Reprinted with permission from the 4th PODC Conference Proceedings. ACM 1985.

Towards a Universal Directory Service

Keith A. Lantz, Judy L. Edighoffer and Bruce L. Hitson
Computer Systems Laboratory
Departments of Computer Science and Electrical Engineering
Stanford University
Stanford, CA 94305

Abstract

Directory services and name servers have been discussed and implemented for a number of distributed systems. Most have been tightly interwoven with the particular distributed systems of which they are a part; a few are more general in nature. In this paper we survey recent work in this area and discuss the advantages and disadvantages of a number of approaches. From this, we are able to extract some fundamental requirements of a naming system capable of handling a wide variety of object types in a heterogeneous environment. We outline how these requirements can be met in a *universal directory service.*

1. Introduction

In recent years, researchers have paid a great deal of attention to data abstraction, modularization, and information hiding. The result has been a clean separation between the *implementation* of an object and the *interface* to that object. Specifically, each object is associated with a *server* or *manager* that implements the object and presents to *clients* an interface that defines the operations that can be performed on the object. This model is particularly appropriate for distributed systems where the physical separation of objects and clients requires that the implementation be on one machine, while the interface is duplicated across multiple machines.

One might think that a coherent model of interfaces and implementations would also lead to common mechanisms for representing, naming, locating, and manipulating objects. By so doing, different types of objects could be manipulated with the same primitives, such that one object — a file, say — could be substituted for another object — a terminal, say — in the manner of UNIX[1] standard I/O [22]. Ironically, the server model frequently has led to implementations — especially monitor-based implementations — that thwart the goal by enforcing overly strong typing. A file server provides a set of primitives that is incompatible with the primitives provided by a mail server, which are in turn incompatible with the primitives for a printer server. Often, each server provides its own naming system. Even where a distinct "name server" is available, it typically is restricted to narrow domains, such as services (how to access a file server) or mailboxes (how to send mail and where to find the mailbox).

In this paper we address some of these problems by developing a *universal directory service* that:

- can span a heterogeneous internetwork of existing naming domains;
- allows us to name, locate, and discover how to manipulate objects (including files, processes, mailboxes, people, and services);
- provides dynamic binding and context mechanisms; and
- can be integrated into most existing systems as a "value-added" feature.

On the one hand, the UDS may be thought of a superimposing a virtual directory structure on top of a multitude of pre-existing directories (name spaces). On the other hand, the UDS is designed in such a way as to admit base-level or native implementations where it provides the only directory structure. Indeed, a prototype base-level implementation has been running at Stanford for over a year [9].

We begin in Section 2 by examining some existing naming systems to understand how they manage objects. Section 3 discusses advantages and disadvantages of the various approaches. In Section 4 we attempt to move beyond existing systems and suggest very general principles on which a general naming service should be based. The architecture for a *universal directory service* (UDS) is presented in Section 5, followed by a brief discussion of implementation issues in Section 6. Section 7 summarizes the major contributions of the UDS. A subsequent paper will present experience with the existing implementation.

2. The State of the Art

As noted above, a number of name services have been implemented for distributed systems. In early message-based systems rudimentary name servers were developed that mapped simple string names for services (such as "File System") into the identifiers for the processes that implemented those services [3, 11, 34]. Similarly restrictive name servers include those that have been developed to map string names for hosts or mailboxes into their network addresses [5, 14, 17, 27] and the "dictionaries" of many database systems [1, 13]. More recently,

[1] UNIX is a trademark of AT&T Bell Laboratories.

several efforts have extended the notion of file system directory to include access to objects other than files, typically by having the directory entry contain a process or port identifier rather than a file identifier [12, 19]. Other efforts have been oriented toward providing access to objects based on the "attributes" of the objects, rather than by a fixed-format name [8, 32].

These last few efforts represent the first steps towards *implementing* name services that can handle all objects. There are, of course, numerous paper designs for similar name services. More importantly, there is a wealth of available research on the fundamental principles of naming (in distributed systems); the reader is referred to [18], [21], [23], [25], [26], and [31], in particular. The universal directory service described in this paper represents an attempt to refine both the concepts of naming and the implementation of viable name services.

In the remainder of this section we examine a number of naming systems that represent the "state of the art". For each system, we briefly introduce the motivation for and background of the system. We cover the key areas of object management by taking the viewpoint of a human user who, upon encountering each system for the first time might attempt to use its naming services.

2.1. V-System

The V-System is a server-based distributed system that runs on a collection of hosts that are connected by a high-speed local-area network [6]. Any server may implement the V-System Name Handling Protocol (VNHP) and so participate in the name service [7]. In fact, the name space is partitioned among servers; each server is expected to implement the objects corresponding to the names it defines.

Object names are structured as a *context* and a *context-specific name* or *CSName*.[2] The *context* portion of the name is used to identify the process supporting that piece of the name space in which the CSName is defined. The name space of CSNames supported by a particular server may take any form, be it flat, hierarchical, or whatever. Even the syntax of the CSName is server-dependent.

This approach allows the servers implementing objects a great deal of autonomy. However, it could also produce a great deal of confusion for clients unless they thoroughly understand the syntax of the names and the semantics of the operations offered by a server. This problem is partially ameliorated by the wide-spread adoption of the V I/O protocol, which defines operations on a large class of file-like objects [4].

2.2. Clearinghouse

The Clearinghouse [17] evolved from the the registration service that was provided in early versions of Grapevine [5]. It is used primarily to name mailboxes, users, and servers (machines), in the context of a local net-based distributed system. The name space of the Clearinghouse is managed by a collection of Clearinghouse servers. Each server manages a portion of the global name space. The name space is not strictly partitioned between servers, as it is in the V-System. Names are organized into a three level hierarchy of the form *L:D:O*, corresponding to the local name, domain name, and organization name, respectively. The syntax for names is uniform over the entire name space. Autonomy is based on the choice of what *D:O* partitions to support within a particular server.

Associated with each object in the Clearinghouse is a set of *properties*, where each property is an ordered tuple consisting of a *⟨PropertyName, PropertyType, PropertyValue⟩*. The *PropertyType* specifies the format of the *PropertyValue*. Only two *PropertyTypes* are supported: an *item* is an uninterpreted string of bits, and a *group* is a set of object names. Lookup mechanisms based on *PropertyNames* are used to implement a simple generic type capability. Each *PropertyName* must be globally registered through a (human) naming authority.

The Clearinghouse can store arbitrary information about any named object. If the proper information were stored, applications could in principle be handed a description of how to operate on an object of some newly created type. However, specification of type is not explicit. The agent getting the description of a named object must recognize which property name refers to the object's type and must then be smart enough to figure out what to do with an object of that type. This forces type knowledge upon the client. Another drawback is that the Clearinghouse doesn't care what information is stored. The information needed to find an object or its manager may not be present at all. Similarly, there is no reason to expect to find any sort of information to help a client understand how to deal with an unfamiliar type. In brief, while the Clearinghouse has the power to provide type-independence if it were properly applied, it lacks the discipline to do so.

2.3. ARPA Domain Name Service

A new name service for the ARPA Internet has been specified and is currently being implemented [14, 15]. It will run on a widely heterogeneous collection of machines running a variety of operating systems. The Domain Name Service is intended to help clients locate servers for common network services and to locate objects managed by such servers.

Name service functions are divided between two classes of "servers": name servers and resolvers. Clients make requests of resolvers, which in turn make requests of name servers. Typically, one name server will not query another name server in order to resolve a name. Instead, it will instruct the resolver which name server, if any, to query next.

The name space is hierarchical, with no limitation on the depth of the hierarchy. The syntax is uniform across the entire name space. Each component of the hierarchy is a domain and typically reflects an administrative or geographical grouping. The design allows an administrative entity to control what names are introduced into domains under its control. In fact, names are generated and approved manually by the (human) administrative entity for each domain.

Associated with each label in a domain name is a (possibly empty) set of *resource records* that contain information about objects within the domain. Each resource record contains a domain name and a number of other fields. The *resource type* defines a standard abstract resource such as "host address" or "mail forwarder". A *class field* defines a system-wide style for interpreting the *resource data* fields. It is typically used to hint at protocol family (e.g., Internet vs. PUP), and in combination with the resource type allows unambiguous interpretation of the resource data field.

Name servers and resolvers are required to contain and use some knowledge of type for their proper operation. First, name servers are expected to recognize that certain type codes represent

[2] *CSName* actually stands for *character string name*, but for purposes of this paper, *context-specific name* is more appropriate.

supertypes of other types. For example, a name server is expected to know that a request for objects of type MAILA (mail agent) can be satisfied by object of either type MF (mail forwarder) or MS (mail server). Second, name servers are expected to use knowledge about the type of an object to provide useful hints. For example, in answer to a query about a mailbox, a name server will typically return not only the name of the ARPANET host supporting that mailbox (the contents of the entry for the mailbox) but will look up and return the ARPANET address of that host. What information is appropriate for each type of object is type-dependent.

2.4. R* Catalog Manager

The R* distributed database management system [33] supports the relational model [30]. The database *catalog* manages information on database objects, including their structure, format, access paths, and access controls [13]. Names are managed by a distributed collection of *catalog managers*. Catalog information about an object is stored at the same site(s) as the object itself. If an object is moved from the site at which it was created, its "birth site", a partial catalog entry is maintained at the birth site indicating where the full catalog entry can be found. The object can be accessed directly at its new site without reference to the birth site, so that access to an object is still possible as long as the site that stores it is operational. (This assumes that the client has learned of the new location of the object before its birth site failed, or, alternatively, that it can discover the new location in other ways).

The name space tends to be more implementation-defined than the prior examples. A name, referred to as a "System Wide Name" (SWN), contains four components: (1) the *user-id* of the object creator, (2) the *user-site* of the object creator, (3) the creator specified *object-name* of the object, and (4) the *object-site* or "birth site" of the object. Autonomous sites can assign the first three components in any way, but site names must be unique. A SWN maps to information such as low-level (storage) format, access information, and object type. In this context, it makes little sense to introduce new types of objects. However, the system may introduce new access methods or recognized data types. As this entails modification to other parts of the sole application, R*, it is perfectly reasonable to change the name service functions as well.

R* supplies some context facilities. Users typically specify only the *object-name* portion of the SWN; simple rules are provided for supplying the missing components. A user's context consists of the *user-id* and site from which the *object-name* was issued; these are used to complete the SWN. Moreover, synonyms are provided on a per user (at a site) basis to allow arbitrary mapping of an *object-name* to a SWN.

2.5. Sesame and Spice

Sesame [10] is the file system for the Spice distributed operating system [2, 20]. As with several other systems, its name service evolved from UNIX – primarily through the addition of interprocess communication ports to the set of objects understood by the directory system [19]. The name service consists of a distributed collection of "Central Name Servers" residing on the file server machines and "Spice Name Servers" residing on each user's workstation.

The name space is organized hierarchically. The name service requires absolute names — from the root — to be specified for all operations. Maintenance responsibility is shared by partitioning the name space along subtree boundaries, such that only one name server has responsibility for a subtree at any time. The names for objects that are intended to be shared should be kept in directories that are always maintained by a Central Name Server. Names for objects that are primarily used by one user may be kept in directories that are maintained by the user's local Spice Name Server.

The name service has a limited facility for user-defined types. The catalog entry associated with user-defined type is fixed length but uninterpreted. There is no support within the name service for guiding applications in the interpretation of user-defined types.

3. Evaluation of the State of the Art

Having presented some of the interesting details of a few systems, we now discuss and evaluate these systems according to different criteria. No attempt is made to be comprehensive; a full discussion of all the relevant issues would fill many volumes. Instead, we hope to raise some important points that will guide us in our approach to developing a more powerful naming system.

3.1. Segregated vs. Integrated

First, we might characterize name services as either *segregated* or *integrated*. Segregated services explicitly separate the name management facilities from the servers that actually implement the objects being named, leading to the notion of "name servers". In integrated services, on the other hand, names for an object are managed by the same object manager that implements the object. All of the example name services except the V-System are segregated.

Integration is an example of the "end-to-end" argument [24]: Let each "application" provide the minimal services it needs, rather than impose a separate general-purpose facility. In the V-System, for example, the global name space of fully-qualified CSNames is strictly partitioned so that each server manages precisely that portion of the name space describing the objects it implements. Maintaining consistency between the object and its directory entry is trivial since both are managed by the same server. Moreover, accessing an object may require one less message exchange — that required in a segregated service to query the name server. Finally, objects are accessible whenever their object manager is; this might not be the case if objects were named through a separate name server and the name server was inaccessible.

On the other hand, servers and clients must duplicate certain functions in the integrated approach. Every server must have its own name parsing code. Moreover, if deemed necessary for reliability or performance, every server must have its own code to replicate the naming information or every application might have to cache names. A segregated approach eliminates this redundancy.

Note that segregation does not imply a centralized implementation. Rather, a segregated name service may be implemented in a distributed fashion. Indeed, all of the example segregated systems are implemented in precisely this manner.

3.2. Scope

Naming systems also can be characterized by whether they name all objects in the system or just some. Again, all systems but the V-System limit the types of objects that can be named. Even the V-System does not ensure that all objects are named consistently, since servers are not *required* to take part in the name service.

As with segregation versus integration, the debate can be thought of in terms of the end-to-end argument, in this case by providing different mechanisms for managing the names of different types of objects. The presumption is that each set of mechanisms will be fine-tuned to its object class, and yield higher performance. In reality, name management appears to require fairly common mechanisms, so the erstwhile "special-purpose" mechanisms are redundant.

3.3. Name Space Structure

The designs discussed all implement some form of global name space. The Clearinghouse, Sesame, and the Domain Name Service specify a hierarchical name space. The V-System specifies only the context portion of a name and leaves the remainder of name parsing operations to individual servers. However, common usage of the V-System naming system is for the server to impose a hierarchical structure.

The fundamental advantages of a hierarchical structure derive from the fact that the name space is partitioned. The size of individual databases (directories) is reduced and each database may be maintained by a different server — perhaps on a different host. Partitioning also provides the means for easily grouping names relative to particular users or administrative domains. On the other hand, such partitioning can result in lower performance than using a flat name space. Consequently, the Clearinghouse restricts the depth of the hierarchy.

3.4. Entry Information

All the designs can provide much more information to the client than a single identifier to which the string name is bound. Each design permits a set of attributes to be returned that further defines the object. In the V-System, these attributes are wired-in at compile-time, once again yielding high performance. In the Clearinghouse and Domain Name Service, it is possible to return attributes that can be interpreted at run-time, yielding greater flexibility at the cost of some performance.

3.5. Context

To deal with users' reluctance to use long absolute names, a name service must be supplemented by a *context facility*. A context facility basically maps relative names into absolute names (see [26] for a detailed discussion of the basic issues). In R*, the context facility is integrated with the name service. The Clearinghouse, on the other hand, provides no context support; other services, especially command interpreters, must do so.

Sesame, the V-System, and the Domain Name Service lie somewhere in between. In the V-System, each workstation has a *context-prefix server* that can be used to define some nicknames. Working directories and search paths are supported as a feature of the command executive, however. Similarly, in Sesame, current directories, search lists, and "logical names" are provided by a separate "environment manager" — of which there is one per user. The Domain Name Service calls for name resolvers to supervise name lookup; a context facility could be built into a resolver or it could be left for outside facilities to implement.

3.6. Wild-carding

Context facilities may be thought of as ways to allow users to use incompletely-specified names — in the sense that the user need not specify the "context" of interest. In some situations, the user may possess (remember) even less information and therefore require a "wild-carding" facility. The Domain Name Service, for example, provides *completion services* in which the set of "best matches" to the partial name is returned. Similar facilities exist in the Clearinghouse and Sesame. Such wild-carding support can reduce the amount of interaction between client and name service required to obtain a complete response to a query, but it also shifts much of the computational burden to the name service. Consequently, the V-System only permits clients to "read" directories and requires them to do any wild-card matching themselves.

3.7. Type Independence

We would like a general name service to handle a wide variety of object types with equal ease. For comparison purposes, we identify three different levels of type-independence — such that the addition of a new object type requires:

1. modifications to applications *and* name servers;
2. modifications to applications, but not to the name servers; or
3. no modifications to applications or name servers.

R* and the Domain Name Service fall in the first class. Sesame and the V-System fall in the second class. The Clearinghouse also falls in the second class, in practice, but could be used as a stepping stone to the third class.

4. Recognition and Synthesis

From the discussion of the previous sections, we can identify a number of areas of commonality across most, if not all of the systems we have discussed. In addition, we identify some capabilities of interest that are not currently provided. Both are summarized in the following points:

- *Decouple service from implementation.*

  A "service" should be provided by a collection of "servers" that adhere to a common protocol. It is not necessary (and in many situations not desirable) that separate servers exist to implement the protocol. Rather, a single physical server can support multiple protocols and participate as a component of several services. In brief, a well-designed service is defined by its interface, and not by a particular implementation.

- *Support multiple media access and object manipulation protocols.*

  Though we encourage the use of common protocols wherever possible, general mechanisms for dealing with heterogeneous services should also be provided. As an example, explicit indication of protocols supported by a client and a server would allow an appropriate protocol translator to be selected (if it was needed). Globally unique identifiers — across all object types — should not be required.

- *Permit easy addition of new object types.*

  We must be able to manipulate large classes of objects using common object manipulation protocols. In addition, we must be able to dynamically create objects of new types and manipulate them to the extent that our object manipulation

protocols allow without major changes to the system — such as recompilation or relinking. In general, such additions should not require human intervention.

- *Support flexible external naming.*

  Largely due to the above two requirements, we should support external names consisting of sets of *(attribute, value)* pairs. By so doing, users may attempt to name objects by any information they have available, rather than relying on a specific positional syntax.

- *Support active name components.*

  Names (or components of names) should be allowed to have an arbitrary action associated with them, which will be invoked whenever the name is resolved. This action is in addition to any other information that may be associated with a name component — such as it's type or access controls. This functionality can be used in combination with object types to implement, for example, extended protection mechanisms, performance monitoring, and cross-domain naming.

- *Provide convenient mechanisms for clients to specify commonly used aliases, nicknames, generic names, and contexts.*

  Traditionally, this has involved simple concepts such as "working directories" or "search lists". We would like to use more powerful mechanisms in which, for example, interpretation depends on either or both the client and the object.

- *Permit autonomy.*

  The failure of any site participating in the naming service must not prevent any other site from accessing information about objects not stored on the failed site. It must be possible for individual system administrators to decide the best implementation technique for their system, in particular, an integrated or segregated implementation. It should be possible to add the naming service to existing operating systems without modifying existing code.

5. An Architecture for a Universal Directory Service

We have ascended from the details of existing systems through a discussion of their strong points, weak points, and omissions, to a very high-level set of principles that we feel are generally applicable to naming systems. In this section, we outline the key features of a *universal directory service* (UDS) that we feel embodies the essential abstractions outlined above. The focus is on architectural issues. For purposes of exposition, any discussion of implementation issues is couched in terms of a segregated implementation. We return to issues of integrated implementation in Section 6.3.

5.1. Target Environment

The environment in which we intend to use a general naming and object location service is best characterized as being heterogeneous. It contains a diverse collection of objects that are implemented and maintained by multiple administrative entities. The environment is also characterized by change; new or improved services will appear continuously. So, objects and even object types will continually be created and destroyed. We must be able to discover and locate the objects that are of interest to our current application.

5.2. Name Space

The UDS uses hierarchical absolute names for all named objects. Syntax is similar to that for UNIX path names [22], but with the (super)root specified as '%'. The collection of all path names constitute the *catalog* of named objects.

A hierarchical name space was chosen for performance and compatibility reasons. Fortunately, it does not preclude the use of attribute-oriented schemes at the level visible to users. For example, a list of *(attribute, value)* pairs might be ordered by sorting first by attribute and then alphabetically within a single attribute. Two reserved characters must be adopted: one to indicate the start of an attribute name and a second to indicate the start of an attribute value. A hierarchical name can be constructed by concatenating the strings for the attributes and their values using a slash followed by the appropriate reserved character as separators. For example, employing **$** and **.** to indicate the start of an attribute and value, respectively:

```
Attribute-oriented:(TOPIC,Thefts) (SITE,Gotham_City)
Hierarchical:      %$SITE/.Gotham_City/$TOPIC/.Thefts
```

While attribute-oriented names can be imposed on any hierarchy, the UDS also defines a special wild-card search to support lookup on such names.

5.3. Catalog Entries

The UDS maps names into entries describing objects. The purpose of these entries is to enable clients to ask appropriate servers to manipulate selected objects. Thus, each entry must contain an identifier for the server implementing the object. Since servers will typically use some form of internal identifiers for the objects they maintain, the UDS should also store such internal identifiers. No assumptions as to format or length of these internal identifiers can be made in a truly heterogeneous environment.

Some servers may want to export type knowledge about objects. For example, a file server may wish to let clients know that certain files are flagged as executable. Therefore, catalog entries contain a type field that can only be interpreted relative to the server implementing the object; a single value for the type field can mean one object type to a file server and a different type to a mail server. Consequently, new object types may be added without modification to the UDS.

The UDS also may cache arbitrary properties associated with an object — such as access control information, last modification time, or annotations [16]. By so doing, the UDS can return useful information to clients on request or can employ the cached information when doing attribute-oriented wild-card searches. However, the information should be regarded strictly as a "hint"; the "truth" can be ascertained only by querying the object's manager.

Architecturally, cached properties consist simply of string data — in the form of (attribute, value) pairs. Consequently, the UDS need only understand the syntax and not the semantics of the data. Indeed, the same observation applies to both server and server-specific object "identifiers", namely, that they consist of an arbitrary string that is not meaningful to the UDS itself.

In this vein, note that the access control information mentioned above is *not* interpreted by the UDS. There is access control information that is interpreted by the UDS, but it is associated with the catalog entry, not with the underlying object. See Section

5.6 for further discussion.

5.4. UDS Object Types

An object manager may enter objects of arbitrary type into the catalog. The UDS is an object manager itself and enters objects into the catalog. The definition of type codes corresponding to the UDS object types must be part of the specification of the UDS interface protocol.

5.4.1. Directory

An object of type *Directory* is used to store a collection of catalog entries. With each directory is associated a particular name prefix. A directory holds entries for all objects whose name consists of that prefix plus some terminal path component. The directory is the fundamental object type of the UDS.

5.4.2. Generic Name

The *GenericName* object type is used to indicate that the named object represents a set of equivalent names. That is, a generic name maps to a variable number of catalog entries. Different approaches to handling generic names may be appropriate, depending on the context in which the generic name is encountered. In certain circumstances, we might just return the list of equivalent entries. In other cases, we might like the UDS to select any one and continue if possible — for example, while parsing a component of a path name. In still other cases, the client or the object manager may wish to specify the criteria to be used in the selection. One useful way to represent a selection function is by identifying a server capable of carrying out the choice (see below). In any event, the catalog entry for a generic name must indicate how to carry out the choice.

5.4.3. Alias

We have just seen how an object of type *GenericName* allows one to perform a selection function (choose one object from a set). An object of type *Alias* allows the inverse mapping, namely, map any one of a set of names to a single object. The UDS identifier for an object of type *Alias* contains the name of the object it is aliasing. This is an example of a *soft* or *symbolic* alias — similar to UNIX 4.2 BSD "symbolic links" — rather than a *hard* or *direct* alias — UNIX "link". Hard aliases are not precluded, however; object managers may choose to register the same object under several different names.

5.4.4. Agent

Authentication can be implemented as a separate service, but there are reasons for supporting the more general concept of an *Agent* in the UDS. First of all, agents other than users need to be identified since objects are typically maintained by programs. Secondly, to protect its catalog entries, the UDS must support the concept of agents (be they users or programs) that may attempt to access objects. Finally, it is convenient to have a notion of user identity that is uniform over the entire name space, rather than trying to handle multiple identities corresponding to different accounts on machines. The catalog entry for an agent must contain a globally unique agent identifier and a password to verify an authentication request. It is also helpful to keep a list of the groups of which the agent is a member.

5.4.5. Server

The primary thrust of the UDS is to allow clients to access arbitrary objects implemented by arbitrary servers. Simply providing the name of the server maintaining an object does not suffice on a heterogeneous system to enable a client to contact a server and properly request it to operate on an object. The UDS must recognize *Server* as a special type of agent and report the necessary additional information. This information includes the various media access protocols by which the server may be accessed and the various object manipulation protocols understood by the server.

To contact a service, the client must know what low level (media) protocol to use to transmit requests to the server and it must know what identifier to use to refer to the server under that protocol. Some servers may be prepared to accept requests in several ways. Thus, the catalog entry for a server must contain a list of *(medium name, identifier-in-medium)* pairs.

Even if a client knows how to contact the server maintaining an object, it must still know how to phrase its requests. That is, it must know which object manipulation protocols are understood by the server. Thus, the catalog entry also contains a list of these protocols.[3]

5.4.6. Protocol

The previous discussion on servers raised the notion of protocols. It is natural and beneficial for a UDS to explicitly support the object type *Protocol*. The benefit is that keeping information on protocols allows greater type independence. Clients do not necessarily know how to use a particular medium or interact using a particular object manipulation protocol. In such cases, they require the aid of a protocol translator. The UDS can keep a list of servers providing translation into a protocol as part of the protocol's catalog entry. By follow-up queries to these servers, a client will then be able to find a server willing to perform protocol translation.

5.5. Parsing Options

The name space is hierarchical, but the parsing process is complicated by the existence of aliases and generic names, and by provisions for "wild-carding". Ideally, the architecture of the UDS should provide transparent handling of these features, while also allowing them to be made visible when desired.

Consider the handling of an alias. The default action that provides transparency is to substitute the alias for the prefix just parsed and restart the parse at the root. If, however, a client wishes to manipulate the catalog entry for an alias, he must have some means for disabling the default action. Thus, the UDS allows clients to give parse control flags as part of their requests. One option prohibits alias substitution.

Generic names raise similar issues. Normally, the reasonable course when a generic is encountered is to invoke the default selection function to pick one choice to continue with. Again, clients will sometimes wish to control the choice made, explore all the choices, or see only a summary indicating a generic entry. Additional parse control options are defined to allow clients to request such special actions.

Another, related architectural issue, is to decide what "name" is

[3]Note that this "type" knowledge need not, in fact, be embedded in the client program itself. See Section 5.9.

returned with a catalog entry (giving the name is important when the request can use wild-card characters). The question is whether to make aliases and generic choices visible. A reasonable course in regards to aliases is to return the primary name: the name that maps directly to the catalog entry without going through any alias. For generic names, a reasonable action is to include a path component reflecting the choice made. The client then has the option of asking for exactly the same catalog entry later if he wishes.

5.6. Protection

Most systems with significant user communities require some reasonable form of protection of one user's information from attempted accesses by another. This need is present whether the users are sharing one timeshared machine, or a collection of computers tied together in a network. Given a UDS, the need for reasonable protection is increased since it is so easy for a program or person on one machine to access the objects and data on another.

A suitable basic protection facility can be derived from any of a number of existing systems. The UDS operations are divided into classes such that an operation in a class may only be performed if the client has been granted the corresponding right. Similarly, clients are divided into four classes: object manager, object owner, privileged users, and everyone else. Ownership is separated from managerial responsibility because while the owner will normally get rights others are denied, the final responsibility for maintaining the object, including its primary name, logically resides with its manager. If desired, special protection mechanisms can be constructed using the portal feature discussed next.

To accomplish this scheme, information about each client class must be added to each catalog entry. The identity of the object owner as well as the manager must likewise be kept. A *privileged user* could be defined in a variety of ways, and could be represented by adding a field to record the name of a user group. Alternatively, it could be implicitly defined as any agent whose list of user groups includes the owner.

5.7. Portals

Thus far we have assumed that all catalog entries are static entities. However, what happens when a systems programmer wishes to monitor accesses or debug access code and therefore wants to filter all accesses to some object (or object group) through another piece of system code — such as a performance monitor or debugger)? What if the system is not booted with all servers running, such that they need to be created at first access? How do we provide for client-specific procedures for generic name resolution or context? How do we introduce extended protection modes? How would we integrate heterogeneous directory systems into our design?

One solution is to allow two types of entries in the catalog structure: *passive* (or static) and *active* (or dynamic). A passive entry designates an existing object requiring no special treatment. An active entry is associated with an action to be taken when the object is referenced. It effectively introduces an indirection in the path name parse. We will refer to the active component of the catalog entry as a *portal*. A portal can be used with any type of object; the type of catalog entry (passive or active) is orthogonal to the type of object (*Directory*, *GenericName*, etc.) about which the entry stores a reference. A portal is invoked every time an attempt is made to map to or continue a parse through a particular catalog entry. Portals can be represented as server identifiers, in which case the UDS interface specification must include the protocol used to communicate with portal servers.

There are three general classes of actions that a portal might take:

1. **monitoring**: Simply "observe" the attempt to access the object, but allow the parse to continue after the portal finishes executing.
2. **access control**: As above, but potentially abort the parse.
3. **domain-switching**: Determine the new name domain (or context) in which the parse should continue *or* complete the parse internal to the portal.

The first class of action handles administrative monitoring and run-time server startup, for example. In the latter case, the UDS is playing a role similar to that of the *listener* or *daemon* processes in many implementations of network architectures.[4]

The second class of action provides, for example, for extended protection modes. The key difference from the first class is that the parse may be aborted by the portal.

The third class of action is perhaps the most powerful. First, it permits object- or user-specific procedures for generic name resolution or context (see the next section). Second, it allows the system to integrate heterogeneous name services; a portal standing in for the "alien" server can forward the as yet unparsed portion of the pathname on to that server for interpretation.

5.8. Context Mechanisms

The UDS name space is a hierarchy in which only absolute names are recognized. Every absolute name is always interpreted in the same manner, with a parse that picks off path name components from left to right, yielding the same catalog entry. There are, however, many benefits to be obtained through the recognition of *context* in the name resolution process. Context facilities can be implemented either directly in the UDS or in separate servers — analogous to Domain Name Service resolvers, Spice environment managers, or UNIX shells.

Many systems provide rudimentary context facilities such as working directories, search lists, logical names, or nicknames. The UDS as currently implemented has no facility for recording a working directory. However, if one were implemented or provided by an external facility, the other common context facilities can be provided using the general primitives already discussed. The effect of multiple search paths can be achieved by setting the "working directory" to be a generic catalog entry instead of an ordinary directory reference. The search paths would appear as a choice of directory entries or aliases pointing to directory entries. To provide the facility of personal nicknames, a UDS client need only create entries under his home directory (or one of the generic search directories just mentioned). The relative name should be the nickname. The catalog entry would then hold as an alias the absolute name for which the nickname stands.

However, these basic context facilities do not satisfy all needs. Consider the problem of resolving "include" file references. If these are stored as absolute pathnames, the same file will always be referred to. That would make it impossible to save access time by using a local version of the same file or to try a compilation with a

[4] There, a *listener* process waits for "requests for connection" and, when one arrives, the listener creates a new *server* process to handle the connection.

personal, test version of code. So, it might seem that the solution is to use relative pathnames and resolve them via a search list. This is not always the answer; in some common circumstances, it is desirable to interpret relative file names with respect to different contexts. For example, formatting another user's files may require resolving the relative file names mentioned in the document root file to the creating user's context, resolving the temporary file names used by the formatter to the invoking user's context, and resolving the commands invoked by the formatter to the context assigned by the maintainer. Ordinary search lists will not necessarily make the proper translations in this case.

Another, slightly less troublesome case is what to do when a file containing file references is moved from one site to another or even from one directory to another. It may attempt to "include" a file **usr/dumbo/foobar** when the directory **usr/dumbo** exists in the new location under the name **common/goofy**. It is possible to define aliases or symbolic links to deal with this problem, but it may result in cluttering top level directories with alias entries. A cleaner solution would be to do the necessary translation through a more powerful context facility.

The UDS provides the means for implementing these more sophisticated context facilities, namely, the portal. The trick is to construct an efficient name map package on a per-name basis that provides the redirection that is appropriate for the context. This package becomes the server implementing a portal. The portal must then be associated with an appropriate catalog entry. For user-defined contexts, an appropriate point would be the entry for the user's home directory. Object-specific contexts, on the other hand, can be created by tagging a particular object's catalog entry. Such an approach yields a logical structure similar to that of property inheritance. It would be convenient under this approach to have a context specification language that can be compiled to produce portal servers automatically.

5.9. Type Independence

At the time an application is written, the writer will have access to existing services through the currently defined, possibly type-dependent object manipulation and media protocols. If these are hardwired in, applications will require continuous modification to keep up with the changing environment. Instead, type-independent applications should be written to handle a general abstract type and an associated object manipulation protocol. Translation to a new type-dependent object manipulation protocols can be handled by protocol translators. The UDS provides explicit support of ***Protocol*** as a basic object type for precisely this reason.

As an example, consider a situation in which the following three servers exist, each with a slightly different object manipulation protocol:

%disk-server speaks **%disk-protocol**

%pipe-server speaks **%pipe-protocol**

%tty-server speaks **%tty-protocol**

Consider also the abstract type, *abstract-file*, with operations OpenFile, ReadCharacter, WriteCharacter, and CloseFile. Call the object manipulation protocol used to specify these operations on the type abstract-file **%abstract-file**. Assume that translators from **%abstract-file** into the protocols above already exist. Finally, for simplicity, assume that all servers and applications exist on a network in which a single form of interprocess communication exists, with network-wide process identifiers. In other words, they are all accessible via the same media access protocols.

In this environment, a typical application would be structured roughly as follows:

1. Look up the name of an object on which the application wishes to do I/O.
2. If the object's manager doesn't speak **%abstract-file**, look up the protocol(s) it does speak.
3. If the protocol has a translator from **%abstract-file**, use it. Otherwise, give up.

Note that it is possible to bury this algorithm in runtime libraries, so that application programmers need not concern themselves.

Now suppose a new type of I/O device was added, managed by the new server **%tape-server** which only speaks **%tape-protocol**. Since **%abstract-file** is a commonly used protocol, the implementor of the new server would most likely supply a new translator that translates from **%abstract-file** to **%tape-protocol**. Once this was done, existing programs would handle tapes without modification.

6. Implementation Issues

In the previous section, the architecture of the UDS was sketched. A complete specification of the UDS interface can be derived by filling in some architectural details. The interface defines the behavior of UDS implementations visible to clients, but leaves implementation details unspecified. In this section, we discuss several such details that are critical to the overall operation of the UDS.

6.1. Replication and Partitioning

To ensure high availability and enhance performance, directories need to be replicated. The availability motivation occurs because while a given directory, say *D*, is stored on a given machine, say *N*, the objects listed in that directory can (and must, in the case of a network transparent system) be stored on various machines. If site *N* crashes or is partitioned away from the other machines, then directory *D* becomes unavailable, and all the objects listed in *D* are inaccessible, even though those objects may be located at the same site as a requesting program.

The performance enhancement motivation occurs because most accesses to directories are look-up, not update. Thus, in principle, multiple copies of a directory distributed around the network permit many look-ups to be local, rather than involving network interaction and delay. Judicious partitioning of information amongst directories can also enhance performance due to locality. Terry [28] analyzes a number of mechanisms for partitioning of information in name servers.

The current UDS implementation uses a modified version of a common voting algorithm [29]. Only updates are voted upon. Requests to read a directory or perform a look-up are done by the directory system to the *nearest* copy (by whatever algorithm is desired to decide nearest). No voting is done to verify that the most recent version of the entry is read; as a result, look-ups should only be treated as "hints". A client can optionally specify that it wants the "truth" (i.e., that a majority read or vote is required).

6.2. Administration and Autonomy

Effective administration of a distributed name domain is essential to a robust system. Sites should remain autonomous to the greatest extent possible for both technical (e.g., performance or availability in the face of partitioning) and non-technical (e.g., accounting or authentication) reasons. In particular, the failure of remote hosts should not prevent local clients from accessing directories that are stored locally.

As discussed thus far, name resolution could involve moving "through" many sites — possibly back to the originating site. If any site along the way is inaccessible, the parse would fail, even if the catalog entry is actually stored on an accessible site. To circumvent this problem, the UDS stores the name prefix associated with each directory stored locally. If an absolute name matches a local prefix, the UDS can (re-)start the parse with the remnant of the name in a local directory.

A related problem is how to achieve autonomy in the sense of being able to enforce local accounting or access control policies. First of all, it must be possible to identify the boundaries between *administrative domains*. A reasonable way to do this is to create a directory structure matching these domains. Under this discipline, directories would be associated with exactly one administrative authority. Special protection at administrative boundaries might be enforced by portals associated with the boundary catalog entries.

Additional administrative autonomy can be achieved by the choice of which UDS servers actually implement a particular portion of the name space. Particular policies imposed by the local authorities can then be coded into the local UDS servers. In addition to accounting and access control policies, local authorities may in this manner enforce other policies, such as dictating which file servers are used for creating new directories. Thus, while UDS servers cooperate in providing a global name space spanning multiple administrative domains, each domain retains the power to control the resources it owns.

6.3. Integration

From the previous discussion, it might appear that the implementation of the UDS would be highly segregated. Indeed, the prototype implementation is segregated [9]. However, the UDS architecture in no way *requires* a segregated implementation.

Rather, the UDS should be thought of as consisting of the collection of servers that adhere to the *universal directory protocol*. These servers may have been written explicitly and exclusively to provide directory service, or they may provide additional services as well. For example, if a mail system was prepared to handle the universal directory protocol, it would classify as both a UDS server and a mail server. In addition, the UDS employs storage servers to store its directories and interprocess communication servers to provide intercommunication between its constituent parts. In fact, it may be quite cost-effective to combine the UDS and storage functions into a single server wherever possible.

In those cases where the UDS is "integrated" into pre-existing servers — such as mail or file servers — catalog entries may take a considerably different form than they would in a segregated implementation. For example, it is no longer necessary to store the identifier for the object manager if the object of interest is maintained by the same server that contains the catalog entry. Also, there is no need to cache properties, since the "master" information is available. Finally, the internal representation of directory entries can be optimized; in particular, it may not be necessary to deal with variable-length information (strings).

7. Summary

In the preceding sections we have touched briefly on a very large number of topics related to naming systems. By examining existing systems, we have attempted to extract and generalize their most useful features. We have tied our ideas together and described how they might be applied to the design of a universal directory service.

Other advanced naming services contain different subsets of important features oriented toward a general naming facility. The UDS integrates all of them, specifically:

- The design focuses on protocol rather than implementation, which admits for implementation as part of existing services or as an additional service — that is, either integrated or segregated.
- The UDS can register name bindings for arbitrary object types — together with information on how to access and manipulate the objects, via media access and object manipulation protocols, respectively.
- The UDS does not interpret the information given by a client as the "meaning" of the name (server-specific identifier, server-specific type code, etc.) Thus, it is type-independent.
- The UDS can both distribute and replicate partitions of its catalog.
- Sites may retain autonomy both in the sense of operating even in isolation and in the sense of enforcing local policies controlling local resources.

The UDS goes farther and makes a number of original contributions in the area of name management, including:

- The UDS design is geared to identifying and providing the information needed to write applications that are type-independent.
- The UDS introduces the conceptually simple, yet powerful extension mechanism of the portal. This enables it to, for example:
  - Incorporate existing name spaces, even those with different name syntax.
  - Create more powerful context facilities.
  - Enforce specialized access control schemes whenever needed.
  - Transparently interpose monitoring facilities.
- The design supports the mapping of attribute-oriented names onto its hierarchical name space by providing a wild-card facility to allow searches based on attribute values. Thus, servers can provide users with a naming interface that is not positional in nature.

Acknowledgements

The UDS has undergone substantial revisions since it was first conceived in the spring of 1981. David Cheriton, Thomas Gross, Michael Kenniston, Charles Kline, Keith Marzullo, Gerald Popek, Michael Powell, and Willy Zwaenepoel participated in various early versions of the design. The prototype implementation is largely the work of Judy Edighoffer, with assistance from Michael Fine and Donald Bate.

This research was supported by the Defense Advanced Research

Projects Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431.

References

1. F.W. Allen, M.E.S. Loomis, and M.V. Mannino. "The integrated dictionary/directory system." *ACM Computing Surveys 14*, 2 (June 1982), 245-286.

2. J.E. Ball, M.R. Barbacci, S.E. Fahlman, S.P. Harbison, P.G. Hibbard, R.F. Rashid, G.G. Robertson, and G.L. Steele Jr. The Spice project. In *1980/1981 Computer Science Research Review*, Department of Computer Science, Carnegie-Mellon University, 1982, pp. 5-36.

3. F. Baskett, J.H. Howard, and J.T. Montague. Task communication in DEMOS. Proc. 6th Symposium on Operating Systems Principles, ACM, November, 1977, pp. 23-32. Published as *Operating Systems Review* 11(5).

4. E.J. Berglund, K.P. Brooks, D.R. Cheriton, D.R. Kaelbling, K.A. Lantz, T.P. Mann, R.J. Nagler, W.I. Nowicki, M. M. Theimer, and W. Zwaenepoel. *V-System Reference Manual.* Distributed Systems Group, Department of Computer Science, Stanford University, 1983.

5. A. Birrell, R. Levin, R. Needham, and M. Schroeder. "Grapevine: An exercise in distributed computing." *Comm. ACM 25*, 4 (April 1982), 260-274. Presented at the 8th Symposium on Operating Systems Principles, ACM, December 1981.

6. D.R. Cheriton. "The V Kernel: A software base for distributed systems." *IEEE Software 1*, 2 (April 1984), 19-42.

7. D.R. Cheriton and T.P. Mann. Uniform access to distributed name interpretation in the V-System. Proc. 4th International Conference on Distributed Computing Systems, IEEE, May, 1984, pp. 290-297.

8. D.H. Craft. Resource management in a decentralized system. Proc. 9th Symposium on Operating Systems Principles, ACM, October, 1983, pp. 11-19. Published as *Operating Systems Review* 17(5).

9. J.L. Edighoffer and K.A. Lantz. Taliesin: A distributed bulletin board system. Submitted to the 2nd International Conference on Computer Message Systems, IFIP, September 1985.

10. M.B. Jones, M.R. Thompson, and R.F. Rashid. Sesame: The Spice file system. Spice Document , Department of Computer Science, Carnegie-Mellon University, October, 1983.

11. K.A. Lantz, K.D. Gradischnig, J.A. Feldman, and R.F. Rashid. "Rochester's Intelligent Gateway." *Computer 15*, 10 (October 1982), 54-68.

12. P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson, and B.L. Stumpf. "The architecture of an integrated local network." *IEEE Journal on Selected Areas in Communication SAC-1*, 5 (November 1983), 842-857.

13. B.G. Lindsay. Object naming and catalog management for a distributed database manager. Proc. 2nd International Conference on Distributed Computing Systems, IEEE, April, 1981, pp. 31-40.

14. P. Mockapetris. Domain names: Concepts and facilities. RFC 882, Network Information Center, SRI International, September, 1983.

15. P. Mockapetris. Domain names: Implementation and specification. RFC 883, Network Information Center, SRI International, September, 1983.

16. J. Mogul. Representing information about files. Proc. 4th International Conference on Distributed Computing Systems, IEEE, May, 1984, pp. 432-439.

17. D.C. Oppen and Y.K. Dalal. "The Clearinghouse: A decentralized agent for locating named objects in a distributed environment." *ACM Transactions on Office Information Systems 1*, 3 (July 1983), 230-253. Earlier, expanded version published as Technical Report OPD-78103, Xerox Office Products Division, Systems Development Department, October 1981.

18. L.L. Peterson. *Defining and naming the fundamental objects in a distributed message system.* Ph.D. Th., Purdue University, 1985.

19. R.F. Rashid. An inter-process communication facility for UNIX. Tech. Rept. CMU-CS-80-124, Department of Computer Science, Carnegie-Mellon University, March, 1980.

20. R.F. Rashid and G.G. Robertson. Accent: A communication oriented network operating system kernel. Proc. 8th Symposium on Operating Systems Principles, ACM, December, 1981, pp. 64-75. Published as *Operating Systems Review* 15(5).

21. D.P. Reed. *Naming and Synchronization in a Decentralized Computer System.* Ph.D. Th., Massachusetts Institute of Technology, 1978.

22. D.M. Ritchie and K. Thompson. "The UNIX timesharing system." *The Bell System Technical Journal 57*, 6 (July/August 1978), 1905-1929.

23. J.H. Saltzer. Naming and binding of objects. In *Operating Systems: An Advanced Course*, R. Bayer, R.M. Graham, and G. Seegmueller, Ed.,Springer-Verlag, 1979, pp. 99-208.

24. J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. Proc. 2nd International Conference on Distributed Computing Systems, INRIA/LRI, April, 1981, pp. 509-512.

25. J.F. Shoch. Inter-network naming, addressing, and routing. Proc. Fall COMPCON, IEEE, September, 1978, pp. 72-79.

26. K. Sollins. *Distributed Name Management.* Ph.D. Th., Massachusetts Institute of Technology, 1985.

27. M. Solomon, L. Landweber, and D. Neuhengen. "The CSNET name server." *Computer Networks 6*, 3 (July 1982), 161-172. Presented at the 7th Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, February 1982.

28. D.B. Terry. An analysis of naming conventions for distributed computer systems. Proc. SIGCOMM '84 Symposium on Communications Architectures and Protocols, ACM, June, 1984, pp. 218-232.

29. R.H. Thomas. A majority consensus approach to concurrency control for multiple copy data bases. Tech. Rept. 3733, Bolt Beranek and Newman, December, 1977.

30. J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

31. R.W. Watson. Identifiers (naming) in distributed systems. In *Distributed Systems - Architecture and Implementation: An Advanced Course*, B.W. Lampson, M. Paul, and H.J. Siegert, Ed.,Springer-Verlag, 1981, pp. 191-210.

32. J.E. White. A user-friendly naming convention for use in communication networks. Proc. Working Conference on Computer Message Services, IFIP Working Group 6.5, May, 1984, pp. 37-57.

33. R. Williams, D. Daniels, L. Haas, G. Lapis, B.G. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost. R*: An overview of the architecture. Proc. 2nd International Conference on Databases, June, 1982, pp. 1-27. Also availables as Research Report RJ3325, IBM San Jose Research Laboratory, December 1981.

34. W. Zwaenepoel and K.A. Lantz. "Perseus: Retrospective on a portable operating system." *Software: Practice and Experience 14*, 1 (January 1984), 31-48. Preliminary version published as Technical Report STAN-CS-83-945, Department of Computer Science, Stanford University, February 1983.