Office Products Division

The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment

by Derek C. Oppen and Yogen K. Dalal

XEROX

The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment

by Derek C. Oppen and Yogen K. Dalal

OPD-T8103 October, 1981

Abstract: We consider the problem of naming and locating objects in a distributed environment, and describe the clearinghouse—a decentralized agent for supporting network-visible objects. Binding is an important architectural component of a distributed system, and the clearinghouse serves the role of "glue" that binds together the many loosely-coupled, network-visible objects.

CR Categories: 3.74, 3.81.

Key words and phrases: Clearinghouse, names, locations, binding, network-visible objects, internetwork, distributed database.

© Copyright 1981 by Xerox Corporation.

XEROX

OFFICE PRODUCTS DIVISION
SYSTEMS DEVELOPMENT DEPARTMENT
3333 Coyote Hill Road / Palo Alto / California 94304

Table of Contents

Preface

We consider the problem of naming and locating objects in a distributed environment, and describe the clearinghouse—a decentralized agent for supporting the naming and locating of distributed objects.

Objects may be *individuals*, such as individual machines, workstations, file servers, or people. A typical use of the clearinghouse is to *locate* individuals. The clearinghouse provides two ways for locating individual objects: by name and by genre. To provide the first, the clearinghouse maintains a database mapping names into locations, and supports at least the following primitive operations using or modifying this database: (1) locating named objects, (2) creating, deleting and changing the locations of objects, (3) creating, deleting and changing the names of objects, and (4) passing the name of an object from one user to another so that others can access the object. To provide the second, the clearinghouse maintains a database mapping generic names (such as "Printers") into objects in the genre.

Objects may also be *groups* of other objects, as in distribution lists or access control lists. The clearinghouse maintains a database mapping names of groups into the sets of names of objects constituting each group, and provides primitives for (1) enumerating names in groups, (2) testing membership in groups, (3) creating, deleting and changing the names of groups, (4) adding and deleting members from groups, and (5) passing the name of a group from one user to another so that others can access the group. The clearinghouse also maintains a database mapping generic names (such as "distribution lists") into groups in the genre.

The objects "known" to the clearinghouse are therefore of many different types, and include workstations, servers (file servers, print servers, mail servers, clearinghouse servers), human users, and groups of these. All objects known to the clearinghouse are named using the same naming convention. The clearinghouse fields requests for information about objects in a uniform fashion, regardless of their type.

The mappings supported by the clearinghouse are richer than those described above. A name is *bound* to a set of *properties* of various types. We can, for instance, associate with the name of a user the location of his local workstation (so that others can send messages to his terminal, say), his local file server (so that he can store and retrieve files), his local mail server (so that he can receive mail), his local printer (so that he can print files), and non-location information such as password and comments. The clearinghouse also supports aliases of names.

The clearinghouse (and its associated database) is decentralized and replicated. That is, instead of one *global* clearinghouse server, there are many *local* clearinghouse servers scattered throughout the internetwork (perhaps, but not necessarily, one per local network), each storing a copy of a portion of the global database. The totality of services supplied by these clearinghouse servers we call "the clearinghouse." Decentralization and replication increase efficiency (it is usually faster to access a clearinghouse server that is physically nearby), security (each organization can control access to its clearinghouse servers), and reliability (if one clearinghouse server is down, perhaps another can respond to a request).

Updates to the various copies of a mapping may occur asynchronously and be interleaved with requests for bindings of names to properties; updates to the various copies are not treated as indivisible transactions. Any resulting inconsistency between the various copies is only transient: the clearinghouse automatically arbitrates between conflicting updates to restore consistency.

A client of the clearinghouse may refer by name to, and query the clearinghouse about, any named object in the distributed environment (subject to access control) regardless of the location of the object, the location of the client or the present distributed configuration of the clearinghouse. No assumptions are made about the physical proximity of clients of the clearinghouse to the objects whose names they present to the clearinghouse. A request to the clearinghouse to bind a name to its set of properties may originate anywhere in the internetwork and be directed to any clearinghouse server. If that clearinghouse server does not have the binding in its local database, it communicates with other clearinghouse servers to get the information. A client of the clearinghouse need not concern itself with the question of which clearinghouse server actually contains the binding—the clearinghouse automatically finds the mapping if it exists.

The clearinghouse described in this paper is the binding agent in Xerox Network Systems (including the Xerox 8010 Star information system), and is one of the key components of the underlying distributed systems architecture.

In Sections 1 and 2 we introduce the subject of this paper and many of the concepts. In Sections 3, 4, and 5 we discuss names, present a uniform naming convention for objects in an internetwork, and describe one particular application of this convention: naming users. In Section 6 we describe the mappings stored by the clearinghouse. In Section 7 we describe the clearinghouse from the client's perspective, and discuss various binding strategies (when the client should bind, or have the clearinghouse bind, a name). In Section 8 we describe the client-clearinghouse interface: what operations are provided to access and manipulate the data stored in the clearinghouse. In Section 9 we discuss the internal structure of the clearinghouse. In Section 10 we describe the algorithm used to find a mapping, the communication between the various clearinghouse servers in response to client requests for database lookups. In Section 11 we describe the distributed update algorithm used to update the clearinghouse database and maintain its consistency. In Section 12 we discuss clearinghouse security. In Section 13 we discuss the decentralized administration of the clearinghouse. In Appendix 1 we discuss in some detail the question of address validation. In Appendix 2 we discuss an alternative internal structure for the clearinghouse.

1. Introduction

Let us introduce the subject matter of this paper by considering the role of the information operator, the "White Pages" and the "Yellow Pages" in the telephone system.

Consider how we telephone a friend. There are two steps we take. First we find the person's telephone number, and then we dial the number. The fact that we consider these to be "steps" rather than "problems" is eloquent testimony to the success of the telephone system. But how do the two steps compare? The second—making the connection once we have the telephone number—is certainly the more mechanical and more predictable, from the user's point of view, and the more automated, from the telephone system's point of view. The first step—finding someone's telephone number given his or her name—is less automatic, less straightforward, and less reliable. If we already know the number or can ask somebody for it, then this is a trivial step. Otherwise, we have to use the telephone system's information system, which we call the *telephone clearinghouse.* If the person lives locally, we telephone information ("411") or look up the telephone number in the White Pages. (The White Pages map names, optionally associated with addresses, into telephone numbers.) If the person's telephone is non-local, we telephone information for the appropriate city ("555-1212"). Once we have accessed the appropriate information operator, we begin our dialogue in search of the telephone number. We present the last name to the operator. If the name is "Oppen" or "Dalal," the operator may immediately give us the telephone number. If the name is "Smith," he or she probably responds, "Can you give me a first name or address?" If we know them, we supply them. If we are lucky, we get the telephone number. Otherwise, we are given a set of telephone numbers, all satisfying the data given the telephone clearinghouse. In any case, we always have to treat whatever information we get from the telephone clearinghouse with a certain amount of suspicion, and treat it as a "hint." We have to accept the possibility that we have been given an incorrect number, perhaps because the person we wish to call has just moved. We are conditioned to this and automatically begin calls with "Is this ...?" to validate the hint.

In other words, although making the connection once we have the correct telephone number offers few surprises, *finding* the telephone number may be a time-consuming and frustrating task. The electrical and mechanical aspects of the telephone system have become so sophisticated that we can easily telephone almost anywhere in the world. The telephone clearinghouse remains unpredictable, and may require considerable interaction between us, as clients, and the information operator. As a result we all maintain our own personal database of telephone numbers (generally a combination of memory, little black books, and pieces of scrap paper) and rely on the telephone system's database only when necessary.

The telephone clearinghouse provides another service: the Yellow Pages. The Yellow Pages map generic names of services (such as "Automobile Dealers") into the names, addresses and telephone numbers of providers of these services. The properties of the information given by the Yellow Pages are the same as the properties described above for the White Pages.

In brief, there are three ways for objects in the telephone system to be found: by name, by number, or by subject. The telephone system prefers to use numbers, but its clients prefer names and generic names. The telephone clearinghouse provides a means for mapping between these various ways of referring to objects in the telephone world.

Let us move from the telephone system to distributed systems and, in particular, to interconnections of local networks of computers. An example might be the distributed "office of the future" consisting of several thousand workstations, assorted file servers, mail servers, communications servers, print servers, and so on, spread over several interconnected networks. Sitting at our local terminal or workstation, we want to send a file to our local printer or to someone else's workstation. Or we want to mail a message to someone elsewhere in the internetwork. The two steps we have to take remain the same: finding out where the printer or workstation or mail server is (that is, what its network address is), and then using this network address to access it.

As with the telephone system, the second step is fast becoming the step to be taken for granted. The design and implementation of internetworks of computers have become increasingly sophisticated, and their performance increasingly reliable. Although the content of this paper does not depend on any particular networking configuration, we will use as an example throughout this paper the *Ethernet* and its associated Pup-based or Xerox Network Systems-based *internetwork* routing machinery ([Metcalfe and Boggs 1976, Boggs *et al.* 1980, Ethernet 1980, Dalal and Printis 1981, Dalal 1981]). The internetwork knows how to use a network address to route a *packet* to the appropriate machine in the internetwork. So the second step—accessing an object once we know its network address—has well-known solutions. It is the first step—finding the location of a distributed object given its name—that we consider here.

An obvious question to ask at this point is: do we need names at all? Why not just refer to an object by its location? Why not just directly use the network address of our local file server or mail server or printer? The reasons are much like those for using names in the telephone system or in a file system. The first is that locations are unappealingly unintuitive; we do not want to refer to our local printer by its network address 5#346#6745 any more than we want to refer to a colleague as 415-494-4763 or to a file by its disk address. The second is that distributed objects change locations much more frequently than they change names. We want a level of indirection between us and the object we wish to access, and that level of indirection is given by a name. (See also [Shoch 1978] and [Abraham and Dalal 1980].)

When a network object is referred to by name, the name must be *bound* to the address of the object. The binding technique used greatly influences the ability of the system to react to changes in the environment. If client software binds names to addresses *statically* (for instance, if software supporting printing has the addresses of the print servers stored in it), the software must be updated if the environment changes (for instance, if new print servers are added or old servers are moved or removed). If client software binds names to addresses *early* (at the moment of system initialization) or *late* (at the moment the software wants to access the service), the system reacts much more gracefully to changes in the environment (they are not necessarily even noticed by the client). There are several possible approaches to binding (and we will discuss them later) but, regardless of the approach, clients need a *clearinghouse*, like the telephone clearinghouse, which maintains mappings from names into addresses and from which clients can request bindings for names.

The problems we address in this paper are therefore the related problems of how to name objects in a distributed computer environment, how to find objects given their names, and how to find objects given their generic names. In other words, how to create an environment similar to the telephone system's with its notions of names, telephone numbers, White Pages and Yellow Pages.

We also consider the administration, rather than just the use, of the internetwork and its clearinghouse. A configuration of several thousand users and their associated workstations, printers, file servers, mail servers, etc., requires considerable management. Administrative tasks include bringing up new networks; adding, changing and deleting services (such as mail services, file services, and even clearinghouse services); adding and deleting users; maintaining users' passwords, the addresses of their chosen local printer, mail and file servers, and so on; and maintaining access lists and other security features of the network. Since our clearinghouse is the main repository of information on users, workstations, and the other components of the internetwork, the clearinghouse provides facilities to aid *system administrators* in the administration of the distributed environment in which it resides. In addition, the clearinghouse "scales upwards" gracefully, and takes in its stride the addition of new networks, the addition of new clearinghouse servers, the interconnection of previously-disjoint networks, and so on.

Our clearinghouse naturally differs from the telephone clearinghouse, not least because of differences in the domains of discourse. The clients of the telephone clearinghouse are people, and the objects known to the telephone clearinghouse are also people (or rather their telephones). The telephone clearinghouse relies on human judgment and human interaction. The clients of our clearinghouse are machines, not people, and so all aspects of client-clearinghouse interaction must be fully automated and predictable.

We faced many questions in designing our clearinghouse; the following are a few of them:

Naming Convention. How shall we name the objects known to the clearinghouse? Should names be hierarchical (as in the Dewey Decimal System), or non-hierarchical (as in the social security numbering system)? Should names be unambiguous like social security numbers (no two people have the same social security number), or ambiguous like surnames (many people can have the same surname)? More generally, what sort of mapping should hold between names and the objects being named: should the mapping be one-to-one (each object has exactly one name and no two objects have the same name; names are unique and unambiguous), many-to-one (no two objects have the same name, but each object can have more than one name; names are non-unique but unambiguous), one-to-many (each object has exactly one name, but many objects may have the same name; names are unique but ambiguous), or many-to-many (names are non-unique and ambiguous)?

Design of the Clearinghouse. How shall we configure the clearinghouse? Is there just one clearinghouse with one monolithic database? Or is there one monolithic database decentralized among many local clearinghouses? If the latter, is the database strictly partitioned among local clearinghouses or can their databases overlap or be replicated? If the latter, are the different copies of the database always consistent? These options all assume that there is just one database, however decentralized. An alternative is that the database is relativized: there are many mappings from names, not just one. This leads to the question of the correctness of information given out by the clearinghouse. If the clearinghouse maps a name into a network address, is that address to be treated as correct or as merely a "hint"?

Management of the Clearinghouse. How is the clearinghouse managed? Are names allocated by the clearinghouse? If so, is there a central *naming authority* which allocates names for the whole internetwork, or is the naming authority decentralized? If not, who allocates a name: the object being named, anyone else who wants the object to be named, or perhaps both? Does the clearinghouse support nicknames, abbreviations and aliases? Who has updating authority over the database: the clearinghouse, its clients, or both?

Access Control. Who may obtain information from the clearinghouse? If the internetwork spans numerous companies, may any client obtain information from any clearinghouse?

In the preface we hinted at some of the design decisions we took, but before describing the options in more detail and our reasons for choosing the ones we did, let us look further at a familiar example of a clearinghouse—that maintained by the telephone system.

2. The Telephone Clearinghouse

The telephone system provides an excellent introduction to the problem of designing a clearinghouse for computer networks. We therefore consider the telephone model in some detail, emphasizing design decisions we refer to later in describing the design of our internetwork clearinghouse. Let us call the whole system provided by the telephone companies for mapping names into telephone numbers the *telephone clearinghouse*, and consider some of its more obvious properties in terms of the four basic operations specified in the preface. We consider only the White Pages component of the telephone clearinghouse; the Yellow Pages are similar.

2.1. Locating Named Objects

In the telephone system, mapping names into telephone numbers is relatively straightforward, at least from the user's point of view. If the person's telephone is local, we look up the telephone number in the telephone book or ask the information operator (by dialing 411) for the number. If the person's telephone is non-local, we telephone the information operator for the appropriate city to find the telephone number.

The database used by the telephone clearinghouse—the "telephone book" (whether printed or online)—is highly decentralized. The decentralization is based on physical locality: each telephone book covers a specific part of the country. It is up to the client of the telephone clearinghouse to know in which telephone book to look or to have the information operator look. (The alternative is exhaustive search.)

This decentralization is partly motivated by size; there are just too many telephones for there to be a common database. It is also motivated by the fact that people's names are ambiguous. Many people may share the same name, and corresponding to one name may be many telephone numbers (even ignoring the case of one person having more than one telephone). Decentralizing the telephone clearinghouse is one way to provide additional information to disambiguate a reference to a person—there may be many John Smiths in the country but hopefully not all are living in the same city as the John Smith whose telephone number I want. However, even by partitioning the database by city and by using other information such as street address, the telephone clearinghouse still may be confronted with a name for which it has several telephone numbers. When this happens it becomes the client's responsibility to disambiguate the reference, perhaps by trying each telephone number until he finds the one he wants. The essential point to note, however, is that the telephone clearinghouse cannot assume that names are unambiguous, and leaves it to the client to resolve ambiguities.

2.2. Creating, Deleting and Changing Locations

Locations may change because a person moves (his name remains the same but the mapping from his name to his telephone number has changed), adds telephone service or cancels telephone service.

Responsibility for *initiating* updates rests with the users of the telephone system. However, the actual updating of the database is done by the telephone company. Users of the telephone clearinghouse have read-only access to the clearinghouse's database. Further, requests for updates may be made only by the provider of the resource (the person who pays for the telephone whose location is being updated) and not by other users of the telephone clearinghouse.

Allocation of telephone numbers is the responsibility of the telephone company; the telephone company provides a *naming authority* to allocate telephone numbers. Users may request a particular number but the telephone company has the final say. There are two reasons for this. First, the telephone company has to guarantee the unambiguity of the number. Second, the telephone number has to conform to the addressing and routing conventions of the telephone system.

The updating process deserves scrutiny because it helps determine the accuracy of the information given out by the telephone clearinghouse. The information is not necessarily "correct." Offline telephone directories ("telephone books") are updated only periodically and so do not contain updates more recent than their date of publication. Even the online telephone directory used by information operators may give information which turns out to be erroneous when used. One reason for this is that asking the operator for a telephone number and using that telephone number to make a call are not treated by the telephone system as an indivisible operation: the directory may be updated between the two events. Another reason is that physically changing a telephone number and updating the database are asynchronous operations. The telephone clearinghouse system is highly parallel with considerable asynchrony.

The partitioning of the telephone clearinghouse's database is not strict. The database is a replicated database. Copies of a directory may appear in different versions, and telephone directories for different cities may overlap in the telephone numbers they cover. Since the updating process is asynchronous, the database used by the telephone company may not be internally consistent.

The effect of this—information given out by the telephone clearinghouse does not necessarily reflect all existing updates—is that the information provided by the telephone clearinghouse can only be used as a *hint*. The user must accept the possibility that he is dialing a wrong number, and *validate* the hint by checking in some way that he has reached the right person. However, the telephone company does provide some mechanisms for helping a user who is relying on an out-of-date directory, memory, or little black book. For instance, if a person moves to another city, his old telephone number is not reassigned to another user for some time, and during that period callers of his old number are either referred to his new number, or are less informatively told that they have reached an out-of-service number.

2.3. Creating, Deleting and Changing Names

Generally, names are added and deleted when service is added or cancelled; names are rarely changed.

What we said above about updating locations generally applies as well to updating names, with one exception. The *choice* of name appearing in the telephone clearinghouse database rests with the holder of the telephone being named, and only the holder can request an update. (That is, you are permitted to choose under what name you will appear in the telephone directory, even if the name is ambiguous.) This raises an interesting issue, that of nicknames, abbreviations and aliases. The above does not mean that I, as a user of the telephone system, cannot choose my own name for you (a *nickname*), but only that the telephone company will not maintain the mapping of my name for you into your telephone number—it will only maintain the mapping of *your* name for yourself into your telephone number. I may have my own "little black book" containing my own relativized version of the telephone clearinghouse, but the telephone company does not try to maintain its accuracy. Similarly, the telephone clearinghouse does not necessarily respond to abbreviations of names. And, finally, the clearinghouse will handle aliases (names I give myself other than my "real" name) only if they are entered in its database. That is, the telephone clearinghouse allows names to be non-unique: a person may have more than one name.

We can summarize some of the differences between telephone numbers (addresses), names, nicknames, aliases and abbreviations—whether they are ambiguous, whether they are chosen by the telephone system or by the owner of the telephone or by others, and whether they are maintained through changes by the telephone system.

	Ambiguous?	**Chosen by**	**Maintained by System?**
Address	No	System	Yes
Name	Yes	Owner	Yes
Alias	Yes	Owner	Yes
Nickname	Yes	Others	No
Abbreviation	Yes	Anyone	No

2.4. Passing Names and Locations

Giving someone else a telephone number (fully expanded to include country and area code) cannot raise problems because telephone numbers are unambiguous. (Of course, the telephone number may be incorrect by the time that person uses it.)

Giving a name to someone else is trickier since names are ambiguous. For instance, because the clearinghouse database is decentralized, giving a name to an information operator in one part of the country may elicit a different response from giving it to one in another part of the country. In the telephone clearinghouse, names are context-dependent. You can ensure that the person to whom you are giving a name will get exactly the same response only if you specify the appropriate clearinghouse as well.

3. Naming Distributed Objects

With this background, we return to the problem of designing a distributed system clearinghouse. A central question in designing such a clearinghouse is how to name the objects known to the clearinghouse.

3.1. Naming Conventions

A *naming convention* describes how *clients* of the naming convention refer to the *objects* named using the convention. The set of clients may overlap with the set of named objects; for instance, people are both clients of, and objects named using the common firstname-middlename-surname naming convention.

Our basic model for describing naming conventions is a directed graph with *vertices* and *edges*. Vertices and edges may be labelled. If vertex *u* has edge labelled *i* leading from it, then *u[i]* denotes the vertex at the end of the edge. (For this to be well-defined, edges leading from any vertex must be unambiguously labelled.) If $u[i_1][i_2]...[i_k]$ = v, then $i_1 \rangle i_2 \rangle ... \rangle i_k$ denotes the (possibly non-unique) *path* from *u* to *v*.

We assume that each named object and each client is represented by exactly one vertex in the graph. With these assumptions, we need not distinguish in the rest of this section between vertices in the name graph, named objects, and clients of the naming system, and our problem becomes: what is the name of one vertex (a named object) relative to another (a client)? There are two fundamental naming conventions, each of which we now describe.

3.2. Absolute Naming

Under the absolute naming convention, the graph consists of labelled vertices but no edges. Each vertex has a unique and unambiguous label. The *distinguished name* of a vertex is its label. Each vertex therefore has an unambiguous distinguished name; the name is the same regardless of the client.

(An equivalent model is the following. The graph has only unlabelled vertices. There is a distinguished vertex called the *directory* or *root vertex*. There is exactly one edge from the directory vertex to each other vertex in the graph; each such edge is uniquely and unambiguously labelled. There are no other edges in the graph. The name of a vertex is the label of the edge leading from the directory vertex to this vertex.)

This is a somewhat precise if rather pedantic definition of what is usually meant by "choosing names from a flat name space." One obvious example of names using absolute naming conventions are Social Security numbers. In the Pilot environment ([Redell *et al.* 1980]), it is possible for each object to have a unique and unambiguous name consisting of a unique and unambiguous processor number (hardwired into the processor at the factory) concatenated to the time of day; this name is called the object's *universal identifier* ([Dalal and Printis 1981]).

3.3. Relative Naming

Under the relative naming convention, the graph has unlabelled vertices but labelled edges. There is either zero or one uniquely-labelled edge from any vertex to any other. If there is an edge labelled i from u to v, then the *distinguished name* of v *relative to* u is i. Here, u is the client and v the named object. Note that names are ambiguous—a relative name is unambiguous only if qualified by some *source* vertex, the client vertex.

Without additional disambiguating information, people's names are relative. One person's use of the name "John Smith" may well differ from another's.

3.4. Comparison of the Absolute and Relative Naming Conventions

There are advantages and disadvantages to each naming convention corresponding to the various tasks of the clearinghouse mentioned in the Preface.

Locating Named Objects. One of the main roles of the clearinghouse is to maintain the mapping *LookUp* from names into objects. If i is the name of an object, then *LookUp(i)* is that object. (The actual form of the righthand side of the mapping will be described in Section 6.) Under the relative naming convention, *LookUp* is relative to each client vertex. That is, if the name of an object v *relative to* u is i, then $LookUp_u(i)$ is v. Under the absolute naming convention, *LookUp* is relative to the whole graph. That is, if the name of an object v is i, then *LookUp(i)* is v; we do not have to qualify *LookUp* with the source vertex. Thus, the database required by the absolute convention may be smaller (since the number of names is exactly the number of vertices) than under the relative convention (where the number of names is on the order of the square of the number of vertices). However, since the relative convention does not require that every vertex be able to directly name every other vertex (there need not be an edge from every vertex to every other), the domain of each *LookUp* under the relative convention will typically be much smaller than the domain for *LookUp* under the absolute convention.

The relative convention encourages decentralization, since the mapping from names to objects is relative to each vertex. The absolute convention encourages centralization, since there is only one mapping for the whole system. Thus the relative convention allows more efficient implementation of the *LookUp* function. Of course, one can use efficient methods such as binary search or hashing with either convention, but these make use only of *syntactic* information in names, not *semantic* information.

Changing Locations or Names. The main considerations here are the size and degree of centralization of the databases. Consider, for instance, the allocation of names. The absolute naming convention requires a centralized naming authority, allocating names for the whole graph. The relative naming convention permits decentralized naming authorities, one for each vertex. The local data base handled by the naming authority under the relative convention will typically be much smaller than the global data base handled by the naming authority under the absolute convention.

Passing Names and Locations. A major advantage of the absolute naming convention is that there is a common way for clients to refer to named objects. It is possible for any client to hand any other client the name of any object in the environment and be guaranteed that the name will mean the same thing to the second client (that is, refer to the same object). This is not the case with the relative addressing convention; if *u* and *v* are vertices, *u[i]* need not equal *v[i]*. Under the relative naming convention, the first client must give the second client the name of the object *relative to the second client*. In practice, this means that the first client has to understand how the second client names objects. This suggests excessive decentralization; it requires too much coordination when objects are to be shared or passed.

3.5. Hierarchical Naming

Neither the absolute nor the relative naming convention is obviously superior to the other; both have advantages and disadvantages. One might imagine combining the two notions, but we can do even better by adding another layer of structure to the basic naming model.

We partition the graph into subgraphs, consisting of subsets of the set of vertices. We assume that each vertex is in exactly one subgraph. The *distinguished name* of a vertex is *vertexname@subgraphname* where *subgraphname* is the name of its containing subgraph and *vertexname* is the name of the vertex in that subgraph. This definition is only well-defined if names are unambiguous within a subgraph; the absolute naming convention must be used within a subgraph. That is, within any subgraph, no two vertices can have the same name. Two different vertices may have names *A@B* and *A@C* however; names need be unambiguous only *within a subgraph*.

The mapping *LookUp* implemented by the clearinghouse becomes a mapping from *Vertexnames x Subgraphnames* into objects. If *B* is the name of a subgraph and *A* is the name of vertex *u* within that subgraph, then *LookUp(A@B) = u*.

The name of a vertex consists of both its name within a subgraph and the name of the subgraph. We have already pointed out that the absolute naming convention must be used for naming vertices within any subgraph. How shall we name subgraphs? They may be named using either the relative or the absolute naming convention. All the remarks made previously about absolute and relative naming conventions hold.

If the absolute naming convention is used, each distinct subgraph has an unambiguous distinguished name. Since the absolute naming convention is also used for naming vertices within each subgraph, it follows that vertices have unambiguous distinguished names. That is, no two vertices have the same name *A@B*. Telephone numbers such as 494-4763 fit into this two-level absolute naming hierarchy. The local exchange is uniquely and unambiguously determined (within each area code) by the exchange number 494; within exchange 494, exactly one telephone has number 4763.

If the relative naming convention is used, each distinct subgraph has an unambiguous distinguished name relative to each other subgraph. And, since we are using the absolute naming convention within subgraphs, it follows that each vertex has an unambiguous distinguished name relative to each source. An example of this is the interface between the Xerox mail transport mechanism [Birrell, Levin, Needham and Schroeder 1981] and the Arpanet mail transport system. A name may be *Oppen.PA* within Xerox but *Oppen@MAXC* outside—the subgraph name has changed.

In either case, the advantages of using a hierarchy is clear: it admits the advantages of absolute naming without barring decentralization. A partitioned name helps suggest the search path to the object being named and encourages a decentralized naming authority.

There is no need to stop at just one level of hierarchy. One can imagine a hierarchy of graphs with corresponding names of the form $i_1@i_2@...@i_k$. Examples include telephone numbers fully expanded to include country and area codes (a four-level hierarchy), or network addresses (a three-level hierarchy of network number, host number, socket number), or booknaming conventions such as the Dewey Decimal System.

We see now that the usual distinction made between "flat" and "hierarchical" is somewhat misleading. The distinctions should be "flat" or "absolute" versus "relative" and "hierarchical" versus "non-hierarchical."

3.6. Abbreviations

The notion of *abbreviation* arises naturally with hierarchical naming. Within subgraph *B*, the name *A@B* can be abbreviated to *A* without ambiguity, given the convention that abbreviations are expanded to include the name of the graph in which the client vertex exists. Abbreviation is a relative notion. (See, for example, [Daley and Neumann 1965] for another approach to abbreviations.)

3.7. Combining Networks

One major advantage of the hierarchical superstructure that we have not considered before, and which is independent of the absolute versus relative naming question, concerns combining networks. One feature that any clearinghouse should be able to handle gracefully is joining its database with the database of another clearinghouse, an event that happens when their respective networks are joined. For instance, consider the telephone model. When the various local telephone systems in North America combined, they did so by adding a superstructure above their existing numbering system, consisting of area codes. Area codes are the names of graphs encompassing various collections of local exchanges. When direct dialing between countries was introduced, yet another layer was added: country codes.

Adding new layers to names is one obvious way to combine networks. The major advantage is that if a name is unambiguous within one network then it is still unambiguous with its network name as prefix, even if the name also appears in some other network (because the latter name is prefixed by the name of *that* network). The major disadvantage is that the software or hardware has to be modified to admit the new level of naming. The latter problem is compounded if abbreviations are allowed.

The alternative to adding a new layer is expanding the existing topmost layer. For instance, the North American area code numbering system is sufficiently flexible that another area code can be added if necessary. The advantage of this is that less change is required to existing software and hardware. The disadvantage, if absolute naming is wanted, is that there has to be a centralized naming authority to ensure that the new area code is unambiguous.

3.8. Levels of Hierarchy

If one chooses to use a hierarchical naming convention, an obvious question is the following: should we agree on a constant number of levels (such as two levels in the Arpanet mailing system or four in the telephone system) or an arbitrary number of levels? If a name is a sequence of the form $i_1@i_2@...@i_k$, should k be constant or arbitrary? There are pros and cons to either scheme. The advantage of the *arbitrary* scheme is that the naming system may evolve (acquire new levels as a result of combining networks) very easily. That is, if we have a network now with names of the form *A@B*, and combine this network (let us call it network *C*) with another network, then we can just change all our names to names of the form *A@B@C* without changing any of the algorithms manipulating names. Allowing arbitrary numbers of levels clearly has an advantage. It also has several non-trivial disadvantages. First, all software must be able to handle an arbitrary number of levels, so software manipulating names will tend to be more complicated than in the *constant* level scheme. Second, abbreviations become very difficult: does *A@B* mean exactly that (an object with a two-level name) or is it an abbreviation for some name *A@B@C*? The disadvantage with the *constant* scheme is that one has to choose a number, and if we later add new levels, we have to do considerably more work.

3.9. Aliases

Our basic model allows each vertex to have exactly one name under the absolute naming convention, and exactly one name relative to any other vertex under the relative naming convention. An obvious extension to this model is to allow *aliases* or alternative names for vertices. To do this, we define an equivalence relation on names; if two names are in the same equivalence class, they are names of the same vertex. Under the relative naming convention, there is one equivalence relation defined on names for each client vertex in the graph. Under the absolute naming convention, there is only one equivalence relation for the whole graph. Each equivalence class has a root or distinguished member, and this we designate the *distinguished name* of the vertex.

The notion of aliasing is easily confused with the notion of relative naming, since each introduces multiple names for objects. The difference lies in the distinction between ambiguity and non-uniqueness. Under the relative naming convention, a name can be *ambiguous* in that it can be the name of more than one node (relative to different source nodes). Under the absolute naming convention, names are unambiguous. In either case, without aliasing, names are *unique*: if a vertex knows another vertex by name, it knows that vertex by exactly one name. With aliasing, names are non-unique; one vertex may know another by several names. Another way of expressing the difference is to consider the mapping from names to vertices. Without aliasing, the mapping is either one-to-one (under the absolute naming convention: each object has exactly one name and no two objects have the same name) or one-to-many (under the relative naming convention: each object has exactly one name relative to any other, but many vertices may have the same name). With aliasing, the mappings become many-to-one or many-to-many. The distinction is subtle. The following table illustrates the various combinations that are possible, in terms of ambiguity and uniqueness:

	Without Aliasing	**With Aliasing**
Absolute Naming Convention	Unambiguous, unique One-to-one	Unambiguous, non-unique Many-to-one
Relative Naming Convention	Ambiguous, unique One-to-many	Ambiguous, non-unique Many-to-many

4. Clearinghouse: Naming Convention

We now describe the naming system supported by our clearinghouse. Recall first that we have a very general notion of the objects being named: an object is anything that has a name known to the clearinghouse and the vague property of "network visibility." We shall give some concrete examples in the following sections.

Objects are named in a uniform fashion. We use the same naming convention for every object, regardless of whether it is a user, a workstation, a server, a distribution list or whatever.

A *name* is a non-null character string of the form *<substring$_1$>@<substring$_2$>@<substring$_3$>*, where *substring$_1$* denotes the *localname*, *substring$_2$* the *domain*, and *substring$_3$* the *organization*. Thus names are of the form *L@D@O* where *L* is the localname, *D* the domain and *O* the organization. None of the substrings may contain occurrences of "@" or "*" (the reasons for the latter exclusion will be given later). The clearinghouse does not attach any meaning to the substrings constituting a name.

Each object has a *distinguished name*. Distinguished names are absolute; no two objects may have the same distinguished name. In addition to its distinguished name, an object may have one or more aliases. Aliases are also absolute; no two objects may have the same alias. A name is either a distinguished name or an alias, but not both.

We have thus divided the world of objects into organizations, and subdivided organizations into domains: a three-level hierarchy. An object is *in organization O* if it has a name of the form *<anything>@<anything>@O*. An object is *in domain D in organization O* or *in D@O* if it has a name of the form *<anything>@D@O*.

This division into organizations and, within them, domains is a logical rather than physical division. An organization will typically be a corporate entity such as Xerox Corporation. The names of all objects within Xerox will be of the form *<anything>@<anything>@Xerox*. Xerox will choose domain names to reflect administrative, geographical, functional or other divisions. Very large corporations may choose to use several organization names if their name space is very, very large. In any case, the fact that two addressable objects have names in the same domain or organization does not imply in any way that they are physically close.

Two names are *equal* if they are identical strings, ignoring case.

4.1. Rationale

We use a uniform naming convention for all objects, regardless of their type. Our approach therefore differs from most systems where different naming conventions are used to name objects of different types: where people, distribution lists, machines, and so on are all named in different fashions. Our approach is much "friendlier" to the user since he has to remember only one convention. A disadvantage is that we cannot tell the type of an object merely by looking at the name. We consider this disadvantage unimportant; the type of the object is easily obtained by checking what its name is mapped into.

Objects known to the clearinghouse have absolute distinguished names and aliases. Thus we favour an absolute naming convention over a relative naming convention. Most systems (including most mail transport systems) have opted for a relative naming convention. However, the advantages of an absolute convention (where a name always denotes the same object regardless of where the name is used) are so clear that we are willing to put up with the burden of some centralization. By choosing the naming convention carefully, we can reduce the pain of this centralization to an acceptable level.

Names are hierarchical. We rejected a non-hierarchical system because, among their other advantages, hierarchical names can be used to help suggest the search path to the mapping.

We have chosen a three-level naming hierarchy, consisting of *organizations*, within them *domains*, and within them *localnames*. We did not choose the arbitrary level scheme because of the greater complexity of the software required to handle names, because we do not think that networks will be combined very often, and because (as with area codes) we will make the name space for organizations large enough so that combinations can generally be made within the three-level hierarchy by adding new organizations. We choose three levels rather than, say, two or four, for pragmatic reasons. A mail system such as the Xerox Laurel-Grapevine system [Birrell, Levin, Needham and Schroeder 1981] works well with only a two-level hierarchy, combining networks across the company's divisional boundaries. We add the third level primarily to facilitate combining networks across company lines. However, the clearinghouse does not give any particular meaning to the partitions; this is why we chose the relatively innocuous names "organization" and "domain." We leave the partitioning of names within an organization to the clearinghouse administrators within the organization (see Section 13); giving them the freedom to partition their name space in the way most convenient to them is clearly desirable.

The usefulness of aliases will be made clear in the next section. Our clearinghouse maintains aliases (that is, modifies them appropriately when updates occur); this is explained later.

The clearinghouse does not support abbreviations. An abbreviated name is a relative, as opposed to absolute, name (for example, *A* abbreviates both *A@B@C* and *A@B@D*) and the clearinghouse concerns itself only with absolute names. Typically, client software will allow its users to abbreviate names, and will add appropriate defaults before presenting them to the clearinghouse.

5. User Names

One important class of "objects" known to the clearinghouse is the set of users. For instance, as we shall see, the clearinghouse may be used to map a user's name into the network address of the mail server where his incoming mail resides. To deliver a piece of mail to a user, an electronic mail system first asks the clearinghouse where the mail server for that user is and then routes the piece of mail to that server.

A major design decision is how users are to be named. We describe our approach (to be used in the Xerox Network Systems product line) to naming users as this will provide further motivation for our naming convention. The following is not part of the design of our clearinghouse, but illustrates one of its important uses.

A *User Name* is a string of the form *<firstname> <blanks> <middlename> <blanks> <lastname>@<domain>@<*organization*>*. Here, *<firstname>*, *<middlename>* and *<lastname>* are strings separated by blanks (they may themselves contain blanks, as in the last name *de Gaulle*). *<firstname>*, *<middlename>* and *<lastname>* are the first name, middle name and last name of the user being named. The following are examples of user names:

Derek Charles Oppen@SDD@Xerox
Yogen Kantilal Dalal@SDD@Xerox

The basic scheme, therefore, is that a legal name consists of the user's three-part localname, domain and organization. No particular semantics are given to domains and organizations; in the above example, the organization name is the name of the company, and the domain name is the name of a logical unit of the company. The reason for making the user name the complete three-part name (rather than just the last name) is to discourage clashes of names and encourage unambiguity. The chance of there being two people with the name *Derek Charles Oppen* in domain *SDD* in organization *Xerox* is hopefully rather remote, and certainly more remote than their being two people with last name *Oppen.*

Our convention for naming users differs from those used in most computer environments in requiring that names be absolute and in using full names to reduce the chance of ambiguity. We have discussed the issue of absolute versus relative naming conventions already, but the second topic deserves attention because it shows the advantages of having a consistent approach to aliases.

The most common way of choosing unambiguous user names in computer environments is to use last names prefixed with however many letters are needed to exclude ambiguity. Thus, if there are two *Oppen's*, one might be *DOppen* and the other *HOppen.* This scheme we find unsatisfactory. It is difficult for users (who have to remember to map their name for the person into the system's name for the person) and difficult for system administrators (who have to manage this rather artificial scheme). Further, it requires users to occasionally change their system names: if a system name is presently *DOppen* and another *D. Oppen* becomes a user, the system name must be changed to avoid ambiguity.

Another scheme is to name users *Oppen-1, Oppen-2*, ... This avoids the problem of names becoming ambiguous, but again is difficult to use and manage.

Our convention is not cumbersome to the user, in theory at least, since we use the same firstname-middlename-lastname convention people are used to already. However, since users would find it very cumbersome to type in full names, various aliases for user names are stored in the clearinghouse. For instance, associated with the user name *Derek Charles Oppen* might be the aliases *Derek Oppen, D Oppen* and *Oppen*. Associated with the name *Robert Allen Mitchell* might be aliases such as the above, together with *Bob Mitchell*, etc. Since our naming convention requires that aliases be absolute, it follows that no two users can have the same alias. For instance, if there are two *Smiths* in *SDD* at *Xerox*, the alias *Smith* cannot be used. More information must be provided with the name, such as initials or first name.

The advantage of using aliases is that it makes the naming convention friendly to the user. The disadvantage is that storage is required to maintain them. Another approach, using pattern matching, is described in Section 8.

The clearinghouse does not explicitly support abbreviations but client software (such as the mail system) may choose to support them, allowing the user to address a message to *Oppen*, say, instead of *Oppen@SDD@Xerox*. The client software adds appropriate defaults to construct the full name.

5.1. Birthmarks

Even with our convention of using a user's full name, there is a possibility that there will be two users with exactly the same name in a domain. Our approach is to disallow this, and let the two users (or a system administrator) choose unambiguous names for each. Another approach is to add as a suffix to each full name a "birthmark." A *<birthmark>* is any string which, together with the user name, the domain name and the organization name, unambiguously identifies the user. The birthmark may be a universal identifier (perhaps the concatenation of the processor number of the workstation on which the name is being added together with the time of day). It might be the social security number of the individual (perhaps not a good idea on privacy grounds). It might be just a positive integer; the naming authority for each domain is responsible for handing out integers. In any case, the combination of the full name and the birthmark must be unambiguous so that no two users can have the same legal name. Again, aliases are used so that users do not need to provide a birthmark unless necessary.

6. Clearinghouse: Mappings

Now that we know how to name the objects known to the clearinghouse, we treat the question of what names are mapped into.

The clearinghouse maps each name into a set of *properties* to be associated with that name. A *property* is an ordered tuple consisting of a *PropertyName*, a *PropertyType* and a *PropertyValue.* The clearinghouse maintains mappings of the form:

$$\begin{aligned} name \rightarrow \{ & \langle PropertyName_1, PropertyType_1, PropertyValue_1 \rangle, \\ & \ldots, \\ & \langle PropertyName_k, PropertyType_k, PropertyValue_k \rangle \}. \end{aligned}$$

More precisely, to admit aliasing, the clearinghouse maps equivalence classes, rather than names, into sets of properties. Each equivalence class consists of a distinguished name and its aliases. However, unless necessary, we will not bother distinguishing between a name and the equivalence class it is in, and so will continue to refer to the clearinghouse mappings as mappings from names to properties.

The value of k is not fixed for any given name. A name may have associated with it any number of properties.

A *PropertyName* identifies a particular property associated with a given name. There may be only one property with a given property name associated with any name; that is, $PropertyName_i \neq PropertyName_j$ if $i \neq j$. (In the examples given in this paper we will use strings for property names; in practice we use integers.) To promote consistency in the use of property names, each property name is registered with the clearinghouse (as discussed in Section 13).

A *PropertyValue* is a datum of type *PropertyType.* There are only two types of property values. The first, of type *individual* or *0*, is "uninterpreted block of data." The clearinghouse attaches no meaning to the contents of this datum, but treats it as just a sequence of bits. The second, of type *group* or *1*, is "set of names," where a name is any name as defined in Section 4. A name may appear only once in the set, but the set may contain any number of different names (including aliases and names of other groups). The names "individual" and "group" reflect the semantics attached by the clearinghouse, whether the property is an individual datum or a group of data; they do not suggest that the object with these properties is an "individual" or a "group."

6.1. Examples

Mapping a name into a network address is an example of a type *individual* mapping, as in the following:

Daisy@SDD@Xerox → *{<"Printer", 0, network address of the printer named Daisy>}.*

or

Oppen@SDD@Xerox → *{<"Workstation", 0, network address of workstation>}.*

Since the value associated with an *individual* property is uninterpreted, it need not be a network address. It might be a name, a comment, or anything else. For example:

> *Tundra@SDD@Xerox → {*
> *<"File Server", 0, [network address of the file server named Tundra,*
> *descriptive comment]>}.*

A distribution list in electronic mail is an example of a mapping of type *group*, as in:

> *ClearinghouseAuthors@SDD@Xerox → {*
> *<"Distribution List", 1, {"Dalal@SDD@Xerox", "Oppen@SDD@Xerox"}>}.*

In each of the above examples, only one property was associated with a name. The following are more realistic examples, where many properties are associated with a name:

> *Oppen@SDD@Xerox → {*
> *<"User", 0, descriptive comment>,*
> *<"Password", 0, password to be used for user authentication>,*
> *<"File Server Name", 0, name of file server containing user's files>,*
> *<"Mail Server Name", 0, name of mail server where user's mail is stored>,*
> *<"Printer Names", 1, set of names of local printers any of which may be used>}.*

In this example, the clearinghouse is used to store the user's "profile." Associated with the user's name is the location of his local file server (so that he can store and retrieve files), his local mail server (so that he can receive mail), his local printer (so that he can print files), and so on. Note that we choose to map the user's name into the *name* of his local file server (and mail server and printer) rather than directly into its network address. The reason for this extra level of indirection is that the name of the file server will perhaps never change but its location certainly will occasionally change, and we do not want a change in a server's location to require a major update of the clearinghouse's database.

Alternatively, a user's profile may be stored as:

> *Yogen K. Dalal@SDD@Xerox → {*
> *<"User Profile", 0, [descriptive comment,*
> *password to be used for user authentication,*
> *name of file server containing user's files,*
> *name of mail server where user's mail is stored,*
> *set of names of local printers any of which may be used]>}*

6.2. Rationale

In the Preface, we separated objects into two broad categories: *individual* objects such as workstations, servers or people whose names are mapped into addresses, and *groups* whose names are mapped into sets of names. As the above shows, the mappings supported by the clearinghouse are more general.

We differentiate between data of type *individual* and data of type *group*, but allow many pieces of data of differing types to be associated with each name. The example given above showing the mapping for a user name shows why. Unlike the simpler telephone model where a single mapping from a user name into a telephone number suffices, we want to map a user's name into a richer collection of information. This applies even to non-user individuals. We may want to associate with a printer's name not only its location (so that files to be printed can be sent to it), but also information describing what fonts the printer supports, if it prints in color, and so on.

The main reason for having "set of names" as a distinct data type is to allow different clients to update the same set simultaneously. For instance, if the set represents an electronic mail distribution list, we want to allow two users to asynchronously add themselves to this list. This is discussed further in Section 8 when we describe the operations supported on elements of a set and in Section 11 when we describe how the clearinghouse automatically arbitrates between conflicting, asynchronous update requests.

7. Clearinghouse: Client's Perspective

We now know how objects are named in the clearinghouse, and what names may be mapped into. Before describing the functions the clearinghouse provides its clients, let us first describe how the clients are to perceive the clearinghouse, repeating many of the points made in the Preface, and the question of when clients should bind names to properties.

Recall first that the clients of the clearinghouse are pieces of software and hardware making use of the clearinghouse client interface. The fact that people are not clients of the clearinghouse (except very indirectly by means of a software interface) immediately introduces an important difference between our clearinghouse and the telephone system's. The telephone system relies on human judgement and human interaction. The clients of our clearinghouse are machines, not people, and so all aspects of client-clearinghouse interaction, including fault-tolerance, must be fully automated.

The clearinghouse (and its associated database) is decentralized and replicated. That is, instead of one *global* clearinghouse, there are many *clearinghouse servers* scattered throughout the internetwork (perhaps, but not necessarily, one per local network), each storing a copy of a portion of the global database. Decentralization and replication increase efficiency (it is faster to access a clearinghouse server physically nearby), security (each organization can control access to its own clearinghouse servers) and reliability (if one clearinghouse server is down, perhaps another can respond to a request). However, we do assume that there is one *global* database (conceptually, that is; physically the database is decentralized). Each clearinghouse server contains a portion of this database. We make no assumptions about how much of the database any particular clearinghouse server stores. The union of all the local databases stored by the clearinghouse servers is assumed to be the global database.

A client of the clearinghouse may refer by name to, and query the clearinghouse about, any named object in the distributed environment (subject to access control) regardless of the location of the object, the location of the client or the present distributed configuration of the clearinghouse. We make no assumptions about the physical proximity of clients of the clearinghouse to the objects whose names they present to the clearinghouse. A request to the clearinghouse to bind a name to its properties may originate anywhere in the internetwork. This makes the internal structure of our clearinghouse considerably more intricate than that of the telephone clearinghouse (where clients have to know which local telephone directory to access), but makes it much easier to use.

In order to provide a uniform way for clients to access the clearinghouse, we assume that all clients contain a (generally very small) clearinghouse component, which we call a *stub clearinghouse*. Stub clearinghouses usually contain little in their databases except pointers to clearinghouse servers, but they provide a uniform way for clients to access the clearinghouse.

A client requests a binding from its stub clearinghouse. The stub communicates with clearinghouse servers to get the information. A client of the clearinghouse need not concern itself with the question of which clearinghouse server actually contains the binding—the clearinghouse automatically finds the mapping if it exists. This differs from many models of distributed environments where one is restricted to local queries or references, and where clearinghouses (or their equivalents) know about objects of specific types only.

Updates to the various copies of a mapping may occur asynchronously and be interleaved with requests for bindings of names to properties. Therefore, clearinghouse server databases may occasionally have incorrect information or be mutually inconsistent. (In this respect, we follow the telephone system's model and not the various models for distributed databases in which there is a notion of "indivisible transaction." We find the latter too complicated for our needs.) Therefore, as in the telephone system, bindings given by clearinghouse servers should be considered by clients to be *hints*. If a client requests the address of a printer, it may wish to check with the server at that address to make sure it is in fact a printer. If not, it must be prepared to find the printer by other means (perhaps the printer will respond to a local broadcast of its name), wait for the clearinghouse to receive the update, or reject the printing request. If the information given out by the clearinghouse is incorrect, it cannot, of course, guarantee that the error in its database will be corrected. It can only hope that whoever has invalidated the information will send (or preferably already has sent) the appropriate update. However, the clearinghouse does guarantee that any inconsistencies between copies of the same portion of the database will be resolved, that any such inconsistency is transient. This guarantee holds even in the case of conflicting updates to the same piece of information; the clearinghouse arbitrates between conflicting updates in a uniform fashion. The updating mechanism is described in Section 11.

Assuming this model of goodwill on the part of its clients—that they will quickly update any clearinghouse entry they have caused to become invalid—and assuming an automatic arbitration mechanism for quickly resolving in a predictable fashion any transient inconsistencies between clearinghouse servers, clients can assume that any information stored by the clearinghouse is either correct or, if not, will soon be corrected. Clients therefore may assume that the clearinghouse either contains the *truth* about any entry, or soon will contain it. It is very important that clients can trust the clearinghouse in this way, because the clearinghouse is often the only source of information available to the client on the locations of servers, on user profiles, and so on.

The fact that the information returned by the clearinghouse is treated by the clients as both the truth (the information is available only from the clearinghouse and so had better be correct) and a hint (the information may be temporarily incorrect) is not self-contradictory. It merely reflects the difference between the long-term and short-term properties of clearinghouse information.

7.1. Binding Strategies

An important consideration to be taken by the client (or, rather, the author of the software comprising the client) is that of *when* to ask the clearinghouse for a binding. The binding technique used greatly influences the ability of the system to react to changes in the environment.

There are three possibilities: *static* binding, in which names are bound at the time of system generation; *early* binding, in which names are bound, say, at the time the system is initialized; and *late* binding, in which names are bound at the time their bindings are to be used. (The boundaries between the three possibilities are somewhat ill-defined; there is a continuum of choices.)

The main tradeoff to be taken into consideration in choosing a binding strategy is *performance* versus *flexibility*.

The later a system binds names, the more gracefully it can react to changes in the environment. If client software binds names statically, the software must be updated whenever the environment changes. For instance, if software supporting printing directly stores the addresses of the print servers (that is; uses a static binding strategy), it must be updated whenever new print servers are added or existing servers are moved or removed. If the software uses a late binding strategy, it will automatically obtain the most up-to-date bindings known to the clearinghouse.

On the other hand, binding requires the resolution of one or more indirect references, and this takes time. Static or early binding increases runtime efficiency since, with either, names are already bound at runtime. Further, late binding requires interaction with the clearinghouse at runtime. Although we have designed the clearinghouse to be very reliable, the possibility exists that a client may occasionally be unable to find any clearinghouse server up and able to resolve a reference.

There are therefore advantages and disadvantages to any binding strategy. A useful compromise combines early and late binding, giving the performance and reliability of the former and the flexibility of the latter. The client uses early binding wherever possible, and uses late binding only if any of these (early) bindings becomes invalid. Thus, software supporting printing stores the addresses of print servers at initialization, and updates these addresses only if they become invalid. Of course, the client must be able to recognize if a stored address is invalid (just as it must accept the possibility that the information received from the clearinghouse is temporarily invalid). We discuss hint validation further in Appendix 1.

7.2. Names versus Generic Names

Allied to the question of when to bind is the question of how many levels of indirection a client should use in referring to an object, in particular the question of whether to use names or generic names. For instance, should printing software know the names of printers at *SDD* (such as *Daisy@SDD@Xerox*) or should it use a generic name (such as *Printers@SDD@Xerox* which perhaps maps into the names of printers at *SDD*)? If the client stores actual names, it must accept the possibility that the name is invalid at runtime or that it is missing a new name—even if the client uses a late binding strategy. The advantage of using generic names (and binding them at runtime) is that the client reacts very gracefully to the addition or deletion of objects. The disadvantage is the introduction of yet another level of indirection.

In the next section, we describe an operation for looking up objects by genre. Our generic lookup differs from the Yellow Pages in that it does not require any explicit mappings from generic names into sets of names.

8. Clearinghouse: Client Interface

The clearinghouse provides a basic set of operations, some of which are exported operations which may be called by clients of the clearinghouse by means of the *stub clearinghouse* resident in the client, and some of which are internal operations used by clearinghouse components to communicate with each other. In this section we describe the exported operations. We describe only the most commonly-used operations and do not, for instance, describe the operations required to add new domains and organizations, to change access control lists, etc.

In the following, a name may be either a distinguished name or an alias. With the exception of some operations which allow "wildcard" characters or which take domain or organization names as arguments, all names presented to the clearinghouse must be full names, of the form *localname@domain@organization.*

The operations abort if there are access control violations, but we defer discussion of access control until Section 12.

Notes on the operations are given at the end of this section.

8.1. Names

AddName(name) adds the mapping *name* $\rightarrow$ *{}*, the mapping into the empty set. It aborts if a mapping for *name* already exists.

DeleteName(name) deletes the mapping *name* $\rightarrow$ *{}*, and *name* and all equivalent names (distinguished name and/or aliases) are released. It aborts if a mapping for *name* does not exist, or if *name* is not mapped into the empty set.

ChangeName(name$_1$, name$_2$) changes the distinguished name of an object with name *name*$_1$ to *name*$_2$. If *name*$_1$ is the distinguished name, it is released. If *name*$_1$ is an alias, the corresponding distinguished name is released. *ChangeName* aborts if *name*$_2$ is already a distinguished name or an alias.

AddAlias(newname, oldname) adds *newname* as an alias of *oldname (oldname* may be either an alias or a distinguished name.*)*. More precisely, *AddAlias* adds *newname* to the equivalence class of *oldname. AddAlias* aborts if *oldname* is not known to the clearinghouse or if *newname* is already a distinguished name or an alias.

DeleteAlias(name) deletes *name* from the equivalence class it has been in. *DeleteAlias* aborts if *name* is not an alias of some distinguished name.

LookupDistinguishedName(name) returns the distinguished name equivalent to *name.* (If *name* is already a distinguished name, *LookupDistinguishedName* returns *name.*)

LookupAliases(name) returns all the aliases for *name*, where *name* may be a distinguished name or itself an alias.

8.2. Individuals

LookupIndividual(name, propertyname) finds the mapping *name* → *{..., <propertyname, 0, propertyvalue>, ...}*, if it exists, and returns *propertyvalue*. It aborts if there is no such mapping (and indicates the reason: whether there is no such name, no property with identifier *propertyname*, or if the property identified by *propertyname* is not an individual).

AddIndividual(name, propertyname, propertyvalue) adds the tuple *<propertyname, 0, propertyvalue>* to the set of properties associated with *name*, if a mapping for *name* already exists. *AddIndividual* aborts if no mapping for *name* already exists or if a mapping *name* → *{..., <propertyname, ...>, ...}* already exists.

DeleteIndividual(name, propertyname) deletes the tuple *<propertyname, 0, propertyvalue>* from the mapping for *name*. *DeleteIndividual* aborts if no mapping *name* → *{..., <propertyname, 0, propertyvalue>, ...}* exists.

ChangeIndividual(name, propertyname, propertyvalue) finds the mapping *name* → *{..., <propertyname, 0, oldpropertyvalue>, ...}*, if it exists, and replaces the existing *oldpropertyvalue* with *propertyvalue*. It aborts if no mapping *name* → *{..., <propertyname, 0, oldpropertyvalue>, ...}* already exists.

8.3. Groups

LookupGroup(name, propertyname) finds the mapping *name* → *{..., <propertyname, 1, propertyvalue>, ...}*, if it exists, and returns *propertyvalue*. It aborts if there is no such mapping (and indicates the reason: whether there is no such name, no property with identifier *propertyname*, or if the property identified by *propertyname* is not a group).

AddGroup(name, propertyname, propertyvalue) adds the tuple *<propertyname, 1, propertyvalue>* to the mapping for *name* if a mapping for *name* already exists. *AddGroup* aborts if no mapping for *name* already exists or if a mapping *name* → *{..., <propertyname, ...>, ...}* already exists.

DeleteGroup(name, propertyname) deletes the tuple *<propertyname, 1, propertyvalue>* from the mapping for *name*. *DeleteGroup* aborts if no mapping *name* → *{..., <propertyname, 1, propertyvalue>, ...}* exists.

ChangeGroup(name, propertyname, propertyvalue) finds the mapping *name* → *{..., <propertyname, 1, oldpropertyvalue>, ...}*, if it exists, and replaces the existing *<propertyname, 1, oldpropertyvalue>* with *<propertyname, 1, propertyvalue>*. It aborts if no mapping *name* → *{..., <propertyname, 1, oldpropertyvalue>, ...}* already exists.

8.4. Group Elements

IsMember(element, name, propertyname) finds the mapping *name* → *{..., <propertyname, 1, propertyvalue>, ...}*, if it exists, and determines if *element* is a member of the group *propertyvalue*. It aborts if there is no such mapping.

IsMemberClosure(element, name, propertyname) finds the mapping *name* → *{..., <propertyname, 1, propertyvalue>, ...}*, if it exists, and determines if *element* is a member of the group *propertyvalue*. If so, it returns with success. If not, it calls *IsMemberClosure(element, name, propertyname)* for each element *x* in the set *propertyvalue* with an associated property named *propertyname*. *IsMemberClosure* aborts if there is no mapping *name* → *{..., <propertyname, 1, propertyvalue>, ...}*.

AddMember(element, name, propertyname) finds the mapping *name* → *{..., <propertyname, 1, propertyvalue>, ...}*, if it exists, and adds *element* to the group *propertyvalue*. It aborts if the mapping *name* → *{..., <propertyname, 1, propertyvalue>, ...}* does not exist or if *element* is already a member of *propertyvalue*.

DeleteMember(element, name, propertyname) finds the mapping *name* → *{..., <propertyname, 1, propertyvalue>, ...}*, if it exists, and deletes *element* from the group *propertyvalue*. It aborts if the mapping *name* → *{..., <propertyname, 1, propertyvalue>, ...}* does not exist or if *element* is not a member of *propertyvalue*.

AddSelf(element, name, propertyname) aborts if the originator of the request does not have name *element*, and otherwise is equivalent to *AddMember(element, name, propertyname)*. (See Section 8.8 below.)

DeleteSelf(element, name, propertyname) aborts if the originator of the request does not have name *element*, and otherwise is equivalent to *DeleteMember(element, name, propertyname)*.

8.5. Generic Names

LookupGeneric(name, propertyname), where *name* is of the form *localname@domain@organization*, returns the set of object names which map into properties of the form *name* → *{..., <propertyname, propertytype, propertyvalue>, ...}*. The set may of course be empty. The *localname* component of *name* may optionally contain one or more wildcard characters "*". Each wildcard character may match zero or more characters. Thus, *name* matches an entry in the database if the entry is equal (ignoring case) to *name* with each occurrence of "*" replaced by any string of characters. If *localname* is the single character "*", then *LookupGeneric* returns the set of all object names which map into properties of the form *name* → *{..., <propertyname, propertytype, propertyvalue>, ...}* in the domain *domain@organization*. If *name* contains no occurrence of "*", then *LookupGeneric* returns either the empty set or the singleton set *{name}*.

8.6. Enumeration

EnumerateObjects(name), where *name* is a domain name *domain@organization*, enumerates all names known to the clearinghouse in this domain.

EnumerateDomains(name), where *name* is an organization name, enumerates all names of domains in this organization.

EnumerateOrganizations() enumerates all names of organizations.

EnumerateProperties(name) returns the set *{<propertyname$_1$, propertytype$_1$, propertyvalue$_1$>, ... <propertyname$_k$, propertytype$_k$, propertyvalue$_k$>}* that *name* maps into.

8.7. Notes on these Operations

Strictly speaking, the clearinghouse requires only a very few commands, for reading, adding, and deleting entries. We provide many different operations, in particular, different commands for different types (for instance, different commands to add an individual and to add a group) and for different levels of granularity (for instance, different commands for adding groups and adding elements to a group). We give different operations for different types to provide a primitive type-checking facility. We give different operations for different levels of granularity for three reasons. First, it minimizes the data that must be transmitted by the clearinghouse or the client when reading or updating an entry. Second, it allows different clients to change different parts of the same entry at the same time. For instance, two clients may add different elements to the same group simultaneously using the *AddMember* command; if each were required to update the whole entry, their two updates would conflict (this is described further in Section 11). Third, as we shall see in Section 12, we make use of the different operations for different levels of granularity in our access control facility. Finally, we provide separate operations for changing an entry or sub-entry although these operations are functionally equivalent to deleting the original entry and adding the changed entry. However, changing an entry constitutes one, indivisible transaction; deleting and adding an entry constitute two transactions separated by a period during which another client may try to read the incorrectly-empty entry.

Names are explicitly, rather than implicitly, registered and deleted. Typically, systems administrators for a particular domain will add and delete names, but may allow users to modify some of the properties associated with names.

IsMemberClosure is the closure of *IsMember*. An example of its use is in the use of membership lists. An example of a membership list might be:

```
ClearinghouseInterest@SDD@Xerox → {
    <"Descriptive Comment", 0, "List of those interested in the Clearinghouse Design">,
    <"Distribution List", 1, {"ClearinghouseDesigners@SDD@Xerox",
        "ClearinghouseSupport@SDD@Xerox"}>}

ClearinghouseDesigners@SDD@Xerox → {
    <"Distribution List", 1, {"Dalal@SDD@Xerox", "Oppen@SDD@Xerox"}>}

ClearinghouseSupport@SDD@Xerox → {
    <"Distribution List", 1, {"ClearinghouseImplementers@SDD@Xerox",
        "ClearinghouseMaintainers@SDD@Xerox"}>},
```

where *ClearinghouseImplementers* and *ClearinghouseMaintainers* map into further distribution lists. To check if *Dalal@SDD@Xerox* is on the *ClearinghouseInterest* distribution list, we call

IsMemberClosure("Dalal@SDD@Xerox", "ClearinghouseInterest@SDD@Xerox", "Distribution List") which automatically checks the *"Distribution List"* group associated with *ClearinghouseInterest*. Since *Dalal* is not an entry, *IsMemberClosure* then checks any subsets of this entry, any of their subsets, and so on. *IsMemberClosure* allows circularities (a list may contain another which in turn contains the first).

AddSelf and *DeleteSelf* provide additional access control to *AddMember* and *DeleteMember*, and are discussed further in Section 12. They are used, for instance, in electronic mail to allow users to add and delete themselves from distribution lists.

LookupGeneric provides a primitive "Yellow Pages" facility. As we shall see in Section 13 on clearinghouse administration, property names are centrally allocated to provide consistency across domain boundaries. For instance, the property name *"Printer"* is reserved as the name of the network address of printers. (Recall our previous example showing the entry for the printer *Daisy: Daisy@SDD@Xerox → {<"Printer", 0, network address of the printer named Daisy>}.)* To find the printers in *SDD@Xerox*, a client calls *LookupGeneric("*@SDD@Xerox", "Printer")*. This "Yellow Pages" facility is fairly simple but considerably less expensive than, for instance, a relational database. *LookupGeneric* suffices for the purposes we envision. For example, when a new user is being registered, part of the registration dialogue involves choosing his preferred local printer, preferred mail server, etc. The *LookupGeneric* operation allows the user (or system administrator) to enumerate all printers or all mail servers, and choose among them. Typically, the mapping for each printer or mail server will also contain descriptive comments (describing in what room the server is located, if it prints in color, etc.) which helps him in making the appropriate choices.

The *LookupGeneric, EnumerateDomains* and *EnumerateOrganizations* operations also provide a "directory" service to users. For instance, the Xerox 8010 Star workstations provide a directory service by means of which a user can list network-based resources. The directory gives the user a window into the clearinghouse database, and is implemented by means of generic lookup.

As we will see in Section 13 on clearinghouse administration, we standardize the use of property names. Thus all clients of the clearinghouse agree on the use of each generic name, in much the same way that the telephone system uses (roughly) the same generic names (such as "Automobile Repair") in all the Yellow Pages it publishes. The advantage of this is that client software can request a service in a standardized fashion, and need not be tailored to a particular environment with a particular set of names for servers supplying this service.

For instance, each user workstation generally has a piece of software that replies to the user command "Help!" This software accesses some server to obtain the information needed to help the user. (The software could store all this information itself, but it is more reasonable to store it in a central location, both to reduce the size of the workstation storage and to make updating the information easier.) Suppose the generic name "Help Service" is agreed upon as the standard property name for such a service. To find the addresses of the servers providing help to users in *SDD@Xerox*, the workstation software calls *LookupGeneric("*@SDD@Xerox", "Help Service")*. The workstation then calls *LookupIndividual* to find the addresses. This piece of code can be used by any workstation, regardless of its location.

The "wildcard" feature of *LookupGeneric* allows clients to find valid names where they have only partial information on or can only guess the name. It is particularly useful in electronic mail and in other uses of user names. Recall our discussion in Section 5 on the use of aliases to make the naming convention as friendly as possible. The problem with using aliases is the tradeoff between the number of aliases stored (to make sure that any reasonable alias works) and the time and space required to find and store these aliases. It is unlikely that we can afford to store all plausible aliases. Further, we want to be able to respond gracefully to a user's use of an ambiguous name. For instance, we want to do more than just reject a piece of mail addressed to "Smith" if there is more than one Smith.

If *LookupIndividual("Smith", "Mail Server")* fails, because "Smith" is ambiguous, the electronic mail system may choose to call *LookupGeneric("*Smith*", "SDD@Xerox", "Mail Server")* to find the set of user names matching this name. It presents this set to the sender of the mail and allows him to choose which unambiguous name is appropriate. A simple algorithm to use in general might be to take any string provided by the user, surround the string with *s, delete any periods, and replace any occurrence of <blank> by *<blank>. Thus *Yogen K. Dalal* becomes **Yogen* K* Dalal**, which matches *Yogen Kantilal Dalal*, as desired.

9. Clearinghouse: Structure

We now describe how the clearinghouse is structured internally. This description will be augmented in Section 12 when we discuss access control.

9.1. Clearinghouse Servers

The database of mappings is decentralized. Copies of portions of the database are contained in *clearinghouse servers* which are servers (or perhaps services on servers) spread throughout the internetwork. We refer to the union of all these clearinghouse servers as "the clearinghouse." Each clearinghouse server is a named object in the internetwork, and so has a distinguished name and possibly aliases as well.

As stated in Section 7, we assume that every client of the clearinghouse contains a clearinghouse component, called a *stub clearinghouse.* Stub clearinghouses provide a uniform way for clients to access the clearinghouse. Stubs provide at least the operations described in Section 8. Stub clearinghouses do not have names (although they will typically be on machines containing named objects). Stubs are required to store the address of at least one clearinghouse server, or at least to be able to find one, perhaps by local or directed broadcast ([Boggs 1981]).

9.2. Domain and Organization Clearinghouses

Each clearinghouse server may contain any portion of the global database (subject, as we shall see, to access control). However, this decentralization is not totally arbitrary. Some clearinghouse servers accept responsibility for maintaining specific portions of the global database.

Corresponding to each domain *D* in each organization *O* are one or more clearinghouse servers each containing a copy of all mappings for every name of the form *⟨anything⟩@D@O*, that is, the mappings for all objects in domain *D@O*. Each such clearinghouse server is called a *domain clearinghouse* for *D@O*. (Each clearinghouse server that is a domain clearinghouse for *D@O* may contain other portions of the database other than just the database for this domain, and each of the domain clearinghouses for *D@O* may differ on what other portions of the global database, if any, they contain.) There is at least one domain clearinghouse for each domain in the distributed environment. Domain clearinghouses are addressable objects in the internetwork and hence have names. Each domain clearinghouse for each domain in organization *O* has a name of the form *⟨anything⟩@O@ClearinghouseServers* which maps into the network address of the server, under property name *Clearinghouse Location.* (*ClearinghouseServers* is a reserved organization name.) Thus, if *L@O@ClearinghouseServers* is the name of a domain clearinghouse for *D@O*, then there is a mapping of the form *L@O@ClearinghouseServers* → *{..., ⟨"Clearinghouse Location", 0, network address⟩, ...}*.

For each domain *D@O*, we require that *D@O@ClearinghouseServers* map into the set of names of domain clearinghouses for *D@O*, under property name *Members*. For example, if the domain clearinghouses for domain *D@O* have names L_1*@O@ClearinghouseServers*, L_2*@O@ClearinghouseServers*, ..., L_k*@O@ClearinghouseServers*, then there is a mapping of the form *D@O@ClearinghouseServers* → *{..., <"Members", 1, {*L_1*@O@ClearinghouseServers,* L_2*@O@ClearinghouseServers, ...,* L_k*@O@ClearinghouseServers}>, ...}*. For each *i* and *j*, L_i*@O@ClearinghouseServers* is a *sibling* of L_j*@O@ClearinghouseServers for domain D@O*. Thus, we have given a name to the set of sibling domain clearinghouses for each domain in organization *O*.

Note that the names of all domain clearinghouses, and all sets of sibling domain clearinghouses, for domains in organization *O* are themselves names in the reserved domain *O@ClearinghouseServers*. We will call each domain clearinghouse for this reserved domain an *organization clearinghouse for O* since it contains the name and address of every domain clearinghouse in the organization. In particular, if L_1*@O@ClearinghouseServers,* L_2*@O@ClearinghouseServers, ...,* L_k*@O@ClearinghouseServers* are the domain clearinghouses for any domain *D@O*, then each organization clearinghouse for *O* contains the mappings *D@O@ClearinghouseServers* → *{..., <"Members", 1, {*L_1*@O@ClearinghouseServers,* L_2*@O@ClearinghouseServers, ...,* L_k*@O@ClearinghouseServers}>, ...},* L_1*@O@ClearinghouseServers* → *{..., <"Clearinghouse Location", 0, network address>, ...}, ...,* L_k*@O@ClearinghouseServers* → *{..., <"Clearinghouse Location", 0, network address>, ...}*.

Since *O@ClearinghouseServers* is a domain, there is at least one domain clearinghouse for *O@ClearinghouseServers* and hence at least one organization clearinghouse for *O*. Each such clearinghouse has a name of the form *<anything>@ClearinghouseServers@ClearinghouseServers* which maps into the network address of the server, under property name *Clearinghouse Location*. Thus, if *L@ClearinghouseServers@ClearinghouseServers* is the name of a domain clearinghouse for *O@ClearinghouseServers* (that is, an organization clearinghouse for *O*), then there is a mapping of the form *L@ClearinghouseServers@ClearinghouseServers* → *{..., <"Clearinghouse Location", 0, network address>, ...}*. For each organization *O*, we require that *O@ClearinghouseServers@ClearinghouseServers* map into the set of names of organization clearinghouses for *O*, under property name *Members*. For example, if the organization clearinghouses for *O* have names L_1*@ClearinghouseServers@ClearinghouseServers,* L_2*@ClearinghouseServers@ClearinghouseServers, ...,* L_k*@ClearinghouseServers@ClearinghouseServers,* then there is a mapping of the form *O@ClearinghouseServers@ClearinghouseServers* → *{..., <"Members", 1, {*L_1*@ClearinghouseServers@ClearinghouseServers,* L_2*@ClearinghouseServers@ClearinghouseServers, ...,* L_k*@ClearinghouseServers@ClearinghouseServers}>, ...}*. Each L_i*@ClearinghouseServers@ClearinghouseServers* is called a *sibling* of L_j*@ClearinghouseServers@ClearinghouseServers for organization O*. Thus, we have given names to the set of sibling organization clearinghouses for each organization *O*.

Note that each organization clearinghouse for *O* points directly to every domain clearinghouse for any domain in *O*, and hence indirectly to every object with a name in *O*.

The clearinghouse reserves the organization name *ClearinghouseServers*, and within it the domain name *ClearinghouseServers*, in order to name clearinghouse servers. The clearinghouse also reserves the property name *Clearinghouse Location*, and uses the property name *Members*.

Note the distinction between *clearinghouse servers* on the one hand and *domain* and *organization* clearinghouses on the other. The former are physical entities that run code, contain databases and are physically resident on network servers. The latter are logical entities, and are a convenience for referring to the clearinghouse servers which contain specific portions of the global database. A particular clearinghouse server may be a domain clearinghouse for zero or more domains, and an organization clearinghouse for one or more organizations.

9.3. Interconnections between Clearinghouse Components

As we have just seen, organization clearinghouses point "downwards" to domain clearinghouses, which point "downwards" to objects. Further interconnection structure is required so that stub clearinghouses can access clearinghouse servers, and clearinghouse servers can access each other. (The communication flow between clearinghouse components is described in Sections 10, 11 and 13.)

First, each clearinghouse server is required to be an organization clearinghouse for the reserved organization *ClearinghouseServers*, and hence each clearinghouse server points "upwards" to *every* organization clearinghouse. Thus, for each organization *O*, each clearinghouse server contains a copy of the mappings *O@ClearinghouseServers@ClearinghouseServers* → *{..., <"Members", 1, {*L_1*@ClearinghouseServers@ClearinghouseServers,* L_2*@ClearinghouseServers@ClearinghouseServers, ...,* L_k*@ClearinghouseServers@ClearinghouseServers}>, ...},* L_1*@ClearinghouseServers@ClearinghouseServers* → *{..., <"Clearinghouse Location", 0, network address>, ...}, ...,* L_k*@ClearinghouseServers@ClearinghouseServers* → *{..., <"Clearinghouse Location", 0, network address>, ...}*. In this way, each clearinghouse server knows the name and address of every organization clearinghouse.

Finally, stub clearinghouses contain the address of at least one clearinghouse server or at least can always find a clearinghouse server, perhaps by local or directed broadcast. If they store an address, it is typically, but not necessarily, the address of the closest clearinghouse server. We do not require a clearinghouse server to keep track of which stub clearinghouses point to it, so these stubs will not be told if the server changes location, and must rely instead on other facilities, such as local or directed broadcast, to find a clearinghouse server if the stored address becomes invalid. The reasons we do not require stubs to follow the same style of interconnection as clearinghouse servers is that the set of stubs may be exceedingly large; these stubs may be in machines which are powered off as often as they are powered on (or are "offline"), making it difficult to update them in reasonable time anyway; and we do not expect clearinghouse servers to be added, deleted or moved very often. Storing the address of a server in a stub versus relying on local broadcast has the same performance-flexibility tradeoff discussed in Section 7.1 on binding strategies.

9.4. Interconnections between Organization Clearinghouses

We have described how organizations are logically connected by the clearinghouse structure. Clearinghouse information may flow freely between all the domain and organization clearinghouses. That is, each domain and organization clearinghouse may request and store a copy of any mapping held by any other domain and organization clearinghouse (subject to access control). This model of *trust* among clearinghouse servers is appropriate within an organization (corresponding to a corporate entity or sub-entity) and between organizations which trust each other and wish to share information freely. (In Section 12.3 we discuss "arm's length" security between organizations.)

Trust is an equivalence relation. An organization trusts itself. If one organization trusts another, then the trust is reciprocated. If one organization trusts another which in turn trusts a third, then the first also trusts the third. (That trust is symmetric is required so that trusted clearinghouses can hear of updates. Making trust an equivalence relation makes implementation of access control easier.)

It is possible for a clearinghouse server that is a domain clearinghouse for a domain in one organization also to be a domain clearinghouse for a domain in another organization. For example, a clearinghouse server may be a domain clearinghouse for one or more domains in the organization *Versatec*, and domain clearinghouse for one or more domains in the organization *Xerox*. It then has, by the above, at least two names (one of them its distinguished name). If the server is physically resident in Versatec, its names might be *Registry@Versatec@ClearinghouseServers* and *SDDRegistryAtVersatec@Xerox@ClearinghouseServers*, with the former its distinguished name.

9.5. Summary

Each clearinghouse server contains mappings for a subset of the set of names. If it is a domain clearinghouse for domain *D* in organization *O*, it contains mappings for all names of the form *<anything>@D@O*. If it is an organization clearinghouse for organization *O*, it contains mappings for all names of the form *<anything>@O@ClearinghouseServers* (names associated with domains in *O*). Each clearinghouse server contains the mappings for all names of the form *<anything>@ClearinghouseServers@ClearinghouseServers* (names associated with organizations); that is, the database associated with the reserved domain *ClearinghouseServers@ClearinghouseServers* is replicated in every clearinghouse server. Stubs point to any clearinghouse server.

For every domain *D* in an organization *O*, *D@O@ClearinghouseServers* names the set of names of sibling domain clearinghouse servers for *D@O*. For every organization *O*, *O@ClearinghouseServers@ClearinghouseServers* names the set of names of sibling organization clearinghouse servers for *O*. (We make no use of the name *ClearinghouseServers@ClearinghouseServers@ClearinghouseServers*, which contains the set of names of *all* clearinghouse servers.)

This clearinghouse structure allows a relatively simple algorithm for managing the decentralized clearinghouse (see Section 13). However, it does require that copies of the mappings for all names of the form *<anything>@ClearinghouseServers@ClearinghouseServers* be stored in *all* clearinghouse servers. In Appendix 2 we describe an alternative structure that localizes information about clearinghouse servers, but which requires a more complicated algorithm for adding or moving clearinghouse servers.

10. Clearinghouse: Distributed Lookup Algorithm

We now briefly describe how clearinghouse components communicate with each other in response to a call to *LookupIndividual* or *LookupGroup.*

Suppose that a stub clearinghouse receives the query *LookupIndividual("A@B@C", propertyname).* The stub clearinghouse follows the following general protocol.

The stub clearinghouse contacts a clearinghouse server and passes it the query. (Recall that each stub clearinghouse stores the address of at least one clearinghouse server or can find one through local or directed broadcast.)

If the clearinghouse server that receives the stub's query is a domain clearinghouse for *B@C*, it can immediately return the answer to the stub who in turn returns it to the client.

Otherwise, the clearinghouse server returns the names and addresses of the organization clearinghouses for *C*, which it is guaranteed to have. The stub contacts any of these clearinghouse servers. If this clearinghouse server happens also to be a domain clearinghouse for *B@C*, it can immediately return the answer to the stub who in turn returns it to the client.

Otherwise the clearinghouse server returns the names and addresses of the domain clearinghouses for *B@C*, which it is guaranteed to have. The stub contacts any of these, since any of them is guaranteed to have the answer.

The domain clearinghouse for *B@C* that returns the answer to the query does so after authenticating the requestor and ascertaining that the requestor has appropriate access rights.

In the worst case, a query conceptually moves "upwards" from a stub clearinghouse to a domain clearinghouse to an organization clearinghouse, and then "downwards" to one of that organization's domain clearinghouse, The number of clearinghouse servers that a stub has to contact will never exceed three: the clearinghouse server whose address it knows, an organization clearinghouse for the organization containing the name in the query, and a domain clearinghouse in that organization.

However, before sending the query "upwards," each clearinghouse component *optionally* first sees if it can shortcut the process by sending the query "sideways," cutting out a level of the hierarchy. (This is similar to the shortcuts used in the routing structure of the telephone system.) These "sideways" pointers are cached pointers, maintained for efficiency. For instance, consider domain clearinghouses for *PARC* and for *SDD*, two logical domains within organization *Xerox*. Depending on the traffic, it may be appropriate for the *PARC* clearinghouse to keep a direct pointer to the *SDD* clearinghouse, and vice versa. This speeds queries that would otherwise go through the *Xerox* organization clearinghouse. To increase the speed of response even further, each clearinghouse server could be a domain clearinghouse for both domains. Alternatively, if the number of domains in *Xerox* is relatively small, it may be appropriate to make each clearinghouse server in *Xerox* an organization clearinghouse for *Xerox*. In this way, each clearinghouse server in *Xerox* always points to every other clearinghouse server in *Xerox.*

Local queries (that is, queries about names "logically near" the stub clearinghouse) will typically be answered more quickly than non-local queries. That is appropriate. The caching mechanism (for storing of "sideways" pointers) can be used to fine-tune clearinghouse servers to respond faster to non-local but frequent queries.

To make the lookup process clearer, let us present it as a quasi-algorithm. To simplify the following, we assume that there is in fact a mapping *A@B@C* → *{..., <propertyname, 0, propertyvalue>, ...}* and that the originator of the request has the appropriate access right to receive it. A clearinghouse component can (1) *Find* the mapping for a name in its own database, or (2) *Contact* a clearinghouse server, or (3) *Return* the mapping for *A@B@C*, a failure message or the name and address of another clearinghouse server.

LookupIndividual("A@B@C", propertyname):

1. ***Stub:***
1.1. *Contact* any clearinghouse server *(Go to 2).*

2. ***Some** clearinghouse server:*
2.1. If this server is a domain clearinghouse for *B@C*, *Return propertyvalue* to the stub, which in turn returns it to the client.
2.2. *Find* the names and addresses of domain clearinghouses for *B@C*. That is, *Find B@C@ClearinghouseServers* → *{..., <"Members", 1, {*L_1*@C@ClearinghouseServers, ...,* L_k*@C@ClearinghouseServers}>, ...}*. If successful, *Find* L_i*@C@ClearinghouseServers* → *{..., <"Clearinghouse Location", 0, networkaddress>, ...}* for all *i* from *1* to *k*. If successful, *Return* those names and addresses found to the stub, which in turn *Contacts* one of the servers (*Go To 4*).
2.3. *Find* the names and addresses of organization clearinghouses for *C*. That is, *Find C@ClearinghouseServers@ClearinghouseServers* → *{..., <"Members", 1, {*L_1*@ClearinghouseServers@ClearinghouseServers, ...,* L_k*@ClearinghouseServers@ClearinghouseServers}>, ...}*. *Find* L_i*@ClearinghouseServers@ClearinghouseServers* → *{..., <"Clearinghouse Location", 0, networkaddress>, ...}* for all *i* from *1* to *k*. This operation is guaranteed to be successful. *Return* the names and addresses found to the stub, which in turn contacts one of the servers (*Go To 3*).

3. ***Organization** clearinghouse for organization C:*
3.1. If this server is a domain clearinghouse for *B@C*, *Return propertyvalue* to the stub, which in turn returns it to client.
3.2. *Find* the names and addresses of domain clearinghouses for *B@C*, as in step 2.2. This operation is guaranteed to be successful. *Return* the names and addresses found to the stub, which in turn *Contacts* one of the servers (*Go To 4.*).

4. ***Domain** clearinghouse for domain B@C:*
4.1. *Find A@B@C* and *Return propertyvalue* to the stub, which in turn returns it to client.

The algorithm is the most-general, worst-case algorithm. Step 2.2 may be deleted if clearinghouse servers do no caching of "sideways" pointers. If *A@B@C* is an alias for *D@E@F*, the domain clearinghouse for *B@C* resolves this indirection without involving the stub. Similarly, if the name of a clearinghouse server (in steps 2.2, 2.3 or 3.2) is an alias, the indirection is resolved before returning the names and addresses to the stub.

10.1. Caching

We allow clearinghouse components to "cache" entries for efficiency. Let us consider briefly when it is appropriate to do so.

As we have seen, it may be useful to cache the address of a clearinghouse server, if the path to that server is often taken and if caching its address reduces the time required to find its address. The latter depends partially on whether it is cheaper to look up a name in a database than it is to communicate with another clearinghouse server, and partially on the probability of finding the entry cached.

There are several ways for a clearinghouse server to decide which addresses to cache. It may rely on the stubs to tell it the addresses of those clearinghouse servers they access most often. Alternatively, it may keep track of the queries it receives from stubs, and find and store the addresses of those clearinghouse servers able to answer the most common requests. For instance, if a clearinghouse server receives many queries of the form *<anything>@B@C*, it may profitably cache the names and addresses of domain clearinghouses for *B@C*.

Cached entries are not maintained by the clearinghouse. Therefore, in the case of clearinghouse server addresses, there is a possibility, if a clearinghouse server caches the address of another clearinghouse server and gives it to a stub, that the address will be invalid by the time it is used. This is not a serious problem; if it occurs, the stub tells the server to delete the cached entry and use the general algorithm described above instead.

In general, it is not appropriate for a stub or clearinghouse server to cache anything other than clearinghouse server addresses since it has to ensure that the client can easily tell if the cached entry is invalid.

11. Clearinghouse: Update Algorithm

The distributed update algorithm we use to alter the clearinghouse database is closely related to the distributed update algorithm used by Grapevine's registration service [Birrell, Levin, Needham, and Schroeder 1981].

The basic model is quite simple. Assume that a client wishes to update the clearinghouse database, for instance by a call to *AddMember("ClearinghouseReviewers@SDD@Xerox", "ClearinghouseInterest@SDD@Xerox", "DistributionList")*. The request is submitted via the stub resident in the client. The stub contacts any domain clearinghouse containing the mapping to be updated (in our example, any domain clearinghouse for *SDD@Xerox*) using *LookUp* to find its address. The domain clearinghouse authenticates the requestor and checks to see if the request is a valid one. If not, it refuses the update request. Otherwise, it updates its own database and acknowledges that it has done so. The interaction with the client is now complete. The domain clearinghouse then propagates the update to its siblings if the database for this domain is replicated in more than one server (see Section 11.3).

The propagation of updates is not treated as an indivisible transaction. Therefore, sibling clearinghouse servers may have databases that are temporarily inconsistent; one server may have updated its database before another has received or acted on an update request. This has ramifications, which we now discuss.

11.1. Choosing among Conflicting Updates

The possibility exists that two clients may submit conflicting update requests to two sibling servers at roughly the same time. For instance, one may ask one server to change a mapping at roughly the same time the other asks another to change it differently. Each server may execute the update and send an acknowledgement to the client, not knowing that the other server is doing the same thing with a conflicting update on behalf of another client. It is only when each server propagates to its siblings the update request that the inconsistency is noticed. We do not mind that the two servers' databases are temporarily inconsistent, but we do want the inconsistencies to be resolved. Therefore, the clearinghouse automatically arbitrates between conflicting updates, as follows. Each server which receives an update request from a client timestamps the request before propagating it to its siblings. Upon receiving an update request, whether directly from a client or from a sibling, a server stores it together with its timestamp. When the server receives a new update request, it compares its timestamp with those of the requests it has already acted upon. If the new request is subsumed by a previously-received request, it is thrown away. Otherwise, it is carried out. (We define "subsumption" in the obvious way. A request to delete a set subsumes any request to alter its contents, a request to change an element of a group subsumes any older request to change the element, and so on.)

An important consideration is the level of granularity of an update. Recall that we specifically introduced "group of names" as a distinct data type, and operations to add and delete elements of groups. A primary reason for introducing operations at the level of "element of group" was to make the handling of distribution lists as easy as possible. For instance, two users may update a distribution list (perhaps by adding themselves using *AddSelf*) at roughly the same time, without having to know that the other is presently updating the same list and without the distributed update algorithm's overriding one update in favor of another (unless they conflict).

Since all update requests are timestamped in a consistent fashion, all sibling servers will agree on their databases as soon as all update requests have been received by all of them. However, there is a slight possibility that an update request directed to a particular server will never reach its destination, so we need one additional mechanism. Every so often, perhaps once a day or once a week, all the sibling clearinghouses for a given portion of the database compare their databases and use the saved update requests with their timestamps to resolve any conflicts. In this way, we maintain consistency (at least among those servers that are up).

The above mechanisms do not ensure that "the most recent update" always wins, because the timestamps are generated by different servers, and their internal clocks may disagree. The disagreement will generally be slight—there are techniques for servers to synchronize their clocks ([Boggs 1981])—but the possibility certainly exists that "the most recent update" does not win. All that is guaranteed is that sibling clearinghouses will eventually agree on their databases as a result of a series of update requests. This suffices. Conflicting updates presented simultaneously is an indication of a problem in coordination outside the clearinghouse.

Thus, the most recent update generally wins. This may not always be appropriate. For instance, if two system administrators register the same name at roughly the same time, presumably the first one to register the name should win. However, we prefer to use one consistent arbitration mechanism, and leave it to the system administrators to improve their coordination.

Storing update requests, including deletion requests, with their timestamps is necessary for consistent arbitration. We naturally want to throw them away as soon as possible. A server can throw away a stored request if the request has been subsumed by a later request (for instance, it can throw away a request to alter an entry if another, later request is received to delete or further alter the entry). It can also throw away a stored request if it and all its siblings have agreed that their databases are consistent.

There remains the possibility that a sibling will be down or inaccessible for an inconsiderate amount of time. We let update requests have only a finite lifespan, of perhaps a week. No request with an expired timestamp will be honored by any server. Further, no server need store update requests with expired timestamps. As a result, if a server resumes service after more than a week of inactivity, it should re-initialize itself.

11.2. Out-of-order Update Requests

We have thus far ignored an important problem—the problem of updates arriving out of order. A clearinghouse server may receive an update request which is inconsistent with its existing database: for instance, a request to delete an item that does not exist. If the request is a direct request from a client, the server refuses the request as described in Section 8. However, if the request is a request propagated by a sibling clearinghouse, the correct response is not so clear. One possibility is to ignore the request, and rely on the once-a-day or once-a-week comparison of databases to resolve the issue. Another possibility is to complain to the originating server and ask it to justify its request. We take a third approach, as follows.

Each server trusts any of its siblings to validate any request relative to its own database before propagating the request. Each request therefore makes sense to at least one sibling. If a server receives a request from a sibling that is inconsistent with its database, it assumes that its sibling is correct, and that the inconsistency is due to one or more lost or late update requests. It therefore carries out the request and updates its database in the appropriate fashion. For instance, if it receives a request to delete a nonexistent entry, it does nothing. If it receives a request to alter a nonexistent entry, it treats the request as a request to add the entry as given. If it receives a request to add an already-existing entry, it treats the request as a request to change the existing entry. When the missing requests finally arrive, they will be ignored since they have been subsumed; their timestamps are outdated.

11.3. Updating Sibling Clearinghouses

An organization clearinghouse knows the names and addresses of all its siblings, while a domain clearinghouse can find the names and addresses of its siblings from any of its organization clearinghouses.

Large distributed systems typically have an electronic mail delivery facility that can be used to propagate update requests among clearinghouse servers. Mail delivery systems allow a message to be sent even though the intended recipient of the message is not ready to receive it; the message is stored in a *mail server* until the recipient is ready. There is therefore a possible possible time-lag between the sending and the receipt of a message. Since our clearinghouse design does not require that updating be an indivisible process, this does not worry us. Using a mail system to propagate update messages has many advantages (see [Birrell, Levin, Needham, and Schroeder 1981]); one of them is that they automatically queue up undeliverable messages and try again later.

Clearinghouse servers, therefore, send their siblings timestamped update *messages* telling them of changes to the databases. We associate with the name of every clearinghouse server the property *<"Mailbox Location", 0, networkaddress>*. Clearinghouse servers must periodically poll their mailboxes for update messages.

12. Clearinghouse: Security

We restrict ourselves to a brief discussion of three issues. The first two concern protecting the clearinghouse from unauthorized access or modification, and involve *authentication* (checking that you are who you say you are) and *access control* (checking that you have the right to do what you want to do). The third concerns inter-organization security: how two mutually-suspicious organizations can allow their respective clearinghouse servers to interact and still keep them at arms' length. We do not discuss how the network ensures (or does not ensure) secure transmission of data.

12.1. Authentication

When a request is sent by a client to a clearinghouse server to read from or write onto a portion of the clearinghouse database, the request is accompanied by the distinguished name of the client and its password. (This is typically the name and password of the human user logged in at the workstation originating the request). If the request is an internal one, from one clearinghouse server to another, the requestor is the name of the originating clearinghouse server and its password. (The clearinghouse thus makes sure that internally-generated updates come only from "trusted" clearinghouse servers.) The clearinghouse maintains a database mapping names of clients and other clearinghouse servers into their valid passwords, and compares the name-password pair appearing in any request against that stored in its database.

The clearinghouse authentication mechanism is available as a service to clients to authenticate *their* clients if they so desire. We do not describe authentication or the operations exported by the clearinghouse to support client authentication further.

12.2. Access Control

Once a client has been authenticated, it is granted certain privileges. Access control is provided at the domain level and at the property level. Access control is not provided at the mapping (set of properties) level nor at the level of element of a group.

Associated with each domain and each property is an *access control list*, which is a set of the form *{<set of names$_1$, set of operations$_1$>, ..., <set of names$_k$, set of operations$_k$>}*. Each tuple consists of a set of names and the set of operations each client in the set may call.

The algorithm for finding a name in an access control list uses *IsMemberClosure*, so names of groups may appear in the set of names. The property name used by *IsMemberClosure* is *Members*. The wildcard character "*" (an asterisk) may be used in the localname, domain name or organization name components of a name in an access control list; it matches any string of characters (ignoring case). The algorithm for finding a name (or asterisk matching a name) in an access control list successively checks each set of names until it finds one which lets the user do the requested operation.

Certain operations that modify the clearinghouse database are protected only at the domain level. These are operations that are typically executed only by *domain system administrators* and other clearinghouse servers. (We assume that there is *system administration software* that is a client of the clearinghouse and provides a friendly user interface to a system administrator.) The operations are:

1. Add a new name *(AddName)*
2. Delete a name *(DeleteName)*
3. Change a name *(ChangeName)*
4. Add an alias for a name *(AddAlias)*
5. Delete an alias for a name *(DeleteAlias)*
6. Add an individual or group *(AddIndividual, AddGroup)*
7. Delete an individual or group *(DeleteIndividual, DeleteGroup)*
8. Enumerate all properties for a name *(EnumerateProperties)*
9. Enumerate all names in a domain *(EnumerateObjects)*

Other clients may perform the following operation:

1. *LookupDistinguishedName*
2. *LookupAliases*
3. *EnumerateDomains*
4. *EnumerateOrganization*

For example, the access control list for the domain *SDD@Xerox* might be:

*{<{DomainSystemAdministrators@SDD@Xerox, *@Xerox@ClearinghouseServers},
{AddName, DeleteName, ChangeName, AddAlias, DeleteAlias, AddIndividual,
AddGroup, DeleteIndividual, DeleteGroup, EnumerateProperties, EnumerateObjects}>,
<{*@*@Xerox}, {LookupDistinguishedName, LookupAliases, EnumerateDomains,
EnumerateOrganization}>}.*

In this example, only a domain system administrator in *SDD@Xerox* (that is, someone whose name is in the *Members* group associated with name *SystemAdministrators@SDD@Xerox*) or a domain clearinghouse in the organization *Xerox* may modify the domain database or enumerate all objects, but anyone in the organization *Xerox* may enumerate other kinds of names.

The remaining operations are protected at the property level. The options for access control depend on whether the property being protected is of type *individual* or of type *group*.

The operations that a client other than a domain system administrator may execute on a property of type *individual* are *LookupIndividual* and *ChangeIndividual*. For example, the access control list for the address of a print server might be:

*{<{DomainSystemAdministrators@SDD@Xerox, PrintServerAdministrators@SDD@Xerox},
{LookupIndividual, ChangeIndividual}>,
<{*@SDD@Xerox}, {LookupIndividual}>}.*

In this example, only a domain system administrator or a print server administrator (that is, someone whose name is in the *Members* group associated with name *PrintServerAdministrators*) may change the address of the print server, but anyone in *SDD* may obtain the address.

The operations that may be executed by clients to read or modify a property of type *group* are as follows:

1. *LookupGroup*
2. *ChangeGroup*
3. *IsMember*
4. *IsMemberClosure*
5. *AddMember*
6. *DeleteMember*
7. *AddSelf*
8. *DeleteSelf*

For example, suppose we have a distribution list whose "owner" (the person in charge of the distribution list) is *Redell@SDD@Xerox*. The access control list for this distribution list might be as follows:

{<{Redell@SDD@Xerox}, {LookupGroup, ReplaceGroup, IsMember, IsMemberClosure, AddMember, DeleteMember, AddSelf, DeleteSelf}>,
<{@*@*}, {LookupGroup, IsMember, IsMemberClosure, AddSelf, DeleteSelf}>}.*

With this access control list, anyone may add or delete himself or herself from the distribution list, but only Redell may add or delete users other than himself. If only Redell may add people to the distribution list, the access control list might be as follows:

{<{Redell@SDD@Xerox}, {LookupGroup, ReplaceGroup, IsMember, IsMemberClosure, AddMember, DeleteMember, AddSelf, DeleteSelf}>,
<{@*@*}, {LookupGroup, IsMember, IsMemberClosure, DeleteSelf}>}.*

The access control list for the name associated with the set of sibling organization clearinghouses in clearinghouse servers in the organization *Xerox* (that is, names of the form *<anything>@ClearinghouseServers@ClearinghouseServers* with property name *Members*), might be as follows:

{<{OrganizationSystemAdministrators@@Xerox}, {LookupGroup, ReplaceGroup, IsMember, IsMemberClosure, AddMember, DeleteMember}>,*
<{@*@ClearinghouseServers}, {LookupGroup, IsMember, IsMemberClosure, AddMember, DeleteMember, AddSelf, DeleteSelf}>*
<{@*@*}, {LookupGroup, IsMember, IsMemberClosure}>}.*

With this access control list, anyone may find the names and addresses associated with organization clearinghouse servers, any clearinghouse server may add or remove itself or any other member from the group, and organization system administrators may add or delete a member from the group. Similarly, the access control list for the name associated with the set of sibling domain clearinghouses in organization clearinghouses in the organization *Xerox* (that is, names of the form

<anything>@Xerox@ClearinghouseServers with property name *Members*), might be as follows:

> *{<{OrganizationSystemAdministrators@*@Xerox}, {LookupGroup, ReplaceGroup, IsMember, IsMemberClosure, AddMember, DeleteMember}>,*
> *<{Xerox@ClearinghouseServers@ClearinghouseServers}, {LookupGroup, IsMember, IsMemberClosure, AddMember, DeleteMember}>*
> *<{*@Xerox@ClearinghouseServers}, {LookupGroup, IsMember, IsMemberClosure, AddSelf, DeleteSelf}>*
> *<{*@*@*}, {LookupGroup, IsMember, IsMemberClosure}>}.*

Finally, if a client calls *LookupGeneric*, it receives back the group of names for which mappings of the form *{..., <propertyname, propertytype, propertyvalue>, ...}* exist, but only those for which it has *LookupIndividual* or *LookupGroup* rights.

Access control lists themselves must have access control. The name and password of a *super system administrator* is associated with access control lists for this purpose. We do not describe the operations exported by the clearinghouse to support the maintenance of access control lists further.

12.3. Arm's Length Security

We have described in Section 9 how two organization clearinghouses access each other, assuming the two organizations trust each other. Their organization clearinghouses contain direct pointers to each other and information flows freely between the two organizations. A clearinghouse component in one can request, obtain (assuming it has the appropriate access rights) and store a copy of any mapping stored by the other.

If two organizations do not wish to trust each other but do wish to join their networks (for instance, to exchange electronic mail, to propagate non-proprietary information, to submit orders for products, etc.), they need a way to keep their clearinghouse databases at arm's length.

One element of security is described above. Each clearinghouse server refuses to give out information (or accept requests for updates) unless the requestor is authenticated and has the appropriate access rights. Unfortunately, it is difficult to be sure that the name and password supplied in a request to a clearinghouse has not been forged or that the requestor is a bonafide clearinghouse component. The following provides a primitive facility for helping avoid the problem of "forged IDs" by providing a level of indirection between the clearinghouses of the two organizations.

An organization suspicious of a second includes a *clearinghouse sentry* as a component in the internetwork router joining the two internetworks, that is, as a component in the internetwork router server at its end of the link. The organization gives out the address of the sentry, rather than the addresses of any of its clearinghouse servers, to the other organization. The other organization therefore has only an indirect pointer to its clearinghouse; all requests must pass through this clearinghouse sentry. The clearinghouse sentry acts as a filter, and may accept or reject the request. If it accepts the request as bonafide, it repackages it, using itself as the requestor, and passes it on to the appropriate clearinghouse server, using the algorithm described in Section 10. Any clearinghouse server receiving a request from a clearinghouse sentry knows the originator of the request is a suspicious organization, and treats the request accordingly. In particular, it may reply "No comment." Whatever the clearinghouse server decides to do, it sends its reply back to the clearinghouse sentry. The clearinghouse sentry repackages the reply and sends it on to the requestor in the other organization.

We modify the Lookup algorithm given in Section 10 to admit clearinghouse_sentries and extend the use of the reserved organization *ClearinghouseServers*. In particular, an organization stores the set of names of *sibling* clearinghouse sentries for another organization *C* and their address under the name *C@ClearinghouseSentries@ClearinghouseServers*. We modify the algorithm so that if an organization clearinghouse fails to find the address of a foreign organization's clearinghouse, it then checks to see if there is a clearinghouse sentry for that organization. If so, the stub contacts it, treating it exactly like an organization clearinghouse. In particular, Step 2.3 in the algorithm given in Section 10 will fail, and will be extended by:

2.4. *Find C@ClearinghouseSentries@ClearinghouseServers* → *{..., <"Members", 1, {L_1@ClearinghouseSentries@ClearinghouseServers, ..., L_k@ClearinghouseSentries@ClearinghouseServers>, ...}*. If successful, *Find L_i@ClearinghouseSentries@ClearinghouseServers* → *{..., <"Clearinghouse Sentry Location", 0, networkaddress>, ...}* for all *i* from *1* to *k*. If successful, *Return* the names and addresses of the sentry to the stub, which in turn *Contacts* one of the server *(Go to 4)*.

2.5. *Return* (unsuccessfully) to the stub, which in turn returns unsuccessfully to the client.

The above mechanisms are still inadequate if one organization already knows the network address of the other's clearinghouse server. In such cases, a security filter must be built at the internetwork level. That is, the internetwork router connecting two organizations permits entry of only those packets from suspect internetworks that are addressed to acceptable destinations. Alternatively, the clearinghouse server checks the return address included with the request to see if it is from an acceptable source. Both schemes, unfortunately, have many operational problems. The subject of internetwork security is still an open research problem, although the use of public-key encryption algorithms has promise ([Needham and Schroeder 1978]).

In many situations the interaction between two mutually-suspicious organizations will be constrained, and access control may be provided directly at the application level without directly involving the clearinghouse for the two organizations. For example, if two mutually-suspicious organizations *Xerox* and *SomeCompany* wish to exchange electronic mail, they use a mail server to connect their internetworks. When a piece of mail is sent from *Dalal@SDD@Xerox* to *Jones@Research@SomeCompany*, a mail server in Xerox uses the generic name *MailServerForSomeCompany@SDD@Xerox* to find the network address of a generic maildrop for outgoing mail to SomeCompany. (SomeCompany does not mind if the network address of this maildrop is known to the world.) This is similar to the usual way mail flows in the postal system.

13. Clearinghouse: Administration

We now consider the administration of the clearinghouse. An internetwork configuration of several thousand users and their associated workstations, printers, file servers, mail servers, etc., requires considerable management. Administrative tasks include managing the name and property name space; bringing up new networks; deciding how to split an organization into domains (reflecting administrative, geographical, functional or other divisional lines); deciding which objects (users, services, etc.) belong to which domains; adding, changing and deleting services (such as mail services, file services, and even clearinghouse services); adding and deleting users; maintaining users' passwords, the addresses of their chosen local printers, mail and file servers, and so on; and maintaining access lists and other security features of the network. We have designed the clearinghouse so that management can be decentralized as much as possible.

We treat the following issues: management of names, adding a new user, adding a new non-clearinghouse server (perhaps a print server or a file server), and adding a new clearinghouse server. These constitute a representative subset of the tasks of the system administrator. The following are suggested scenarios for clearinghouse administration; they. will typically be tailored to the needs of any particular organization.

13.1. Managing the Name Space

The allocation of names is managed by a *naming authority* which is responsible for making sure that different objects have different names. There are naming authorities for object names, domain names, organization names and property names.

Object names Object names are proposed by system administrators and validated by domain clearinghouses. For example, a domain system administrator for domain *B@C* proposes a name *A@B@C* and adds it to the database (indirectly using *AddName*). A domain clearinghouse for that domain checks that the requestor is a registered system administrator for domain *B@C* and that the name is not already in use. If the clearinghouse succeeds in validating the requestor and the request, it adds the mapping *A@B@C* → *{}*, thus registering the name. Thereafter, the name remains registered until a system administrator deletes it (indirectly, using *DeleteName*). Management of object names is thus decentralized to the domain level.

Domain names Each organization chooses one or more people to be *organization system administrators*. They are responsible for choosing domain names within the organization. They check to see if a domain name is already in use (by querying the clearinghouse), but the clearinghouse provides no special primitives for directly registering domain names. Domain names become registered indirectly when the name of the first domain clearinghouse for a domain is registered. Domains will be added relatively infrequently.

Organization names The organizations which choose to make use of our naming convention will agree on a central naming authority to validate new organization names (in much the same way the various telephone companies agree on the allocation of area code numbers). The only conflicts likely to occur will be among regional companies with the same name. They will have to agree on organization names to avoid ambiguity. Given goodwill on the part of the companies involved, the central naming authority need be nothing more than someone keeping track of organization names. The central naming authority checks to see if an organization name is already in use (by querying the clearinghouse), but the clearinghouse provides no special primitives for directly registering organization names. Organization names become registered indirectly when the name of the first organization clearinghouse for an organization is registered. Organizations will be added infrequently.

Note that if two organizations call themselves by the same organization name, they will not necessarily run into any problems as long as the networks on which their clearinghouse servers reside are never interconnected. However, if their networks are joined, they may contain conflicting mappings for the same name. The clearinghouse updating algorithm will automatically arbitrate in favor of the most recent mapping, which is hardly appropriate. So it is clearly essential for organizations to choose different names if they plan on ever joining their networks to other networks.

Property names Since property names must also be unambiguous, we need a naming authority for property names. We could follow the same model as we have above, and decentralize this naming authority as much as possible. However, as one of our goals is to design the clearinghouse so that networks can be joined smoothly, we prefer to have one centralized naming authority for property names responsible for validating property names throughout the world. In this way, everyone will agree on the property names for mailboxes, etc. The use of such a naming authority is voluntary; organizations may choose to let their domain system administrators choose property names as they wish. The clearinghouse does not enforce property name coordination among organizations, but it is clearly in their interests to do so if they plan on joining their respective networks.

13.2. Adding a New User

The system administrator first registers the user's full name by adding it to the database (that is, software calls *AddName* on behalf of the system administrator). (In the unlikely event that the name is already in use, the system administrator asks the user to choose another closely-related name.) The system administrator, in collaboration with the user, next adds appropriate aliases for the user's full name and deletes any existing aliases that are now likely to cause confusion. The system administrator, in collaboration with the new user, then registers, in the user's profile, his or her password, his or her privileges, the names of his or her preferred mail servers, file servers, etc. (These are, of course, only representative of the tasks needed to add a new user.)

13.3. Adding a New Server

Suppose we wish to add a new non-clearinghouse server (perhaps a file server or a print server) to an existing logical network which has a clearinghouse server installed.

We assume that the server has been installed, that it knows its network address (perhaps because it has been told its network address by the system administrator, or has acquired it from its operating system), that it has a stub clearinghouse, and that the stub either knows the address of a clearinghouse server or can find one, perhaps by local or directed broadcast.

The system administrator carries on the following dialogue. The system administrator tells the server its distinguished name, and instructs the stub to register the server with the appropriate domain clearinghouse server. Registration involves storing at least the name of the server, its password and its network address. (That is, software makes the appropriate calls to functions such as *AddName* and *AddIndividual* on behalf of the system administrator.)

13.4. Adding a New Clearinghouse Server

We may wish to add a new clearinghouse server (or a "service" on an existing server) to serve as an organizational and/or domain clearinghouse for a new organizational installation or to serve as a sibling (replicated server) for one or more existing domains or organizations.

We assume that the server has been installed, that it initially has nothing registered in it, and that it knows its network address (perhaps because it has been told its address by the system administrator, or has acquired it from its operating system).

If the new clearinghouse server is in a network containing other clearinghouse servers, then the new clearinghouse server is told the address of an existing server. This address may be obtained through local or directed broadcast (in much the same way that a stub discovers the network address of a clearinghouse server), or the system administrator may supply it. The new server stores this address and uses it in much the same way a stub clearinghouse uses the address of a clearinghouse server: as a means of making a connection with the clearinghouse system.

The next step is to give the clearinghouse server a name. If the clearinghouse server is to be in a new organization, the new organization name is first registered with the central naming authority, as described above. If the server is to be in a new domain, the domain name is similarly registered with the organization naming authority.

The system administrator then gives the clearinghouse a name. In order to give a name to the clearinghouse server the system administrator decides what portion of the clearinghouse database this clearinghouse server is to contain. Suppose that it is to be an organization clearinghouse for the existing organization *Xerox* and domain clearinghouse for the new domain *ASD@Xerox.*

The system administrator picks *NewServer@ClearinghouseServers@ClearinghouseServers* and *NewServer@Xerox@ClearinghouseServers* for the organization and domain clearinghouse names, and registers the names, network addresses, passwords, and so on with the clearinghouse. (Software makes appropriate calls to *AddName* and *AddIndividual* for each name on behalf of the system administrator.)

The system administrator then tells each clearinghouse server in the system to add the name *NewServer@ClearinghouseServers@ClearinghouseServers* to its group *Xerox@ClearinghouseServers@ClearinghouseServers* (the set of names of organization clearinghouses for Xerox). That is, software on the system administrator's behalf tells each server to execute

AddMember("NewServer@ClearinghouseServers@ClearinghouseServers", "Xerox@ClearinghouseServers@ClearinghouseServers", "Members") on their own database. The system administrator next tells each organization clearinghouse server in *Xerox* to add to its database the existence of the new domain, by adding the property *<"Members", 1, {NewServer@Xerox@ClearinghouseServers}>* to the name *ASD@Xerox@ClearinghouseServers*. That is, software on behalf of the system administrator tells each organization clearinghouse server in *Xerox* to execute *AddName("ASD@Xerox@ClearinghouseServers)* followed by *AddGroup("ASD@Xerox@ClearinghouseServers", "Members", "NewServer@Xerox@ClearinghouseServers")* on their own database. They already have the server's address. The updating algorithm automatically will check to make sure that the new name has not already been registered. The new domain *ASD@Xerox* has been registered with the clearinghouse indirectly. The new server is now known to all appropriate clearinghouse servers that should know about it, and will receive any updates requested from this moment on. The system administrator now instructs the server to obtain a copy of the database for the organization *Xerox* and the domain *SDD@Xerox*. Some care must be taken to ensure that the new server receives any updates presently being propagated (each server receiving an update request makes sure that it is being sent to the most up-to-date list it knows).

To make the process of adding a new clearinghouse server clearer, let us present it as a quasi-algorithm. To simplify the following, we assume that there is no conflict in naming, and that the originator of the request has the appropriate access right to make additions. A clearinghouse component can (1) *Find* the mapping for a name in its own database, or (2) *Contact* a clearinghouse server, or (3) *Add* a name or the mapping for a name in its own database, or (4) *Mail* an update message to another clearinghouse server, or (5) *Read* an update message from another clearinghouse and modify its database.

Creating a domain clearinghouse for a new domain B@C:

1. ***New*** *clearinghouse server:*

1.1. The system administrator gives the domain clearinghouse the name *NewServer@C@ClearinghouseServers.*

1.2. *Contact* any organization clearinghouse server for C *(Go to 2).*

2. ***Organization*** *clearinghouse for C:*

2.1. *Add* the name *NewServer@C@ClearinghouseServers* to the database. (Recall that an organization clearinghouse for C contain mappings for *<anything>@C@ClearinghouseServers*).

2.2. *Add* the property *<Clearinghouse Location, 0, network address>* to the name *NewServer@C@ClearinghouseServers.*

2.3. *Add* the name *B@C@ClearinghouseServers* to the database.

2.4. *Add* the property *<Members, 1, {NewServer@C@ClearinghouseServers}>* to the name *B@C@ClearinghouseServers.*

2.5. *Find* the names and addresses of sibling organization clearinghouse for *C*. That is, *Find C@ClearinghouseServers@ClearinghouseServers* → *{..., <"Members", 1, {L_1@ClearinghouseServers@ClearinghouseServers, ..., L_k@ClearinghouseServers@ClearinghouseServers}>, ...}. Find*

L_i@ClearinghouseServers@ClearinghouseServers → {..., <"Clearinghouse Location", 0, networkaddress>, ...} for all *i* from *1* to *k*. This operation is guaranteed to be successful. *Mail* each server, which is an organization clearinghouse for *C*, an update message containing the actions specified in steps 2.1, 2.2, 2.3, and 2.4.

3. ***Sibling** organization clearinghouse for C:*
3.1. *Read* update message to *Add* the names and properties as in steps 2.1, 2.2, 2.3, and 2.4.
3.2. *Go to 3.1.*

Creating a sibling domain clearinghouse for an existing domain B@C:

The same set of actions are performed as described above except that Step 2.3 is omitted and 2.4 is modified to add *NewServer@C@ClearinghouseServers* as a new member to the group *B@C@ClearinghouseServers*.

Creating an organization clearinghouse for a new organization C:

1. *New clearinghouse server:*
1.1. The system administrator gives the organization clearinghouse the name *NewServer@ClearinghouseServers@ClearinghouseServers*.
1.2. *Contact* any clearinghouse server *(Go to 2)*.

2. ***Some** clearinghouse server:*
2.1. *Add* the name *NewServer@ClearinghouseServers@ClearinghouseServers* to the database. (Recall that all clearinghouse servers contain mappings for *<anything>@ClearinghouseServers@ClearinghouseServers*).
2.2. *Add* the property *<Clearinghouse Location, 0, network address>* to the name *NewServer@ClearinghouseServers@ClearinghouseServers*.
2.3. *Add* the name *C@ClearinghouseServers@ClearinghouseServers* to the database.
2.4. *Add* the property *<Members, 1, {NewServer@ClearinghouseServers@ClearinghouseServers}>* to the name *C@ClearinghouseServers@ClearinghouseServers*.
2.5. *Find* the name and address of one organization clearinghouses for each organization. That is, *Find <anything>@ClearinghouseServers@ClearinghouseServers → {..., <"Members", 1, {L_1@ClearinghouseServers@ClearinghouseServers, ..., L_k@ClearinghouseServers@ClearinghouseServers}>, ...}. Find L_i@ClearinghouseServers@ClearinghouseServers → {..., <"Clearinghouse Location", 0, networkaddress>, ...}* for any *i* from *1* to *k*. This operation is guaranteed to be successful. *Mail* the servers, which is an organization clearinghouse, an update message containing the actions specified in steps 2.1, 2.2, 2.3, and 2.4, and ask the server to propagate the update to its siblings and domain clearinghouses.

3. ***Organization** clearinghouse for organization F, say:*
3.1. *Read* update message to *Add* the names and properties as in steps 2.1, 2.2, 2.3, and 2.4. If message does not require propagation of updates *Go to 3.1*.

3.2. *Find* the names and addresses of sibling organization clearinghouse for *F*. That is, *Find F@ClearinghouseServers@ClearinghouseServers* → *{..., <"Members", 1, {L_1@ClearinghouseServers@ClearinghouseServers, ..., L_k@ClearinghouseServers@ClearinghouseServers}>, ...}*. *Find L_i@ClearinghouseServers@ClearinghouseServers* → *{..., <"Clearinghouse Location", 0, networkaddress>, ...}* for all *i* from *1* to *k*. *Mail* each server, which is an organization clearinghouse for *F*, an update message containing the actions specified in steps 2.1, 2.2, 2.3, and 2.4.

3.3. *Find* the names and addresses of all domain clearinghouses in the organization *F*. That is, *Find <anything>@F@ClearinghouseServers* → *{..., <"Members", 1, {L_1@F@ClearinghouseServers, ..., L_k@F@ClearinghouseServers}>, ...}*. If successful, *Find L_i@F@ClearinghouseServers* → *{..., <"Clearinghouse Location", 0, networkaddress>, ...}* for all *i* from *1* to *k*. *Mail* each server, which is a domain clearinghouse for *F*, an update message containing the actions specified in in steps 2.1, 2.2, 2.3, and 2.4.

3.4. *Go to 3.1.*

4. ***Domain** clearinghouse, say for E@F:*

4.1. *Read* update message to *Add* the names and properties as in steps 2.1, 2.2, 2.3, and 2.4.

4.2. *Go to 4.1.*

Creating a sibling organization clearinghouse for an existing organization C:

The same set of actions are performed as described above except that Step 2.3 is omitted and 2.4 is modified to add *NewServer@ClearinghouseServers@ClearinghouseServers* as a new member to the group *C@ClearinghouseServers@ClearinghouseServers*. Note that in step 2.5, the clearinghouse server will send an update message to any one organization clearinghouse for *C*, thereby propagating the update to all clearinghouse servers in *C*.

13.5. System Administration Facilities

To aid the system administrator manage the clearinghouse and the distributed system it supports, a sophisticated *system administration service* provides a user interface that allows the system administrator to manipulate the clearinghouse (and other services). This service provides facilities to manage distribution lists and access control lists, partition the clearinghouse database over a number of servers, and so on.

14. Summary and Conclusions

A powerful binding mechanism that brings together the various network-visible objects of a distributed system is an essential component of any large network-based system. The clearinghouse provides such a mechanism.

Since we do not know how distributed systems will evolve, we have designed the clearinghouse to be as open-ended as possible. We did not design the clearinghouse to be a general-purpose, relational, distributed database nor a distributed file system, although the functions it provides are superficially similar. It is not clear that network binding agents, relational databases, and file systems should be thought of as manifestations of the same basic object; their implementations may well require different properties. In any case, we certainly did not try to solve the "general database problem," but rather attempted to design a system which is implementable now within the existing technology yet which can evolve as distributed systems evolve.

A future paper will describe the network protocols (both connection-oriented and datagram-based) used in implementing the clearinghouse.

Acknowledgments

Many people have examined the requirements and structure of clearinghouse-like systems; and we have profited from the earlier efforts by Steven Abraham, Doug Brotz, Will Crowther, Bob Metcalfe, Hal Murray, and John Shoch. The organization of the clearinghouse, and, in particular, the lookup and updating algorithms, have been very heavily influenced by the work of the Grapevine project; we thank Andrew Birrell, Roy Levin, and Michael Schroeder for many stimulating discussions. The primary client for the clearinghouse is the Xerox Network Systems product line, and we are indebted to Marney Beard, Charles Irby, and Ralph Kimball for their considerable input on what the Star project needed from the clearinghouse. David Redell, who is responsible for Star's underlying electronic mail system, was a constant source of inspiration and convinced us to use the naming convention for clearinghouse servers that is described in Section 9 over the one described in Appendix 2. Jim White carefully criticized numerous earlier versions of this document; we are grateful to him. Larry Garlick, Lance Kayashima, and Bob Lyon implemented the clearinghouse, and Mike Trigoboff implemented the Star Directory Icon; we thank them for turning the clearinghouse design into reality.

References

[Abraham and Dalal 1980]
S. M. Abraham and Y. K. Dalal, "Techniques for Decentralized Management of Distributed Systems," *20th IEEE Computer Society International Conference (Compcon)*, February 1980, pp. 430-436.

[Birrell, Levin, Needham, and Schroeder 1981]
A. D. Birrell, R. Levin, R. M. Needham and M. D. Schroeder, "Grapevine: an Exercise in Distributed Computing," submitted for publication.

[Boggs 1981]
D. R. Boggs, "Internet Broadcasting," Ph.D. Thesis, Stanford University, 1981, in preparation, (will be available from Xerox Palo Alto Research Center).

[Boggs, *et al.* 1980]
D. R. Boggs, J. F. Shoch, E. A. Taft, and R. M. Metcalfe, "PUP: An internetwork architecture," *IEEE Transactions on Communications*, com-28:4, April 1980, pp. 612-624.

[Dalal 1981]
Y. K. Dalal, "The Information Outlet: A new tool for office organization," *Proceedings of the Online Conference on Local Networks & Distributed Office Systems*, London, 11-13 May, 1981, pp. 11-19. Also Xerox Office Products Division, Palo Alto, OPD-T8104, October 1981.

[Dalal and Printis 1981]
Y. K. Dalal and R. S. Printis, "48-bit Absolute Internet and Ethernet Host Numbers," to be published in *Proceedings of the 7th Data Communications Conference*, October 1981. Also Xerox Office Products Division, Palo Alto, OPD-T8101, July 1981.

[Daley and Neumann 1965]
R. C. Daley and P. G. Neumann, "A general-purpose file system for secondary storage," *Proc. Fall Joint Computer Conf.*, 1965, AFIPS Press, pp. 213-228.

[Dawes et al 1981]
N. Dawes, S. Harris, M. Magoon, S. Maveety, D. Petty, "The Design and Service Impact of COCOS—An Electronic Office System," *Proc. IFIP International Symposium on Computer Message Systems.*

[Ethernet 1980]
The Ethernet, A Local Area Network: Data Link Layer and Physical Link Layer Specifications, Version 1.0, September 30, 1980.

[Galler and Fischer 1964]
B. A. Galler and M. J. Fischer, "An Improved Equivalence Algorithm," *CACM*, 7:5, pp. 301-303.

[Levin and Schroeder 1979]
R. Levin and M. D. Schroeder, "Transport of Electronic Messages Through a Network," Xerox PARC Technical Report CSL-79-4, April 1979.

[Metcalfe and Boggs 1976]
R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *CACM*, 19:7, July 1976, pp. 395-404.

[Needham and Schroeder 1979]
R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computer," *CACM*, 21:12, December 1978, pp. 993-999.

[Pickens, Feinler, and Mathis 1979]
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*, August 1979.

[Redell, *et al.* 1980]
D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell, "Pilot: An Operating System for a Personal Computer," *CACM*, 23:2, February 1980, pp. 81-91.

[Shoch 1978]
J. F. Shoch, "Internetwork Naming Addressing and Routing," *17th IEEE Computer Society International Conference (Compcon)*, September 1978.

[Tarjan 1975]
R. E. Tarjan, "Efficiency of a Good but not Linear Set Union Algorithm," *JCSS*, 9:3, pp. 355-365

[Thomas 1976]
R. H. Thomas, "A Solution to the Update Problem for Multiple Copy Data Bases which use Distributed Control," Bolt, Beranek and Newman technical report No. 3340.

Appendix 1: Network Addresses and Address Verification

The first use of our clearinghouse will be as the primary binding agent for the Xerox Network Systems product line, including the Xerox 8010 Star information system. We discuss in this appendix some of the issues concerning addresses in the Ethernet environment. We assume for concreteness that the internetwork is a collection of Ethernets and private and public data networks, with associated Xerox Network Systems-based internetwork routing machinery ([Boggs, *et al.* 1980, Dalal 1981]). A *network address* is a triple consisting of a *network number*, a *host number*, and a *socket number*. The internetwork links *machines*.

There is no clear correspondence between machines and the addressable objects known to the clearinghouse. One machine on an internetwork may contain many named objects; for instance, a machine may support a file service and a printer service (a *server* may contain many *services*). These different objects resident on the same machine may use the same network address even though they are separate objects logically and have different names. This introduces no problems since the clearinghouse does not check for uniqueness of addresses associated with names. Alternatively, different objects physically resident on one machine may have different network addresses, since a machine may have many different socket numbers. To allow both possibilities, we map the names of addressable objects into network addresses without worrying about the configurations of the machines in which they are resident.

However, it may be that one machine has more than one network address, since it may be physically part of more than one network. Therefore, the name of an addressable object such as printer or file server may be mapped into a set of network addresses, rather than a single address. However, these addresses may differ only in their network numbers: objects may be physically resident in one machine only.

Since the addresses given out by the clearinghouse may be transiently incorrect, clients need a way to check the accuracy of the network addresses given out by the clearinghouse. One way is to insist that each addressable object have a *uniqueid*, an absolute name which might, for example, consist of a unique processor number (hardwired into the processor at the factory) concatenated to the time of day. The uniqueid is used to check the accuracy of the network addresses supplied by the clearinghouse. This uniqueid is stored with the network addresses in the clearinghouse. When a client receives a set of addresses from the clearinghouse which are allegedly the addresses of some object, it checks with the object to make sure the uniqueid supplied by the clearinghouse agrees with the uniqueid stored by the object.

In summary, in the Xerox internetwork environment, the address of an object is stored as a tuple consisting of a set of network addresses and a uniqueid. The different network addresses can differ only in their network number. Thus, the name of an addressable object may be mapped into its addresses as follows: $L@D@R \rightarrow \{\ldots, \langle \text{"Address"}, 0, \langle \{networkaddress_1, networkaddress_2, \ldots, networkaddress_k\}, uniqueid\rangle\rangle, \ldots\}$.

Appendix 2: Another Structure for Clearinghouse Servers

We describe another structure for naming and locating clearinghouse servers. With this structure, one searches for a name by traversing the clearinghouse tree first bottom-up, and then top-down, as opposed to the system described in Sections 9 through 13, which always searches the tree top-down.

A2.1. Domain and Organization Clearinghouses

Corresponding to each domain *D* in each organization *O* are one or more clearinghouse servers each containing a copy of all mappings for every name of the form *<anything>@D@O*, that is, the mappings for all objects in domain *D* in *O*. Each such clearinghouse server is called a *domain clearinghouse* for *D* in *O*. (Each clearinghouse server that is a domain clearinghouse for *D* in *O* may contain other portions of the database other than just the database for this domain, and each of the domain clearinghouses for *D* in *O* may differ on what other portions of the global database, if any, they contain.) There is at least one domain clearinghouse for each domain in the distributed environment. Domain clearinghouses are addressable objects in the internetwork and hence have names $L_1@D_1@O_1$, $L_2@D_2@O_2$, ..., $L_k@D_k@O_k$, say. The set of names of the domain clearinghouses for *D* in *O* has name *Clearinghouses@D@O*, and various clearinghouse servers will be required to contain the mapping *Clearinghouses@D@O* → {..., <"Clearinghouse Names", 1, $\{L_1@D_1@O_1, L_2@D_2@O_2, ..., L_k@D_k@O_k\}$>, ...}.

Corresponding to each organization *O* are one or more clearinghouse servers each containing a copy of all mappings for every name of the form *Clearinghouses@<anything>@O*, that is, the mappings for all domain clearinghouses in organization *O*. Each such clearinghouse server is called an *organization clearinghouse* for *O*. Further, for each such mapping *Clearinghouses@<anything>@O* → {..., <"Clearinghouse Names", 1, $\{L_1@D_1@O_1, L_2@D_2@O_2, ..., L_k@D_k@O_k\}$>, ...}, each organization clearinghouse for *O* contains the mapping $L_i@D_i@O_i$ → {..., <"Clearinghouse Location", 0, *network address*>, ...} for every *i* from *1* to *k*. (They are not required to contain any other mappings for any of the domains $D_i@O_i$, although they may do so.) Each organization clearinghouse for *O* therefore knows the name and the address of every domain clearinghouse in *O*, and hence points directly or indirectly to every object in organization *O*. There is at least one organization clearinghouse for each organization in the distributed environment. Organization clearinghouses are addressable objects in the internetwork and have names $L_1@D_1@O_1$, $L_2@D_2@O_2$, ..., $L_k@D_k@O_k$, say. The set of names of the organization clearinghouses for *O* has name *Clearinghouses@Clearinghouses@O*, and various clearinghouse servers will be required to contain the mapping *Clearinghouses@Clearinghouses@O* → {..., <"Clearinghouse Names", 1, $\{L_1@D_1@O_1, L_2@D_2@O_2, ..., L_k@D_k@O_k\}$>, ...}.

The clearinghouse reserves the localname *Clearinghouses* in every domain, the domain name *Clearinghouses* in every organization, and the property names *Clearinghouse Location* and *Clearinghouse Names*. Further, by convention, the organization clearinghouses for *O* are also the domain clearinghouses for the domain *Clearinghouses@O*, for every organization *O*. This avoids introducing an ambiguity in the use of the name *Clearinghouses@Clearinghouses@O*. Each domain *Clearinghouses@O* is reserved for the use of the clearinghouse, and contains no names other than *Clearinghouses@Clearinghouses@O*.

We do not require domain or organization clearinghouses to be in the domains or organizations whose databases they house. For instance, a clearinghouse server with name *ClearinghouseServer@SDD@Xerox* may be a domain clearinghouse for domain *PARC@Xerox* even though it is logically "in" another domain *SDD*.

A2.2. Interconnections between Clearinghouse Components

As we have just seen, organization clearinghouses point downwards to domain clearinghouses, which point downwards to objects. Considerably more interconnection structure is required so that stub clearinghouses can access clearinghouse servers, and clearinghouse servers can access each other, both to receive and to exchange information.

Domain clearinghouses point upwards to their containing organization clearinghouses, as follows. Each domain clearinghouse for organization *O* is required to contain the mapping *Clearinghouses@Clearinghouses@O* → *{..., <"Clearinghouse Names", 1,* $\{L_1@D_1@O_1, L_2@D_2@O_2, ..., L_k@D_k@O_k\}$*>, ...}*, say, and the mappings $L_i@D_i@O_i$ → *{..., <"Clearinghouse Location", 0, networkaddress>, ...}* for every *i* from *1* to *k*. In this way, domain clearinghouses know the names and addresses of their containing organization clearinghouses. Domain clearinghouses have to be able to access their containing organization clearinghouses since only the latter need contain the addresses of all domain clearinghouses in the organization, and the addresses of other organization clearinghouses.

Domain clearinghouses also point to their *siblings*, the other clearinghouse servers which are domain clearinghouses for their domain. That is, a domain clearinghouse for domain *D@O* contains the mapping *Clearinghouses@D@O* → *{..., <"Clearinghouse Names", 1,* $\{L_1@D_1@O_1, L_2@D_2@O_2, ..., L_k@D_k@O_k\}$*>, ...}*, and the mappings $L_i@D_i@O_i$ → *{..., <"Clearinghouse Location", 0, network address>, ...}* for every *i* from *1* to *k*. (It contains the mapping for *Clearinghouses@D@O* anyway since that name is in domain *D@O*.) Thus it points to all the other clearinghouse servers which are domain clearinghouses for *D@O*—its siblings. This information is redundant (it could be obtained from the containing organization clearinghouse) but makes updating domain clearinghouses somewhat faster.

Similarly, organization clearinghouses point to their *siblings*, the other clearinghouse servers which are organization clearinghouses for their organization. In particular, each organization clearinghouse for organization *O* contains the mapping *Clearinghouses@Clearinghouses@O* → *{..., <"Clearinghouse Names", 1,* $\{L_1@D_1@O_1, L_2@D_2@O_2, ..., L_k@D_k@O_k\}$*>, ...}*, and the mappings $L_i@D_i@O_i$ → *{..., <"Clearinghouse Location", 0, network address>, ...}* for every *i* from *1* to *k*. Thus it points to all the other clearinghouse servers which are organization clearinghouses for *O*. Again, it is redundant for an organization clearinghouse to point to all its siblings, since any of its contained domain clearinghouses point to them, but again it makes updating organization clearinghouses faster.

Finally, stub clearinghouses contain the address of at least one clearinghouse server or at least can always find a clearinghouse server, perhaps by local or directed broadcast. If they store an address, it is typically, but not necessarily, the address of the closest clearinghouse server.

Recall again that a clearinghouse server may contain any portion of the whole database (subject to access control). Thus it may be a domain clearinghouse for one or more domains, and/or an organization clearinghouse for one or more organizations. On the other hand, as we have pointed out, it is not necessary for a clearinghouse component to contain a sizable database.

A2.3. Interconnections between Organization Clearinghouses

We have described how the clearinghouse is structured within an organization. Clearinghouse information may flow freely among all the domain and organization clearinghouses within a particular organization. That is, each domain and organization clearinghouse may request and store a copy of any mapping held by any other domain and organization clearinghouse within that organization. Since an "organization" is presumed to be a corporate entity or sub-entity, this model of *trust* among clearinghouse servers is appropriate within an organization.

If the internetworks of two companies are joined, they may wish to keep their respective domain and organization clearinghouses at "arm's length." How they do this is described in Section 12. Here we restrict ourselves to describing how organizations, which trust each other and wish to share information freely, are logically connected by the clearinghouse mechanism.

Each organization clearinghouse contains the mapping *Clearinghouses@Clearinghouses@O* → *{..., <"Clearinghouse Names", 1,* $\{L_1@D_1@O_1, L_2@D_2@O_2, \ldots, L_k@D_k@O_k\}$*>, ...}* for each organization *O* it trusts. Furthermore, for each such name $L_i@D_i@O_i$, it contains the mapping $L_i@D_i@O_i$ → *{..., <"Clearinghouse Location", 0, networkaddress>, ...}*. Each organization clearinghouse therefore knows the name and the address of every organization clearinghouse of every organization it trusts.

A2.4. Summary

Each clearinghouse server contains mappings for a subset of the set of names. If it is a domain clearinghouse for domain *D* in *O*, it contains mappings for all names in that domain. If it is a organization clearinghouse for organization *O*, it contains mappings for the names of all domain clearinghouses in that organization. Each organization clearinghouse also contains mappings for the names of all organization clearinghouses of those organizations with which it is friendly. Each domain and organization clearinghouse points to its sibling clearinghouses. Domain clearinghouses point to their containing organization clearinghouses, and stubs point to any clearinghouse server.

XEROX

Xerox Corporation
Office Products Division
3333 Coyote Hill Road
Palo Alto, California 94304

XEROX® is a trademark of XEROX CORPORATION Printed in U.S.A.