names using alternative mappings like Namecoin. Operating systems didn't understand TCP/IP either in the distant past. I used to regularly patch Windows 3.1 by adding a TCP/IP stack. Windows 95 included it due to popular demand. FreeSpeechMe provides a browser plugin that can resolve `.bit` domains.

What's the advantage of `windley.bit` over `windley.com`? Simply that the mapping is completely distributed. There is no single point of failure. You can turn off all the TLD servers and `windley.bit` will still work. A `.bit` domain name can't be censored in the same ways a `.com` domain name can. For example, even if the Stop Online Privacy Act

[4] Namecoin is actually a general-purpose distributed key-value store. So, domain names are just one thing you can use it for.

were to become the law of the land, domain names based on the `.bit` directory would be immune from its censorship provisions.

As computers are incorporated into many more facets of our lives than they currently are, we will need to find ways to trust those computers and avoid giving up autonomy to centralized authorities. Heterarchical directories will be a key technology. I don't think it's going to far to say that our natural rights as human beings are based on a world that is heterarchical (at the global level) and that we are fooling ourselves if we believe we can engineer virtual systems that respect or protect those rights using hierarchies and centralized authorities.

Conclusion

Directories are one of the fundamental pieces of a digital identity infrastructure. Many organizations can put themselves far down the path to having an effective enterprise identity management by attacking their directory problems first.