Name Space Models for Locating Services *

Nigel Hinds and C. V. Ravishankar

Abstract

Much of recent work on computer systems has focused on providing transparent resource-sharing in a distributed computing environment. Many of these systems use the server-client model to provide access to data and services. As more distributed services are offered and the demand for sharing increases in these environments, efficient management and accessing schemes become crucial. Locating *services* makes *name service* a critical part of access management.

This report describes some of the work in progress as part of the Universal Name Semantics (UNS) project at the University of Michigan. UNS was intended to explore issues involved in providing client programs with seamless naming of objects across heterogeneous name spaces. UNS is distinguished from other heterogeneous naming systems in its attempt to partially automate the translation process by exploiting *abstract similarity* between name spaces.

1 Introduction

Descriptive naming systems have become more popular in recent years [11]. Their flexibility facilitates item *discovery* by allowing incomplete queries. However, there are currently no data models for describing arbitrary services, i.e., there is no *complete* set of attributes that is capable of describing any service. A print server would have far different attributes than an authentication or name server. The print server could be described by its *lines or pages produces per minute, command language, and paper size*. An authentication server could be described by its *administrative domain or people it will authenticate, and the life time of its keys*. Representing multiple services in a single model would require using either the union or intersection of these attributes. Using the union would lead to considerable waste. Taking a print and an authentication service as an example, the waste might appear insignificant, but as the number of service types increases, so will the waste. Alternatively, using the intersection would most certainly underspecify both services, i.e., the attributes that the services have in common may not be adequate to uniquely identify either service.

As more distributed services are offered and the demand for sharing increases, efficient management and access schemes become crucial. Also, as the environment evolves, heterogeneity is usually unavoidable. This makes name translation an issue. The Universal Name Semantics (UNS) project is a preliminary attempt to design a semantic service description model using *abstract similarity* between services and name spaces. The finished Name Semantic Model (NSM) would be used to implement translators between heterogeneous name spaces. This

*IBM contact for this paper is Jacob Slonim, Centre for Advanced Studies, IBM Canada Ltd., Dept. 61/894, 895 Don Mills Road, North York, Ontario M3C 1W3

This paper is available as IBM Canada Laboratory Technical Report 74.074

paper describes some results of that work.

The remainder of this paper is structured as follows. In section 2, basic attribute-naming concepts are introduced. In section 3, the service naming abilities of a number of state-of-the-art naming systems are critiqued using evaluation techniques introduced in section 2. Section 4 describes techniques for providing heterogeneous name spaces. Section 5 describes the work toward the UNS service naming architecture. Finally, in section 6, we evaluate the UNS work and discuss future directions.

2 Naming Concepts

Fundamental naming concepts are introduced in this section. Much of the information presented here is contained in [16]; the reader is directed there for a more thorough treatment.

A computer or software system can be viewed, at some level, as an object manager. Printing systems treat printers and files as objects; operating systems treat users, processes, and files as objects. Names are given to objects so that users may specify these objects in the system. For example, in UNIX one can think of *inodes* as file object data structures understood and managed by the operating system. Users identify files (objects) by referring to file names, which are strings of characters intelligible to humans.

A *naming architecture* [1] is a model for providing and administering names in a computer system. One component of a naming architecture is the *name space* or *domain*: the universe of all possible names for a particular naming architecture. For example, the Internet mail system name space is made up of strings such as 'ravi@zip.eecs.umich.edu', where the right side of the '@' symbol denotes the computer name (zip.eecs.umich.edu) and the left side identifies a user (ravi) on that computer. The computer name portion of an email address is itself part of the internet naming architecture name space. The Internet host name space consists of computers whose names are constructed from a concatenation of hierarchical domain names strings delimited by '.'. For example, zip.eecs.umich.edu is the name for a computer 'zip' in the 'eecs' subdomain, which is in the 'umich' subdomain, which is in the 'edu' subdomain.

A name resolution system or *name server* is another (functional) component of a naming architecture. The primary function of the name server is to store and *bind/map* names in the name space to objects.

In the UNIX file system example, the naming architecture provides a mechanism for mapping file names in the user domain to inodes in the system domain. Inode data can then be used to physically locate and access the file. In this example, inodes can be considered *names* in another level of naming.

The *context* [16] is an (environment) object that contains a particular mapping from names to objects. A context is usually associated with the name or the environment in which the name is used, or the scope in which the name is valid. Using different contexts will result in different bindings. In addition, *relative mapping* may involve a number of indirect bindings to reach the actual object desired. In this situation many contexts could be involved, each containing a (partial) mapping to an object/name in another context. The construction of Internet host names above is an example of a *hierarchical names space* that uses multiple contexts

or domains.

The UNIX file system again serves as a good example of how context is used. In the case of a hierarchical file system, the current working directory serves as the *current context*. File names beginning with "/" are *absolute* names and interpreted relative to the root directory. All other files are interpreted relative to the current working directory: i.e., each name/directory in the pathname of an object serves as a context in which to search for the next name/directory.

A flat name space can be considered a hierarchical name space with only one directory. The list of ethernet addresses on a local network is an example of a flat name space.

A *resolver* is the user interface component of the naming architecture. Given a name, the resolver will attempt to find the corresponding object (to which the name is bound) by consulting one or more name servers. Resolvers are typically library routines which are linked into client code.

The UNIX file system example in this section illustrates a typical name mapping issue involving two domains; the user-specified file names and the file system inodes. Naming issues, however, are encountered at all levels of a computer system. Even subroutine calls involve them, since each procedure called must be located and address information substituted for the name. In most systems the linker performs this *static* binding immediately after compilation. Newer systems allow *dynamic* binding to routines at runtime. The latter approach has the advantages of allowing libraries of commonly used routines to be shared among programs at runtime, thereby providing greater flexibility. Naming is also an issue in computer hardware. This report, however, will focus on higher level service naming issues (e.g., file system names). For more information on language and hardware level naming, the reader is directed to [16] and [5].

2.1 Models for Scalability

It is less useful to discuss name service in small nondistributed environments, because the name server's primary function is to facilitate communication in a widely distributed environment. Furthermore, heterogeneous naming requires somehow managing or merging a number of name spaces into one. Even the merger of small name spaces will result in a large distributed system. These factors make scalability a major issue in name space architecture design.

Below are some principal methods of providing distributed system scalability [7][17]:

Hierarchical Administration: One of the best distributed administration structures is the hierarchy. Many of the name systems presented will have hierarchical name spaces and administrative models. In such models, the administrative domains typically correspond to the name space domains. At least one name server is responsible for mapping all the names in a particular domain. This distributes the load across servers. As domains grow, they can be partitioned into subdomains. The hierarchical model also distributes authority or responsibility, so conceptually separate domains can be maintained by independent organizations.

Another advantage of hierarchical name

spaces is that they tend to isolate faults. Service disruptions in one branch of the domain will not necessarily affect service in another domain.

Partial Replication: This technique distributes multiple copies of frequently accessed information. Replication prevents access bottlenecks at a single server as the number of users increases. It also provides a level of fault tolerance to any system. In the event of server failure, all name service requests can be redirected to the replica.

Relaxed Consistency Constraints: As updates are made to information in the replicated systems, small periods of data inconsistency exist. Inconsistency is tolerable for many types of information, as long as all the replicas eventually converge.

Caching This is a general technique where clients of a service keep local copies of information retrieved from previous queries.

Dynamic updates also affects system scalability. As the system grows, propagating the effects of dynamic update degrades performance.

2.2 Descriptive Naming Issues

Descriptive or attribute-based naming systems have become more popular in recent years due to their flexibility. Descriptive naming can be used to provide white-pages services to identify users based on imprecise or incomplete descriptions [3]; more importantly, it can also provide yellow-page services for locating computational resources based on properties or attributes of the desired resource [3]. Locating objects based on imprecise descriptions moves into the realm of *name discovery*. If the name of an object is not known, it can be discovered by describing the attributes of the object. This is particularly useful for systems like Cygnus at the University of Michigan [15] [4], where users provide a description to locate and use the desired service.

Because attribute-based name spaces are flat, a user need not be concerned with precise pathname interpretation order, unlike in hierarchical systems. Neufeld explores techniques that provide the flexibility of descriptive naming within a hierarchical name space (Appendix A).

One major issue to be resolved in descriptive naming systems is whether it is necessary to provide a set of attributes that completely describes all objects in the name space, and if so what these attributes are.

The following are descriptive names for three printers:

```
Printer1:
   (language=PS, type=laser,
   pages-per-min=60)

Printer2:
   (language=PS, type=laser,
   pages-per-min=60, ink=blue,
   paper-size=8.5-by-11)

Printer3:
   (language=ascii,
   type=inkjet,
   pages-per-min=60, ink=blue)
```

If we consider Printer2 as having a *complete* set of attributes, then the descriptions of the other printers are *incomplete*. There are cases where a subset of the complete attributes might be enough to describe a desired object. A user wishing to print might query a name server asking for a printer with the following attributes:

```
(language=PS, type=laser)
```

In this example the user does not care about any other attributes, so a number of printers can provide the service. If there is a specific attribute that is not wanted, the user could explicitly say 'attr≠value'. This, however, requires him or her to know all the possible attributes of the object. Furthermore, ambiguities may arise when new objects are added whose attributes are a superset of the attributes of old objects. For example, adding the new printer

```
Printer4:
   (language=PS, type=laser,
   pages-per-min=60, ink=blue,
   paper-size=8.5-by-11,
   location=EECS-4418)
```

might cause ambiguities for a user who in the past used a complete attribute query and had `Printer2` returned.

2.3 Naming: People vs. Services

Previously, no distinction was made between the various types of named data and the impact of that data on the naming architecture. In this section we will distinguish architectural requirements for naming people (white-pages data) and naming services (yellow-pages data). We will later use these requirements for evaluating the naming systems presented in section 3.

- Both WP and YP data usually only require a scaled-down version of a database system.
- Modification patterns:

  WP data infrequently modified. The YP data is frequently subject to change with the addition of a new services.

- Security requirements:

  Are there differences between WP and YP security needs? The name server can be used to store access rights data for any object. System administrators can verify user names and vital WP data when entries are made. YP entries are more dynamic and harder to verify. During the service development process, programmers may require frequent update access to the YP database in order to test new systems. It would be harder to have a single administrator oversee the YP database update process.

  If either YP or WP data are compromised, the data returned by the name server cannot be trusted. This means, (1) the user may unwittingly divulge sensitive data to an intruder, or (2) the data returned to the user is most likely erroneous. For example, a compromised WP service may return bad mail addresses, in order to intercept mail; a compromised YP service may return false host address data for a remote file system, thereby leading the user to the intruder's erroneous file system. If the service used has some authentication procedure of its own, this will lessen the effects of a compromised name service. These issues would stress the need for name-service-wide security to guard against receiving false data and tampering with local data.

- Access pattern differences:
  WP and hostname data typically have a wider remote access pattern than YP data. User names and mail addresses are required for worldwide mail delivery.

  In contrast, services are primarily accessed by local users. This affects distributed security requirements. For example, if YP were strictly for local use, there would be no need for security measures to exchange data with foreign YP servers. Even with restricted access to YP data, some form of data distribution might be needed for fault tolerance or administrative efficiency. In the future, remote access to services is likely to become more common. The number of distributed file systems and databases as well as utility services for managing data is increasing.

- Discovery pattern differences:
  Discovery differs from *lookup* in the nature of the query performed. When a name lookup is performed, the unique (named) object is usually known to whoever is performing the lookup. The aim of the lookup is to determine the value of some attribute(s) of the object.

  When a discovery query is performed, the subject of the query is typically unknown. The aim of discovery is to determine the set of objects that have a certain set of attribute values.

  The discovery requirements for WP and YP data are similar. Attribute-based architectures have proven superior for both types of data [12] [13]. Although the lookup and discovery queries for WP and YP may be similar, the results may be handled differently. For example, when duplicate objects are returned from a WP query, the user may need to further disambiguate the results to find the one true object. In the YP case, the user is often satisfied with any response and needs only to choose one.

All of the above factors affect the naming architecture design process. The ideal YP name server will have the following features:

- Attribute-based:
  Objects are in a flat name space and the user is able to specify queries on any field contained in an object data record.

- Modifiable:
  The system should allow easy online modification by the administrator and possibly by service programmers.

- Distribution:
  The system should provide replication and distribution to facilitate remote access.

- Security:
  The system security should control update access to information contained in the system. In addition, read restrictions to some information might also be required. As part of security, some form of authentication will also be necessary.

3 Previous Work

In this section we review a number of naming systems, and evaluate them using the criteria introduced in section 2.3.

3.1 Grapevine

Grapevine [2] was one of the first large-scale distributed naming systems, and has made many contributions to the general field of distributed computing. Grapevine is a distributed systems environment that provides facilities for delivery of electronic mail, and also offers authentication and access control services for clients.

The Grapevine name server maintains a *registration data base* for mapping names (of users, machines, or services) to information about the name. The name of an entry in the registration database is called an *RName*. The Grapevine namespace of RNames is a two-level hierarchy, where every RName is a string of the form *F.R*. *R* is the *registry* name and *F* is the object name within the registry. Registries represent organizational, geographic, or other potential administrative partions, and are the unit of replication and distribution in Grapevine.

Grapevine's two-level hierarchical name space limits scalability. Another scaling problem is the *pseudo registry*, *gv* that contains information on all other registries. In a large distributed system, maintaining such a list centrally (with or without replication) would be unfeasible.

The Grapevine registry commands are specific to mailing applications. Names in the database are mailing addresses, lists, and other associated mail delivery objects. There are no general mechanism for creating and modifying arbitrary attributes.

3.2 Clearinghouse

Clearinghouse (CH) is a distributed naming service for arbitrary objects [10] that provides a three-level hierarchical name space CH names are of the form *L:D:O*, where *L* is the *localname*, *D* is the *domain*, and *O* is the *organization*. An *organization* will typically be an autonomous institution or corporate entity. *Domain* can be used to distinguish administrative, geographical, or other logical divisions. The *localname* is the domain-unique name of the object. A CH lookup returns a list of object properties of the form <*PropertyName, PropertyType, PropertyValue*>. Property names and values are strings. There are two types of *PropertyTypes*: *item*, which is an uninterpreted block of data, and *group*, which is a set of CH names.

CH has many of the scaling problems associated with Grapevine; for example, update propagation delays will mean inconsistencies in some replicated databases. It does provide the ability to define arbitrary objects and properties (attributes), and appears to also provide the ability to perform attribute-based queries. Finally, CH provides authentication and access control lists to ensure database integrity.

3.3 Sun Yellow-Pages

Sun's yellow-pages (Sun-YP) is a replicated – but not distributed – key-value lookup service. It is a three-level hierarchical name space of the form *Key, Map, Domain*. The Domain represents the autonomous administration boundary. Within a domain, there are an arbitrary number of maps, which correspond to object structure types. For example, the host map contains a binding from hostname (the *Key*)

to the host IP address data.

Although Sun-YP allows replication, it does not provide an update propagation protocol: the user must design the propagation mechanism to provide consistency. Even though Sun-YP allows arbitrary attributes in the maps, lookups only use the designated single value key when searching.

3.4 Univers

Univers is a lightweight relational database that implements attribute-based naming techniques. It is strictly concerned with resource management, and does not implement any particular naming architecture. Systems such as Profile [13] build a naming architecture using Univers and the interpret function translator it provides.

Motivated by the need to maintain ambiguous names in dynamic environments, Univers introduces the notion of an *interpret function*, which is used to define search strategies during resolution. Univers database entries are LISP-like list structures. An example is:

```
((address 35.1.128.16)
(architecture 68020)
(service display))
```

The access routines are also list structures, which allow users to define *interpret functions*. The following example defines a YP-like interpret function to find a set of hosts in the `systemdb` context that match the set of `mandatory` attributes. The results are then ordered by the `ypprime` routine (not shown). `Select_type` returns the named context, which is similar to a table in a relational database system.

```
#yp(type mandatory optional)
(define yp
    (ypprime
    (equal (arg 2)
           (select_type
             (get_context systemdb)
             (arg 1) )
    )
    (arg 3)
    )
)
```

Univers provides a subset of the traditional relational database management mechanisms. It cannot be updated online; all changes must be made to human-readable database files, and then the system is restarted. Furthermore, there are no provisions for distribution or replication.

3.5 Discussion

The advantages of a general attribute-based naming mechanism should be clear. However, none of the systems meets all the requirements described in section 2.3.

Grapevine has a number of scaling problems. Further, the user must provide a registry name (context) as part of the query. This technique limits the registries searched, but might not be suitable for services. Finally, Grapevine does not allow queries on arbitrary attributes.

Clearinghouse also requires users to provide organizational and domain contexts in which to limit the search.

Sun-YP has no provisions for distribution. Replica updates are not designed as part of the system, and therefore typically require complete map transfers.

Univers is only the database portion of a name server, and provides no distributed man-

agement.

Of all the systems, Clearinghouse comes the closest to the ideal YP name server described in section 2.3. However, it lacks scaling and the ability to query on arbitrary attributes.

In the next section we will introduce techniques for managing a heterogeneous name space, followed by a review of the UNS work.

4 Heterogeneous Naming

Managing a global name space in a heterogeneous distributed naming system can be a major issue. Most research in heterogeneity is concerned with machine architecture and network differences. As one considers high-level applications, the list of alternatives becomes long. For example, there are few transport layer protocol standards compared to the number of user interface protocols.

High-level object naming is an important part of distributed computing. The location/access transparency goals of most distributed systems requires merging multiple name spaces and naming mechanisms to provide a global name space. This heterogeneity brings all parts of the naming architecture into conflict.

A number of methods for providing global name spaces are discussed in this section, after which we describe the Heterogeneous Name Service project.

Standardization: A single naming architecture is imposed on all members of the network community who desire naming services. The standard would include a description of the name space, data structures, and resolver routines. As with any standard, there are pros and cons. If all members of the network use the same systems, problems are reduced and maintenance is simplified. On the other hand, standardization tends to entrench bad ideas and designs. If standards cannot be easily changed, the technology must be fairly mature in order to develop a good standard. Typically, by the time the technology has completely matured, one or more organizations have independently developed systems of their own. Often, one of these independently developed systems (with all its features and faults) will be chosen as the standard. Finally, standards-making politics can also reduce the technology to the *least common denominator*.

Clearly some form of standardization is needed, so servers can at least know on which ports to begin looking for peers. The question is, how far standards should go?

Reregistration: Reregistration, like standardization, creates a new global naming architecture, but other naming architectures may co-exist with the new one. Servers may *reregister* (enter) their names with the new system. Clients who wish to may use the new server and gain access to the larger name space.

Reregistration gives new clients of the global server access to names in all the reregistered servers. However, as names are updated and added to the old native servers, these names must also be reregistered with the global server. So, reregistration is a continuous process. Old clients must be modified to take advantage of the

new global name server.

Fallback: In this method, new names are added to the new global name server. Native servers continue to handle queries, but are modified so names that are not found are passed to the new server.

Fallback leaves old clients unchanged. The drawback is that the native servers must be modified to forward queries to the global name server. Name inconsistency can arise if the same name exists in more than one old server. In this case, each user will access his or her respective (differing) local bindings of the name.

Direct Access: The Direct Access method is similar to reregistration. A new global name space is created. Each name in the new name space is mapped, usually by the new server, to a name in one of the existing server name spaces. When queries come to the new global servers they are directed to the native name servers.

Direct Access avoids the inconsistency problem associated with fallback, but requires all queries to perform time-consuming accesses to the underlying native servers.

Heterogeneous Name Service (HNS) is the name server for the larger Heterogeneous Computing System (HCS) project at the University of Washington [18]. HNS uses *direct access*, naming which allows existing name service systems to manage their local data. It then functions as a global name service over all the local name services.

The HNS global name space has two parts. A *context* identifies the native name server domain where the data can be found, and an *individual name* identifies the name of the object in the native server, e.g., 'BIND,fiji.cs.washington.edu', defines a host name that can be resolved by the BIND name service. The right side of the comma is a valid name in the BIND naming architecture. The context portion maps to a single name server or name space.

To query, a client would use a resolver to pass a global name to the local HNS. The client would expect a particular data type as a response: this data type is called the *query class*. Upon receiving the query, HNS would map the (context, query class) pair to a *name semantic manager* (NSM) to handle the specific retrieval of data from the native server determined by the context. The client would then complete the query by making calls to the NSM. Using an NSM, clients can query for names and service without being concerned about the underlying name service used to resolve the query. Note that the NSM is separated from both the client and the HNS, which allows reconfiguration without altering either. Adding new systems requires building a new NSM and registering its existence with HNS.

5 Universal Name Space Approach

In this section we describe the initial approach to designing a generic name space model. We introduce the Name Semantic Model (NSM), then discuss the difficulties that arose during its design and implementation. The original aim of the NSM project was to facilitate name translation in a heterogeneous environment.

5.1 Naming Semantics Approach

This work is based on techniques of identifying the generic attributes of heterogeneous systems – *abstract similarity*. The method is based on software classification [14] and language translation techniques described in [8]. The idea is to describe all the things that are named as objects in an object hierarchy.

We have ignored many of the implementation details regarding network packet translation, e.g., data type and structure layout translation. Although this translation is necessary, it will not be discussed here. In this work we are only concerned with the fundamental issues of name translation. The goal is to design a language for specifying names in the various existing naming architectures. At this point, the work is still in progress. The next sections will describe the preliminary work toward defining an abstract similarity between naming systems and names in general.

5.1.1 Name Space Structure

In this section, we propose a generic name space architecture. We first make a distinction between the structure of the name space (name space structure) and the manner in which names are composed at each context in the structure (name structure).

There is one basic name structure type: hierarchical. In what is traditionally called a flat structure, the entire name space is visible from a single context. This can be considered a single-level hierarchy. In this work, hierarchical names are constructed by concatenating valid names from subdomains or branches, which can be thought of as contexts. In some hierarchical structures, links are allowed, forming a Directed Acyclic Graph (DAG). At any level in the name space structure, objects are uniquely identified by a *context-sensitive name* (CSN), which is a *name* within the context or subdomain of that level.

We further propose to use only one name structure: attribute-based. This is possible because nondescriptive naming systems can be considered descriptive by defining the name as an attribute.

A *completely qualified name* (CQN) is a sequence of context-sensitive names. In the case of the flat name structure, a CQN would be the CSN at the top (and only) level. In a hierarchical structure, a CQN would be an ordered n-tuple of CSNs.

5.1.2 Syntactical Issues

The first step in the translation is to find a way to generalize the syntax used in the various type of naming architectures. This initial grammar allows us to define attribute-based hierarchical systems. This language, however, could not be used to describe Profile names. Profile names only contain values without the associated attributes; the values in this version must be associated with attributes.

```
name
   ::= level ldel csn rdel
   | ldel csn rdel level

level
   ::= '+' |  '+' integer | nil

csn
   ::= name '=' value, csn
   | name '=' value

integer
```

```
   ::= ''numeric integer value''

name
   ::= ''attribute name string''

value
   ::= ''attribute value string''

ldel
   ::=  ''character string''

rdel
   ::=  ''character string''
```

The language is similar to regular expression syntax. The plus-sign meaning 'one or more instances', and the side on which the plus-sign appears indicates the direction of the name hierarchy. With a plus-sign on the left side, the root of the hierarchy would be the far-right CSN. Also as part of the name space definition, the designer would supply delimiter character strings (ldel and rdel).

In the prototype language, the names in the X.500 name space (e.g., "{C=UK}, {O=Telecom}, {OU=Sales, L=Ipswitch}, {CN=Smith}") would be represented as `(value=value)+`, with the delimiter strings, "{" and "}".

5.1.3 Semantic Issues

We move now to semantics, the most difficult and important part of the translation process. There are a number of obstacles to overcome in this area. In this work, we identify some of the semantic translation issues specific to naming systems, and offer structures and techniques for handling name semantics.

The semantics problem can be divided into two cases: translation between similar naming architectures, but with different name space instances (Figure 1), and translation between different naming architectures (Figure 2). In Figure 1, both naming architectures are the same. However, the actual name spaces are different. The challenge is how to convert xclock in NS_1 to xclock in NS_2. In Figure 2, the architectures of the name space are different: NS_1 is hierarchical, while NS_2 is flat. This architectural difference poses additional problems, which will not be discussed in this paper.

Another example of the semantic translation problem (between similar architectures) would be translating between two flat attribute-based name spaces. If the first name space had types login, name, room, and phone, and the second name space had the types signon, name, office, and phone, it is clear that in general, translating between the two spaces is impossible. Universal Name Semantic (UNS) uses classification techniques to define common abstract types.

Classification methods are used in many areas. Classification systems describe groupings of similar objects and show relationships between dissimilar groups [14]. A simple classification structure looks like a hierarchy, but more complex relationships can generate network structures. *Hierarchical classification* relationships are based on the principles of subordination and inclusion, while *syntactical relationships* relate two or more concepts belonging to different hierarchies. The Dewey decimal system is a typical enumerative hierarchical classification system. The system we propose is an enumerative hierarchy, where the universe of objects is divided into successively narrower classes.

Our work is also based on Lee and Malone's

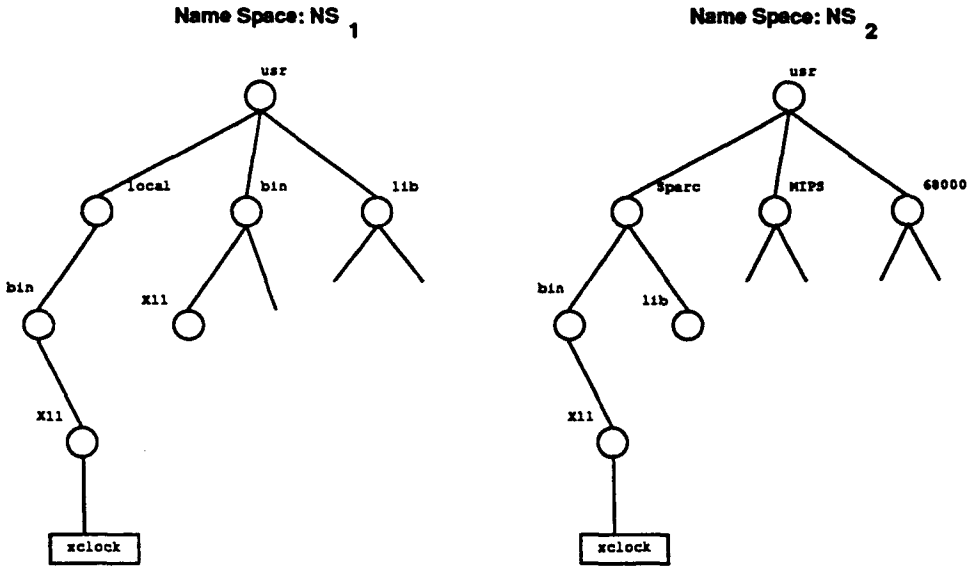

Figure 1: Similar Name Architecture Translation

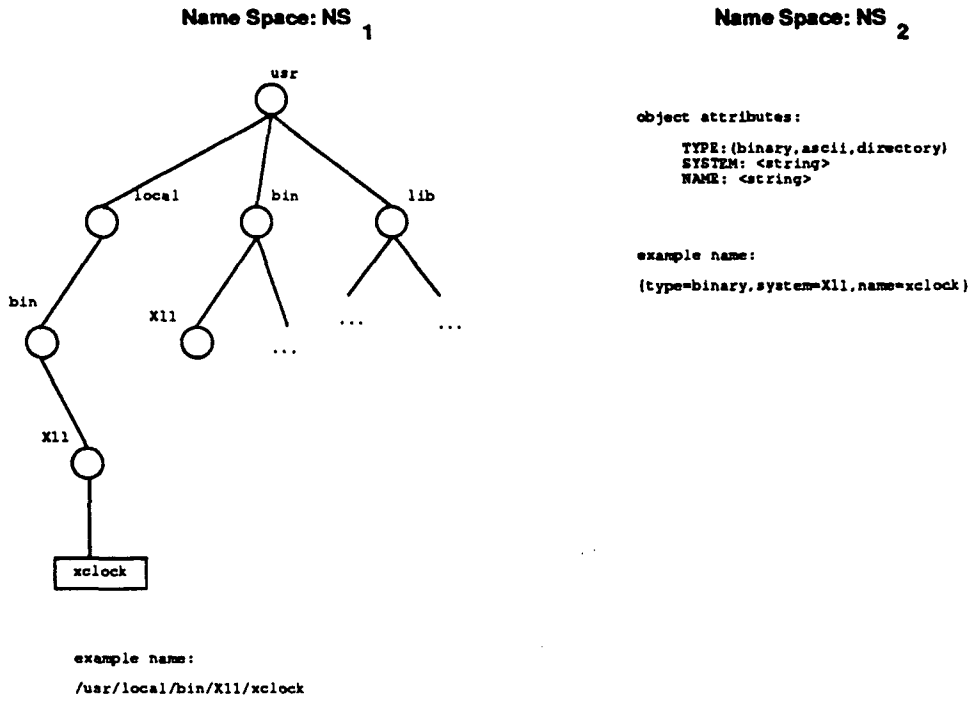

Figure 2: Architectural Differences

work on type hierarchies for communication. In their technical report [8] they discuss possible translation schemes for object systems from a set theoretic view-point. For our purposes, the *languages* of the communicating groups can be considered sets of named objects. A group language G_i is used by a subset of users who work together. Languages may vary in syntax, as well as in semantics captured by the structures. A set overlap represents syntactic or semantic similarity. A *common language*, C, is defined as a language that all language groups use to communicate with one another.

The translation solution space is then described as set relations between the common language C and any group language, G_i. Translation involves converting the objects from one language to semantically equivalent objects in the target languages.

Considering the case where every group language has the same relationship with the common language, there are six relationships that will be evaluated with respect to the following objectives:

Maximize Expressive Adequacy:
Presumably, each distinct group language provides some function not provided by the other languages. G_i must be powerful enough to express varying structures and semantics.

Minimize Need for Consensus: Redesign of G_i type structures should require minimal consensus from other language groups.

Minimize Need for Updates:
Somewhat related to the consensus objective, updates are required when a modification to one language affects translation information maintained by another language. Propagating these effects is costly from a communication and programming standpoint, and should be minimized.

Five of the six set relationships are presented and evaluated below:

$C = \emptyset$: No Common Language: If there is no common language, then the groups wishing to communicate must perform pair-wise translations. This requires that each language maintain a translation "dictionary" for all other languages.

This method provides flexibility to design detail translations between any two languages, but suffers from the classic $n(n-1)$ translators problem.

Expressive Adequacy: Since each language performs pair-wise translations, the separate languages G_i are free to represent structures in any manner.

Consensus: There is no need for consensus, but this makes translation very costly.

Updates: Although consensus is not required for structure modifications, changes must be propagated and customized to the dictionaries maintained by the other languages.

One difficulty that all translations will have in fully preserving the semantics is managing ambiguity:

If there is a one-to-many mapping from a language G_i structure X to a G_j structure Y, then there should be other characteristics of X that allow unambiguous selection of the Y to which X should be mapped.

$C = G_i$: Identical Group Languages:
This is the standardization case, where all groups use the same language.

Expressive Adequacy and Consensus: Expressiveness is limited and the need for consensus is high. All groups must agree on all changes.

Updates: Update must be propagated to all languages, but this really amounts to a wholesale replacement and is more manageable than the pair-wise case.

$C \cap G_i = \emptyset$: External Common Languages: In this scheme, the languages are disjointed from one another. Communication between languages is performed by first translating into the common language. With this method, each group only needs to know how to translate its own language into and out of the common language.

This method suffers primarily from the fact that the one-to-many problem is now twofold. Translation from G_i to C, as well as translation from C to G_j, must be unambiguous.

Expressive Adequacy: Separate group languages can once again be designed with as much power as needed. The common language design and modifications require consensus, but typically changes to the common language will be infrequent, reducing the number of updates.

$C \subset G_i$: Internal Common Languages: Every group has a portion of its language that is shared with all the other groups. Object hierarchies provide a good example of this scheme. Base object types can be shared at higher abstract levels. Objects are described as basic types with further distinctions made within a group language. For example, a *file* object might have basic attributes shared by all groups, such as creation date and length. Any group may create a subclass of the type *file*, with more attributes to further distinguish *files*. When communicating within the group, all the descriptive information accompanies the object. Inter-group communications will only recognize the shared object types; objects are translated to their nearest common "ancestor".

Expressive Adequacy: Groups have much flexibility in expressiveness, and are allowed to add as many subtypes as they require. However, groups cannot ignore distinctions made in the common hierarchy.

Consensus: Consensus is required only among the objects in the common hierarchy.

Updates: Updates are required only of the objects in the common hierarchy.

$C \supset G_i$: Superset: The common language is a superset of the group languages. If different groups maintain different type hierarchies, when one group receives an object of unknown type, that type must be imported. This scheme can be very similar to having an external common language.

Expressive Adequacy: Expressive freedom is excellent.

Consensus: There is little need for consensus.

Updates: As with the external language case, updates are costly because a change to one language might require changes to the $(n-1)$ translators.

$C \cap G_i \neq \emptyset, C \neq G_i$: Intersection:

Intersection is not discussed because it can be considered as a combination of the five schemes.

5.2 The Name Semantic Model

In this section, we outline a prototype Name Semantic Model (NSM). The main concern here is to construct a simple test case to determine if such a hierarchy is a feasible approach to managing heterogeneous name architectures. The NSM model is an enumerative classification hierarchy [14] that defines objects with inheritance. For translation purposes the model is very similar to Lee's *Internal Common Language* model. We first define the *common* objects in the NSM, and then describe a sample translation.

At the most basic level, we name *objects*. Further, the group of objects is divided into (elementary) subsets of *person*, *place*, or *system*. For simplicity, we will only consider objects needed to describe a portion of the UNIX file system. Figure 3 illustrates a sample NSM. Each node has a label that represents a *type*. Arrows from the node are *type attributes*, and lines between nodes form the class or type hierarchy. An initial set of important NSM types and attributes is described below.

place: For the moment the only type here is *directory*. Place describes objects used to hold other objects. *Country* would be an example of a place. This model has no attributes associated with *place*.

directory: This is the traditional file system notion of a directory. It has attributes:

name: character string.

contents: one or more *object* types. For example, the directory *X11* might contain documentation as well as binaries, `contents=exec,file`.

system: This is a broad category used to describe large multi-component software or hardware packages.

admin: person or organization charged with managing the system.

software: This describes entities other than directories that exist in the file system.

file: Although executables exist as files, a distinction has been made between the two.

format: of type (ASCII, EBCDIC, ...)

(exec)utable: This is a *machine-interpretable* sequence of instructions.

(interp)reter: type of the interpreter (machine).

hardware: This is a physical device, possibly with a processor.

(arch)itecture: processor name.

To demonstrate the use of the model, we will use the UNIX file system name space, and analyze two similar name space structures with differing name space instances (Figure 4). In both name spaces, each node or name in a directory must be defined as an instance of an object in the NSM. In this example, the nodes are either *directories* or *executables*. It would also be necessary to provide values for the attributes inherited from the NSM type.

The translation process is outlined below:

1. A Name Semantic Object (NSO) (an object in the name space viewed as an object in the NSM) is constructed by traversing the name space nodes on the way to the

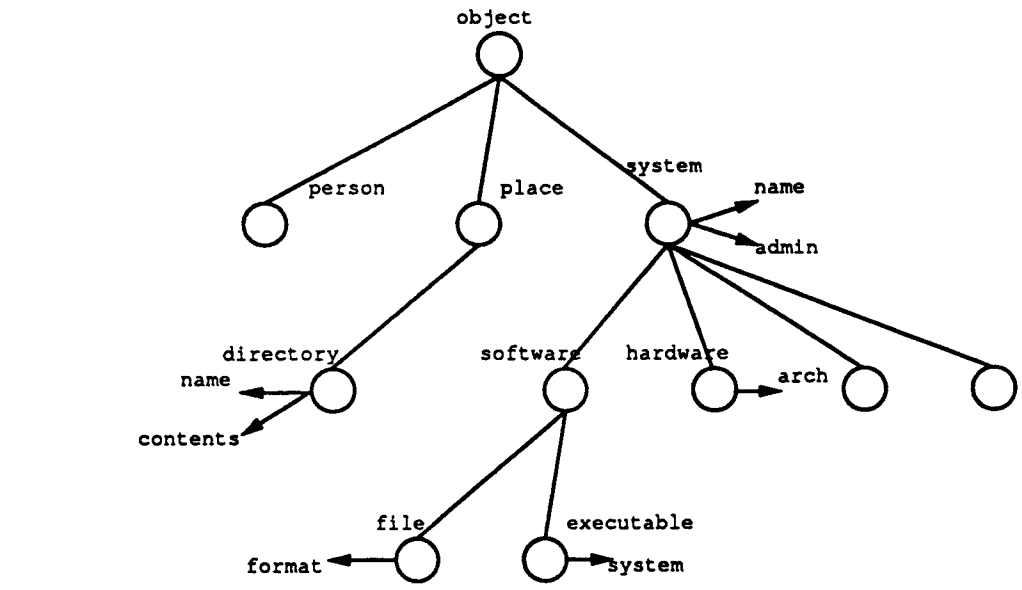

Figure 3: Sample Name Semantic Model

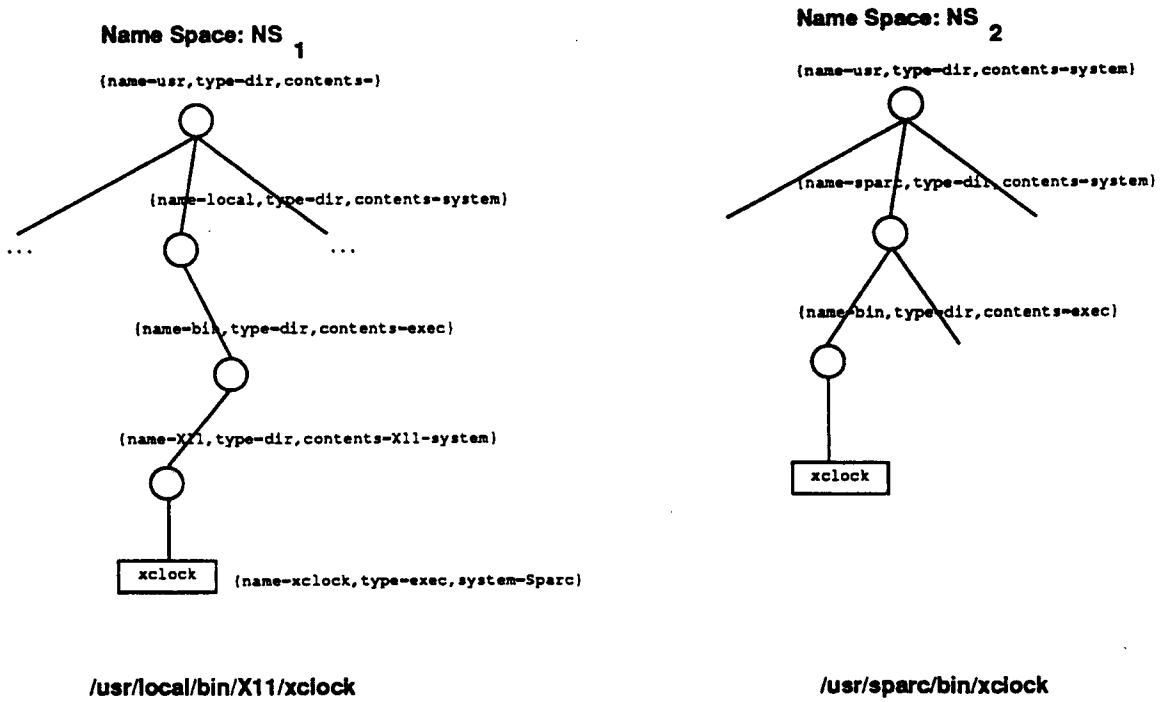

Figure 4: Sample Name Spaces NS_1 & NS_2

object being translated. Starting from the root of the name space, traverse each CSN accessing its *type* data in the NSM.

2. NSO_1 is translated into NSM_2 by processing each object in NSO_1 [8]:

   (a) *For any object, if it is an instance of a type shared between NSM_1 and NSM_2, then the object's type remains the same.*

   (b) *If the object does not belong to a type in NSM_2, then the object is automatically translated into the most specific supertype shared between NSM_1 and NSM_2.*

3. Construct a name in NS_2. Traverse NS_2 referring back to NSO_2 to create CSN at each level.

Sample Translation

NSO_1 is constructed by traversing the pathname in NS_1 (in Figure 4). At each node in the path, *type* information from NSM_1 is used to add a new node to NSO_1. When the traversal is complete, the structure in Figure 5 will have been created. In NSO_1, the new subtype *machine* represents the addition of a nonshared type (not shown in Figure 3). There are two system nodes, one for the software object being named and one for the hardware architecture on which it runs. Although there are a number of *directories*, to save space the *place* supertype has not been replicated.

NSO_1 is then passed to the name server for NS_2. Each object in NSO_1 is translated into an object in NSM_2. The result will be an NSO (NSO_2) in NSM_2. One notable translation in this example is that of *machine* in NSM_1/NSO_1 to *executable* in NSM_2. Since NSM_2 does not have the type *machine*, the objects below it were translated into the nearest shared type, i.e., *executable*. Attributes from *machine* were moved to *executable* as well.

Finally, NS_2 is traversed to create a name in that name space. At each node in the name space, type information for that node is combined with type information from NSO_2 to determine which subnodes/directories to include in the path. If the process works correctly, the object being translated should be located (and the translation terminated) at or before a leaf directory in NS_2. At directory /usr in NS_2, the contents of each subdirectory would be a *system*. At this point the translator would look in the NSO_2 for any *system* names that match those in the directory. The **sparc** system is chosen at this level. At the next level the **bin** directory (containing *executables*) is chosen because the translator knows that the NSO_2 describes an *executable* object. At the last level, the translator will search the leaf directory for a file with the name of the desired object.

5.2.1 Implementation

The current implementation includes a yacc name syntax grammar and graph manipulation package. This version of the grammar is capable of processing name architectures with the following forms:

```
name
   ::= csn_name ''/'' name
   | csn_name

csn_name
   ::= value | name ''='' value
   | csn_name

name
```

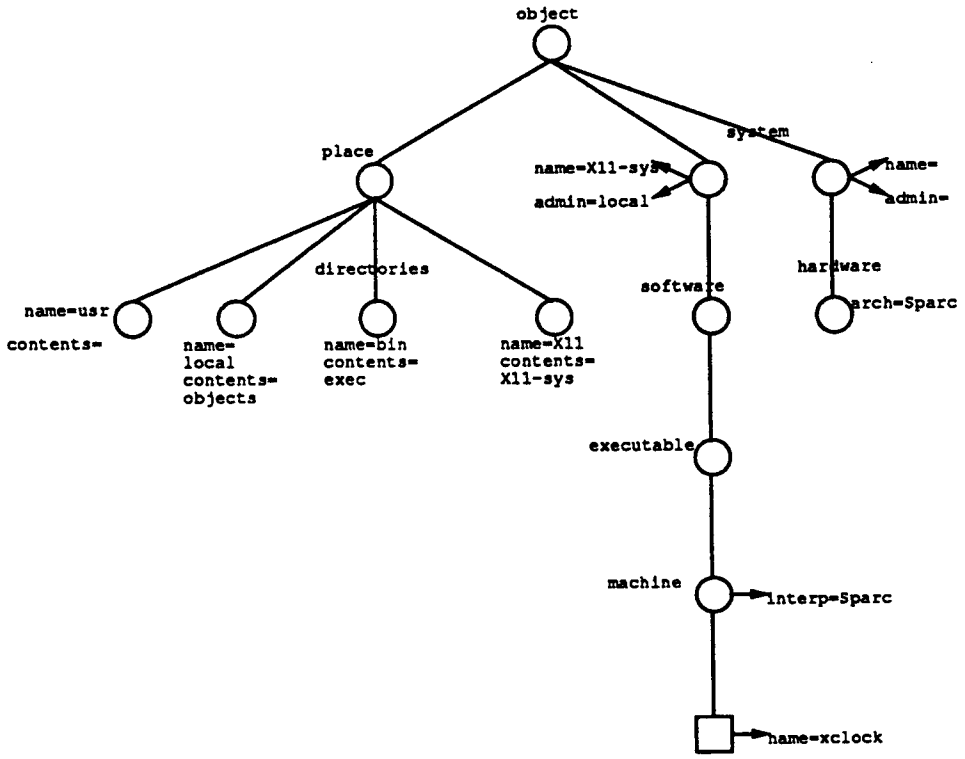

Figure 5: NSO_1

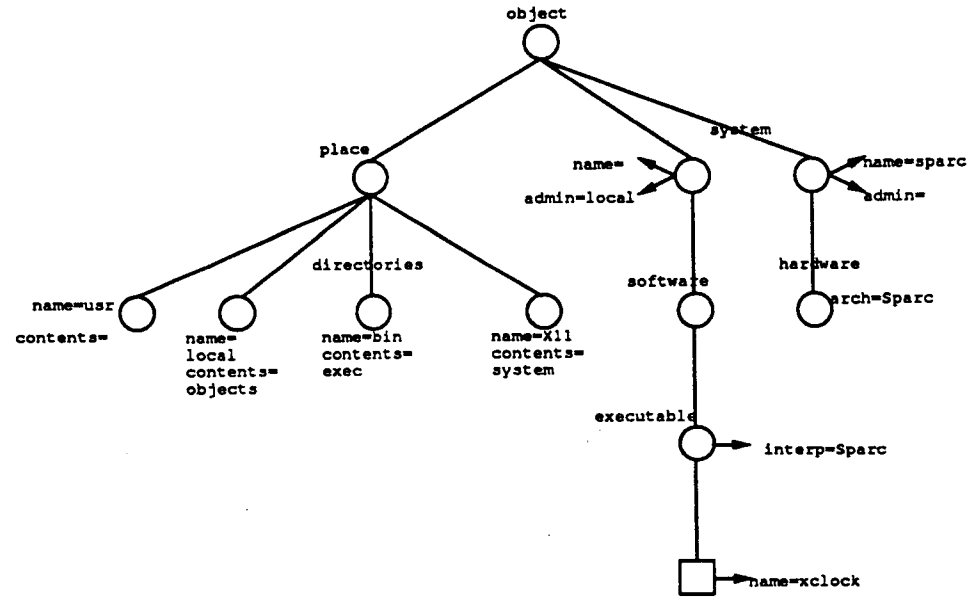

Figure 6: NSO_2

```
    ::= ''attribute name string''

value
    ::= ''attribute value string''
```

Using the language described in section 5.1.2, the system is capable of processing names of the form:

```
(value|name-value)+
```

The graph package can be used to construct an NSM as well as a name space structure. The NSM objects can include annotations or type data (name-value data within brackets). For example, the NSM in Figure 3 would be entered into the system as follows:

```
create nsm
    object/system[name=<default>,
     admin...]/software/executable
add object/system/software/file
        [owner=<default>, ...]
add object/place/directory
```

There are number of other implementation issues, which we discuss briefly below:

Defining the Name Space: The graph package can be used to define the NSM and the valid attributes. However, instead of defining the name space in the graph package (as was done in the examples), a UNIX file system name space can be used. The translation program can be made to operate directly on files in UNIX, e.g., attributes can be associated with directories and files by including dot-files.

Exporting the Service: An RPC interface can be used to export this service to users. An alternative is to have resolvers connect to a *well-known* port.

Queries Packets: As mentioned in section 5.1, there are packet format issues to overcome. The focus here has been on the translation of the information contained in name service packet. One solution would be to design a message format for the UNS system and perform this conversion as part of the translation process.

Discovery: Users can discover named objects by providing partial queries to the name server.

6 Evaluation and Future Work

The major problem with the UNS work has been the lack of a clearly defined NSM. YP objects have diverse types. Attempting to relate all the possible types and their attributes in one semantic hierarchy has proven fruitless thus far.

In looking at the requirements of providing an attribute-based system, two critical issues arise.

1. Systems such as Profile and Univers provide attribute-based naming for flat name spaces, similar to relational databases. We would like to provide the user with an interface that accepts any unordered set of attributes in a query, i.e., provide the illusion of a flat name space for query purposes. This would either mean checking multiple name spaces (possibly one for each service type) or searching a single very large name space – assuming there is a way to combine the disparate service types into one name space. In either case a considerable amount of time is spent

searching the flat name space.

2. To address the problem of combining the disparate *types* into one name space, the dynamic nature of service types is not supported by most relational database schema design techniques. Considerable waste can be introduced if a single table is used to describe all services.

Based on these two issues, YP server design becomes a data organization problem. Neufeld [9] (cf. Appendix A) has explored some techniques for combining the efficiency of hierarchical naming spaces with the simplicity and flexibility of attribute-based naming. The Neufeld methods have been used almost exclusively for user or WP data. Future work will begin by examining the Neufeld techniques to determine their applicability to YP name spaces.

References

[1] M. A. Bauer. Naming and name management systems: A survey of the state of the art. Technical Report #241, The University of Western Ontario, June 1989.

[2] A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An exercise in distributed computing. *Communication of the ACM*, 25(4), April 1982.

[3] M. Bowman, L. Peterson, and A. Yeatts. Univers: An attribute-based name server. Technical Report, University of Arizona, 1989.

[4] R. N. Chang and C. V. Ravishankar. Language support for an abstract view of network service. Technical Report, University of Michigan, Ann Arbor, 1989.

[5] B. Furht and V. Milutinovic. Microprocessor architectures for virtual memory management. In *Tutorial: Computer Architecture*. IEEE Computer Society Press, New York, NY, 1987.

[6] S. Kille. The design of quipu (version 2). Research Note RN/89/19, Department of Computer Science, University College London, March 1988.

[7] B. W. Lampson and H. Sturgis. Hints for computer system design. In *Proceedings of 9th ACM Symposium on Operating Systems Principles*, pages 33–48, July 1983.

[8] J. Lee and T. Malone. Partially shared views: A scheme of communicating among groups that use different type hierarchies. CCSTR #111, MIT, Sloan School of Management, 1989.

[9] G. Neufeld. Descriptive naming in X.500. In *SIGCOMM '89 Symposium, Communications Architectures and Protocols*, pages 64–71, September 1989.

[10] D. Oppen and Y. Dalal. The clearinghouse: A decentralized agent for locating named objects in a distributed enviroment. *ACM Transactions on Office Information Systems (TOIS)*, 1(3), July 1983.

[11] L. Peterson. An architecture for naming resources in large internet. TR 87-7, University of Arizona, Tucson, November 1987.

[12] L. Peterson. A yellow-pages service. In *Proceedings of ACM SIGCOMM '87 Workshop*, August 1987.

[13] L. Peterson. The profile naming service. *ACM Transactions on Computer Systems*, 6(4), November 1988.

[14] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1), January 1987.

[15] C. V. Ravishankar and R. N. Chang. An attribute-based service-request mechanism for heterogeneous distributed systems. Technical Report, University of Michigan, Ann Arbor, 1988.

[16] J. H. Saltzer. Naming and binding of objects. In *Operating Systems: An Advanced Course*, pages 99–208. Springer-Verlag, Berlin, 1979.

[17] M. F. Schwartz. *Naming in Large, Heterogeneous Systems.* PhD thesis, University of Washington, August 1987.

[18] M. F. Schwartz, J. Zahorjan, and D. Notkin. A name service for evolving, heterogeneous systems. In *Proceedings of Operating Systems Conference '87*, 1987.

About the authors

Chinya V. Ravishankar is presently Assistant Professor in the Electrical Engineering and Computer Sciences Department at the University of Michigan. His research interests include distributed systems, programming language design and implementation, and software tools.

Ravishankar received his B.Tech. degree in Chemical Engineering from the Indian Institute of Technology–Bombay, in 1975, and his M.S. and Ph.D. degrees in Computer Science from the University of Wisconsin–Madison in 1986 and 1987, respectively. He is a member of ACM and IEEE. He can be reached at the EECS Department, University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109-2122, or by e-mail at ravi@eecs.umich.edu.

Nigel Hinds is presently a Ph.D. precandidate in Computer Science at the University of Michigan. His research interests include distributed systems, databases, and name service.

Nigel received his B.S. degree in Computer Science from the University of Michigan, Ann Arbor, in 1986. He can be reached at the EECS Department, University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109-2122, or by e-mail at nigel@eecs.umich.edu.

A Appendix: Neufeld X.500

The X.500 model [9] combines a hierarchical name space with descriptive naming. An X.500 name space is a directory database or Directory Information Tree (DIT). In a DIT, each node or *entry* represents an ISO object such as country, organization, person, or application. Each entry contains a set of attributes as defined by the entry class. For any entry a number of the attributes are defined as *distinguished*. Distinguished attributes are used to ensure uniqueness among sibling entries. Together with their values, the set of distinguished attributes form a *Relative Distinguished Name* (RDN). A *distinguished name* for an object is a sequence of RDNs.

X.500 descriptive names can be used to describe just enough about the object so as to distinguish it from all other objects in the name space. In an example from [9] (Figure 7, an RDN for 'Peter Smith' is {C=CA, Org=UBC, CN='Peter Smith'}. In this case neither Organizational units 'Science' nor 'CS' were needed to describe the person.

To reduce the search space during name resolution, X.500 uses the concept of a *registered name*. A registered name is a set of RDNs. Again from [9], in Figure 7, {C=CA, Org=UBC} is a registered name and no object below {C=CA} may have org=UBC as an RDN attribute. The resolution algorithm first finds the object denoted by the largest registered subset of the attributes (in a query), and then uses the remaining attributes to search below that object.

X.500 also provides replication and partitioning within a DIT. A Directory Service Agent

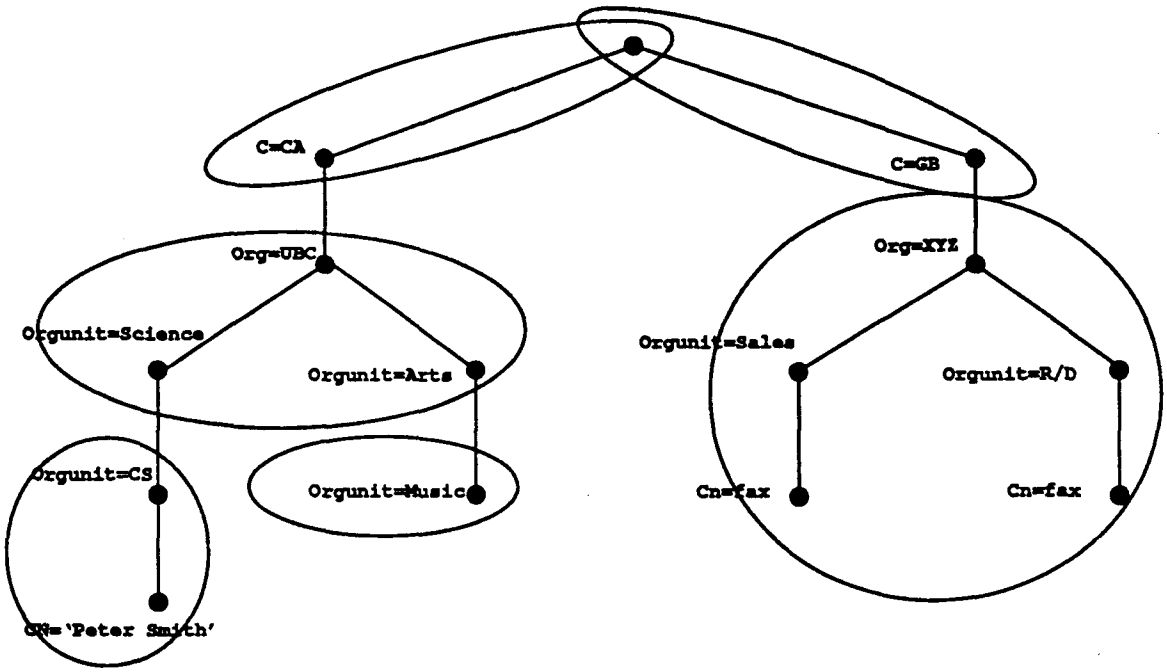

Figure 7: X.500 Directory Information Tree

(DSA) may contain a portion of a DIT called a *naming context*.

The QUIPU implementation of X.500 [6] provides authentication between the DSAs and the Directory User Agents (DUAs) or resolver through which the DSAs and DIT are accessed. QUIPU further provides Access Control Lists for each entry.