1984

A Name Resolution Model for Distributed Systems

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Larry J. Peterson

Report Number:
84-491

A Name Resolution Model for Distributed Systems

Tilde Report CSD-TR-491

(revised February 1985)

Douglas E. Comer
Larry L. Peterson

ABSTRACT

Naming is an important issue in any programming system, especially when the system spans multiple computers. We present a model that describes names and name resolution in distributed systems, in which we view names to be purely syntactic entities, and name resolution to be a syntax-driven operation. We use the model to formally define familiar concepts and terminology involved in naming. We also describe the role of name servers in name resolution. Examples from computer mail, a distributed application that uses many forms of names, illustrate our model.

This project is supported in part by grants from the National Science Foundation (MCS-8219178) and SUN Microsystems Incorporated.

1. INTRODUCTION

A significant characteristic of any computing system is the way in which objects in the system are named. The mapping of names into the objects they represent is the responsibility of the computing system's *name resolution mechanism*. Because the naming mechanism supports references to objects, it directly influences both the ease with which users refer to objects, and the degree of object sharing allowed in the system.

Names in programming languages and operating systems are well understood. For example, Johnston [JOHN71] defines the contour model for resolving names in Algol-like languages. Daley and Dennis [DALE68], Fabry [FABR74], Denning [DENN70], Henderson [HEND75], and Ritche and Thompson [RITC74] describe various paradigms for referencing and sharing segments and files in operating systems. In this paper, we present a model for discussing names and name resolution in distributed systems.

Although no precise definition of distributed systems exists, they can be characterized as consisting of multiple processing components that are physically distributed and therefore must interact through a communication network. In addition, a set of services unifies the distributed system and provides access to the resources comprising the system [ECKH78, ENSL78, LELA81]. In practice, distributed systems range from tightly coupled distributed operating systems, to very loosely coupled mail systems.

For example, Distributed Operating Systems (DOS), such as LOCUS [WALK83], Accent [RASH81], TILDE [COME84], EDEN [ALME83], and the V-System [CHER83], consist of a collection of computers connected by a local area network. The DOS provides users of the system with access to the same types of objects as centralized operating systems: files, compilers, printers, etc. Users identify objects as though they are local, even if they are implemented on remote computers; the DOS hides the distributed nature of the environment by providing transparent access to the component resources.

On a larger scale, XEROX interconnects individual local area networks to form an internet. Users work on workstations connected to the internet, and request services from server processors that are also connected to the internet. *GRAPEVINE* [BIRR82], for example, provides a mechanism for users to send messages to each other by implementing a message service and a registration service. Users register themselves with a certain registry. Programs running on behalf of message senders access the registration service to locate a message server willing to accept the message for the recipient. Users identify each other for the purpose of sending mail with names of the form $F.R$, where R identifies a registry, and F denotes a user that is registered at that registry.

Even though the distributed system supported by XEROX is physically larger than than those that implement DOS, it is still under the auspices of a single authority. The *DARPA* Internet is an example of a distributed system in which the component processors have a larger degree of autonomy. In the *DARPA* Internet, the primary shared resource is the computer, called a *host*. Users may remotly log into a host via the TELNET protocol, access files on a remote host with the FTP protocol, and send mail to a user on a remote host indirectly through the SMTP protocol. (See [REYN84] for *DARPA* Internet protocols.) Hosts in the *DARPA* Internet are currently have names of the form *host*1. The domain-name project [MOCK83], however, is replacing flat names with a hierarchical scheme. For example, *host*1 might be known as *host*1.*anycity*.*anystate*.*usa* in the domain-name system.

While processors in many distributed systems cooperate so all resources in the system can be accessed by all users in the system, processors in computer mail systems only cooperate to the degree necessary to exchange mail. That is, the only namable resource is the computer *mailbox*. The computer mail transport system consists of a confederation of interconnected delivery systems.

Users on a single computer system share a *local* mail delivery system [SHOE79]. If computer systems communicate through shared protocols they may cooperate to form a *network-wide* delivery system [POST82]. When computer systems are connected to more than one network, network delivery systems overlap to form a *global* mail system [ALLM83].

On networks like the DARPA Internet [POST82], mailbox addresses contain information identifying a user on a specific host. For example, a sender might identify John Doe, a computer scientist at the National Laboratory, as *jxd@nlabs*, where *jxd* is John Doe's login identifier, and *nlabs* is the official DARPA host name for the computer he uses. On networks like UUCP [NOWI80], however, the mailbox address must specify all the intermediate computers through which the message must travel to reach the destination computer. For example, if the sender's computer is connected to a computer named *host*1, which is connected to another computer named *host*1, which is in turn connected to to *nlabs*, the sender might specify John Doe's UUCP mailbox address as *host*1!*host*2!*nlabs*!*jxd*. UUCP style addresses are called *source-route* addresses [SUNS77] because they specify a route through the network from the source computer to the destination computer. Finally, when the recipient's mailbox exists on a remote network, the address must identify the bridges (i.e. hosts that connect network mail systems), that the message needs to cross. For example, if we consider *nlabs* to be a mail bridge between the UUCP and DARPA Internet networks, and John Smith is a user with login identifier *jxs* on the DARPA host *stateu*, then the sender in the previous example might address John Smith at State University as *host*1!*host*2!*nlabs*!*jxs@stateu*.

Because users identify each other with a mailbox address, networks often provide *name servers* (also called *white-pages directories*), that translate information about a person (e.g. name, place of employment, etc.), into a corresponding mailbox address. For example the DARPA Internet provides a white-pages directory called NICNAME [PICK79]. The CSNET name server provides a similar service [SOLO82, DENN81]. A naming scheme developed by the IFIP Working Group 6.5 combines name servers and the mail service by replacing mailbox addresses with *human-oriented names* [CUNN82].

In the rest of this paper, we investigate issues involved in naming and name resolution in distributed systems. First, we review existing models for names in Section 2. Next, we define the underlying components of a distributed system in Section 3, and we present a new model for name resolution in Sections 4 and 5. Finally, we present a conceptual discussion of our naming model in Section 6, and we investigate the role of name servers in Section 7.

A final note. Examples from computer mail illustrate our model. Using computer mail allows us to give concrete examples for a single, intuitively obvious, object type. In addition, mail contains a rich set of names, which unlike many low-level names, are easily readable.

2. RELATED WORK

Because naming is a fundamental issue in distributed systems, it is important to understand the underlying principles. Existing work on naming point to four basic ideas: (1) multiple levels of names exist for objects, where high-level names are mapped into low-level names; (2) there exist various kinds of names, such as hierarchical, absolute and relative; (3) names contain information used to locate objects on remote computers; and (4) naming systems are implemented by name servers.

Specifically, Watson [WATS81b] describes naming with a general model in which the *identifier* for an object is mapped into a lower level identifier for the object, as illustrated in Figure 1. Subsequent mappings eventually yield the object itself. Watson also describes a specific naming scheme that involves both high-level human-readable identifiers, and low-level identifiers that specify objects' locations.

$$\begin{tikzcd}
\textbf{Identifier}_{level\ n} \arrow[d] \\
\textbf{Identifier}_{level\ n-1} \arrow[d, dotted, no head] \\
\textbf{Identifier}_{level\ 1} \arrow[d] \\
\textbf{Object}
\end{tikzcd}$$

Figure 1
Multi-Level Mapping of Identifiers to Objects

In another model, Shoch [SHOC78] makes a distinction between the nature of different levels by suggesting that

> The *name* of a resource indicates **what** we seek,
> an *address* indicates **where** it is,
> and a *route* tells us **how to get there**.

For example, in a *DOS*, the named resource might be a file, the address is the current location of the file in the distributed system, and the route is the sequence of network connections through which a request must travel to reach the file.

Oppen and Dalal [OPPE81] construct a naming model that describes how *clients* refer to *objects*. They represent each client and object in the distributed system by exactly one vertex in a directed graph. If vertex u has an outgoing edge labelled i, then $u[i]$ denotes the vertex at the end of the edge. In addition, if $u[i_1] \ldots [i_k] = v$ then $i_1 > \ldots > i_k$ denotes a *path* from u to v. The model characterizes the nature of various types of names. For example, a graph consisting of labelled vertices but no edges models *absolute* names, where the unique label of each vertex corresponds to the object's name. Similarly, a graph with unlabelled vertices but labelled edges represents *relative* names. The distinguishing name of an object is then given by a sequence of edge labels. Furthermore, if the graph is partitioned into subgraphs, an object is named by a combination of the subgraph name and the vertex label, and *hierarchical* names result.

Based on their abstract model, Oppen and Dalal describe the Clearinghouse name server that implements a universal name resolution service. The Clearinghouse provides a coherent naming system that supports uniform and transparent access to all objects, independent of the object's type. It resolves names for objects into information that the client uses to access the object (i.e. the address of the object's manager).

A significant characteristic of the Clearinghouse, is that although the name server database is distributed, the name server is logically centralized. Cheriton and Mann [CHER84], describe an alternative approach to naming in a distributed system, in which the name server service is logically distributed across the servers for a given object. In their model, the name of an object is resolvable by the server that implements the object. The approach uses a centralized name server only when absolutely necessary to locate the server that completes the name resolution. Terry [TERR84] presents a general model for analyzing the name server mechanism for a given naming convention.

Finally, in work on the *IFIP* directory service for computer mail, Sirbu and Sutherland [SIRB84] define two methods of accessing directories (i.e. name servers). In the first approach, called *direct access*, a client that wishes to translate a name is responsible for directly accessing all necessary directories; if one directory is unable to completely resolve the name, the client must contact additional directories. In contrast, if the directory assumes the burden of accessing additional directories for the client, then the system supports *indirect access*.

In addition, Sirbu and Sutherland discuss the relationship between resolving the name of a message recipient, and delivering the message. In the *absolute* approach, the recipient's name is completely resolved into the address of a mailbox, and the message is then delivered to that mailbox, while in the *incremental* method, the message is moved towards the destination in conjunction with name resolution.

3. FRAMEWORK FOR DISCUSSION

This section establishes an intuitive understanding of the fundamental components of a distributed system, and formally defines the terminology needed in later sections. It also presents a notational foundation for naming in distributed systems, and describes an example distributed system for computer mail.

3.1 Fundamental Components of a Distributed System

We conceptually view the underlying distributed system in terms of an *object model* [JONE78], in which the system is said to consist of a collection of *objects*, denoted O. An object is either a physical resource (e.g. a disk or a processor), or an abstract resource (e.g. a file or a process). Objects are further characterized as being either *passive* or *active*, where passive objects correspond to stored data, and active objects correspond to processes that act upon passive resources. For the purpose of our discussion, we use the term object to denote only passive objects, and we treat the behavior of processes separately. The objects in a distributed system are partitioned into *types*, Associated with each object type is a *manager* that implements the object type, and presents *clients* throughout the distributed system with an interface to the object. The interface is defined by the set of *operations* that may be applied to the object.

Because clients and managers that invoke and implement operations are physically implemented in terms of a set of cooperating processes, they can be described by models of distributed processing and concurrent programming (e.g. *remote procedure calls (RPC)*, and *inter-process communication (IPC)* [BIRR83, WATS81a]). However, because we are not concerned with process synchronization and data protection in this chapter, we do not limit our view of distributed systems to either the *RPC* or the *IPC* approach. Instead, we think of all the processing involved in carrying out an operation on behalf of a client as a single *computation* that encompasses both the work involved in actually performing the operation, as well as the overhead incurred by the system in locating the manager and identifying the object. Furthermore, we view all of the processors that comprise a distributed system as forming a single virtual processor, and we think of a computation as being implemented in terms of a single process, called an *operation–process*, that runs on the virtual processor. We now consider the distributed nature of objects in the distributed system.

First, we view a distributed system as being composed of a finite set of *environments*, denoted E, where an environment defines the subset of objects currently available or accessible to a computation (i.e. O is partitioned across E). Although no inherent relationship exists between environments and physical processors, a single physical processor and the locally implemented objects could be represented by a single environment. If we view environments as corresponding to *address spaces*, then several environments would be associated with a single processor. An operation-process executes in a single environment at any given time. In other words, we view a computation as being defined by an operation-process and the set of objects accessible to that operation-process, where we let $\epsilon \in E$ denote the environment in which a given operation-process currently executes. Also, we ignore the possibility of operation-processes interfering with each other.

Second, the environments in the system are connected by a set of directed *links*, denoted L, such that an operation-process is allowed to move from its current environment to a new environment, if and only if a link leads from the current environment to the new environment. We visualize the relationship between environments and links with a directed graph, denoted $D = (E, L)$, where the vertices of D correspond to the environments in the system, and the edges represent links tha connect environments. Because links do not necessarily exist between all environments in the distributed system, we also define a *path* from one environment to another to be a sequence of one or more links passing through zero or more intermediate environments. Figure 2 schematically depicts a distributed system, where the vertex containing ϵ marks the current environment of some computation.

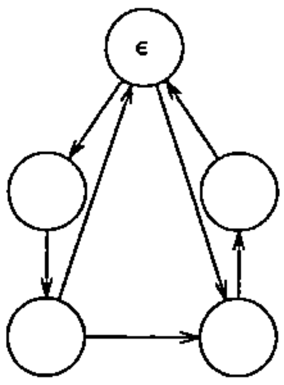

Figure 2
Schematic View of a Distributed System

Thus, instead of viewing a distributed system in terms of the physically distinct processors, we think of a distributed system as containing a set of environments and a single virtual processor, and we view the computation encompassing an operation as a single process that moves from environment to environment over links, thereby gaining access needed objects. When a client invokes an operation on an object, it in effect initiates an operation-process that executes in some starting environment, while the manager completes the operation-process by performing the operation on the object in a possibly remote environment.

3.2 Naming

Now consider how clients reference objects. Informally, an object is identified with a *name*, where a name is a *string* composed of a set of *symbols* chosen from a finite *alphabet*. Formally, we define the general form of names used to identify objects with a *language* over the alphabet Σ, denoted N, which we express in terms of the following regular expression

$$R = \Delta^* A^+ (\Delta^+ A^+)^* \Delta^*$$

such that A and Δ are disjoint subsets of Σ. The set of strings in A^+ are called *simple names*, and the set of strings in Δ^+ are called *delimiters*, where we consider simple names and delimiters to be *tokens*. In addition, a name that contains more than one simple name is said to be a *compound name*. (Note that a name may be preceded or followed by a delimiter.) Thus, a name in N is a sequence of one or more simple names separated by delimiters.

For example, the compound name *host*1!*user*@*host*2 consists of the five tokens "*host*1", "!", "*user*", "@", and "*host*2", where "*user*" is the second simple name component. Throughout the paper, we let ν represent an arbitrary name in N, and we let ν_s denote a simple name in A^+. In addition, we represent the empty string as λ.

We think of N as being partitioned into three sets: a set of names that are assumed to be *resolvable*, a set of *unresolved* names, and a set of *unresolvable* names. We assume that the set of resolvable names, denoted $\Omega \subseteq N$, do not require any further interpretation, and that the manager for each object type is able access the appropriate object given its resolvable name. Thus, if we let $\omega \in \Omega$ denote a resolvable name for object $o \in O$, such that o is implemented in environment $e \in E$, then we say that ω is resolvable and understood in environment e. Furthermore, we say that a client identifies an object with an arbitrary name in N.

Finally, the distributed system provides a *name resolution mechanism* that translates the arbitrary name specified by the client into a resolvable name that the manager understands. The mechanism also moves the operation-process from the initial environment to the environment in which the object is defined. For example, if a client identifies object o with an unresolved name ν while executing in environment e_{client}, such that o is implemented in $e_{manager}$ and ω is a resolvable name for o in $e_{manager}$, then the name resolution mechanism translates ν into ω and moves the operation-process from e_{client} to $e_{manager}$.

We discuss the translation of arbitrary names into resolvable names in Section 4, while we describe the movement of operation-processes from an initial environment to the environment that implements the object in Section 5.

3.3 Example

From the user's perspective, a distributed system provides a set of *services* that are implemented in terms of the objects defined in the system. That is, a service is defined by a *user interface* that provides users with a set of *commands*. Each command is in turn implemented by a set of operations that are performed on objects in the system. Thus, when the user invokes a command, the user interface becomes a client of some object manager in the system.

We now consider the computer mail service that provides users with the ability to exchange and archive memos. (We use this example throughout this paper to demonstrate our model.) Specifically, the service supports commands to *view*, *send*, *save*, and *delete* messages, where the commands are implemented in terms of two principle objects: *mailboxes* and *files*.

The mailbox manager supports *append* and *retrieve* operations. Because mailboxes are implemented by file objects (where the file manager supports *read* and *write* operations), and a file system is associated with each computing system, we think of each computer in a mail system as corresponding to an environment. That is, each environment contains objects corresponding to the files in a computer's file system, as well as the mailboxes implemented in that file system. In addition, the network connections in the global mail system corresponds to links.

Finally, the work carried out by the hierarchy of mailers described in Section 1 corresponds roughly to the single operation-process associated with the append operation (we call it the *delivery-process*). That is, when the user instructs the user interface to send a message to a recipient, the interface becomes a client running in the sender's environment. The mail system's name resolution mechanism then resolves the mailbox name of the recipient, and causes the delivery-process to move toward the environment in which the mailbox manager implements the recipient's mailbox.

4. NAME RESOLUTION MODEL

In this and the next section, we present a model for name resolution that extend Saltzer's model of names in operating systems [SALT78] and computer networks [SALT82]; we focus on the resolution of arbitrary names for objects into resolvable names for objects in this section, and we discuss the movement of operation-processes in the next section.

We define Φ to be a partial function that performs the mapping

$$\Phi: \mathrm{N} \longrightarrow \Omega$$

Our model does not distinguish between "names" and "addresses" as defined by Shoch. Instead, it treats various ways of identifying an object as different *forms* of names, and it describes the bindings between forms in a hierarchy. At each level in the hierarchy, names are defined by a uniform set of rules; we assume that there exists a single name resolution function to interpret a name, given a representation of the name and the set of rules for a level. In other words, we view names as purely syntactic entities, and we think of name resolution as a syntax-driven operation.

Informally, Φ is defined by three components: a finite set of *contexts*, each of which defines an interpretation of a set of names, a universal function that *resolves* names relative to an interpretation, and a mechanism that defines the *closure* of a name and a context. Because we view all names as being resolved relative to a given context, we say that a context and a name pair form a *qualified name*. To translate an unresolved name into a resolvable name, Φ first forms a qualified name from the given name and a selected context, where the closure mechanism selects the initial context based upon the current *state* of the operation-process that invoked Φ. The universal resolution function then translates the initial qualified name into a resolvable name by successively interpretting a name according to a context, where a context interprets a name to be either another qualified name or a resolvable name, (i.e. given an arbitrary name as an argument, a context either returns a new name and a new context to interpret that name, or it returns a resolvable name).

Formally, let $C = \{c_i | i > 0\}$ denote the set of contexts available in a distributed system, such that each context in C is a partial function that performs the mapping

$$c_i: \mathrm{N} \longrightarrow \Psi$$

where Ψ represents the set of all qualified names, and each $\psi \in \Psi$ is a pair $<c_i, \nu>$ for $c_i \in C$ and $\nu \in \mathrm{N}$. (We allow the context component to be undefined, denoted $\perp$.) Furthermore, let f_R denote the universal resolution function that performs the mapping

$$f_R: \Psi \longrightarrow \Omega$$

and let $S = \{s_i \mid i > 0\}$ denote the finite set of possible states that may be assumed by an operation-process, such that the closure mechanism is defined by the function

$$f_C\colon S \longrightarrow C$$

Finally, define the name resolution mechanism Φ as follows

$$\Phi(\nu) = \begin{cases} \nu & \text{if } \nu \in \Omega \\ f_R(<f_C(\sigma), \nu>) & \textit{otherwise} \end{cases}$$

where $\sigma \in S$ represents the current state of the operation-process that invoked Φ.

Although we think of Φ as representing the name resolution mechanism for a given distributed system, f_R is a general function that resolves names in any distributed system; it is the set of contexts and the closure function that uniquely define names in a particular system. We say that C and f_C define the *naming system* for a distributed system.

The rest of this section describes the three components of a distributed system's name resolution mechanism in detail: Section 4.1 defines contexts, Section 4.2 discusses the closure function, and Section 4.3 describes the resolution function.

4.1 Contexts

A context defines an interpretation of a set of names, where by "interpretation" we mean a mapping of arbitrary names into qualified names. We think of the mapping performed by a context in two stages: a *translating* function selects a simple name component from a given name, and a table of *bindings* define a mapping of simple names to qualified names.

First, let f_T^i denote a translating function associated with context c_i. It performs the mapping

$$f_T^i\colon \nu \longrightarrow <\nu_s, \nu'>$$

for $\nu, \nu' \in N$ and $\nu_s \in A^+$. That is, the function takes a name in N as an argument, and returns a pair of names: the first is a simple name and the second is an arbitrary name.

Recall that a name is a syntactic entity described by a language N that we represent with the regular expression R over the alphabet Σ. The strings defined by R contain a sequence of simple names separated by delimiters. We intuitively think of the translating function as building a *parse tree* out of the simple name and delimiter tokens that comprise a given name [HOPC79]. It then extracts a simple name from the tree, and returns a pair consisting of the selected simple name and the name corresponding to the rest of the tree. For example, the translating function might return the pair $<$*host*1, *host*2!*user* $>$ when applied to the name *host*1!*host*2!*user*. (It is also possible that f_T^i returns a simple name that is not a component of the name it was given as input, in which case the second name in the returned pair is the original name.) Thus, while the language N defines the sequence of simple name and delimiter tokens that comprise names throughout the entire naming system, the tokens are parsed on a context by context basis, according to the precedence assigned to delimiters by each context's translating function (i.e. the delimiters behave like operators).

Second, let $\beta = <\nu_s, \psi>$ denote a single binding of simple name $\nu_s \in A^+$ to qualified name $\psi \in \Psi$. A finite set of bindings are associated with context c_i, where for convenience we denote the set of bindings as c_i, and let the term "context" refer to a table of bindings. Figure 3 schematically depicts the table of bindings associated with a context.

ν_s	c_j	γ

Figure 3
Context Viewed as a Table

To map an arbitrary name into a qualified name, c_i looks up the simple name returned by its translating function in the table of bindings, and returns the corresponding qualified name. Not all the bindings in the table, however, contain fully specified qualified names; therefore, the context may augment the qualified name before returning it. Specifically, there exist two types of bindings: *complete* bindings in which the qualified name contains both a name and a context, and *partial* bindings in which the qualified name contains only a context, and the name component is assumed to be the original name minus the simple name just interpretted.

Formally, we define the mapping performed by each c_i with the following definition.

Definition: For context $c_i \in C$ applied to the arbitrary name ν, suppose $f_T^i(\nu)$ returns the pair $<\nu_s, \nu'>$, and $\beta = <\nu_s, \psi>$ is a binding in c_i.

(i) If $\psi = <c_j, \gamma>$ for $c_j \in C$ and $\gamma \in N$, then β is a *complete* binding and c_i returns $<c_j, \gamma>$.

(ii) If $\psi = <c_j, \lambda>$ for $c_j \in C$, then β is a *partial* binding and c_i returns $<c_j, \nu'>$. □

If the simple name is not bound in the context, then the mapping is undefined. Also, we assume that contexts contain only valid bindings as described in the definition. In addition, a *final* binding is a special case of a complete binding, in which the context component of the qualified name is undefined; final bindings signify that the name component of the qualified name is a resolvable name (i.e. $\psi = <\bot, \omega>$).

We conceptually view the relationship among the contexts that comprise a naming system with a directed graph, denoted K, called a *naming network*. The set of vertices in K correspond to the union of the set of contexts C and the set of resolvable names Ω. The set of edges in K correspond to the bindings contained in the contexts of C, such that (c_i, c_j) is an edge labelled "$\nu_s \Rightarrow \gamma$" if and only if context c_i contains the complete binding $<\nu_s, <c_j, \gamma>>$, (c_i, c_j) is an edge labelled "ν_s" if and only if context c_i contains the partial binding $<\nu_s, <c_j, \lambda>>$, and (c_i, ω) is an edge labelled "ν_s" if and only if context c_i contains the final binding $<\nu_s, <\bot, \omega>>$ for resolvable name $\omega \in \Omega$.

For example, consider the naming system for a distributed system similar to the DARPA Internet that supports names for mailboxes. Let $O = \{o_{jxd}, o_{jxs}\}$ denote the set of mailbox objects, and let $\Omega = \{\omega_{jxd}, \omega_{jxs}\}$ represent the set of resolvable names for mailboxes. Figure 4 depicts the naming network that supports names of the form *jxd@nlabs*, while Figure 5 illustrates the naming network that supports three forms of names for mailboxes: high-level names of the form *Doe.Research.NLabs*, unique identifiers of the form *UID–Doe*, and names of the form *jxd@nlabs*. We distinguish between vertices that correspond to contexts and vertices that correspond to resolvable names by representing context vertices with ellipses and object vertices with boxes.

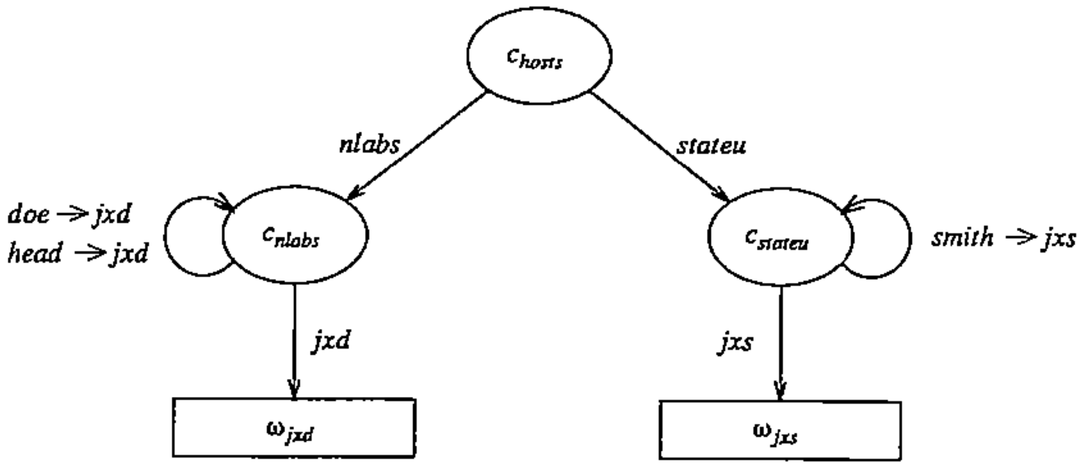

Figure 4
Naming Network for Names of the Form *user@host*

4.2 Closure Function

The closure function f_C maps the state of an operation-process into a context in C. Let U denote the set of all users in the distributed system. Each state $s_i \in S$ is defined by the pair $<e_k, u_x>$, where $e_k \in E$ is the current environment of the operation-process, and $u_x \in U$ is the user on who's behalf the operation-process is executing (i.e. the user that invoked the client). We discuss the mapping performed by f_C in detail in Section 6.3; for now, we think of the function as being implemented by a table of state/context pairs, where each user defines his or her own entry in the table.

4.3 Resolution Function

The universal resolution function f_R resolves a qualified name into a resolvable name by successively applying a context to a name. Formally stated,

$$f_R(<c_i, v>) = \begin{cases} f_R(c_i(v)) & \text{if } f_1(c_i(v)) \in C \\ f_2(c_i(v)) & \text{if } f_1(c_i(v)) = \bot \\ \bot & \textit{otherwise} \end{cases}$$

where f_i represents a *selector* function that returns the i^{th} component of a given tuple (for integer i).

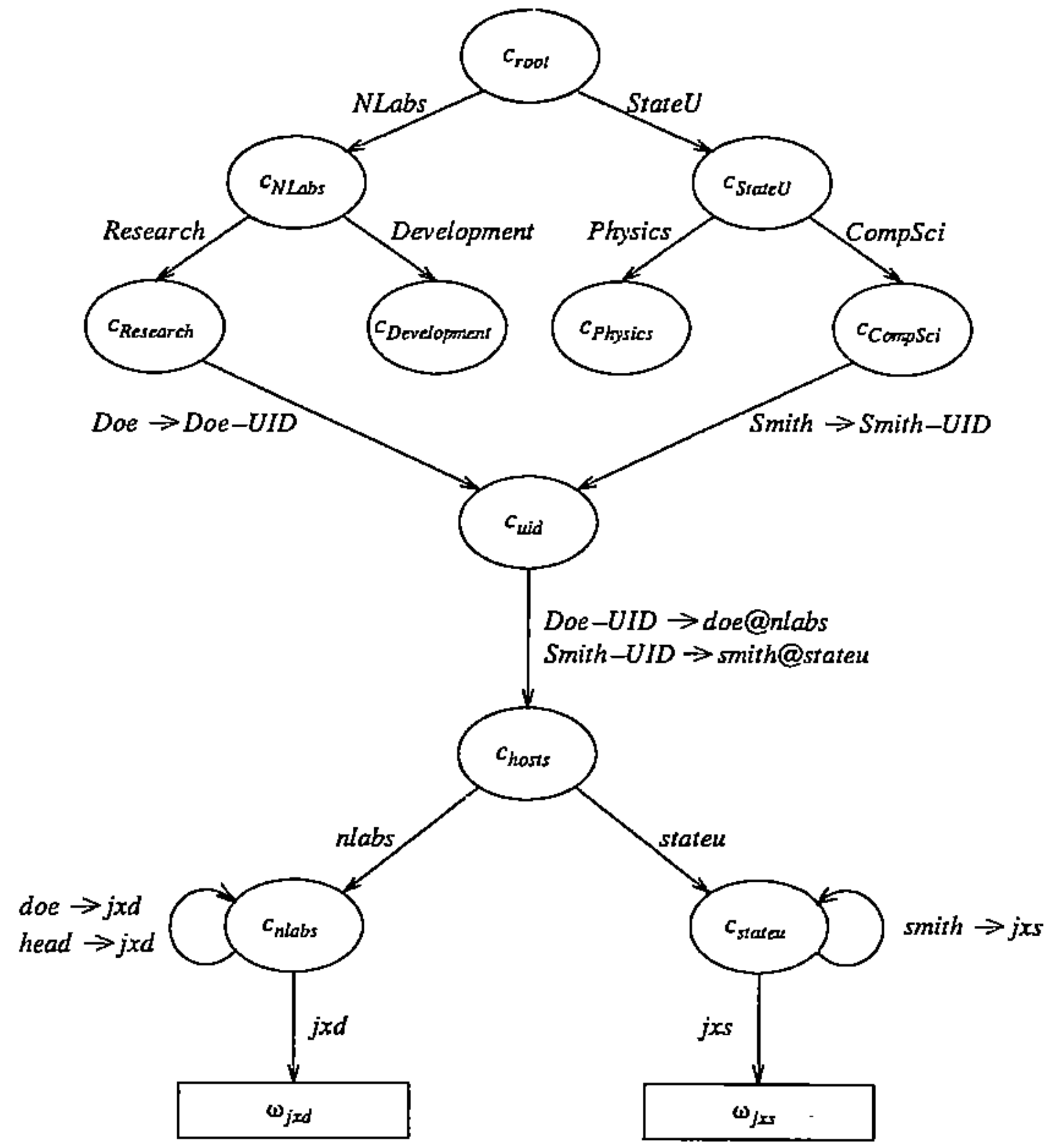

Figure 5
Naming Network for Naming System with Multiple Forms of Names

From an operational perspective, f_R translates one qualified name into another, based on the interpretation of the name in the specified context, until it produces a resolvable name, where a qualified name with an undefined context component signifies a resolvable name. In other words, we think of the function as traversing the naming network, based upon matches between the simple name components of the name being resolved and the edge labels, until a vertex labelled by a resolvable name is reached.

To fully understand the process of name resolution, it is helpful to trace the operation of Φ with a state transition diagram. Let the qualified name $\psi = <c_i, \nu>$ represent a state in the operation of Φ, and let each individual step in the operation of Φ correspond to the transition of state ψ to state ψ' as defined by a single application of f_R, denoted

$$\psi \xrightarrow[f_R]{} \psi'$$

Also, let $\psi_f = <\perp, \omega>$ for $\omega \in \Omega$ denote a final state in the execution of Φ, and let $\psi_0 = <f_C(\sigma), \nu>$ represent the starting state of Φ.

Thus, if $\psi_0 = <c_{root}, Doe.Research.NLabs>$, then Figure 6 depicts the resolution of the arbitrary name *Doe.Research.NLabs* into resolvable name ω_{jxd} in the naming system defined in Figure 5.

$$\begin{tikzcd}
< c_{root}, Doe.Research.NLabs > \arrow[d] \\
< c_{NLabs}, Doe.Research > \arrow[d] \\
< c_{Research}, Doe > \arrow[d] \\
< c_{uid}, Doe\text{–}UID > \arrow[d] \\
< c_{hosts}, doe@nlabs > \arrow[d] \\
< c_{nlabs}, doe > \arrow[d] \\
< c_{nlabs}, jxd > \arrow[d] \\
< \perp, \omega_{jxd} >
\end{tikzcd}$$

Figure 6
State Transition View of Name Resolution

5. MOVING THROUGH THE DISTRIBUTED SYSTEM

This section completes our model of a distributed system's name resolution mechanism by defining Φ in a system that contains more than one environment, where in addition to mapping arbitrary names that identify objects into resolvable names for objects, Φ has a *side–effect* of moving an operation-process from a initial environment to an environment that defines the object. That is, for an operation-process that references object $o \in O$ in environment e_{client} with the name ν, Φ maps ν into ω, and moves the operation-process from e_{client} to $e_{manager}$, where $o \in e_{manager}$ and ω is a resolvable name in $e_{manager}$.

A distributed system similar to the DARPA Internet illustrates our model. (Recall that the directed graph $D = (E, L)$ models a distributed system.) Let each environment in E represent a host in the system, and let each link in L correspond to a network connection between hosts, where $l_{Addr-nlabs}$ denotes a link leading to environment e_{nlabs} (corresponding to host *nlabs*). Furthermore, assume that a network link connects all hosts in the system (i.e. D forms a completely connected graph), corresponding to the situation in the DARPA Internet where hosts are logically connected by the TCP/IP protocols [ISI81a, ISI81b]). Finally, because files implement mailbox objects, let $\Omega = \{/usr/mail/jxd,\ /usr/mail/jxd\}$ represent the set of resolvable names for mailbox objects. Figure 7 schematically depicts the example distributed system D.

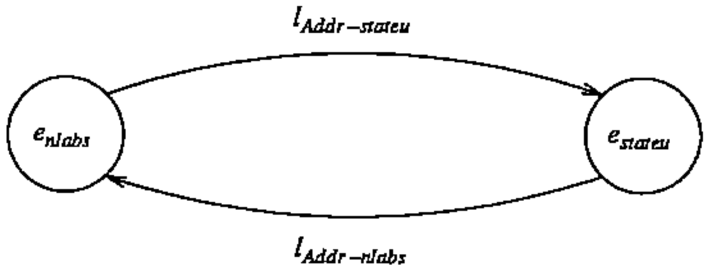

Figure 7
DARPA Internet Represented by Environments and Links

Intuitively, we not only view the interpretation of a name defined by a context as a mapping of arbitrary names to qualified names, but we also think of the interpretation as specifying how operation-processes are to move through the environments in the distributed system. Formally, let $\beta = <\nu_s, \psi, l_j>$ for $l_j \in L$, represent an extended binding in the contexts of C, where simple name ν_s is said to be bound to the qualified name ψ and the link l_j. (Let the triple $\beta = <\nu_s, \tau, \perp>$ denote a binding that does not specify any movement through the system.) Figure 8 illustrates the addition of links to the naming system originally depicted in Figure 4, by augmenting the edge labels of the graph with links.

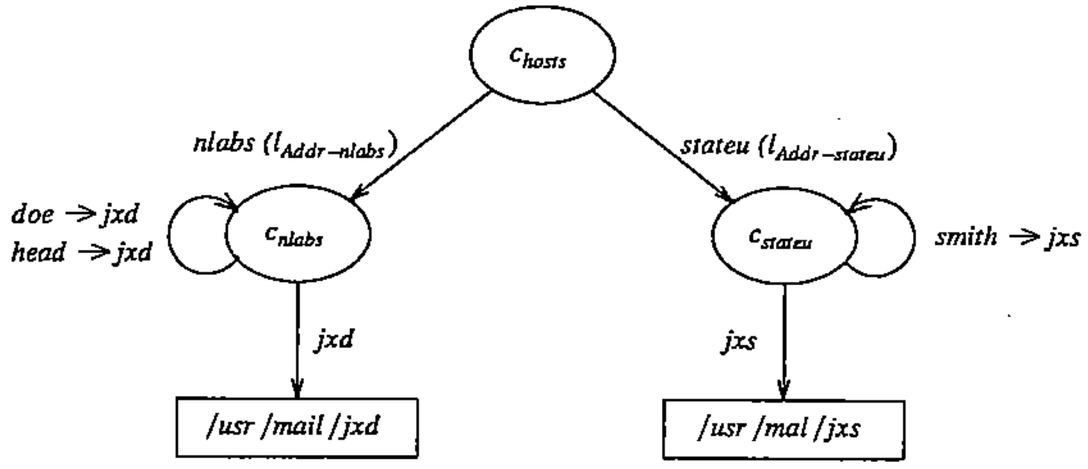

Figure 8
Naming Network with Links

Furthermore, let Φ_M represent a system mechanism that *moves* an operation-process from one environment to another over a specified link. That is, if $\epsilon \in E$ denotes the current environment of an operation-process, and the operation-process applies Φ_M to link l_j, then the operation-process is moved from ϵ to ϵ' (i.e. the state of the operation-process changes from $\sigma = <\epsilon, u_x>$ to $\sigma' = <\epsilon', u_x>$), where l_j is a link that leads from ϵ to ϵ'.

Φ employs Φ_M to move an operation-process from its current environment to a new environment based upon the links encountered in the bindings of contexts.[†] Because we think of the movement of an operation-process as a side-effect of the name resolution mechanism Φ, we do not alter the definition of the mapping performed by each context. Instead, we operationally define the affect of Φ on the environment in which an operation-process executes in Algorithm 1, where *lookup* and *type* are primitive operations that return a pair containing the qualified name and link bound to a simple name, and the type of the binding associated with a simple name, respectively.

```
$c_i(v)$  =  begin
                 $<v_s, v'> = f_T^i(v)$
                 $<\psi, l> = lookup(v_s)$
                 if $(l \neq \perp)$
                     $\Phi_M(l)$
                 if $(type(v_s) = complete)$
                     return($\psi$)
                 if $(type(v_s) = partial)$
                     return($<f_1(\psi), v'>$)
                 else
                     return($\perp$)
             end
```

Algorithm 1
Operational Description of Context c_i

Note that it is not our objective to fully specify the operation of Φ_M, which is an abstraction of the transport mechanism of a distributed system's underlying network. Instead, we emphasize that the name resolution mechanism and the transport mechanism are intertwined, and that in fact, the name resolution mechanism drives the transport mechanism based on the name it is resolving.

6. CONCEPTUAL VIEW OF NAME RESOLUTION

The previous two sections define a formal model for name resolution. This section discusses name resolution from an intuitive perspective, and formally defines familiar (but informally described), concepts and terminology. Specifically, Sections 6.1 and 6.2 consider the behavior of groups of contexts, while Section 6.3 investigates the closure function. Finally, Section 6.4 discusses the bindings of simple names to qualified names.

[†]Because an operation-process encompasses Φ, its current environment is affected when Φ employs Φ_M.

6.1 Name Spaces

As mentioned in Section 4, we think of the different forms of names that exist to identify objects as corresponding to multiple levels of a hierarchy, where each level defines a set of rules that specify how to resolve a set of names into a form defined by another level; a view of name resolution similar to the one schematically depicted in Figure 1.

Although the model presented in Section 4 describes the resolution of names on a context by context basis, we understand that there exists a fundamental relationship between groups of contexts that treat names in a uniform manner; we say that such a collection of contexts form a *name space*. Formally stated,

Definition: A *name space* is a partition of C, denoted C_n, such that

(i) The translating functions for each context in C_n are functionally equivalent, and

(ii) The contexts in C_n are connected by partial bindings. □

We graphically depict name space C_n with a subgraph of K, denoted K_n, that includes vertices corresponding to the contexts in C_n, and edges that correspond to the partial bindings between contexts in C_n.[†] (Subgraph K_n is called the naming network for name space C_n.) For example, Figure 9 illustrates the naming network depicted in Figure 5 partitioned into three separate name spaces; the uppermost name space, denoted C_H, defines high-level names, the middle name space, denoted C_U, defines unique-identifiers, and the bottom name space, denoted C_D, defines names that correspond to mailbox names in the DARPA Internet.

Partitioning the naming system into name spaces allows us to say that there exists a single abstract translating function associated with an entire name space, where the translating function may always select the *left–most* simple name (as in name space C_H), or the *right–most* simple name (as with *GRAPEVINE* and DARPA Internet domain names). Also, instead of a single language N that defines the form of names throughout the entire naming system, we say that there exists a language that defines the form of names for each name space. Thus, the set of delimiters may be different from one name space to another, and the language for each name space may enforce additional restrictions on the placement of delimiters in a compound name.

Furthermore, the names defined by a given name space can be characterized by the structure of the naming network that abstractly represents the name space. Consider for example, three forms of names used throughout this paper: unique-identifiers such as those defined by name space C_U, DARPA domain names of the form *jxd.nlabs.anycity.anystate.usa*, and *UUCP* names like *host*1!*host*2!*host*3*user*. Unique-identifiers are simple names, while the latter two forms are compound names. In addition, there is a hierarchical relationship between the component simple names in the domain names (i.e. *anycity* is located in *anystate* and *anystate* is located in the *usa*), while the components of the *UUCP* name are understood to be non-hierarchical. (In fact, they are adjacent to each other when viewed as hosts in the underlying distributed system.) Formally, we make the following three definitions.

Definition: Name space C_n for which naming network K_n contains a single vertex is said to be a *flat* name space. Names defined in a flat name space are called *unique–identifiers*. □

Definition: Name space C_n for which naming network K_n forms a directed acyclic graph with

[†]Our definition does not preclude the possibility that complete bindings may exist within a given name space. Such intra-name space bindings are a special case of complete bindings called *indirect* or *recursive* bindings, that form a cycle in the corresponding subgraph of the naming network.

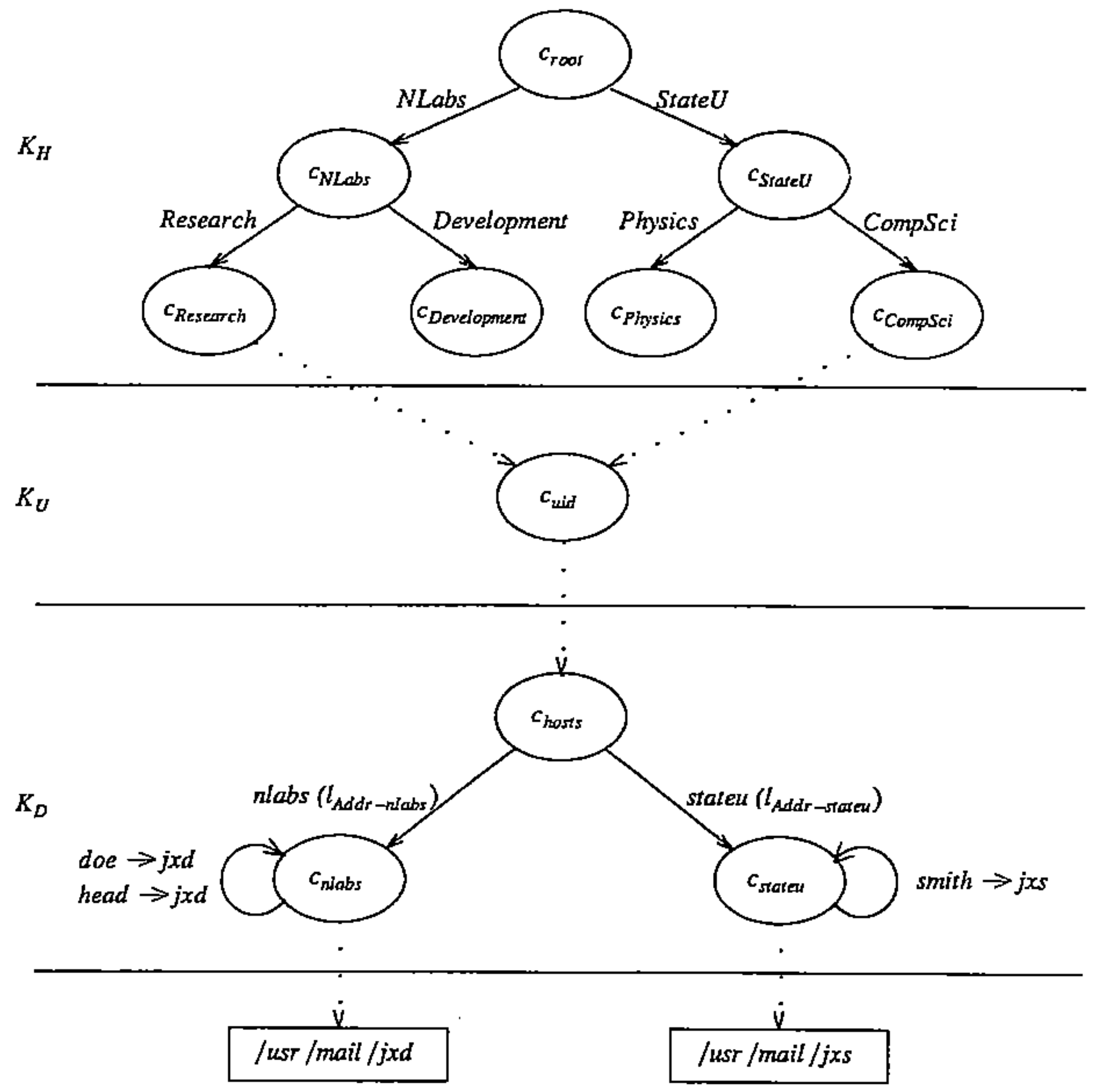

Figure 9
Naming Network Partitioned into Name Spaces

more than one node is said to be a *hierarchical* name space. Names defined in a hierarchical name space are called *hierarchical* names. □

Definition: Names defined in name space C_n for which there exists a subgraph of K_n, denoted $\hat{K}_n$, such that $\hat{K}_n$ and D are isomorphic, (where $D = (E, L)$ is the directed graph representing the distributed system), are called *source-route* names. □

Thus, domain names are hierarchical because they are described by an acyclic naming network, while names defined in name spaces C_H and C_D, as well as GRAPEVINE names of the form $F.R$, are a special case of hierarchical names that are modelled by a graph that forms a tree. In the UUCP name space, however, there is an intuitive relationship between the naming network that defines a set of names, and the underlying distributed system; when the user must explicitly specify a path from the starting environment to the final environment in a name, then the corresponding naming network must mirror the structure of the system to correctly move the operation-process from environment to environment. Figure 10 schematically depicts the name space for the UUCP-like distributed system illustrated in Figure 11.

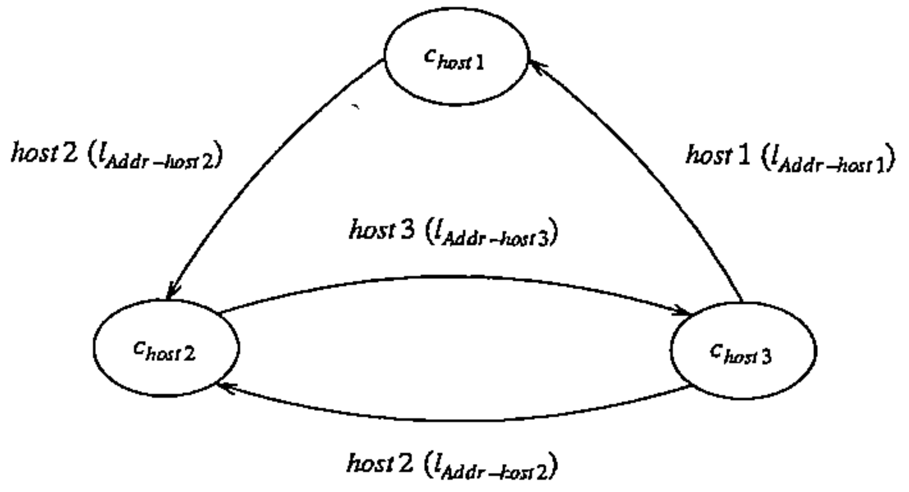
c_{host1}
$host2\ (l_{Addr-host2})$
$host1\ (l_{Addr-host1})$
$host3\ (l_{Addr-host3})$
c_{host2}
c_{host3}
$host2\ (l_{Addr-host2})$

Figure 10
Naming Network for Names in a UUCP-like System

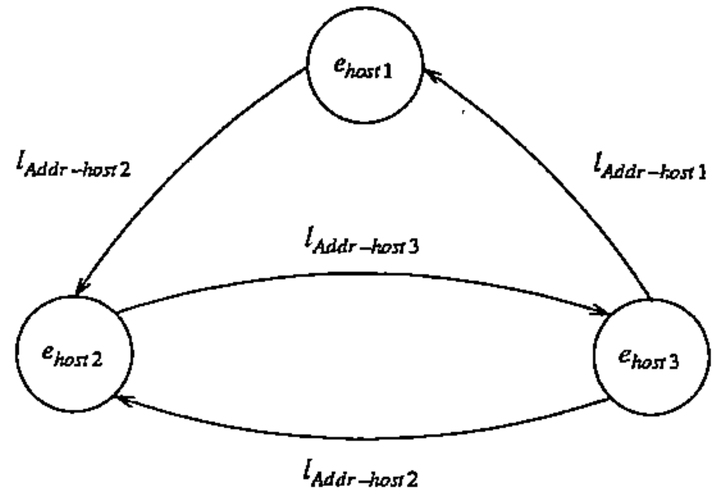

Figure 11
Underlying UUCP-like Distributed System

Finally, consider the relationship between the names defined in a name space, and the presence of links in the name space. Because names ultimately denote objects, name spaces that include links not only distinguish between objects, but they also specify where the object is located in the distributed system. Name spaces that do not contain links, however, do not aid in locating an object; they exist for the convenience of the client that must identify the object. Thus,

Definition: Names defined in name space C_n such that for all $c_i \in C_n$, the constituent bindings are of the form $<v_s, \psi, \perp>$, are called *logical* names. □

Definition: Name space C_n for which there exist a context $c_i \in C_n$ that contains bindings of the form $<v_s, \psi, l_j>$ for $l_j \in L$, is called a *physical* name space. Names defined in a routing name space that have simple name components bound to links are known as *addresses*. □

For example, DARPA Internet names of the form *user@host* direct the operation-process to the correct host, and thus constitute addresses, while names defined by C_H and C_U are logical. Resolving an address moves the operation-process closer to the destination environment, similar to the idea of incremental resolution described by Sirbu and Sutherland. In contrast, resolving a logical name corresponds to their notion of absolute resolution.

6.2 Levels

In addition to partitioning contexts into name spaces, we view a collection of one or more name spaces as forming a *level*, where a level is loosely defined to be a collection of contexts that define names of a given form. Consider, for example, the global mail system in which there exists names of the form *user@host*, *host*1!*host*2!*user*, and *host*2!*user@host*1. The first two forms are defined by name spaces that correspond to the DARPA Internet (depicted by K_D in Figure 9), and the UUCP network (depicted in Figure 10), respectively. The last form, however, is defined by a combination of the two name spaces. Formally,

Definition: A *level* is a collection of contexts connected by partial bindings that do not necessarily have equivalent translating functions. □

Because "mailbox addresses" described in Section 1 constitute a form of names, we say that there exists a level in the naming system composed of contexts drawn from the set of name spaces that define names for each network mailing system in the global system, as well as contexts that roughly correspond to mail bridges (i.e. they contain bindings that connect them to contexts in separate name spaces). Figure 12 schematically depicts the level that corresponds to the global mail system, where each context is collapsed into a point, and a dotted ellipse represents the collection of contexts that comprise a name space.

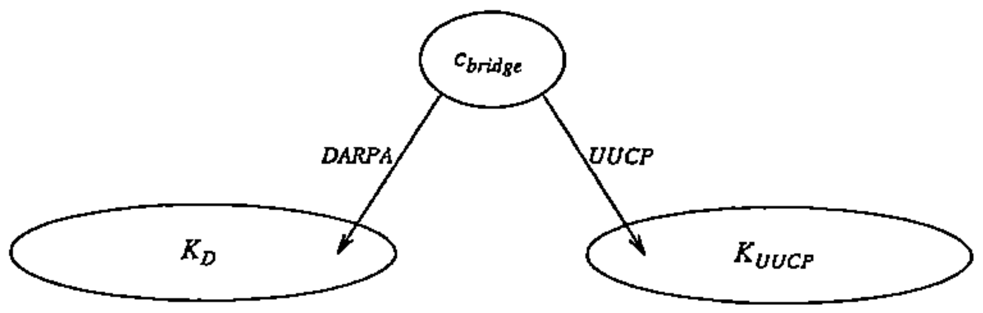

Figure 12
Level Composed of Multiple Name Spaces

Thus, we abstractly view the overall process of name resolution as being partitioned into two orthogonal dimensions. In one dimension, names of one form (as defined by one level), are translated into names of another form (as defined by another level), while in the second dimension, names of a given form are interpretted within a single level. That is, we visualize the the subgraphs corresponding to each level in the naming network as being in separate planes, where the edges between contexts in different levels connect contexts in adjacent planes. Similarly, we think of the state transition representation of the operation of Φ as two dimensional: involving transitions between states within a given level, and transitions between states in different levels.

6.3 Closure Function

As defined in Section 4, the closure function selects an initial context based on the state of an operation-process, where the state consists of the current environment and the user on who's behalf the operation-process is executing. We now investigate the effect of the closure function f_C upon names defined by a naming system.

First, because the bindings in a naming system are well defined, and the behavior of Φ is deterministic, if names are always resolved starting at the same context, then a given name must always resolve into the same resolvable name. If, however, resolution begins at different contexts, then different names may resolve into the same resolvable name (or a given name may resolve into different resolvable names), depending on the starting context. Formally, we make the following two definitions, in which we view the naming system as begin partitioned into name spaces, and we focus on name spaces that Φ enters via the closure function rather than by a complete binding from contexts in other name spaces.

Definition: Names defined in name space C_n, such that $f_C(s_i) = f_C(s_j)$ for all $s_i, s_j \in S$, are called *absolute* names. □

Definition: Names defined in name space C_n, for which there exists states $s_i, s_j \in S$ such that $f_C(s_i) \neq f_C(<s_j)$, are called *relative* names. □

Because the two definitions are mutually exclusive, we think of a given name space as either supporting absolute names or relative names, but not both.

Next, we consider the mapping between states and contexts in more detail.

Definition: If there exists an environment $e_k \in E$ and a user $u_x \in U$ such that $f_C(<e_k, u_x>) = c_i$, and $f_C(<e_k, u_x>) \neq f_C(<e_l, u_y>)$ for all $u_x \neq u_y$ and all $e_l \in E$, then c_i is called a *private* context. Names defined by a private context are called *aliases*. □

Definition: A context $c_i \in C$, such that c_i is not a private context is called a *public* context. Names defined by a public context are called *public* names. □

In other words, aliases are names used by a specific user to identify objects, while public names are available to many users. Figure 13 illustrates a portion of the example naming system that includes a private context, denoted $c_{private}$, for some user in the Research division of National Labs. Note that in practice, the private contexts that define aliases are roots nodes in the naming network that are connected to the rest of the naming network by complete or final bindings. Therefore, aliases are generally simple names.

Finally, we consider an extension to the closure function that allows it to return a set of contexts at which name resolution may begin. In the following definition, let $\hat{C}$ denote the subset of C returned by f_C.

Definition: A naming system for which $f_C(\sigma) = \hat{C}$, such that $\hat{C}$ contains more than one context, is said to support *multiple-relative* names. Furthermore,

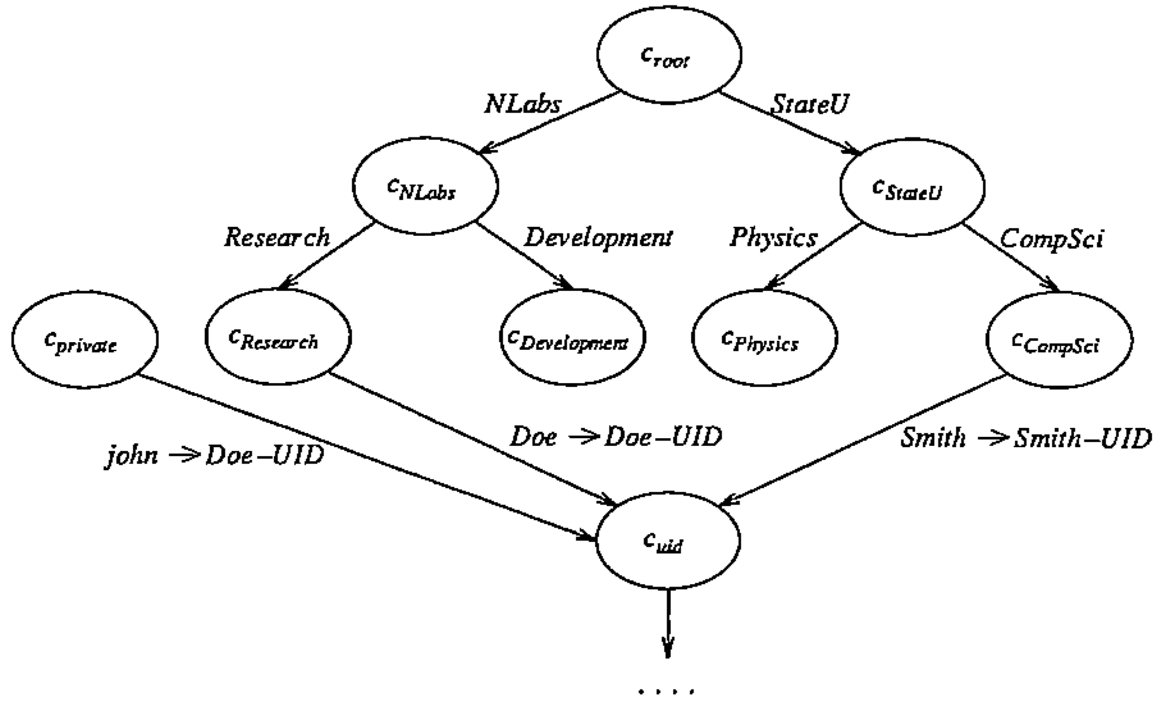

Figure 13
Naming Network with a Private Context

(i) If all the contexts in $\hat{C}$ are in the same name space (i.e. $\hat{C} \subseteq C_n$ for some name space C_n), and the name space is hierarchical, then names resolved relative to the root of the naming network are called *full* names,[†] while all other names are considered *abbreviated* names.

(ii) If the contexts in $\hat{C}$ are in two or more levels, the naming system is said to support *multiple−views*. □

For example, in the set of name spaces illustrated in Figure 9, Φ must begin the resolution of *jxd* and *postmaster* in context c_{nlabs}, but Φ must start at c_{hosts} to resolve *jxd@nlabs* and *postmaster@nlabs*. Similarly, if f_C returns the set of contexts $\{c_{root}, c_{hosts}\}$, then John Doe's mailbox can be identified with names from different levels (e.g. *Doe.Research.NLabs* and *jxd@nlabs*). Finally, if f_C returns the set of contexts $\{c_{private}, c_{Research}, c_{NLabs}, c_{root}\}$ for the naming network illustrated in Figure 13, then John Doe's mailbox can be identified as *john*, *Doe*, *Doe.Research*, or *Doe.Research.NLabs*, where the first name is an alias, the next two are abbreviated names, and the last name is a full name.

In practice, either the contexts in the set $\hat{C}$ returned by f_C are tried in order until one is found that leads to successful resolution of the name (similar to a *search path* in UNIX [RITC74]), or a selector function is associated with the naming system to pick one of the contexts, based perhaps on the syntax of the name being resolved. For example, if f_C returns the set $\{c_{hosts}, c_{nlabs}\}$ when applied to state $<e_{nlabs}, u_x>$, then the selector function would pick c_{hosts} if the name being resolved is a compound name (e.g. *jxd@nlabs*), and it would select c_{nlabs} if the name being resolved is a simple name (e.g. *jxd*).

[†]Full names are often called absolute names, but we use the term full to distinguish between a special case of multiple-relative names and absolute names.

6.4 Bindings in a Given Context

Finally, we discuss the relationship between the simple name components, and the qualified name components of the bindings contained in the contexts of a naming system. (For simplicity, assume the definition of binding pairs from Section 4.1, and ignore links.) First, we make the following definition regarding the possibility of several simple names being bound to the same qualified name.

Definition: A set of simple names, denoted $N_{syn} = \{v_{s_k} | k > 0\}$, such that for some context $c_i \in C$, there exists a binding $< v_{s_k}, \psi >$ for each $v_{s_k} \in N_{syn}$, is called a set of *synonyms* (or *nicknames*). □

In contrast, if we extend the form of bindings contained in a context to allow a simple name to be bound to a set of qualified names, where either any one of the qualified names, or all of the qualified names may be selected (denoted by separating the qualified names in the set with "|" and "&" respectively), then the following types of names result.

Definition: A name v defined in name space C_n, such that for some simple name component v_k, there exists a context $c_i \in C_n$ that contains the binding $< v_k, \{\psi_1 \mid \cdots \mid \psi_n\} >$, is called a *generic* name. □

Definition: A name v defined in name space C_n, such that for some simple name component v_k, there exists a context $c_i \in C_n$ that contains the binding $< v_k, \{\psi_1 \ \& \cdots \& \ \psi_n\} >$, is called a *group* name (or *distribution list*). □

Informally, if a simple name is bound to the set $\{\psi_1 \mid \cdots \mid \psi_n\}$, and the context returns a single qualified name ψ_i from the set, where a qualified name is selected based some factor outside the realm of name resolution (i.e. the availability or ease of access of the object it denotes), then the naming mechanism is said to support generic names. Similarly, if a simple name is bound to the set $\{\psi_1 \ \& \cdots \& \ \psi_n\}$, and the name resolution mechanism conceptually splits into n separate occurrences, each of which proceeds with one of the qualified names ψ_i from the set, then the name resolution mechanism is said to support group names. For example, name space C_H supports the group *all* by having the binding

$$< all, \{< c_{UID}, Doe\text{-}UID > \ \& \ < c_{UID}, Smith\text{-}UID >\} >$$

in context c_{root}. (Additional combinations of bindings support the same group.)

7. NAME SERVERS

This section describes name servers that implement our abstract model of name resolution. Conceptually, a name server corresponds roughly to a context; it consists of a data structure that contains a set of bindings, and a program that returns the binding associated with a given name. In the framework for distributed systems used throughout this paper, there exists a name server object type, denoted $O_{NS} = \{o^j_{NS} | j > 0\}$, upon which a *lookup* operation may be performed. (For simplicity, we refer to a name server object and its manager collectively as a "name server", and we let the term "name server object" denote the set of bindings implemented by a given name server.) Furthermore, a *name server service* provides a *resolve* command that is implemented by the set of name servers, in the same way as the computer mail service is implemented by a set of mailboxes and files.[†] Thus, we think of name servers as implementing contexts, and the name server service

[†]The object manager and the service interface also support operations/commands that establish and

as implementing the resolution function f_R.

The rest of this section discusses distinctions between the abstract components of the name resolution mechanism, and the name servers that implement those components. Specifically, Section 7.1 explains how the bindings in a naming system are distributed over a set of name server objects, and Section 7.2 describes how objects that define names are themselves named in a distributed system. Finally, Section 7.3 contains an intuitive discussion of name servers.

7.1 Distribution of Bindings Over Name Server Objects

Each name server object employed by the name server service implements some subset of the bindings defined by the contexts of a naming system, where we consider the set of bindings in the system to be distributed over the name server objects on context boundaries.† This section defines the relationship between contexts and name server objects, where a limited naming system, consisting only of the name space C_H, illustrates our discussion.

Formally, the set of contexts C is partitioned into a set of *binding-groups*, denoted $G = \{g_j | j>0\}$ (i.e. each $g_j \subseteq C$), such that name server object o_{NS}^j *implements* the bindings defined in the contexts of group g_j. In other words, the partition of contexts defines the distribution of bindings over name server objects. (We ignore the related question of how name server objects are distributed throughout the system.) We now describe three different partitionings of C.

First, each binding-group g_j may contain exactly one context c_i, such that the bindings contained in each context c_i are implemented by a single name server object o_{NS}^i, and each name server object o_{NS}^i implements only those bindings contained in context c_i. Figure 14 schematically depicts the implementation of a naming system in which each name server corresponds to a single context. (A box represents a binding-group that defines the set of bindings implemented in a single name server object.)

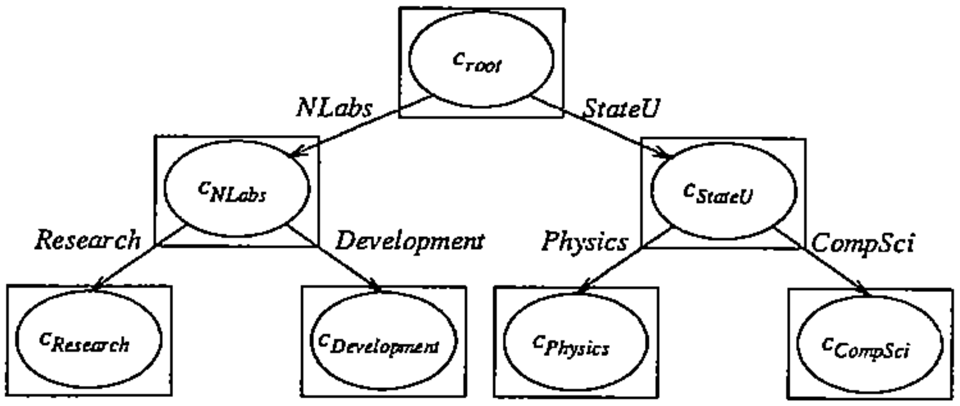

Figure 14
One Context per Binding-group Partitioning of C_H

change bindings.

†We do not consider the case in which bindings are distributed over name servers across context boundaries; one name server object implements a subset of the bindings from more than one context.

Second, a single binding-group may contain more than one context, such that a single name server object implements the bindings defined by several contexts. Figure 15 illustrates the naming system in which a single name server object exists for all of the contexts corresponding to National Labs and State University, respectively.

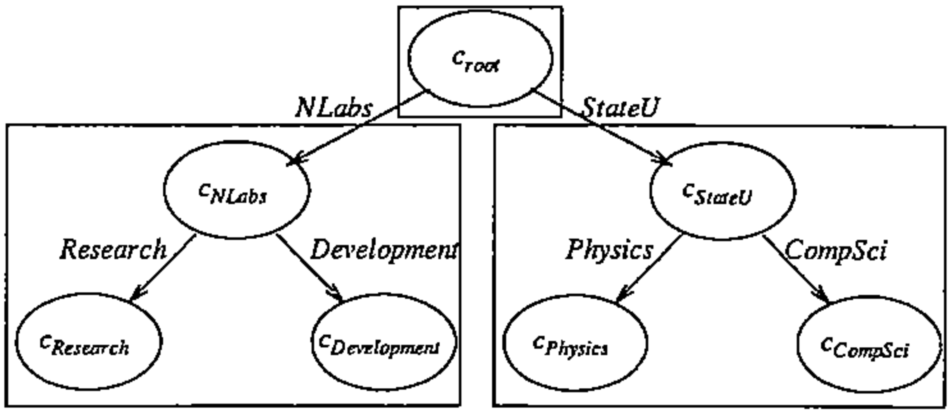

Figure 15
Multiple Contexts per Binding-group Partitioning of C_H

Finally, as illustrated in Figure 16, there may be a single name server associated with an entire name space (or naming system).

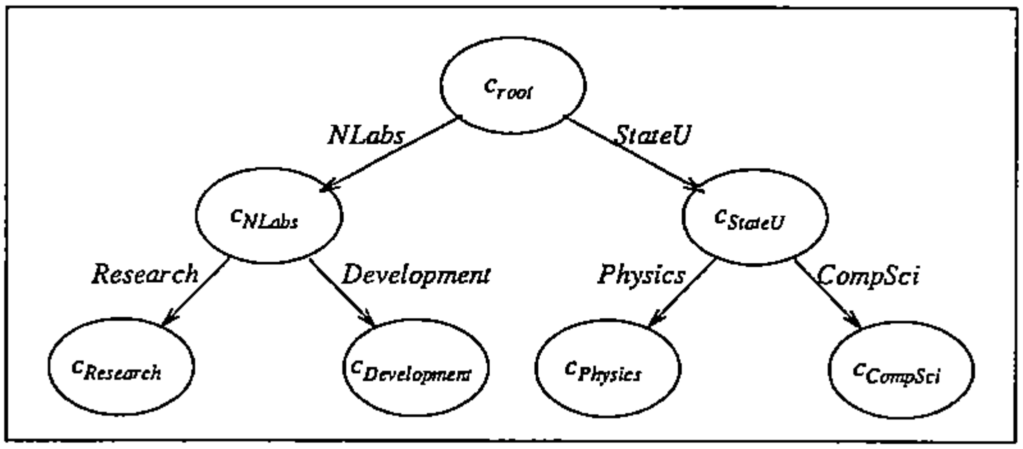

Figure 16
A Single Binding-group Encompassing All of C_H

Whether or not each name server maintains the structure of the component contexts is a question of internal representation, and does not affect our abstract view of a naming system. For example, rather than take advantage of the hierarchical relationship between the contexts implemented by the name server depicted in Figure 16, the name server could compress the names into a flat data structure, such that instead of matching one simple name component at a time, it does a string comparison of entire compound names. Also, multiple copies of a name server object may exist in the system, in which case the name server is said to be *replicated*.

7.2 Naming Name Server Objects

We now turn our attention to the problem of naming name server objects. Recall that the name server service is implemented by a set of name servers, each of which implements some subset of the bindings that comprise the naming system. The service moves from name server to name server in the same way as we visualize the function f_R iterating over a sequence of contexts. Thus, to resolve a given name, the service performs the *lookup* operation on a set of name server objects, requiring that the service be able to identify each object with a name. For the purpose of the following discussion, we assume that each context in a name space is implemented by a single name server, similar to the example illustrated in Figure 14.

To gain an intuitive understanding of the implications of naming name server objects, consider an analogy from the UNIX file system [RITC74]. The names of files in the system are defined by *directories*, where a directory is an abstraction similar to a context. Directories, however, are implemented as files. Thus, file objects are named by entities that are themselves implemented as files, corresponding to the situation in which the names of objects in a distributed system are implemented by other objects in the system.

Our mathematical model avoids the problem by identifying the abstract version of a name server object (i.e. a context), with a meta-symbol (e.g. c_i). Thus, the binding of simple name v_s to context c_i is represented by a binding of the form $<v_s, <c_i, \lambda>>$, and implicitly depicted in the naming network with a directed edge.

When implementing contexts in name servers, however, the name of the next name server object must be explicitly supplied. In the UNIX file system, this is accomplished by identifying a file that implements a directory with a lower-level name for that file (called an *i–number*). Similarly, the bindings contained in name server object o_{NS}^i identify other name server objects with names that are resolvable by still other name servers. For example, if we let *UID–NS–NLabs* be the unique identifier for the name server object that implements context c_{NLabs}, then the name server that implements context c_{root} would contain a binding of the form $<NLabs, <UID\text{–}NS\text{–}NLabs, \lambda>>$, implying that the service should next access the name server object named *UID–NS–NLabs*.

In addition to naming the next name server that should be queried, the name server service must also have knowledge of an initial name server that it can access, loosely corresponding to the starting context defined by the closure function f_C. That is, the name server service must know a priori the names of some set of name servers at which it can begin resolving a name. In our example naming system, the service might know about name server object o_{NS}^{root}, or it might know the names of a set of name server objects that it should try in order (e.g. $\{o_{NS}^{Research}, o_{NS}^{NLabs}, o_{NS}^{root}\}$).

7.3 Discussion

We conclude this section by making two observations regarding the role of name servers in name resolution. First, because Φ defines a name resolution mechanism for objects that are distributed throughout the distributed system, and name server objects may themselves be distributed throughout the system, there exists a relationship between the name servers that define the name of objects, and the objects themselves. For example, the name server service for GRAPEVINE [BIRR82], whether physically distributed over multiple environments or not, is understood to be logically centralized because it defines logical names. In contrast, if the distribution of the name server matches the distribution of objects as defined by a physical name space (e.g. name space C_D that defines names of the form *user@host*), then the objects and the name servers are located close together (i.e. in the same environment). Chertion [CHER84] considers name servers that are located close to the objects for which they define names to be logically distributed.

Second, because the object being named by the client, as well as the name server objects needed to resolve the name for that object, are potentially located in remote environments of the distributed system, Φ may need to employ the transport mechanism Φ_M to reach both the object and name server objects. That is, while a client initiates an operation-process that employs Φ to direct it from environment to environment, Φ in turn becomes a client of the name server objects in the system by invoking the *lookup* operation, thereby initiating its own operation-process. Thus, we conceptually think of the original operation-process as moving from a starting environment to a final environment, while Φ takes "side-trips" to possibly remote name server objects to determine the next move for the original operation-process.

8. SUMMARY

This paper presents a formal model for name resolution in distributed systems that views names to be purely syntactic entities, and name resolution to be a syntax-driven operation. Specifically, a distributed system's name resolution mechanism is described by a set of contexts that define interpretations of names (including moves through the environments in the system), and a universal function that resolves names according to an interpretation defined by a context.

We then use the model to define familiar terminology, including hierarchical names, unique-identifiers, source-route names, logical names, addresses, absolute names, relative names, and aliases. Furthermore, it explains how groups of contexts that behave in a uniform manner correspond to levels, where each level defines names of a certain form. Thus, our model both explains the partitioning of names into levels (as do Shoch's and Watson's models), as well as the nature of names corresponding to any given level. It also illustrates the relationship between the naming system and the underlying distributed system (as do the Oppen and Dalal, and Sirbu and Sutherland models). Finally, we describe the role of name servers in name resolution by defining the relationship between name servers and contexts.

REFERENCES

ALLM83 Eric Allman, "SENDMAIL — An Internetwork Mail Router," *UNIX Programmer's Manual*, 4.2 Berkeley Software Distribution, Vol. 2 (Aug. 1983).

ALME83 Guy Almes, Andrew Black, Edward Lazowska, and Jerre Noe, *The Eden System: A Technical Review*, Technical Report 83-10-05, University of Washington (Oct. 1983).

ANDR83 Gregory R. Andrews and Fred B. Schneider, "Concepts and Notations for Concurrent Programming," *Computing Surveys 15*, 1 (March 1983), 3-43.

BIRR82 Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder, "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM 25*, 4 (Apr. 1982), 260-273.

BIRR83 Andrew Birrell and Bruce Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems 2*, 1 (February 1984), 39-59.

CHER83	David Cheriton and Willy Zwaenepoel, "The Distributed V Kernal and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating System Principles* (Oct. 1983), 129-139.
CHER84	David Cheriton and Timothy Mann, "Uniform Access to Distributed Name Interpretation," *The Fourth International Conference on Distributed Computing Systems* (May 1984).
COME84	Douglas Comer, *Transparent Integrated Local and Distant Environment (TILDE): Project Overview,* Tilde Report CSD-TR-466, Purdue University (Mar. 1984).
CUNN82	I. Cunningham, et. al., "Emerging Protocols for Global Message Exchange," *Fall COMPCON* (Sep. 1982), 153-161.
DALE68	R. C. Daley and J. B. Dennis, "Virtual Memory, Processes, and Sharing in MULTICS," *Communications of ACM 11* 5 (May 1968).
DENN70	Peter Denning, "Virtural Memory," *Computing Surveys* 2, 3 (Sep. 1970).
DENN81	Peter Denning and Douglas Comer, *The CSNET User Environment,* CSD-TR-456, Purdue University (July 1981).
DENS68	Jack B. Dennis and Earl C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM 9,* 3 (March 1966), 143-155.
ECKH78	Richard Eckhouse and John Stankovic, "Issues in Distributed Processing — An Overview of Two Workshops," *Computer* (Jan. 1978), 22-26.
ENSL78	Philip Enslow, "What is a 'Distributed' Data Processing System?" *Computer* (Jan. 1978), 13-21.
FABR74	R.S. Fabry, "Capability-Based Addressing", *CACM,* Vol. 17, No. 7 (July 1974), 403-411.
HEND75	D.A. Henderson, *The Binding Model: A Semantic Base for Modular Programming Systems,* MIT/LCS/TR-145 (February 1975).
HOPC79	John Hopcroft and Jeffrey Ullman, *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, Reading Mass. (1979).
ISI81a	Information Sciences Institute, "Internet Protocol," *Request For Comments 791,* Edited by Jon Postel (Sept. 1981).
ISI81b	Information Sciences Institute, "Transmission Control Protocol," *Request For Comments 793,* Edited by Jon Postel (Sept. 1981).

JOHN71 John Johnston, "The Contour Model of Block Structured Processes," *Proceedings of a Symposium on Data Structures in Programming Languages* (February 1971), 55-82.

JONE78 Anita Jones, "The Object Model: A Conceptual Tool for Structuring Software," *Lecture Notes in Computer Science: Operating Systems,* Edited by R. Bayer, R.M. Graham, and G. Seegmuller, Spinger-Verlay, New York (1978), 7-16.

LELA81 Gerard LeLann, "Motivations, Objectives, and Characterization of Distributed Systems," *Lecture Notes in Computer Science: Distributed Systems — Architecture and Implementation,* Edited by B. W. Lampson, M. Paul, and H. J. Siegert, Spinger-Verlay, New York (1981), 1-9.

MOCK83 P. Mockapetris, "Domain Names — Concepts and Facilities," *Request For Comments 882* (Nov. 1983).

NOWI80 D. A. Nowitz and M. E. Lesk, "Implementation of a Dial-up Network of UNIX Systems," *UNIX Programmer's Manual,* 4.1 Berkeley Software Distribution, Vol. 2 (1979).

OPPE81 Derek Oppen and Yoger Dala, *The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment,* Office Products Division, XEROX (October 1981).

PICK79 J. R. Pickens, E. J. Feinler, and J. E. Mathis, "The NIC Name Server — A Datagram Based Information Utility", *Proceedings 4th Berkeley Workshop on Distributed Data Management and Computer Networks* (Aug. 1979).

POST82 Jonathan B. Postel "Simple Mail Transfer Protocol," *Request For Comments 821* (Aug. 1982).

RASH81 Richard F. Rashid and George Robertson, "Accent: A communication oriented network operating system kernal," *Proceedings of the Eighth ACM Symposium on Operating System Principles* (Dec. 1981), 64-75.

REYN84 J. Reynolds and J. Postel, "Official Arpa-Internet Protocols," *Request For Comments 924* (October 1984).

RITC74 Dennis Ritchie and Ken Thompson, "The UNIX Time-Sharing System," *Communications of the ACM 17,* 7 (July 1974), 365-375.

SALT78 Jerome Saltzer, "Naming and Binding of Objects," *Lecture Notes in Computer Science,* Vol. 60, Springer Verlag, New York, Ch. 3 (1978), 99-208.

SALT82 Jerome Saltzer, "On the Naming and Binding of Network Destinations," *International Symposium on Local Computer Networks, IFIP/T.C.6* (Apr. 1982), 311-317.

SHOC78 John F. Shoch, "Inter—Network Naming, Addressing, and Routing," *Proceedings 17th IEEE Computer Society International Conference (COMPCON)* (Sep. 1978), 72-79.

SHOE79 Kurt Shoens, "Mail Reference Manual," *UNIX Programmer's Manual*, 4.1 Berkeley Software Distribution, Vol. 2 (1979).

SIRB84 Marvin Sirbu and Juliet Sutherland, "Naming and Directory Issues in Message Transfer Systems," *Computer-Based Message Services*, Edited by H.T. Smith, North-Holland, New York (1984), 15-37.

SOLO82 Marvin Solomon, Lawrence Landweber, and Donald Neuhegen, "The CSNET Name Server," *Computer Networks 6* (1982), 161-172.

SUNS77 Carl Sunshine, "Source Routing in Computer Networks," *ACM SIGCOMM Computer Communication Rev. 7*, 1 (Jan. 1977), 29-33.

TERR84 D.B. Terry, "An Analysis of Naming Conventions for Distributed Computing Systems," *SIGCOMM Tutorials and Symposium on Communications Architecture and Protocols* (June 1984), 218-224.

WALK83 Bruce Walker, Gerald Pepek, Robert English, Charles Kline, and Greg Thiel, "The LOCUS Distributed Operating System," *Proceedings of the Ninth ACM Symposium on Operating System Principles* (Oct. 1983), 49-70.

WATS81a Richard Watson, "Distributed System Architecture Model," *Lecture Notes in Computer Science: Distributed Systems — Architecture and Implementation*, Edited by B. W. Lampson, M. Paul, and H. J. Siegert, Spinger-Verlay, New York (1981), 10-43.

WATS81b Richard Watson, "Identifiers (Naming) in Distributed Systems," *Lecture Notes in Computer Science: Distributed Systems — Architecture and Implementation*, Edited by B. W. Lampson, M. Paul, and H. J. Siegert, Spinger-Verlay, New York (1981), 191-210.