MIT/LCS/TR-331

DISTRIBUTED NAME MANAGEMENT

Karen Rosin Sollins

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract numbers N00014-75-C-0661 and N00014-83-K-0125.

2

This blank page was inserted to preserve pagination.

Distributed Name Management

by

Karen Rosin Sollins

Submitted to the
Department of Electrical Engineering and Computer Science
on February 14, 1985 in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

This research was support by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract Nos. N00014-75-C-0661 and N00014-85-K-0125.

Distributed Name Management

by
Karen Rosin Sollins

Submitted to the
Department of Electrical Engineering and Computer Science
on February 14, 1985 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

The problem being addressed in this research is the design of a naming facility achieving the following goals. First, two functions on names must be supported: accessing a named object, and acting as a place holder for the named object. Second, it must be possible to share those names. Third, communication of the names as well as communication by use of the names must be possible. Finally, feasibility of implementation is a goal. In this research a name is defined to be an object that can be associated with another object and has an equality operation defined on it. Two functions are defined for a name; it can be used both to provide access to the named object and as a place holder for the named object. The assumed system model is a loosely coupled, distributed system.

The research addresses this problem with: (1) a detailed analysis of the naming problem and the nature of names themselves; (2) a proposal for a set of mechanisms that addresses the problem above, including the proposal of two new types of objects and the mechanisms for their use; and (3) two examples of uses of the model. The model consists of private views of shared, local namespaces allowing shared use of names and supporting shared responsibility for management of the namespace. In addition the model provides for the acceptance and deletion of names in stages.

The contributions of the research include an investigation into the nature of names, an analysis of naming as a social process especially recognizing both the joint management of names by the users of those names and the fact that acceptance and possibly deletion occur in degrees, and the proposal for a mechanism to address these issues.

Key words: naming, distributed system, sharing, cooperation, software environment, strong typing.

Acknowledgments

I would not have succeeded in this research project without the support, guidance, and caring of a great many people:

Mike Sollins, my husband, who, more than anyone else, supported, listened to, and coaxed me through the many highs and lows of such a project,

David Reed, my advisor, who, more than anyone else, has showed me how and encouraged me to think and question and has worked tirelessly with me through many versions of explaining my ideas,

Peter Sollins, my son, who has come with me through many emotional highs and lows, always ready with a hug, a kiss, and a word of encouragement,

David Clark, a reader, who in friendship has put much more into this project than can be asked of a reader and kept bringing me back to reality,

J. C. R. Licklider, a reader, who has shown me how to look at the world from new perspectives and always with enthusiasm,

Deborah Estrin and Sam Hsu, two special friends, who, in many long conversations, have helped me through the trials and tribulations of being a graduate student while maintaining some perspective on self and life,

Axel and Kathy Rosin and Susanna Bergtold, my parents and sister, who have supported and encouraged me in things that they may believe they cannot understand, but could with a good explanation,

The many members of the computer systems research groups, especially Mark Rosenstein for his code and support in the mail implementation, Larry Allen, Jerry Saltzer, and John Romkey for technical discussions of my research, the users of my mail implementation, and the other members of the groups, who all have given me their support and shared their sense of humor,

The women, other female students and staff members, especially Deborah Estrin, Muriel Webber, Debby Fagin, and Toby Bloom, who have given me a better understanding of myself and MIT and how we can all work together.

Thank you

For my husband Mike, to whom a great deal of credit goes for this,

with all my love

Table of Contents

Table of Figures

Chapter One

Introduction

1.1 The Issues

Names are a critical part of communication, both among humans and between humans and computers. In order to communicate with another human, the human must be able to name objects and actions in such a way that both humans understand the names. Analogously, in order to communicate with a computer, the human must be able to name operations and objects in a way meaningful to both the human and the computer. Therefore, what can be named and how is a central issue in designing a computer system useful to humans.

There are three concepts that form the basis of this research project. The first of these ideas is that many, perhaps most, computer environments today consist of federations of fairly autonomous computers connected by networks and internets[1]. Such a federation leads to issues of independence in defining names, reliability of service, replication of data, redundancy, and many others.

The second idea is that, in addition to providing excellent storage for information and arithmetic and decision-making capabilities, computer systems provide a medium of communication and cooperation both between people and computers and among people. Such communication and cooperation may be achieved through sending and receiving electronic messages, sharing and working within a large, possibly distributed, database management system, cooperative text or program preparation, or a number of other activities.

---

[1] An internet is a network of networks, allowing for communication across network boundaries.

The third idea is that imitating human naming patterns in a naming facility will lead to a more useful naming facility. Observations about human naming are considered in this research for two reasons. First, humans are autonomous beings forming and reforming federations in which they effectively communicate and cooperate with each other. Second, computer systems designers and builders have created naming facilities that are frequently adequate for computer use, but often not for human use. It should be noted that most of these observations can be found separately as goals of various naming facilities, although they have not been assembled to form the goals of a single naming facility. The observations are:

1. **Communication:** *Names are part of the basis for communication. Therefore sets of names used by individuals should be sharable, reflecting common interests and communication patterns.*

2. **Individuality:** *Part of the social process of naming is that each individual brings personal experiences and unique decision making to the process. Those experiences may be shared with others, but no two people will have had exactly the same set of experiences, and no two people will make exactly the same choices at all times.*

3. **Multiplicity of names:**

   - *Different people use the same name for different things.*

   - *Different people use different names for the same thing.*

   - *A single user uses different names for the same thing.*

   - *A single user uses the same name for different things in different situations or at different times.*

4. **Locality of names:** *A person uses a small set of local names to reflect his or her focus of interest.*

5. **Flexibility of usage of names:** *Humans use several sorts of names. For example, names are often descriptive. In addition, descriptions that have not been previously chosen as names may be used. Humans also use*

generic names to label classes of objects. These generic names may be labels or descriptions. In fact, humans often use combinations of generic names and descriptive names in order to narrow the set of objects that are identified.

6. **Manifest meaning of names:** *The words used by humans for names have meanings constrained by human languages. These meanings are understood by other humans as well.*

7. **Usability of names:** *Humans are able rapidly to define or redefine names and shift contexts on the basis of conversational cues. They also have mechanisms for disambiguating names, such as querying the source of a name for further information.*

8. **Unification:** *Humans often use various naming schemes, not limiting the naming of objects to special schemes based solely on the type of the object. Rather, the various schemes are generally applicable.*

The goal of this work is to investigate a framework for a naming facility that allows for communication, cooperation, and more human-like naming based on the list of observations above. Part of this investigation is a study of those aspects of naming that are common to many or all applications and those aspects that are not, and therefore must be application specific.

The underlying model of a federation of computers is discussed in Section 1.2, followed by a brief investigation of the problem being posed in this research in Section 1.3. A brief introduction to the proposed framework for a naming facility is contained in Section 1.4. Section 1.5 discusses related work, first considering some philosophical, linguistic and sociological work that has influenced this research. It then presents a representative sample of work in computer science that has investigated the ideas that are being brought together in this work. Finally, the last section of this chapter describes how the investigation will proceed through the remainder of the thesis.

1.2 The Assumption of Federation

Of the three ideas mentioned in Section 1.1, federation is an underlying assumption of this research, while the concepts of communication and cooperation and the concept of more human-like naming are goals to be achieved. Since computational federation is an assumption, in addition to defining it, the implications of federation on naming and name management must be carefully considered. A conclusion will be that federation complements concepts of communication and cooperation and human-like naming. The goals define a large problem area, that must be limited in order to make this solution feasible. These limitations will be discussed, followed by a brief description of the proposed mechanisms that comprise the solution.

The direction in which computer systems have been moving is toward a multiplicity of machines interconnected by networks providing a communication medium. The concerns of privacy and independence from other users have always been issues among computer administrators and users, but the nature of those concerns has changed somewhat as smaller cheaper computers have become available. In many cases, administrators purchase such computers and put them into service in isolation. At some later time, the administrators decide to connect the computers under their management. From here, the collection may continue to grow with little control or consensus among the participants in such a "system". A computer is *autonomous* if all the activities on it are isolated from the activities of any other; for all intents and purposes, it is not connected to any other computer. Many administrators have pursued this option in order to escape large time-sharing systems. A *federation* is a loose coupling of computers to allow some degree of cooperation, while at the same time preserving a degree of autonomy. In a federation, there is some agreement on behavior and protocols to be utilized, but the barriers apparent in the isolated machine are still available to anyone who wants to enforce them. If the administrator or user wants to disconnect the computer from

the network by simply not accepting messages, that is possible. If that computer provides a service to the participants in the network, they must understand that such a service will not always be available. On the other hand, federation provides the common ground for communication (such as agreement about protocols and services to be available) should it be desired. Federation includes autonomous behavior, a relatively easy problem to address, while allowing for unplanned interconnection and cooperation as needed. Allowing for cooperation is more difficult to address, and frequently ignored or disallowed. The loose coupling labelled federation is taken as the system model in this research.

Federation brings with it the fact that communication may only be available on an irregular and unpredictable basis, both because the humans involved may choose it and because communication links are physically unavailable. For example, two networks may be created independently and only later connected. The connection may come and go, or particular machines may be available only at certain times. These irregular communication patterns have several implications. First, uniform agreement cannot be assumed, affecting naming. In general, most naming schemes today assume that there will be an agreement on a naming service. In the large Arpanet community, the Network Information Center (NIC) [15] provides that service, although there is a plan for distributing this responsibility to some extent to address this problem of a central service [31]. The creators of Grapevine [5] and Clearinghouse [36] distributed this responsibility among managers or administrators, but still require a local external service to register names. Neither Grapevine nor Clearinghouse allows for graceful merging of two of their environments when namespaces overlap.

There are two implications of federations; their effects on naming are worth noting at this point. First, the assumption of independent initialization implies that once two systems have joined in a federation, unique identifiers are not available unless

some prior arrangement was made. Since the two systems were initialized and operating independently, they may have overlapping sets of identifiers in use. If a merged set of names is not to have duplicates, it is possible that names must be changed and future agreement must be coordinated. The fact that particular namespaces are assumed to contain only unique names may have far-reaching consequences if this assumption has been built into application subsystems and programs as well as the operating systems. The problem may be especially insidious if the merger is occurring between two distributed systems of the same type, where such dependencies may be well hidden from the user. This issue was addressed both in SNA [3] where the solution was to build a wall between two such cooperating, but independent networks, and by Rom [41] who proposed algorithms for merging namespaces of networks at the time of merging.

The second result of assuming federation is an unpredicatable lack of availability of participants in the federation. For naming, names needing non-local resolution may not always be resolvable. Any functions which are to be usable whenever a local node is available must not be dependent on auxiliary remote services that might not be available. For instance, if a remote printing service should be available to the local machine whenever the printing server and the communications medium are available, then accessing the printing server must not be dependent on a remote name or authentication service. This assumption may have far-reaching effects, for instance in compiling code with remote procedure calls, using a distributed database management system, sending and receiving mail and many other distributed applications. Such applications may be designed on the assumption that certain auxiliary information is available, although it is possible to perform certain functions without that information. Needless to say, when the time comes to perform the remote procedure call or access the non-local data, the non-local site involved must be accessible.

Compare briefly the human situation with the assumed model of federation. There are many similarities. Humans will often think and function independently and then discuss or operate cooperatively. An individual may develop ideas privately before sharing them. Then a group may form to address them. Humans certainly function both without joint initialization and in the face of possibly intermittent communication. Humans, beginning with some basic shared means of communicating (which may be as basic as facial and hand expressions), negotiate further means of communication. They also generally use names without requiring or even wanting access to the named entity. In fact, part of the function of a name is as a place holder. It is the sharing and joint management of names that this research is addressing.

The following section will briefly present a model for a set of mechanisms that adhere to the eight observations listed above. The model will be addressed further in Chapter 3 and succeeding chapters.

1.3 The naming problem

The problem in naming that this research is addressing can be stated simply and then subdivided into three subproblems. Each of these in turn can be subdivided again. This structure of the problem will be examined in this section.

Names allow the *users* of objects to identify and access those objects *jointly.* Although joint naming is not always used, the fact that naming is used frequently for communication among users must be supported. The naming problem is that currently available naming facilities in computers do not support joint naming among people adequately, in many cases because the full extent of the problem has not been recognized. In addition, feasibility of implementation must also be a goal of the design of a naming facility. Three words were highlighted because they identify the three subproblems that are addressed in this research.

The nature of *names* will be studied in order to understand both the inherent characteristics of them and the uses of names. This research identifies five characteristics of names. All have an impact on use or understanding of names. Three of the characteristics reflect roles in naming: who assigns names, who resolves them, and who uses them. These three properties of names determine the namespace from which names are chosen, within which they are associated and therefore can be resolved, and within which they will be used.[2] The other two properties of names identify the degree of ambiguity or uniqueness of a particular name and its degree of meaningfulness. Name, as defined in this research, have two basic functions. First, they provide access to the named objects; and, second, they can be used as place holders for the objects.

Understanding the nature of names and naming is closely related to recognizing and identifying the aspects of how *users* or people name. Eight observations about human naming have been identified in Section 1.1. Various of those eight aspects of naming can be found in various computer based naming facilities, but no single facility allows for all of them. Naming in computer systems has generally been more restrictive for humans than direct interpersonal communication allows.

Joint naming implies two subproblems. The first is that communication using names must be supported, requiring sharing an understanding of names. The second subproblem is that negotiation must take place in order to reach an understanding about what is to be shared. Negotiation may also involve acceptance of names by degrees or stages. Because federation is an underlying assumption, dependency on an external decision maker cannot be built into the support

---

[2] *Namespace* is a general term for an object that remembers the association between a name and an object and provides translation between names and objects. Chapter 2 investigates the relationship between names and namespaces further and Chapter 3 presents the formal model, called a context, of a namespace proposed in this research.

mechanisms. Whatever joint understanding exists can only be defined by the participants in the understanding.

The problem being addressed in the model in the next section is *to mirror people using names jointly to identify and use objects.* Names can be understood better by studying both their inherent characteristics and their uses. People using names can be understood better by recognizing the various aspects of human naming, both characteristics and uses. And finally, the joint naming that people do can be better understood by recognizing that it is a form of communication and sharing and that a structured negotiation must take place in order to reach agreement and allow for communication and sharing.

1.4 Model for a Solution

The previous sections presented an assumption of federation and the problem areas of communication and human naming. The solution in this research is based on defining two new types of objects, **contexts** and **aggregates**. Aggregates are composed of contexts and, therefore, will be considered later.

The basis for this proposal is a simple type of object called a context. A context translates names into either objects or other names and is the model for a namespace in this research. A name is an object assigned to another object within a namespace or context that allows the user either to use the name as a place holder for the named object in the context or to access the named object through the context. In some cases, a name will be translated into another name less meaningful to or less easily used by the user of the original name. Further translation in the same or another context may then be requested. In the remaining cases, the user or program will use the resulting translation as is. Whether further translation is needed or not, the decision is not made within the context but by the client, whether user or program, requesting the translation.

A context is a shared object and therefore has two further properties, both related to the fact that the most basic operations on contexts are name assignment and translation. First, a context contains a model of the fact that the associations between names and the objects they are naming may occur by degrees. For example, once a name has been selected, more uses of it will probably make it more easily understood. With disuse a name may be forgotten. In contexts, this is modelled as a series of states. Chapter 4 addresses this set of issues in detail. The final property of contexts is a set of participants, some representation of those sharing responsibility for a context or namespace. This information is needed for two reasons. First, identification of the context may include some means of identifying the participants. This a reflection of a human pattern of identifying subject matter, by including recognition of who is involved. The second reason is that different participants may have different roles in the selection of names. Again this will be discussed in Chapter 4. Thus, in addition to the actual translations between names and objects, a context also contains some means of identifying participants and a representation of the states of translations.

The other mechanism proposed here is the aggregate, the individual's naming window onto the world. Names can be assigned and used only through aggregates. An aggregate has two parts, the **current context** and the **environment**. The aggregate itself is not shared, although its current context is shared. When two people communicate, there is a small set of names that they use regularly and to which they may add new names needed in that conversation; it is this current context that they share. They each also have a pool of other contexts on which to draw. These pools may be different for each participant in the conversation. The pools, which are called their environments, consist of collections of contexts, which may or may not be partially ordered, but which are used to translate names not in the current context. The current context is shared by the participants. Other contexts may also

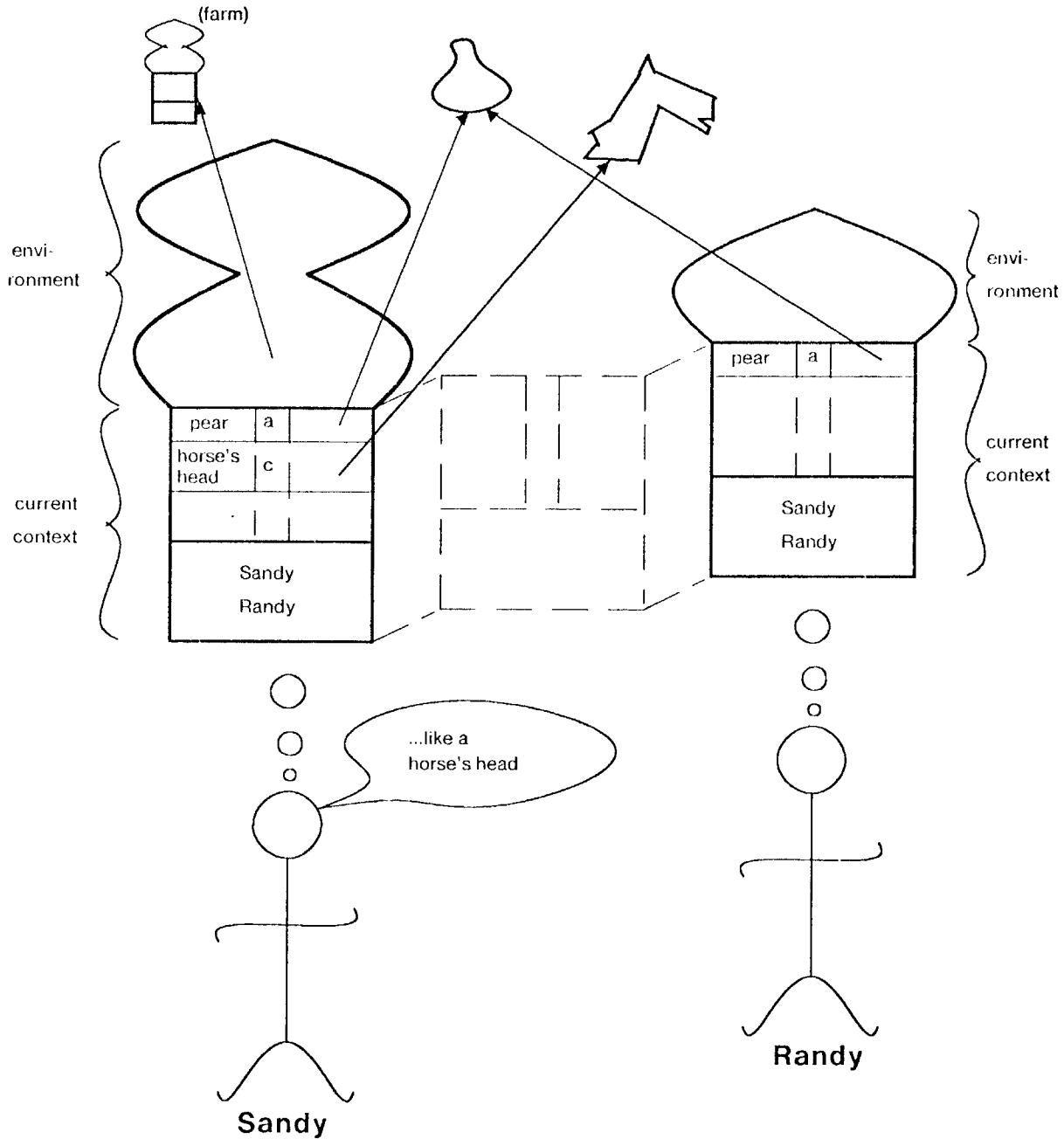

Figure 1-1: Aggregates containing private copies of a shared current context

be shared. A user may include any context in which he or she is a participant in the environment of an aggregate that does not have that context as its current context.

Figure 1-1 is provided as a visualization of a shared context and two aggregates representing individuals sharing it. In that figure, Sandy and Randy are identifying shapes. They have labelled one shape "pear" and now Sandy is proposing to name the second "horse's head." Sandy was recently on a farm, so farm animal shapes come to mind. With Sandy's proposal, the name becomes a candidate. If Randy agrees, the name "horse's head" will be accepted in their shared context reflecting the naming of these shapes.

The figure represents this situation as follows. Sandy and Randy each have an aggregate. Each aggregate contains a copy of the context that they share and each has a private environment. Sandy's aggregate has two rules in its environment and Randy's has one. The first rule in Sandy's environment contains only the current context of the aggregate known as "farm". The other rules are not depicted in the figure. The copies of the shared context need not be, and are not in this case, in synchrony. Both copies contain the fact that Sandy and Randy are the participants sharing this context. The fact that agreement has been reached about the name assignment for "pear" is reflected in the letter "a" in the entry, representing an entry accepted into the context. The entry for "horse's head" is being proposed by Sandy and therefore is in the "candidate" state represented by the letter "c". The information about this candidate entry has not yet propagated to Randy's copy of the context and therefore does not appear in Randy's copy of the shared context.

The reader should be aware that although the aggregate mechanism is based on the idea of human conversation, it will have a more general use. The attempt here is to model human behavior, not to provide any sort of explanation for how humans behave. The concepts of current context and aggregate are extensions and

modifications of the ideas of working directory and search rules used in many file systems. This is one of the aspects of the work of others that is discussed in the next section.

1.5 Related Work

According to Lampson [26]:

> Basically, there are only two ways [that] are known of doing naming. One way is to use hierarchical names, where you work your way down some structure like a tree-structured directory system, or an arrangement of nested records. If you apply an appropriate discipline of not generating two subnames that are the same at any level, then you have an unambiguous naming scheme. This is inconvenient, because you have to give this long structured name. The other method is to have some more-or-less aimless collection of scopes that you wander through, using something that is a search path or a scope inheritance rule or call it what you will. This has the advantage that if you're lucky, it will be convenient and give you what you want, and the disadvantage that you'll never really be quite sure of what it is you're going to get. You can basically pay your money and take your choice. Perhaps it's unfortunate that there's not any systematic way to decide exactly what search rule will be followed. There's not much uniformity either in the specifying of search rules or in the arrangement of hierarchical naming systems, but there are really only those two basic ideas. The whole subject, in my opinion, is much simpler than it's generally made out to be.

Fortunately for the users of computer systems, the set of solutions to naming problems is much richer than Lampson suggests. Exploration of various problems has proceeded in many of the subfields of computer science. In fact so much has been done, in many cases as a side effect of other research and development on other problems, that this report can only touch on a sampling of the work that has been done. The related research will be addressed in a non-traditional fashion in this thesis. This chapter will consider those works that have direct influence on this research. In addition, in each succeeding chapter, there will be a discussion of other

research related to the topic of that chapter. Therefore, what is traditionally a section on related works in a thesis will be distributed throughout this thesis.

The philosopher Quine [39] provides a masterful study of the relationships between names, the objects being named and the meanings of the names. Much can be learned much that is directly applicable to naming facilities that impose the thinking of the designers and builders of such a facility on its users. Naming forms the basis of thinking and communication. In a more practical sense, types or styles of names are not limited to types of objects being named. In particular, in the work here, Quine's idea of *general* names has been simplified and transformed into the idea of *generic* names.

Carroll of IBM as part of his work on names and naming has done sociological studies of human naming patterns both in conversation [54, 7] and in communicating with computer systems [6]. From Carroll's work, four important lessons can be learned. First, in communication between two people, there is a form of negotiating that takes place in proposing and accepting names that will be used by the two in the future. This idea of cooperative name management will be addressed in detail in Chapter 4. Second, Carroll teaches that naming is done on the basis of conversations, topics of mutual interest, and, in addition, based on the participants involved. It should be noted that conversations cannot necessarily be organized in a hierarchical fashion, but humans have mechanisms for distinguishing them without such hierarchical structures. Third, the individual, in bringing past experiences to a conversation, plays an important role in determining the names that will be chosen through those personal experiences. Fourth, Carroll re-enforces the concept learned from Quine that naming is universal. Objects are not necessarily distinguished by the types of names they have, but rather use the same naming mechanisms for naming all sorts of objects. Much of what can be learned from Quine and Carroll has not been built into computer systems, although many systems

begin to recognize in different ways from each other that the problems are not as simple as Lampson said.

This report will now review briefly those particular projects that have strongly influenced this research and what those influences have been. Beginning with Saltzer's work on naming [42], there are two ideas that have been taken from that. The first is the need for local and modular namespaces. Saltzer provides a detailed and careful analysis of why both locality and modularity are important.

The second idea inherited from Saltzer, reinforced by the work of Birrell et al. on Grapevine [5], Oppen and Dalal on Clearinghouse [36], and Lantz and Edighoffer on UDS [28], is that a naming facility can and should be universal. Naming problems and facilities cannot be split along the boundaries of the types of objects being named. Saltzer presents his model and then applies it to both a file system and memory management. The Grapevine experience was that their facility was originally used for naming mail recipients but the same naming facility could be and was used by the mail service itself to name and locate the services it needed to operate. In addition, other communities had other plans for it as a naming service. Both Clearinghouse and UDS were designed initially as universal naming services, in recognition that such universality was beneficial and efficient. This idea of universality was also reinforced by Saltzer [44] and Shoch [46] in which they distinguish names based on the objects being named. These papers only reinforced the idea that such efforts were creating artificial and unnecessary boundaries in naming.

Multics [37] has contributed several ideas to this work. There are two important influences. The first is in the structure of an aggregate. As mentioned, this is based on the idea of search rules and a working directory. Of course, other operating systems have incorporated these ideas as well, but it was Multics with which the

author was familiar. The second is the observation that even within the restrictions on segment names there are attempts to allow names to reflect meanings and as much as possible reflect names that might be used outside the system. Again this can be seen repeatedly in other operating systems as well. Directories have certain meanings. Component names have meanings. Both reflect external names as much as possible. In addition, as will be seen later, the Multics known segment table provides per process local naming and that is a large component of this work.

There are two final influences that bear mentioning here. The first is Lindsay's set of goals in his work on the catalog and object naming in R* [29]. Those goals have much in common with the earlier observations about human naming, although Lindsay did not emphasize communication and sharing as is done here. The final influence is a negative one, and to some extent work is progressing in an attempt to address it. The situation is the one found in the Arpanet, where a global, hierarchical namespace with a central administration is the only choice. At the level of internet addresses there is a hierarchy administered by the NIC [15]. A hierarchy is convenient but it does not reflect reality. Many hosts are on several networks or subnets and the structure of the internet is not hierarchical. At the level of naming hosts and users, work at moving away from a flat, global namespace again centrally managed by the NIC is progressing. The work of Mockapetris [31, 32] sets the standard to be a global hierarchical structure with a hierarchical administration. This addresses the problems of a flat namespace and a central authority, but does not address the fact that the administrative entities that will manage such a namespace do not form a hierarchy. In addition, the administrative structure will be reflected in the names, despite the fact that this has little to do with the names that people might want to use.

As mentioned previously, there is a great deal of work related to naming. What has been provided here is a summary of those works that had the strongest influence on

this research as it developed. Throughout the remainder of the thesis a sampling of other work will be noted where relevant. What is important to note here is that although the influence of others can be found in many aspects of this work, none has pulled the set of ideas together into one place.

1.6 The Plan

As part of a research project, it is necessary to identify the methodology used as a basis for the research. There are three parts to this methodology: (1) identification of the problem, (2) the tools used both in analyzing the problem and in providing a solution, and (3) testing the results for adequacy. The problem itself is recognizable as a problem because although humans have a very rich and flexible naming capability, computer systems do not and the problem becomes accentuated in a federated computing facility. The problem can best be explained as is done in Chapter 2 by comparison with human naming. Three tools are used in addressing the problem. The first is to examine human behavior, to gain an understanding of one approach to solving the problem. The second is to design a model. By nature, the model can only be an approximation because total human behavior is quite complex and frequently unpredictable, especially in new situations. The third tool is an implementation. The implementation of the model allows for study of the feasibility of the model and examination of the behavior of the model. The final part of the methodology of a research project is verification of adequacy of the results. First, the value of the issues can only be judged by the audience, although the fact that the work is novel can be argued by reviewing other work in the field. Second, implementability must be evaluated. This can be achieved most directly by an implementation, or if not, a design indicating the details needed for an implementation. Such an argument leaves the final decision to the audience again. The final measure that one can apply to a model for a solution is simplicity. This determination must also be left to the audience.

This report investigates the problems of naming a large variety of objects in a federated world of computational resources cooperatively among groups of humans in such a way as to mirror as best possible the naming that the humans would do among themselves without the medium of computers. Returning to the analysis of the problem in Section 1.3, it is investigated in depth in Chapter 2, including definitions of the problem itself, as well the definition of the term "name" as it is used in this research. That discussion is also concerned with the general issues of naming and how humans use names. The set of observations is examined in more depth than in this chapter, complemented by a study of attributes and functions of names. Chapters 3 and 4 together present a model for a naming facility. Chapter 3 defines and discusses contexts and aggregates in detail, followed by a discussion of joint management and name assignment in Chapter 4. Chapters 5 and 6 discuss implementations in two domains, in order to verify both that the problems presented are real and that the recommended framework can be used to build a naming facility in the two domains. Chapter 5 discusses an implementation in an electronic mail system and Chapter 6 presents a design for an implementation in a programming support environment. The thesis concludes in Chapter 7 with a review of what has been developed pointing to further research to be done as well. It concludes with a discussion of the contributions of this research.

This thesis addresses a large collection of issues surrounding naming and as such is an attempt to bring some order to that area. It presents a model, used in designing implementations, but neither the model nor the designs is an end, but rather they are a beginning. This research is a step forward in providing a more usable environment for clients of computer systems by improving the naming facilities and thereby the operating systems on those computers.

Chapter Two

The Nature of Names

2.1 Introduction to the Problem

The problem being addressed in this research is how to design a naming facility under the assumption of a federated system and achieving the following goals:

- support of names as defined below,
- provision of sharing and communication of and by use of those names,
- feasibility of implementing such a naming facility.

Federation provides benefits over both centralized computing facilities and decentralized but more tightly coupled distributed computing facilities. It both allows for a local tolerance to partial failures elsewhere and supports local isolation if that is desired. Continued operation in the face of separation due to remote failures or the choice of isolation require local functionality. Enough information and processing ability must be available to allow for the continuation of local operations, such as accessing local objects using local names for them. In addition, creation of new local names for local objects should be possible, without the need to access a remote name server or administrator. Of course, for those activities that require remote access, such as reaching agreement with remote sites on a shared name for something, one must have access to the remote participants, and such activities must await reconnection. This line of reasoning leads to the conclusion that local naming and name management must occur in order to benefit from federation.

Humans provide a good paradigm for studying cooperative naming in a federation, because they jointly define and use names as they are described in this research. In addition, they form federations with local facilities for name management within each person's mind, and with no sharing except in the form of the information that flows through various media of communication between them. Therefore, frequently throughout this research humans and human naming are used as examples both for understanding names and naming and also for where problems may continue to exist.

This chapter analyzes in depth the problem as identified above, by examining various aspects of names and naming. The first step in this analysis is to provide an operational definition of names. The definition is simple, in order to capture the essence of naming. Others have assumed more complex definitions, often in order to provide additional functionality that may be needed in particular applications. The definition is followed by discussions of aspects of names and observations about how names are used. The investigation of aspects of names provides the reader with a deeper understanding of names themselves, while the observations about uses explore patterns of cooperative usage within the definition of names. In addition, as part of the investigation of names this chapter presents a list of other potential uses for names to be found in other naming facilities, but excluded from this one because they are not consistent with the definition of names chosen here. Implementability and consideration of those problems found in other similar facilities that are not part of naming as defined here are left to later chapters of this document.

Figure 2-1 provides a simple example of a number of the issues to be addressed here. The Green family consists of five members, three of whom are children. The two older children, named Samantha and Samuel, may be given the same nickname "Sammy" at times. The baby, Sandy, cannot pronounce the names "Samantha" and "Samuel" given to the older children by their parents. This example will be used in a number of cases to illustrate points in the remainder of this chapter.

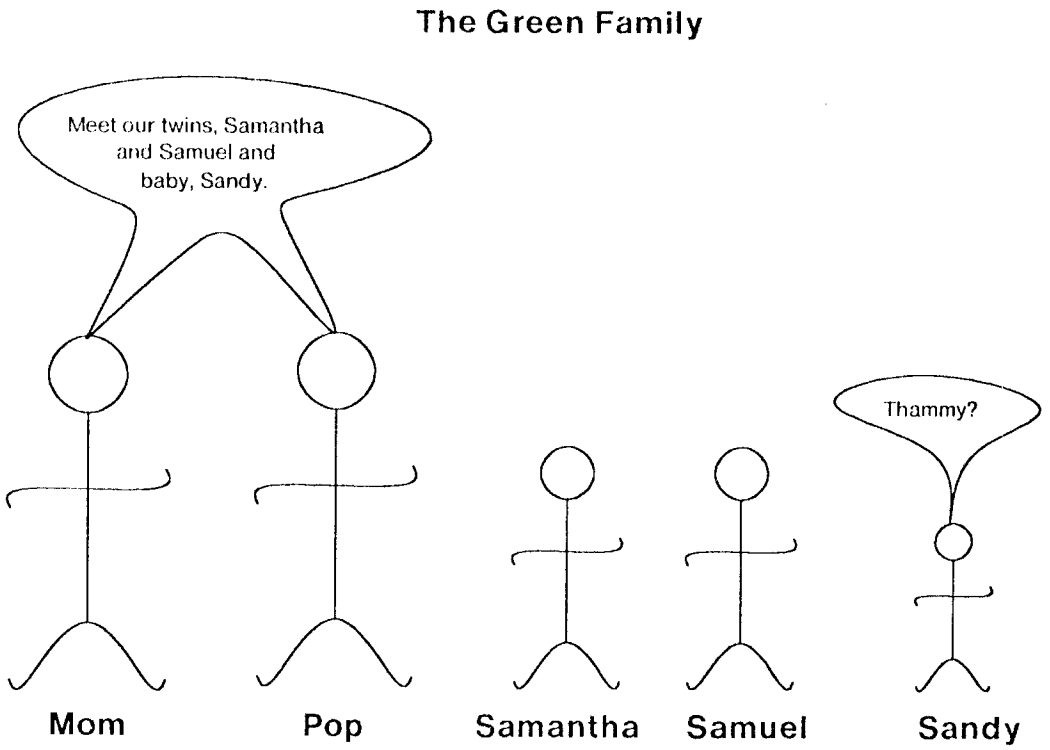

Figure 2-1: Examples of naming issues

2.2 The Definition of a Name

Definition: *A name is an object that can be associated with another object and has an equality operation that is reflexive, transitive, and symmetric. It has two uses. First, it may provide access to the object with which it has been associated. Second, it may act as a place holder for the object with which it has been associated.*

Association of a name with an object is a function of the namespace within which the name is defined. A name can be defined in different ways in different

namespaces, resulting in accessing different objects by use of the name. The equality operation in the definition of a name is an operation on names, whereas assignment or the act of associating a name with an object is an operation on a namespace. Therefore the function of providing access to a named object is also a function of the namespace. In order to understand the definition of a name better, the two uses of names are investigated separately. This section concludes with a discussion of the function that is the reverse of accessing an object, an additional possible function of a naming facility, although not a function of names themselves.

Access

The function that is most commonly considered in naming is the resolution of names. The desired response must be recognized when a user requests that a name be resolved in a particular namespace. First, in most naming facilities it is assumed that there must be a single response in most situations in order that the name be resolved correctly and that it be considered a valid name. This is certainly not true when humans doing the naming. Consider the baby Sandy asking "Thammy" for help. After doing it once, the child learns that several people may respond despite that fact that only one person may have been intended. Humans have developed many techniques for disambiguating, when that is important. But they also may take advantage of the ambiguity. The point here is that a single or a particular resolution is not always what is most useful. In this research, the possibility of multiple resolutions for a single name is not excluded. In cases where multiple resolutions occur and a single one is needed, further resolution or selection using non-naming operations will be required.

A second aspect of name resolution is the actual translation of a name. There are two sorts of entities that can be returned to the user of the name. The first is an object or what appears to be an object to the user. In this case, the user does something with the object such as hand it to a service that will print it, copy it,

modify it or perform some other operation with it. The other alternative is that another name is returned to the user of the name. Not all systems allow for this. Those that provide linking, aliasing, or other forms of synonyms may be prepared for the return of names instead of objects in at least some situations. The names are simply a form of indirection. In their most general form, such translations provide another name in another namespace. A common form of this can be found in the telephone book. A name is resolved to a telephone number, not the person; further resolution is needed. The telephone number is a name that the telephone system understands. To review, the naming facility will allow for one or more responses to a request for name resolution and those responses may be either objects or other names, that may or may not need further resolution.

Place holder

The other use of a name is as a place holder for an object or indirect reference. Names provide one of the same facilities in communication that pronouns do in grammar. They allow for identifying something without actually having the object in question. The situations in which such a facility is useful are those in which containment of the object is impractical or impossible. For example, the object may not yet exist or when the time comes, one of several objects will be chosen by some other selection criteria to be used as well. The flexibility of delaying the binding of name to object may also be important. In addition names allow for multiple, physically disparate references to the same object. If names did not exist, it might be necessary to have two copies of the object, making sharing impossible. Thus, names serve an important function of standing in for the objects they are naming, to both provide sharing and allow for delayed binding.

Finding a nickname

Consider a situation in which one of the parents sees one of the twins doing something dangerous. The parent says, "Samantha, no, Samuel, watch out!" The

parent is searching through the set of names relevant in that context to match the person being warned. This operation is the reverse of accessing an object given its name. In this case, a name is needed for an object. The same issues are relevant to this "untranslate" function as to the access or "translate" function. If multiple names have been assigned to the object, as with a name being assigned to several objects, it is possible that one will need to be selected. The naming facility cannot know which one to select; this function is outside the naming facility. The object for which a name is being sought may be either a different type of object or another, possibly less meaningful name. Finally, if the untranslate function is to be supported, an equality operation is needed on objects, in order to implement the comparison of the object for which untranslation is sought and the objects named in the naming facility. The untranslate function will recur in discussions of both the model and the implementations.

2.3 Aspects of Names

A set of aspects of names, by which names can be characterized, can be derived from the definition of a name. These characteristics fall into two categories, some identifying the participants in name management and others relating to use of names. As listed here, the first three fall into the former category and the fourth and fifth into the latter category. In order to provide a preliminary understanding of these five aspects of names, an example from Figure 2-1 is given here. Each aspect is then discussed below in further detail, including when relevant the general form of appropriate operations.

- *Assignment*: Mom and Pop chose the names "Samantha" and "Samuel."

- *Resolution*: Samantha and Samuel recognized the name "Thammy."

- *Scope of use*: Although "Samantha" and "Samuel" are the names given to the twins, these are not names that Sandy can pronouce and therefore use. As a result, *Sandy tries "Thammy"* instead.

- *Uniqueness/Ambiguity*: Sandy tries "Thammy" but it might be applicable to either twin. This may or may not be the desired effect.

- *Meaningfulness*: Sandy Green is possibly sandy (perhaps indicating hair color), probably not green, but is a member of the family named Green.

These examples are only that. Each of the points listed above also needs further explanation and discussion. They are discussed separately below.

Assignment

One of the three sorts of participants in name management is the name assigner, the other two being the name resolver and the user of names. The generic form of the operation used for assignment is this research is **add_name** (name, object). There are three possible sources for name assignments: an external naming authority, the object itself or some representative of the object such as its owner, and the users of the names Each is discussed separately.

In many examples, such as Grapevine [5], Clearinghouse [36] and the Arpanet [15, 31, 32], naming authorities are hierarchically organized to allow for distribution of responsibilities. Registration of a new name in Grapevine requires contacting an administrator who will add the name. The hierarchical structure reflects a distribution of the responsibility in recognition that a single authority cannot manage such a job alone. Distribution of name assignment responsibilities is also one of the reasons for the move from a network information center being the sole allocation authority for names of networks and hosts on the Arpanet, to the domain scheme, in which the authority is delegated hierarchically. Unfortunately, neither the central authority, nor even the hierarchically structured set of authorities addresses all the needs of a community of name users. A hierarchy does not reflect multiple overlapping groups, nor does it allow for the individual to play a role except in the extreme situation in which every individual is a separate naming authority.

A second source of names is the object itself or someone directly responsible for it. Two examples of this are people choosing their own names for themselves, and the creators of files choosing names for their files. The individual will understand his or her own needs, but may not realize implications of choices of names on the rest of the community. For instance, a programmer may write a new archival facility that uses data compression. The programmer may also have written a special data compression procedure unwittingly choosing the name "compress", although other procedures were available by the same name. A question about which compression algorithm is used must be resolved. Such a decision often uses name resolution and may have surprising consequences for the user. Thus, although privately chosen names solve some of the problems and hierarchies solve others, neither suffices.

A third source of names can be the users of the names. Consider the following situation. A group forms to discuss a problem. They discover that there are two Alexes in the group. In order to distinguish the two in conversation, as a group they decide that they will use middle names for each of them. Thus one is called "Brown" and the other "Harrington." Neither of these is a name that would have been chosen by an authority nor by the individuals although the two Alexes realize that if even one of them is called "Alex" there might be confusion. This is a problem that neither the naming authority nor the individual might consider, but it is important in the area of naming and relevant to the question of how names are assigned.

Resolution

Name resolution involves translating names into objects by recording name assignments at earlier times. Therefore the name resolver is that entity that performs the **add_name** operation previously mentioned as well as the **translate** (name) operation. The name resolver will make use of the equality operation on names in order to achieve translation. There are many examples of name resolution.

In the example above, Samantha and Samuel are performing name resolution by translating the name "Thammy" into themselves. A file system is a name resolver. In the Arpanet, the IMPS that translate net addresses into routes are name resolvers. The list is endless.

Scope of names

The third aspect in considering the management of names is their scope or who can use them. In this case, the two uses of names come into play. It is the user of a name that will invoke **translate**. It is also the user who may use a name as a place holder for an object. There is no operation involved here in the use of the names itself. One can ask whether a name has a global scope, in which case it has been assigned and its resolution is the same everywhere. Or is it private to an individual? As in the case of the two Alexes, is it of interest to a group of users, although not to the whole universe? There are examples of attempts to create global names. This was the situation originally in SNA [10]. SNA is representative of a collection of similar situations, in which it is assumed that there is a single, global namespace or domain within which names are used. At some point, the developers discover that there is a need to connect two of these global namespaces. Each has the idea of unique names in a global namespace so embedded in it, that a very difficult problem confronts the architects. In SNA, the choice was to maintain the separate namespaces, and build a wall between the two, never allowing names from one to move to the other, but only providing translation at the boundary [3]. The idea was to present to the user of such names the appearance of a single, global namespace. This is only a facade, and the user may discover by moving across that boundary that the namespace is indeed not a global namespace in which names have the same meaning everywhere. Source routing [43, 35] provides the other extreme from a global namespace, in which a particular name for a particular object must be completely local and dependent on the user of the name. This situation has the

problem that names cannot be shared, thus obviating one of the main uses of names. But there is a third possibility, a middle area, in which groups share names and their resolutions. Rom's [41] proposal falls into this middle area. In his scheme those who need to know the names do, and, for those who do not, there is no problem if the namespaces overlap. He proposes an algorithm for changing names within each scope so that all names within that scope are unique. He recognizes that this need be carried no further than the boundaries of use of a name.

Uniqueness/Ambiguity

Orthogonal to the determination of the participants in name management is the issue of uniqueness of names. There are three issues to consider when discussing uniqueness. The first is the desirability of it. The designer of a naming scheme must determine whether any form of unique naming is needed. The second is the degree of uniqueness needed. It may be that a name should be used no more than once, but that synonyms, multiple names for the same object, would be useful. On the other hand, it may that each name can be assigned no more than once and that each object can have no more than one name assigned to it. Finally, feasibility must be considered. This was discussed in relation to federation earlier. It is possible that regardless of the decisions made on the desirability of uniquess and the degree of uniqueness needed, it is impossible. The uniqueness/ambiguity characteristic of names is observable in the two operations mentioned above, **add_name** and **translate**. If names must be unique then **add_name** may fail due to duplication, while if ambiguity is permitted **translate** may return more than one object. In this latter situation, further selection may be needed, either by inquiring about additional names for the objects in question or by considering other aspects of the object, such as its type or state.

Both ambiguity and uniqueness have their uses. It is frequently important to be able to identify or select exactly one object within a set. In fact, it is often assumed

that each object within a set is distinguishable by name from all others. Executing a piece of code and specifying on which data object it must operate requires identifying each, distinguishing them from all other possibilities. The simplest form of such identification is to use names, avoiding the use of selection procedures sometimes used to create or distinguish objects based on other information. Such a name needs to be unique within a namespace. If the universe is small enough, it may be simpler to use a global or universal namespace rather than dividing or modularizing the namespace, as is often done to create manageable sized namespaces within which names can be unique.

In contrast, there are situations in which a lack of uniqueness is important. Consider briefly Figure 2-1. The baby Sandy may say, "Help, Thammy, I'm lost." To Sandy, it is more important that a familiar face be found than whether it is Samantha's or Samuel's. In a technical example, if data is replicated in a distributed system, the user may not need or want to know which copy is being used and would prefer that the system determine which copy is most easily accessible at the moment. Both uniqueness and useful ambiguity can be seen simply in a file system such as Multics [37] where a name may be a complete path name to distinguish a particular segment or a short name, allowing the search rule mechanism and Known Segment Table to provide the final resolution of the name at the time of use.

A further extension of the idea of ambiguity or lack of uniqueness can be found in the concept of a *generic* name. Such a name identifies a class of objects that have some set of attributes in common. The generic name allows for identification of objects based on that shared set of characteristics by being a label or place holder for the set. This is a direct adaptation of Quine's [39] concept of *general* naming.

The problems of feasibility must also be considered, especially in a federated computing facility. If there is an authority that can guarantee uniqueness of names

either by generating unique names or by verifying the uniqueness of names presented to it, then it is feasible to base various schemes on the fact that unique names are available. A centralized computing facility can probably make such a guarantee, although even in this case, it is difficult. One technique for generating unique identifiers is to use sequential numbers, reliably remembering the previous number that was used. This is feasible only if the numbers can be generated quickly enough and if the means of remembering is reliable enough. Some systems have used the clock to generate names, assuming that it is both reliable and fine grained enough. Another scheme is to subdivide the set of names, allowing each of a collection of authorities to manage a subset of the names. This provides some relief for the problem of a single authority being a bottleneck, but it increases the probability of duplicate names if unreliability is a problem. For example, if there is a power failure, instead of a single authority possibly handing out a duplicate name, each authority may hand out a duplicate name.

The problem of feasibility becomes more complex with consideration of merging namespaces in which the names have been selected by independent naming authorities each of which assumes that it is choosing globally unique names. The problem in this case is how to deal with unexpected duplicate names. Both Rom [41] and the architects of SNA [10, 3] dealt with the possibility of duplicate names because it was important to each of the underlying architectures that names be unique. Rom's decision was to replace duplicate names invisibly, while the SNA solution was to keep two namespaces separate, but gloss over that fact at a higher level. In fact, this is not how humans address the problem in their communication. Instead, they live with the possibility of ambiguity, recognizing that globally unique names are not possible, and they manage without them, relying, when necessary, on locally unique names.

Meaningfulness

From the points of view of the assigners and users of names, those names can fall anywhere in a range from those that have no relevant meaning to those that also carry a great deal of information about the named object. The simplest names carry no meaning and are only labels. One example of these is the set of numbers generated by a random number generator and used for labelling objects. Any relationship between any two such names is purely accidental. A user of the name "Sandy Green" is unlikely to assume the named person is in any way green, but may assume blonde hair from the name "Sandy". The nickname "Teach" may not only be a name for a person, but also carries the information that the person so labelled is probably a teacher. The name "President" not only identifies an individual, but also indicates the relationship between that person and other members of an organization. Furthermore, humans sometimes associate an attribute that is to be used as a name with an object, e.g. "position: president" so that in the future one can identify the person with that name. It still is the case that the name must have been assigned as a name in order to be one. This is separable from whether or not it is meaningful.

A further extension of the idea of identifying an object by information leads to identification of an object by aspects of the object that may not have been preassigned, but have meaning in relation to that object. For example, consider a situation in which family names have been recorded for people, but not substrings of those names. Then, selecting those people whose family names contain the string "ollins" but for whom that is not their full family name is not naming. In addition, information about an object may take a form similar to that of an attribute. An example might be a timestamp of creation for an object, in milliseconds since the beginning of the century, such as "CreationTimestamp:27162241234". It is improbable that anyone will ever use that information as it stands as a name,

although the information may be used as part of the selection process of finding an object. For instance, one might want to find all the objects created before a particular time. This sort of identification and selection is not within the bounds of what is identified as naming in this research.

Recognized structure in names is another form of manifest meaning. If a structure is understood, components of that structure are recognized as having meaning. The simplest structure is a flat namespace in which case each name is composed of a single component. Two examples of flat namespaces in networks are RSCS [17, 16] from IBM and the older form of naming hosts on the Arpanet [31, 32]. In addition numerous simple file systems and user identification schemes as well as other examples support only flat naming. A second common structure is the hierarchy in which the nested components may reflect meaning or another one of the issues discussed in this section. A third form of organization is the directed graph, where each node may have more than one parent and more than one offspring. The schemes used in R* [29] and the IFIP WG6.5 proposal [18, 59] fall into this category. In these cases a set of name components may be presented to the user as a choice of hierarchies or as an unordered set of components. It is this third possibility that seems to reflect the structure of names that humans use most often.

The manifestation of meaning is an unstated issue in the work of Saltzer [44] and Shoch [46]. Both realized that different names manifest different sorts and degrees of meaning to different assigners and users of names and each author based his characterizations of names on the views of those assigners and users of names.

These five attributes of names allow for comparison among different naming schemes along orthogonal axes. The three roles in terms of choice and use of names address the questions of who plays those roles. The choices can be related to each other or independent of each other. The degree of uniqueness or ambiguity

determines repeatability of assignment. Finally, the degree of meaningfulness determines how much and which information can be conveyed by using a name as a place holder. None of these aspects of names needs to be dependent on the others.

2.4 Aspects of Human Naming

As mentioned earlier, humans provide a useful paradigm for investigating naming in a federated computing facility. Therefore, it is useful to understand how humans name. The following is a list of observations about human naming that were listed briefly in Chapter 1. Each will be considered here in more detail. In addition, where relevant, related literature will be noted. These eight observations form the basis of a further understanding of the goals of this research in relation to supporting naming in a federated system and providing sharing and communication of and through names.

1. **Communication:** There are two aspects of communication. One aspect of communication is cooperative use of names. In addition, information related to named objects may be shared and passed between the user of a name and the recipient of the name by passing meaningful names. The individuality of each communicant is closely related to joint naming and shared responsibilities for names, although that has been separated here as a distinct issue.

   Examples of sharing namespaces can be found in many other works. The most common place where operating systems provide sharing is in their file systems. Hierarchical structures such as those of Multics [37] and Unix [40, 57] provide sharing by the use of working directories and search rules. Non-hierarchical systems such as OS6 [48, 49], Eden [1, 19], and CAP [33, 34, 60] also allow for similar means of switching namespaces or resolving names in other name spaces.[3]

---

[3]The Alto operating system [25] also provides a non-hierarchical structure, although it is a single user system and apparently little use was made of any facilities for dividing the namespace into directories or subdirectories.

Multics also provides an interesting example of local shared naming, that was designed with a particular issue in mind. For each process, there is a Known Segment Table that maps a nickname into a particular segment on a per process basis. The table is shared by all procedures running within that process. When a local, short name is used in a procedure, the system checks the search rules for the means of resolving it. Normally, the first entry in the search rules is the known segment table, followed in any order by the directory of the calling procedure, the working directory, the user's home directory and any other directories specified by the user. None of these is required and they can be in any order although some orderings will lead to unpredictable behavior.[4] The idea behind this mechanism was that if a nickname were used in a number of procedures, it should be resolved to the same segment, so that, for instance, if one were working on a database, all the procedures would share the database. On the other hand, it also can provide for anomalous behavior, when the programmer of a procedure had a different resolution of the name in mind. For instance, it is possible that two different procedures may have the same name, but provide different functionality and different results using different arguments. Despite this potential problem, the shared nicknaming facility is commonly used in Multics.

2. **Individuality:** Each creator of names is different. Those differences are manifest both in the individual's set of experiences and decisions based on those experiences. No two individuals have had exactly the same set of experiences. In addition, in the same situation two individuals will make different choices.[5] Therefore, in any joint decision such as choosing names, individuality also plays a role.

   Various forms of private nicknaming, linking, aliasing and synonyms support the individual as distinct from the community. In Multics and Unix, local linking to segments or files in other directories supports private names for these objects on a per directory basis. In addition,

---

[4]For example consider not putting the known segment table first. This can lead to multiple occurrences of a name in the known segment table. If the known segment table is used to resolve the name, which resolution is used will be implementation dependent.

[5]No implication of a causal relationship between experiences and choices should be interpreted from this.

both systems support aliasing on a per user basis, allowing the individual to personalize the names used for invocation and other forms of naming as well. Synonyms can also be found in many systems. For example, in R* as part of a more complex naming and cataloguing scheme [29], Lindsay has proposed private synonyms. These lists are on the basis of an individual user at a particular site. Many other systems (such as mail systems providing private templates) also support individuality to one degree or another. Just a sample has been discussed here.

3. **Multiplicity of names:** Allowing for a particular name to identify different objects and for different names to identify a particular object, provides a flexibility present in human naming, but often not in computer systems. For example, many people have the same nickname. It is often advantageous to name people having the same family name by referring to them by their family name. In addition, in some cases name assignment varying with the situation and time may be useful. For example, the title "Chair of the committee" will be resolved differently depending on which committee is being discussed and when. The other side of that situation is that such duplication in names may sometimes be confusing. In those cases, locally unique names such as nicknames may be created.

   Again, there are many examples of multiplicity in the literature. Source routing [43, 50] provides an important one. As its name implies source routing is a mechanism by which an object is named at the source of the name by the route from the source to the object. One distinguishing characteristic of source routes is that they are dependent on the source and therefore imply multiple names. In addition, the forms of naming mentioned under individuality also support multiple names, although there are other forms of multiple names as well. They can be found for instance in IBM's SNA in the mechanism for joining two SNA networks [10, 3]. SNA provides a static hierarchical structure for internetworking and aliasing local to each single network, providing multiple names for hosts, although from any location only one name is accessible. The aliases may not escape the local network and are shared by all users of the local network. Within a single network the namespace, including aliases, is flat. Thus, in an internetwork of SNA networks, there may be a different name on each network for a particular host. Both Clearinghouse [36] and the IFIP Working Group 6.5 work on names and directories [18, 59] support multiple names

explicitly. In Clearinghouse one name for each object is more important than all other names for that object, while the WG6.5 work has no such mechanism. All are equally valid as long as they define a complete set of component names, one component from each naming authority on a directed path from the root to the destination. In fact, multiple names fall into two categories. The first category contains those names that allow only different names from different perspectives, such as links in a hierarchy in which any object can only be named at most once from any directory. The second is synonyms within a single namespace such as Lindsay's set of synonyms for R*.

4. **Locality of names:** Conversations are a common source of local naming. Within a particular conversation, the participants will define the names that they are using locally in that conversation. As they move to other conversations, those names may have different meanings. For instance, the name "Alex" may identify one person in one conversation, and someone else in another. If both Alexes participate in a single conversation, the group of participants may agree on different names for each of them, or find other ways to distinguish them. Locality is used by humans constantly in order to avoid having to provide unique names over all experiences.

   Directories, whether in hierarchical or non-hierarchical file systems, are one of the most common forms of providing local naming. This can be seen in Multics and Unix in their hierarchical file systems as well as those previously mentioned non-hierarchical file systems. The need for local naming can also be found in networks. In SNA [3], although the attempt has been to provide an image of a single namespace to the user, in fact what is provided is a collection of local namespaces each consisting of an SNA network. To move from one namespace to another the user must move from one SNA net to another. The domain naming project in the Internet [31, 32] is aimed at providing local namespaces by dividing a single namespace into a hierarchy. In R* [29], full names consist of four components: the creator's name, the creator's site, the site of creation, and a name that is unique given the other three components. Local naming is supported by supporting defaulting of any of the first three components. Saltzer [42] in his treatise on names discusses the need for locality in naming even in a centralized facility in order to achieve modularity and provide for sharing.

5. **Flexibility of usage:** There are several sorts of names that humans use in addition to unique, or relatively unique, names. For example, names that reflect role or position, reflecting relation to others, form one group of names. The names "Cousin" and "Chair of the committee" are two such. These fall into the category of generic names. An example of a different sort of name is "the green one." In this case, the name is descriptive. It reflects something of the inherent nature of the object being named. The different sorts of names implied here reflect different means of incorporating meaning into names.

   There is not much work on supporting different sorts of names for the same object other than in Clearinghouse [36] and the IFIP Working Group 6.5 [18, 59]. In Clearinghouse an object can be named both by its unique name that may carry no meaning and by a set of properties having values. The WG6.5 project supports the possibility of multiple paths through a rooted directed graph, allowing for name components ranging from those that are simply unique within a set, but otherwise have no particular meaning to names that are attribute pairs and have meanings.

6. **Manifest meaning of names:** When objects are given names that have meaning as well as providing identification, and those names are shared among a group of people, it is assumed that those names also will be understood by the whole group. If people do not understand those meanings, they will have difficulty remembering the names. In addition, as seen in several other works such as the WG6.5 project [59] and Multics [37][6] when a namespace is divided, one of the goals is that the components of the name be meaningful and therefore guessable by the potential users of the name.

   Communicating and sharing meaning is often provided as part of the structure of names. This can be seen clearly in some file systems. Multics and Unix again provide an example. The hierarchical structure of directories is often used to provide part of the name of an object and allow that part to have some meaning. An example from Unix might be "/usr/sollins/lib/mail.ml". This identifies a library written in Mock Lisp [14] that supports a mail system, and belongs to the user "sollins".

---

[6] These are only examples.

Multics supports a similar syntax. Supporting the sharing of the meanings of names was also one of the goals of the IFIP Working Group 6.5 in the mechanisms provided there and described previously in this work. The property lists of Cocos [11, 20] and Clearinghouse [36] also have the same effect of allowing users of names to share meanings by incorporating a means of allowing for meaningful names into the naming facility.

7. **Usability of names:** It is easy for people, talking to each other, to define and redefine names thus providing multiple names, if one does not suffice. In addition, without appearing to think, people can reflect upon the choices of names and select the ones they want. This must all be easy to do when communicating with and through a computer system, as well.

   Providing usability in naming facilities is generally not one of the primary goals in designing naming mechanisms. Lindsay [29] in R* worked toward a naming facility that would make name resolution simple for the user. His defaulting mechanisms certainly were a step in that direction. In fact linking and the default name resolution provided by allowing the user to specify both a working or current directory and a set of search rules are also a step toward making naming facilities more useful without adding to the burden placed on the user of names. These facilities have already been discussed in other contexts. In a more general sense, all naming facilities are trying to make computational facilities more usable.

8. **Unification:** Finally, although several researchers have recognized that the mechanisms used for naming one class of boject are also useful for others, there is an added argument in favor of a unified naming facility. In discussing flexibility it was suggested that generic names may be useful. A generic name may reflect an entity that is not recognized as a single type of object in the computer system. Instead humans apply the name to a collection of objects, each of which may be a different type. This is essentially what was done in Clearinghouse, with properties. A user has a set of properties, that may, for instance, reflect different ways of reaching the user, such as a list of electronic mailboxes, a phone number, and a US postal address. In fact, these are all different objects, that have been organized hierarchically, presumably because access to the information is to be based on property names within user names.

> Clearly those researchers designing and building general name servers such as Dalal and Oppen in Clearinghouse [36] and Lantz and Edighoffer in UDS [28] recognize the general applicability of solving certain naming problems in such a way that the solutions are usable in many domains. In addition, several researchers have discovered after the fact that their solutions were applicable to other problems. An example of this can be found in the Grapevine project [5], where although it was not planned this way, the authors found that the mechanisms that they developed for naming mailboxes also served their own needs of naming other services needed by the mail service itself. So Grapevine uses its own mechanisms behind the scenes to provide some of the user level services. In addition, Grapevine registration servers that keep track of names are used for non-mail applications as well, although the details of those uses are not in the published literature.

With this list of observations, the discussion of the problem addressed in this research is complete. The final section of this chapter discusses a further set of problems. Some of these problem are generally considered unsolved while solutions to others are often sought in naming facilities.

2.5 Additional Problems

The definition of names and the goals for a naming facility assumed in this research are broad and simple. The reason for this choice was to provide the common functionality needed for many different sorts of applications. Frequently, when a naming facility is built for a specific application or subsystem, greater functionality is required of the naming facility. Therefore certain naming facilities address problems that may not be addressed by the facility proposed in this research. This section contains a list of the most common of these additional problems solved by some naming facilities. In some cases, the problems identified here represent problems that even humans with their much more sophisticated naming mechanisms cannot always solve satisfactorily. This list of problems will recur in Chapter 4 in a discussion of how the proposed model addresses some of these problems, in spite of their not being goals of the research.

The reply-to problem

When a message or some other information is delivered to a user, it is often tagged with a name for the sender or source of the information. There are many situations in which that name is either ambiguously defined or undefined in the receiving namespace. For example, at MIT, one of the computers is named "Comet". In addition, one the computers at Symbolics is also named "Comet" and the networks of the two organizations are interconnected. If someone at Symbolics on Comet sends mail to someone at MIT, unless the mail systems change the name Comet to SCRC-Comet (for Symbolics), the recipient will not be able to respond to the sender, since the name "Comet" within MIT identifies a computer on which that sender does not have an account. In a more aggravated form of this problem, there may be different users with the same name, one on the MIT "Comet" and one on the Symbolics "Comet." The reply-to problem is that one cannot always reply to a name, despite the fact that mail arrived from a person with that name. When this problem is specific to networking it is often labelled as the problem of source route translation.

The name-equality problem

The name-equality problem arises in trying to answer the following question; *given two names do they identify the same object?* This is a particularly difficult question, and although names are often used to answer it, they do not provide the whole answer. In a world where every object has a system-wide unique name (possibly in addition to other names), and access to that unique name is provided, given two names they can be resolved to their respective objects. By discovering their system-wide unique names and comparing them the question can be addressed. In other cases, the objects themselves may support an equality operation.

In addition, there are other considerations that come into play. For example, in an environment where objects are strongly typed, an object may be wrapped in layers

of typing. Consider comparing an object with the object that is its representation[7]. It is not clear whether the two are the same or different objects since underneath it all they are represented by the same collection of bits, but at the higher level they may not be accessed by the same mechanisms and the user may appear to be very different from each other.

A reverse sort of situation may arise, in which an object consists of multiple copies kept in different places and reliably maintained in a consistent state. It is certainly possible to find two different names for different copies of the object, but at some level, even though the names are different the two may be considered to represent the same object. In this situation, two different collections of bits may represent the same object.

The question of identity and how it relates to names is complex, and simply answering the question of whether or not two names resolve to objects that have the same or different globally unique identifiers may not in fact answer the deeper question that is being asked. The problem here is that although the assumption may be that the question to be answered is the one posed above, in fact there is a collection of more specific questions that need to be answered, and a function that answers the one above does not answer the more complex ones. In fact, all the possible questions cannot be enumerated, because there will be at least one for each type of object, and all types of objects cannot be enumerated. In addition, the number of questions will be dependent on the uses of those types, again impossible to enumerate. Thus, the name-equality problem persists.

---

[7]This is the terminology that is used in Clu [30]. An object is of a particular type defined by the type name and the names of the operations and their arguments and is realized by being represented by another object of another type. The system provides a small number of basic types.

The who-is problem

The who-is problem is similar to the name-equality problem but reaches beyond the bounds of the computer system, and is therefore related to the goal of providing for the manifest nature of names. The problem here is the following; a person has received a name inside the computer system, and knows about an object or person outside the system. The recipient of the name would like to test for equality between the inside and outside worlds. This is an especially difficult problem, because outside the computer system, humans will use a large array of other facilities, perhaps making use of the five senses as well, in order to address the problem, and those are not available inside the computer.

The mobile-name problem

Part of the goal of multiplicity is to allow a name to be used for more than one object, but there is a problem that can arise from this. In some cases, such as "Chair of the committee" the name must be assigned to no more than one object at a time, but which object is being named may change over time. The mobile-name problem reflects this mobility of a name. The problem may be compounded in spanning multiple computers.

Location transparency

It is very difficult to separate a naming facility from location of the user of the names. If a user has access to a set of names in one location, when he or she travels across the country, the names that he or she uses should be the same. The person is the same and the objects being named are the same, but in too many situations, the host through which the user is accessing the computational resources has a strong influence on the names that are available. This problem is labelled location transparency. It makes naming much more difficult for the user.

The "a" vs. "the" problem

This problem can best be understood by considering a person asking for "a book about genetics" initially and then following that with future requests for "the book about genetics." In the initial request, one of a collection might have sufficed. After the name was bound once to a particular book, that one was the only one that would suffice. A first step toward addressing this problem can be seen in the Known Segment Table in Multics, but generally this is not a problem that has been addressed thoroughly in naming facilities.

Selection

Both the goals of multiplicity of names and the recognition of generic names will lead to the problem that a name may not map into a single object in a situation in which a single object is needed. This problem is common for humans who have a large array of mechanisms to call into play to address it. They may ask about other names assigned to the possible choices. They may call defaulting procedures in to play. They may ask about the nature of the objects. They may ask whether any of the choices is one that they have chosen previously. They may ask for recommendations from others. And the list goes on. The problem is not a simple one, nor are the potential solutions. Selection functions appear not to be generalizable and are best left to specific applications to handle.

Persistence

Many facilities have a short-term and a long-term naming mechanism for objects. Programming systems are a prime example of this. Consider the runtime system for Clu [30]. In this case, objects can be named as typed objects within the language, but such typed objects are not persistent; they cease to exist with the completion of execution of the code. The file system is another naming facility for naming persistent objects. In order to make an object persistent it is translated from its runtime form into a form that is stored in a file, which in turn is identified through

the file system. Clu provides a facility, albeit somewhat awkward, for retaining some type information when an object is transformed using the "gc-dump" facility to save an object in a file.

A second example naming persistent objects can be found in the Macintosh operating system [2], in which files containing data have associated with them the program that created them. When a file is "invoked" that program is invoked operating on the data in the file. The Eden system [1, 19] provides a third example, although it is still in the prototype stage. Finally, the Swallow repository [52, 51] was a prototypical storage facility designed to support objects rather than files. There are other such research projects, but the idea of persistent objects is not widely accepted yet, and it will be a long time before the small step taken by Apple in the Macintosh will move even the small set of researchers, much less the larger group of programmers, to the recognition that all objects should have persistence as they do outside the computer system.

This concludes the discussion of the problems that are and are not being addressed by the naming facility modelled in the following two chapters.

2.6 Summary

The emphasis of this chapter has been on the problem being addressed in this research. The problem itself can be stated simply as the design of a naming facility that supports names and the functions for which they are used, allows for communication both of the names themselves and of information by means of the names, and is implementable. In order to design such a naming facility, one must understand names, the definition of them, what their functions are, and how they are used. The definition of a name is simple. A name is an object that can be associated with another object of any type and that has an equality operation on it.

A name has two possible uses. It can be used as handle providing access to the object named by it, or it can serve as a place holder for that object.

There is more to understanding the naming problem than these simple definitions. The assumption of a federated computing facility means that not only will cooperative activity occur at the convenience of the communicants, but also that it will be intermingled with periods of isolated activity. It is the need for cooperation while allowing for autonomy that makes the problem more difficult. Human interactions provide a useful paradigm for understanding the patterns of communication and autonomy in a federation of computers; therefore, human interaction and naming was explored in order to understand the problem in a federation better. Section 2.4 presented a list of observations about human naming that are taken as subproblems of this research project.

In addition, there are a number facets of naming that can be used to understand and compare naming schemes including the one to be proposed in this work. They include identification of the participants in the naming activities, the assigner of a name, the resolver of the name if it is being used for access to an object, and the user of the name or the scope over which the name is known. Furthermore, two additional attributes of names are the degree of uniqueness of a name and the degree of meaningfulness. The degree of uniqueness is reflected in whether or not a name can be assigned to more than one object or not. Meaningfulness reflects the information that is inherent in the name and therefore can be carried in the name itself when the name passes from one user to another.

The definitions of names and the problem being defined in this research are somewhat different from past related work. Others have often imposed a greater functionality on names and naming facilities, losing generality by including functions that are application specific. The definitions chosen here were selected for

their generality and therefore the assumption that a solution that addresses them will be of general applicability. The next two chapters present the proposed solution, a model for a naming facility. Chapters 5 and 6 address the issue of implementability.

Chapter Three

Sharing and Individuality: The Model, Part I

3.1 Introduction

This chapter and the next together describe the model for a solution to naming in a computer federation. Chapter 2 investigated the computer naming problem posed in this research in detail by comparing it with human naming. This comparison led to a fuller description of part of the problem based on the observations of human naming as well as discussions of the uses of names and a better understanding of an orthogonal set of characteristics of names. Human naming is a complex and rich set of mechanisms. In order to create a mechanism that is currently implementable, the model proposed here is an approximation. It is not presented itself as a proposal for the mechanisms used by humans, but rather it is a mechanism that exhibits an approximation to human behavior in order to meet the goals of this research.

The method for discussing the model is as follows. The model consists of two newly defined types of objects. One new type, **context**, supports sharing of names and name management among a group. The other new type, **aggregate**, provides an individual's viewpoint on those shared objects. Each type is discussed separately, although the two discussions follow the same pattern. The set of issues related to joint management and shared responsibility for shared contexts is separated and discussed in Chapter 4 in order to simplify presentation of the material. These two chapters together describe the model. Therefore, a summary of how the model achieves the goals is left to the end of Chapter 4.

The discussion of each of the two new types proceeds along the following lines. The

presentation begins with a definition and discussion of motivation and use of the type including such issues as naming objects of this type, initialization, and containment of objects of this type in other objects of the same type. The discussion proceeds with identification of the basic operations on the type. A more complete list of possible operations is included in Appendix A. Finally, implementation issues relating to each type are discussed, including management of multiple copies, synchronization of distributed information, communication media for such distributed information, and a review of initialization questions. The chapter concludes with three examples of the use of the two proposed types of objects, first in a human interaction, and then naming facilities in two existing systems.

3.2 The Context

Definition and Discussion

Definition: *A context is a shared object that maps names into either objects or other names. These mappings are in one of a series of states ranging from unknown or deleted to fully accepted. In addition to the mapping information, a context contains information reflecting the identity of the participants in the sharing and joint management of the context. Any information in a context may vary over time. There are two functions on names supported by contexts: access to a named object and substitution of one name for another.*

In the approach in this research of modelling human use of names, a context represents a focus of interest, and as such may be shared among a group of users of the names. In its simplest form it is based on the idea of a working directory in a file system such as Multics [37]. In such a system, the user can change working directories explicitly to reflect a change in the set of name mappings that is to be used. The idea of names being mapped into other names is a direct extension of the idea of links in a file system that allow a name in one directory to be mapped into another name in another directory.

There are two issues that will be discussed further in Chapter 4 but are worth mentioning here. First, one component of a context that does not have a counterpart in a working directory is the list of participants reflecting the shared nature of contexts. The group of participants is not only the users of names, but also the group sharing responsibility for managing the context. Therefore, as a group they will add and delete names, decide when the context should merge with another or perhaps when it should divide into several. Second, a mapping in a context may be in one of a number of states, reflecting its previous use in that context. Prior to any assignment or use in a context, a name will be unknown in the context. Usage may cause it to move through a series of states until it is fully accepted as a name in that context. Disuse or explicit deletion operations may cause a name to pass through a series of states until it is deleted. Continued investigation of joint management and the states of mappings will be delayed to Chapter 4.

There is one further aspect of the functionality of a context that must be mentioned. A name may be reserved without it being assigned to another name or object. There are many uses for such a possibility. A name might be reserved but not assigned either because the object to which it will be assigned does not yet exist or is unknown or because the name has been deassigned until some further event. An example of the first situation may arise in programming, when a procedure calls another procedure that has not yet been written. The second situation may arise, for example, when a procedure provides a printing service, but the code is found to contain so many bugs that it is temporarily taken out of service. The name by which it was invoked should remain reserved for the time when the code is back in service or a substitute is found.

Figure 3-1 provides one possible depiction of a context. It has five entries including three names for one object, one of which is indirect. Two objects are named. There are three users participating in sharing the context. In addition, there is one name

that is unassigned. Each entry in the context is in one of several states, represented by the letters, "c", "a", and "d", for "candidate", "accepted", and "deleted".

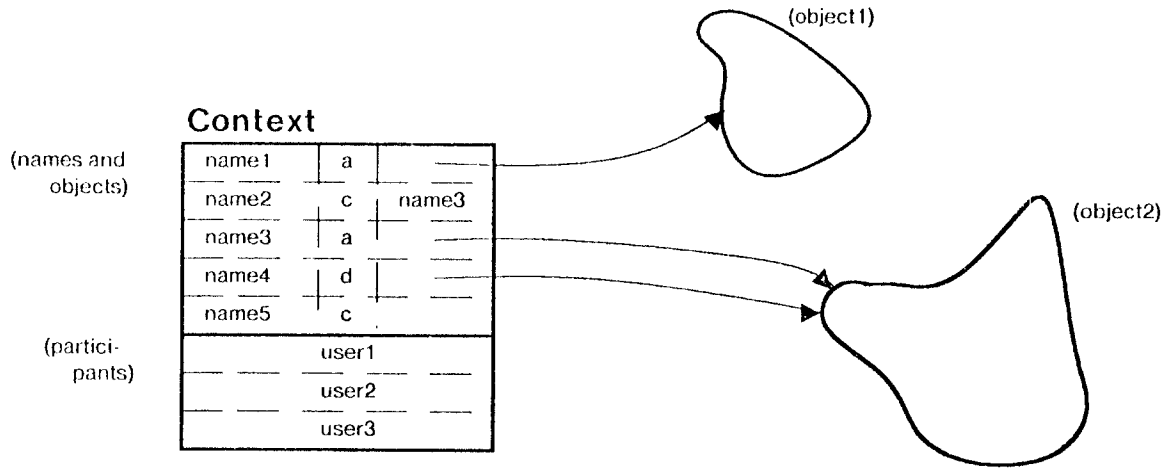

Figure 3-1: Depiction of a context

Operations

There are four operations of primary importance on contexts. In addition, many others are needed to make contexts usable. Only the five basic operations are discussed here; a more complete list is included in Appendix A.1.

create = proc () returns (context)

This operation is the local operation creating a local copy of a context. It creates a context containing no names and only the creator indicated as a participant. Prior to creation of a shared context, negotiation must take place. This negotiation is considered further in the discussion of implementation issues.

add_name = proc (context, name, [object])

A name is added to a particular context. The addition procedure must

take into consideration the issues to be discussed in Chapter 4, reflecting usage of the name and the degree of sharing of responsibility for name assignment. The object argument may be another name or some other type of object. In addition, the object argument is optional because a name may either be assigned to an object through this operation or reserved for future assignment. In this latter case the assign_object_to__reserved_name operation will also be needed. In Appendix A.1 two operations have been provided, one with the object, add__name, and one without, reserve_name. In addition, an operation is then needed to assign an object to a reserved name, assign__object_to_reserved_name.

translate = proc (context, name) returns (set[object])

This is probably the most commonly used operation on contexts. The translation operation takes a name and returns all the objects and names into which the first name is translated with the context provided. The invoker of the operation must be prepared for several possibilities. First, the name may not exist in the context. Second, it may exist but not be assigned to an existing object. Third, it might be translated into another name in another context, and fourth, it might be translated into an object. Furthermore, the invoker must be prepared for more than one translation; the set may consist of representatives from any of the four possibilities.

untranslate = proc (context, object/name) returns (set[name])

As discussed in Section 2.2, this operation is the reverse of translate, although the values returned by this operation are more predictable than for translate. In this case the only response is a possibly empty set of names. Again, the invoker must be prepared for the response being a set of more than one name. This operation was found to be especially useful in the electronic mail implementation because mail would often arrive from senders not using this mail system, but rather their own.

add_participant = proc (context, participant)

This operation is needed in order to define the list of participants sharing a context. The means of identifying participants has been excluded from the naming facility and this research. The reason for this decision is that identifying participants may involve complex activities that certainly do not fall within the bounds of naming as defined here. For example, participant identification may include sophisticated authentication

procedures. All that will be said here is that a mechanism for identifying participants must be available and it will vary at least from one system to another and possibly from one subsystem to another.

In order to use contexts, many additional operations are needed. Appendix A.1 contains such a list. These operations include operations for deletion of various pieces of information, such as names, bindings, participants.

Implementation Issues

The implementation issues for contexts fall into three categories, effects of federation, communication, and naming of contexts. In order to provide service in the face of discontinuities in cooperation in a federated computer facility, a context that is shared across such a federation must be implemented as multiple copies. The reason for this is that if a name has been defined in a locally known context for a local object that name must be usable for that object even if the remainder of the federation is not in communication. In addition, there is a further complication. It is possible to define a context in such a way that any individual participant is allowed to define new names in the context. In this case, if the federation is in a disconnected state, the local user should still be able to define new names in the context. This also points to the need for a local version of the context. On the other hand, local versions or copies require synchronization.

The synchronization need not be perfect. As a result of federation, copies of the context need not be kept in perfect synchrony. In fact, for a human interface such behavior is probably both unnecessary and undesirable. As long as mutual agreement on the contents of the context is eventually reached, it need not occur instantaneously or even atomically. Modifications to a local copy need only occur by the time of next use after their arrival at the local site. This may appear to cause problems, for example, if two users attempt to define the same name in a situation in which each name may have only one translation. Such a situation should occur. If a

context is created with the restriction that a name occur at most once in it and all users have equal responsibility for assigning names, no user can be allowed to define a new name unilaterally. Communication with the other copies of the context is a necessity and such a proposal for a new name can be at best only tentative, pending synchronization with all other participants. The issue of synchronization will be discussed further in the consideration of implementation issues for aggregates.

In building a naming facility, one must consider what information needs to be communicated and how that will be achieved. The second area of concern in implementing contexts is communication. There are two sorts of information that must be communicated in relation to using contexts. The first is the names themselves and the second is the negotiation information related to management of the shared context. Closely tied to this is a determination of the medium of communication. As will be seen in Chapter 5, in the electronic mail system, the medium of communication was the mail itself. The medium of communication and the use of the names will determine the representation form of the names that are passed among participants. In addition, the medium of communication and the objects being communicated will determine the form of communication that is available for the information needed to manage a context. Management information is needed in order to reach agreement on initializing a new context as well as to make decisions about adding and deleting information in the context. There is an underlying assumption in this discussion of communication and initialization that there is some basis for initiating communication. There must be some agreement among the participants on a communications protocol. In talking to someone one has never met before, there will probably be an assumption of a common language and possibly some common experiences. Lacking that there may be an assumption of understanding certain facial and hand expressions. Without some basis from which to begin, negotiation and communication cannot be established.

The final implementation issue in relation to contexts is how contexts are identified. Contexts must be identifiable in order both to manage the information in them and to use them in name translation, accessing objects given names. Since a context is an object it can be named in another context just as any other object can be named. This quickly reduces to a problem of initialization, that was discussed above. Agreement must be reached not only on the fact that a context will be created, but also how it will be identified. Interestingly, humans use more than a name to identify a context. They also use participants. Since participant information is part of every context, it can easily used in the selection process in choosing a context from within which to use names. Because participant identification may not be by name, selection of a context based on participant information does not fall under the responsibilities of the naming facility. This issue of selection versus naming arises in an important role in a programming support environment and therefore is discussed further in Chapter 6.

To review, in this section an object type called **context** has been proposed as the basis for shared naming. It is jointly managed by a set of participants and contain not only the relevant naming information but also some form of identification of the participants. Name translations in a context can be in one of a number of states reflecting previous usage of the name. The basic operations on a context are to create a context, add names and participants to the context and to translate names into objects. In addition a number of other operations are needed for general use and management of contexts. The assumption of a federated computing facility leads to the implementation requirement that multiple copies of a shared context exist, one for each independently operating entity. Further issues that must be considered in any implementation are synchronization of those multiple copies, how communication occurs and what is communicated, the basis for communication, and how selection that is not straightforward naming, such as in selecting a context on the basis of participants as well as an agreed upon name, is to occur.

3.3 The Aggregate

Definition and Discussion

Definition: *An aggregate is a private object that consists of a current context and an environment. The current context is shared among aggregates belonging to the several participants of the context. An environment is a partially ordered set of contexts used in the partial ordering specified to translate names not known in the current context. Any information in an aggregate may vary over time. The functions on names supported by aggregates are access to a name object and substitution of one name for another.*

The view taken in this research is that all naming is done through the naming facility. This is not to say that there are not other ways of identifying and accessing an object, but only that all naming is to be through the naming facility. Each namespace of a user is an **aggregate**. The aggregate is a private view of a shared context. The context is the namespace shared by a group for a particular purpose, with a particular focus. In addition, each participant has his or her private view of the sharing. If a group of people have a conversation, they will jointly define terms and use nicknames on which they have agreed. In addition, the issue of the participants' individuality must be considered. In order to capture these ideas, an aggregate is composed of two components. The first is the **current context** which is the shared context representing the focus of the group. The second component is the **environment**, a partially ordered set of other contexts in which the individual is also a participant and from which he or she may wish to draw information. The idea for the structure of an aggregate is derived from the concepts of working directories and search rules. The current context is derived from the working directory and the environment, from search rules. The user of names would like to be able to draw on other experiences without having to be explicit about it. Unlike the search rules of Multics or Unix, in this research a partial rather than a complete ordering is

permissible. This decision is in keeping with the fact that names may be resolvable to more than one object. If there are several contexts at the same priority in an aggregate, then all resolutions of a particular name in those contexts have equal priority within that aggregate. A "rule" is a set of contexts at a single priority in an environment. Figure 3-2 is one possible visualization of an aggregate. It has the two part current context and an environment with three rules. The first contains two contexts, the second, one.

Operations

The operations on aggregates fall into two categories, those that have counterparts in contexts and those that do not. Even the operations in the first category are not identical to the comparable operations on contexts. The operations on environments, adding contexts to rules and adding rules, are completely new here.

create = proc () returns (aggregate)

Creation of a new aggregate involves creation of a new context as described for contexts as well as creation of an environment. Although this operation involves creating a new context as the current context, in the mail implementation, as will be seen in Chapter 5, creation may involve using a pre-existing context as the current context.

add_name = proc (aggregate, name, [object])

This operation is quite similar to the comparable operation on a context except that an aggregate is identified and the addition is made to the current context of that aggregate.

translate = proc (aggregate, name) returns (set[object])

The translate operation on an aggregate is somewhat different from translation on a context, above and beyond the fact that one of its arguments is an aggregate. The net result is similar, return of a set of objects having the name assigned to them. The difference is in the aggregate's resources used. First, the current context is checked. If there is no translation there, the highest priority set of contexts in the environment is checked (the first rule in the environment), and so on until a rule in the environment is found having at least one translation. All translations at a particular rule are considered equally valid. Thus, the

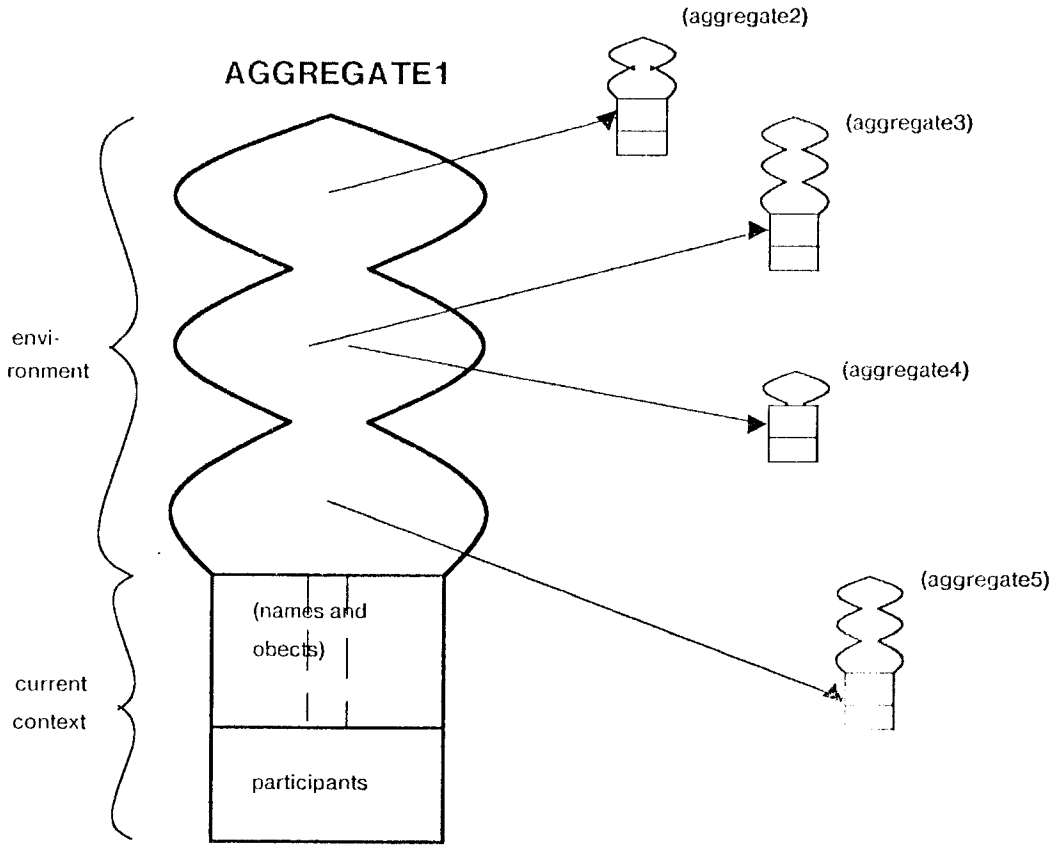

Figure 3-2: Depiction of an aggregate

order in which contexts are checked within a rule and the order of the returned values are meaningless.

untranslate = proc (aggregate, object/name) returns (set[name])

The untranslate operation is also somewhat different from untranslation on a context, in the same way that translate is different. If the object is not named in the current context, then the environment is used. Again, all untranslations within a particular rule are considered of equal importance. The untranslate operation was found especially useful in the

electronic mail implementation presented in Chapter 5. Because incoming mail might have been generated using a mail program not implementing aggregates and contexts, the untranslate helped provide the user with a more uniform interface. The add_aggregate operation also allows the user to assign the incoming message to an aggregate in order that the untranslation operation occur in that aggregate.

add_participant = proc (aggregate, participant)

This operation is identical to the operation of the same name for contexts except that it adds a participant to the current context of the aggregate provided.

insert_rule = proc (aggregate1, rule#, aggregate2)

This operation affects aggregate1, by inserting the current context of aggregate2 as a new rule at the specified number. The reason that an aggregate is specified for addition is that it would be possible, as will be noted in the implementation discussion, to name only aggregates and identify contexts only as the current context of an aggregate. In order for this operation to succeed the current context of aggregate2 cannot be in some other rule.

add_to_rule = proc (aggregate1, rule#, aggregate2)

This operation is similar to insert_rule except that it adds the current context of aggregate2 to the specified rule in aggregate1. Again, it does not succeed if the context is already in another rule.

The additional operations needed to make aggregates usable are listed in Appendix A.2. These operations include a selection of operations for management of the environment as well as those operations inherited from contexts.

Implementation Issues

Two of the issues discussed with respect to contexts must be reconsidered in discussing aggregates. The first is the synchronization of copies of a shared context, each of which is the current context of an aggregate. The second is naming aggregates. In addition, a different form of initialization must be considered.

An aggregate reflects the owner's private view of a shared context. It is possible to use that advantageously by recognizing that changes to a private copy of the shared context need not occur until the owner of the aggregate actually uses the context. Therefore, delaying such changes is feasible. This allows for a relaxation in synchronization of the multiple copies of a context with the understanding that such delays in updates not be visible to the owner of the aggregate. The electronic mail facility takes advantage of this by having the bearers of new information be the messages themselves. Updates to a current context only occur as new mail items containing any new information are read. Other synchronization mechanisms are possible and can be based on the medium of communication. What is important to note here is that it is not necessary to provide any form of update atomicity because the level of cooperation among participants is not close.

Naming of aggregates is the second implementation issue. In the discussion of contexts, the suggestion was made that contexts be named through the naming mechanism. The same holds true for aggregates. There is a further question related to naming aggregates and contexts, that of whether separate names are needed for aggregates and contexts. The approach that is taken in this research is that a context can be named simply by identifying it as the current context of some aggregate. This implies that a context can be the current context of at most one aggregate for each participant involved in sharing the current context. It also implies that a context cannot be divorced from its aggregate. An alternative would be to allow a user of the naming facility to create a new aggregate that would have a current context that was already the current context of another aggregate owned by that same user, but having a different environment. Uses for such a facility are not obvious and it therefore adds unnecessary complexity. Such a facility is available in the electronic mail facility, but no use was ever found for it. If a use is found, a cleaner solution to the problem may be that the user who wants to use a context

twice in different aggregates create two identities as different participants. This latter alternative allows the user of the context to distinguish between the two aggregates.

The final implementation issue to be addressed here relates to initialization. In addition to the discussion related to contexts, one must consider how a user gets started. The proposal here is that each user start with some basic aggregate that is the private world of the individual. That private aggregate would contain a current context of private names. In addition, the individual may want to include more recent sets of names in the environment of that aggregate. The environment of the user's basic aggregate may change more frequently than most other environments reflecting recent experiences. The set of contexts in the environment may be fairly stable, but their arrangement into rules may vary. In addition, although this was not discussed earlier, an enhancement to the creation operation for aggregates would be to insert a single context, the current context of the user's basic aggregate, into any newly created environment. In the electronic mail facility, the first time someone uses the facility a basic aggregate containing a private, unshared context is created. When a new aggregate is created it is completely empty.

To summarize the contents of this section, an aggregate is the only interface that the user has to the naming facility, although it is composed of contexts. The aggregate is not shared, but consists of one jointly managed current context that is the focus of most of the activity in the aggregate and a private environment within which names used in relation to the current context but not defined there may be recognized. In addition to the operations provided for contexts, the only additional operations needed for aggregates are those to manage the environment. Aggregates can be named using the naming abilities of aggregates themselves. In addition, since from each user's viewpoint a context is in exactly one aggregate, the context need not have a name separate from the name of the aggregate in which it is contained. The

fact that updates to a shared context need not occur until the user next sees the context makes careful and immediate synchronization of multiple copies unnecessary. Finally, each user will have a private set of names managed in a private or basic aggregate. The current context of that aggregate is not shared.

3.4 Examples of Uses of Contexts and Aggregates

With the definitions and discussions of names, contexts, and aggregates in place, a presentation of how they can be used to describe several existing situations is in order. Three examples are discussed here. They will also reappear in Chapter 4. The three are a conversation between two people, the Known Segment Table in Multics mentioned earlier, and the cataloguing facility in R*.

The particular example of a human interaction used here is one of a large number presented by Carroll [7]. Carroll was using data collected by Krauss[8], although it was analyzed further by Carroll and his colleagues and presented in the Appendix of Carroll's work. The situation was as follows. Eighteen subject pairs were observed. For each pair, the two subjects were arranged so that they could not see each other, but could communicate. They were shown a collection of graphical patterns in different spatial arrangements for each of the two subjects. The subjects were to identify jointly all the figures. The complete conversations were originally recorded. Carroll and his colleagues extracted all the references to the figures, sorting them by reference to each figure, resulting in 212 different situations. The analysis of this data presents the subjects reaching an agreement in most cases about a name and then later using that name. Just one of these will be presented here to exemplify some of the procedures of joint definition and use of names. Carroll used the data

---

[8]According to Carroll, these data were originally discussed in the literature by Krauss and Weinheimer [21], and later again by Krauss and his colleagues in [22, 23, 24]

to study the sorts of names that were chosen and the procedures by which they were selected. The example chosen is in Figure 3-3[9]. The Arabic numerals refer to page numbers of the original observations and the Roman numerals identify the subject. The page numbers were included to indicate the distribution of the references.

Considering this example in the terms the model presented in this research, the two subjects have a shared context predefined for them. When their discussion is complete it will contain names for all the objects shown to them. In addition, each has a private view of the shared context. Perhaps, subject I was recently on a farm and therefore a context defining farm animal names may have been high on the environment list for this subject. On the other hand subject II may have had nothing unusual occur recently leading to the suggestion of "horse's head". (See Figure 1-1 on page 21.) In this example, it is clear in addition to the shared context used for defining names for the figures being shown to the subjects, they assume that they have other experiences in common, in this case experiences that would give them both the knowledge of the shape of both a seahorse and a horse's head. Those experiences may well not be shared experiences, but each will have contexts in which those names are defined and the assumption is that they are defined in similar ways. Before the series begins for these two subjects, they will have some set of contexts that they will bring with them to the interaction, those contexts forming their environments. The shared context will be empty until they begin defining terms. The negotiation process through which they go will be discussed further in Chapter 4, in considering how agreement on names is reached.

The Multics Known Segment Table (KST) [37] was described earlier in Section 2.4. Normally, when a process is initialized the KST is empty. It is generally the first

---

[9]This dialog is from p. 13 of Carroll [7]. It is between the second subject pair and is discussing the figure labelled B by the experimenters.

1.I sort of like a head on it, an animals head, sort of like a horses head
1.II horses head
1.I two points on the top
1.II sort of like it's got two points on the top
1.I a seahorse
1.II and it comes real narrow at the bottom
1.I like a seahorses head
2.I same seahorse
3.I seahorses head
3.I seahorse sort of thing
4.I seahorse
5.I seahorse
6.I seahorse
6.I seahorse
6.II seahorse
6.II seahorse
6.I seahorse
7.II seahorse
7.I seahorse
7.I seahorse
8.I seahorse sort of thing
8.I seahorse
9.I seahorse
10.I seahorse

Figure 3-3: Example of joint selection of a name

entry in the search rules. When a name needs resolution in the process and that name is not in the KST, another rule is used to resolve the name and then an entry is made into the KST. From that point forward, any reference to that name is resolved in the KST, assuming the KST has highest priority in the search rules. Thus all occurrences of that name in any segment used in that process will be resolved in the same way. The search rules can easily be compared with the environment of an aggregate and the KST, when it is at the top of the list in the search rules, can be

compared with the current context. The architects and designers of Multics were aware when this mechanism was created that there is a potential for incorrect resolution of names, but it was decided that that cost was worth the beneficial tradeoff. Once in a great while, the mechanism surprises a programmer or user, but in general the mechanism provides the desired and expected behavior. The same tradeoff will exist in the mechanisms proposed here and the same choice is made. The idea missing from the KST is any representation of participants, since by design there was only one shared context and participation was not an issue.

In the catalog of R*, a distributed database management system [29], Lindsay made a similar choice. In that case, each user at a site has a set of single component nicknames. A system name consists of four components, the creator's name, the creator's site, the creation site, and a name for the object that is unique when combined with the other three components. If any of the first three components is not specified there are mechanisms for choosing default names. In addition, if only a single component name is specified, the user's local table of synonyms will be used for possible name translation prior to any other defaulting that may come into play. In this case, the system-wide catalogue that translates system wide names into objects is a single shared context. The private, local synonym tables provide private views on that. In addition, another mechanism, the defaulting mechanism is inserted in the middle. It provides a non-naming function, in terms of naming as defined in this research. The combination of mechanisms in R* as described by Lindsay provide a tradeoff similar to that of the Multics KST. Again, translations will be made using a common table, with possibly undesirable effects, but in most cases acceptable and even desirable effects.

These three examples point out that not only does the model describe patterns of human naming, but also choices similar to those of this research have been made in other computer systems with similar tradeoffs. The choices were made knowingly

and successfully. Chapter 4 will return to the seahorse example in discussing in detail the problems of candidacy and joint management of names. These are an important part of the proposed mechanism and therefore were separated in order to give them a more thorough discussion.

Chapter Four

Joint Management and Name Assignment: The Model, Part II

4.1 Introduction

This chapter completes the discussion of the model. The aspects of the model presented in this chapter are the joint selection of names to be in a shared context and representation of state changes with patterns of usage of names. Chapter 3 addressed the fact that names have two uses, as handles for accessing the objects to which they are assigned and as place holders for those object. Since a name is anything that fits the definition presented there, exactly how a name is contained or passed between users is not specified. That is an implementation issue, not part of the model. The issues addressed here are how and where names are entered into a context and which names are chosen. Although these issues involve possibly distributed decision making, for simplicity it will be assumed that lack of synchronization and accessibility are not a problem. The issues of synchronization and multiple copies will recur in several places. The problems discussed in this chapter involve agreement at a different level of abstraction from multiple copies of a context.

The problem of name selection can be decomposed into two separate problems. The first is the determination of which names are proposed for entry into a particular context. The naming facility puts no restrictions on these choices other than requiring that names fit the definition of names in Chapter 2 and they are supported in the implementation of contexts and aggregates. They are solely the responsibility of the proposers of names. The second problem of name selection is

determination of how and when a name becomes part of a context. There are two means by which a name can be entered into a context. The first is as a proposal from one or more participants and the second is as the result of merging two contexts, thus creating a new one. Thus the participants sharing in the use of a context are also the proposers of new potential name assignments.

Direct proposal of a name by participants leads to recognition that there are many possible factors that may come into play in determining whether or not a name will be chosen by a group of communicants. Some of those will in fact be influenced by the form of the name and possibly its relationship to other names that have already been accepted or rejected. Which factors are relevant to a particular context for both addition and deletion will determine part of the nature of that context. Therefore the functions of acceptance and deletion must be parameters of the type of a context.

When a name is proposed as a candidate for acceptance, it is transformed from being unknown to being tentatively accepted. In this model, the degree of acceptance or deletion is represented as one of a series of states. That series can be depicted by a state diagram including transitions between the states. A name may pass through a number of candidate states before being fully accepted. The transitions from one such state to another will occur when certain factors arise during use of the name. For example, it may be that anyone within a group can propose a name, moving it to the first candidate state. As it is used repeatedly, it moves through states toward the accepted state. Many factors, one of which is frequency of use, may affect progress through the candidate states. Perhaps, it can only be truly accepted when it is used by the organizer of the context in which it is being proposed. Figure 4-1 is a depiction of an example of a state diagram.

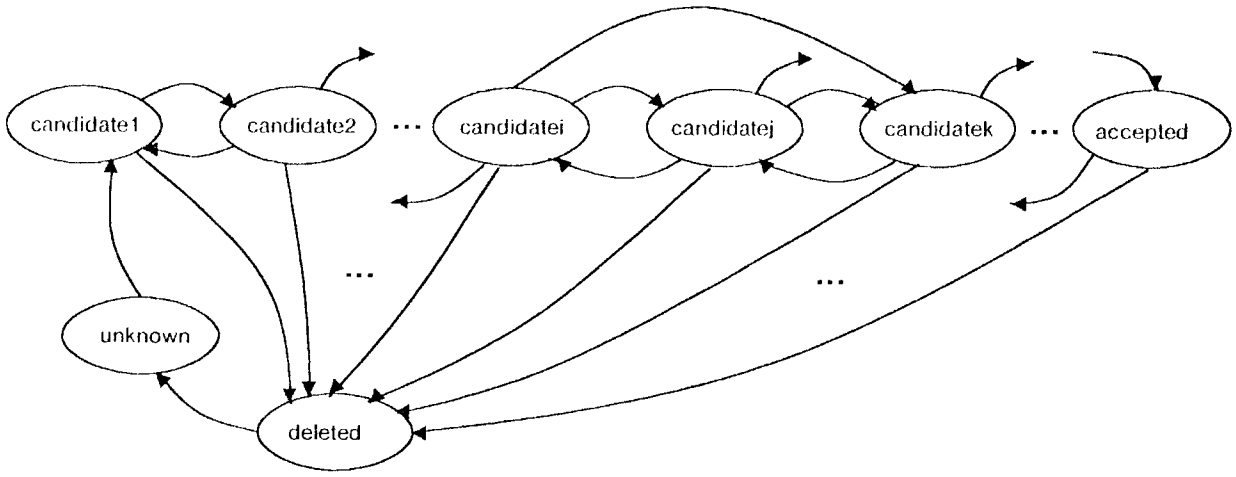

Figure 4-1: An example of a state diagram of the transitions of context entries

The second means of entering names into a context is through merging. In this research a proposal was made for a collection of separate namespaces called contexts. There will be occasions on which it will be necessary to merge two contexts to form a third. Even if the contexts are parameterized by the same acceptance and deletion procedures, merging two contexts may be complicated. A table can be used to indicate the state of each entry in the new context based on its state in the original contexts. Figure 4-2 presents one such example. In such a table choices must be made about the state of an entry in the merged context given its possible states in the two contexts being merged. The fact that a name is in a particular state in a particular context is the result of the history of its use in that context. If the two source contexts contain different states reflecting different aspects of the history of use of names, the choice of states in the newly merged context will be especially difficult to determine, and probably cannot be handled by any general procedure.

	u	d	c1	c2	...	cn	a
u	u	d	c1	c1		c1	c1
d	d	d	c1	c1		cm	cn
c1	c1	c1	c1	c1		cm	cn
c2	c1	c1	c1	c2		cm	cn
...							
cn	c1	cm	cm	cm		cn	a
a	c1	cn	cn	cn		a	a

u	unknown
d	deleted
c1	candidate1
c2	candidate2
cn	candidaten
a	accepted

Figure 4-2: An example of a table for merging contexts

Given this background the factors that may play a role in the functions of acceptance and deletion can be investigated. Section 4.2 discusses a simple example to highlight some of the factors and how they come into play in accepting a single name. A larger list of factors is discussed in Section 4.3. Such a list cannot be complete because one cannot predict all the possible uses of names nor the joint decisions among participants of criteria for accepting and deleting. The most that can be done is present a well thought out set of likely factors. This will be followed in Section 4.4 by a discussion of how the factors might come into play as parameters to the acceptance and deletion operations. A sample set of choices with respect to those factors appears in Section 4.5. Merging is discussed in more detail in Section 4.6 and the chapter concludes with a review of the model as presented in this and

the preceding chapters and how the model as whole addresses the goals presented in Chapter 2.

4.2 A Simple Example

There are many possible factors that may affect the set of names in use in a current context. There are probably different factors that affect acceptance than deletion. Deletion is considered here to be less important than acceptance because a name need not be used even if it is in a context, although there may be special situations in which deletion is important. Such a situation might occur if each object were allowed only one name in a context. If a name fell into disuse, it might be that the name itself was causing a problem. For instance, it might be difficult to use, causing an undesirable modification of behavior of the users. Therefore, it would be useful to have such a name deleted, allowing for a new one. The reverse situation in which a name can be assigned to no more than one object may also cause a problem of name conflict. In this situation, a name cannot be reused and assigned to an object unless it is not naming anything else. Although deletion is of frequent concern, acceptance is considered here to be even more important and, therefore, the focus here will be on acceptance.

Three examples were presented in Section 3.4. Of those only one involves negotiated responsibility for choosing names. That one, the conversation between two experimental subjects, also reflects degrees of acceptance of a name, not found in the other two. Since the human interaction provides an example of a set of factors that may come into play in such decision-making, it will provide the starting point for the discussion of factors involved in such joint decision making. Those factors are also relevant to non-human interactions.

This discussion returns to Figure 3-3 on page 73. There are several things to notice about the interaction presented in this figure. The first is the degree to which negotiation is taking place. I makes the initial comment, II picks up with "horses head", then I modifies it, and II picks up on the modification, I proposes "seahorse", II adds to the modification, I uses the head idea once more, and they settle into "seahorse", both using it from then on. The second and third points stem from noticing that all this negotiation happens on the first page. There is a rather intense period of negotiation consisting of seven references to the figure, after which agreement has been reached. The total number of references before agreement is reached is not high, in this case seven, although in many other examples it is even lower. In addition, because this occurs in a short period of time, the frequency of reference is high. Fourth, the name passes through several mutations, beginning with a comparison to a "horse's or other animal's head" to assuming just the term "horse's head", through the stages of "seahorse". Carroll [7] discusses various forms of mutation that may take place, that will be discussed further in Section 4.3. The fifth point is a little more obscure. Although the researchers chose the label "B" for this shape, the subjects chose a name that has some meaning to them; it describes a shape that they both understand. It is something that each assumes the other will know and understand. Such a name is something that the participants realize that they share with each other in a different context.

Attention must be given to the fact that only a single example was discussed above. One cannot make generalizations based on it, but rather use it to exemplify some of the sorts of factors that are considered to be important in studying the procedures used for jointly agreeing upon names to be shared. This particular example was chosen to reflect several of those factors. Other examples may reflect other factors, but most did not seem to reflect them as clearly. The next section will discuss a non-exhaustive collection of factors that affect joint agreement on names.

4.3 Factors in Joint Management

Given the five factors that played a role in the example presented above of two participants agreeing upon a shared name, a larger set of factors will now be considered. These factors are derived from a variety of sources and modifications of observations about them. One obvious source is the work by Carroll [7, 54]. The other major source is information that is considered important to record for files in various file systems. Initially in this chapter a distinction was made between the content of a name and the mechanism by which agreement is reached in selecting the name. In fact the two can be closely tied to each other.

Factors:

- **The user's relationship to the group:** The user of a name may play an important role in reaching an agreement on a name. The user may be in some sort of either dominant or subordinate role in relation to the recipients of the name. As will be seen in the programming support environment, a librarian may have special privileges when it comes to defining names in a shared context, while the individual programmer may only be allowed to make suggestions to the librarian.

- **The recipients' relationships to the group:** As with the user of a name, the role of the recipients may make a difference as well. For instance, it may be that, if the dominant participant is among the recipients, the usage will carry more weight in upgrading the state of the entry in the current context than if only subordinate participants see a name. In addition, the number of recipients may be significant.

- **The application's usage of the name and relationship to other applications:** How the name is used, by which application, may determine how much weight the usage of a newly proposed name or a name in a candidate state will have. It may well be that a context is used by several applications, such as one that is used both for source code and compiled code. It may be that for proposing a new name for source code, agreement is needed among the various participants, but once that has been decided, naming a compiled object that is derived from such a source code object can be done without any further negotiation. In addition, an application program may use names in various functions,

some more important than others. This factor may be tied closely to the factor of previous choices.

- **Time of usage:** The time at which a name is used may have an effect on its state. For example, it may be that at certain times of the year, usage becomes much heavier and, in order to avoid definition of many names that will not be used much again, this fact may influence the way the other factors are taken into account.

- **Number of uses:** This factor may alone be the most important. In the example the word "seahorse" was used in conjunction with other words four times after its original proposal before it was accepted. In the electronic mail implementation, number of uses is the sole criterion. This factor may take on numerical values up to a limiting value. In addition this factor may be used in conjunction with others such as the user or the recipients.

- **Frequency of use within a period:** This factor has two important aspects. The first is the frequency of usage. It may be that a name that is used once a day is less likely to be accepted than a name that is used once an hour. The other aspect of this factor is the period over which the frequency extends. It may be important that a name not only by used at least once an hour, but also that this usage pattern be maintained for at least two days, or some similar requirement. It should be clear that this factor cannot become relevant until a name has passed the initial proposing stage and has become a candidate for acceptance.

- **Mutation:** Mutation was mentioned in the discussion of the example in the previous section. There identification changed from comparison to an animal's or horse's head to a seahorse's head to a seahorse. These changes are not very great. If the changes had been less closely related to each other, perhaps more uses or more negotiation would have been needed to reach agreement. Mutation is also related to the next factor as well.

- **Relationship between a description and the final choice of a name:** If the original description was "like a seahorse" and the final name was "seahorse", arriving at that agreement might be easier and quicker than if the original name was "like a horse's head". In turn this latter might be easier than if the original had been "like an animal's head". Carroll

analyzed the 212 different joint identifications presenting a set of conclusions about possible strategies used to arrive at a name given a description. He also analyzed the data for number of occurrences of each. The following is simply a list of them in decreasing order of frequency:

1. *The Whole-Description Strategy* in which the whole description (which may be a single word or small number of words) is used as the name.

2. *The Content Strategy* in which the final name comprises the content of the original description.

3. *The Content-Noun Strategy* in which the major noun of the description becomes the name.

4. *Minor Literal Strategies* in which the name finally chosen plays a minor role in the initial description.

5. *Nonliteral Strategies* into which all other examples that reached agreement on a name fall. This includes strategies such as use of synonyms or other semantic relationships in combination with one of the previous strategies.

Depending on which strategy is being used in arriving at a name, the period of negotiation before acceptance may be shorter or longer. This factor, as many of the others, is likely to be used in conjunction with other factors.

- **Previous choices:** This factor was mentioned in the example. It is based on ideas both of Carroll [54] in his work on human factors and observation of operating systems throughout this research. Many systems provide for similar character strings to be used in situations to indicate relationships among the named objects. In addition, Carroll suggests that names displaying what he calls *congruence* are easier for people to handle. What Carroll is describing is complementary terms, or opposites, such as using the term "down" rather than "return" for the motion that is the opposite of that labelled "up" or in the electronic mail example using the names "sender" and "recipient" rather than "sender" and "reader".

- **Sharing in other contexts:** This factor was also discussed in relation to the example. If the proposer of a name and the recipients of the proposal recognize it from another shared context, perhaps it should be more easily accepted than if the recipients have never seen the name before.

Ten factors have been suggested here. In different situations different factors may be more or less important. In the example only five of them were identified. The proposal in this research is that the factors be specified on a per-context basis. In fact, the proposal here is that the type *context* not be a type but rather a type generator and that the acceptance and deletion factors and their interrelationships form the basis of the parameterization. Parameterization is discussed further in the next two sections.

4.4 Parameterization of Joint Management

This section addresses the means for using the factors listed in the previous section. First, the implementor using the context type generator must understand how those factors will be evaluated by the context type for both the acceptance and deletion operations. In addition, the implementor must identify the states through which a name may pass in moving from unknown in the context to perhaps accepted as part of the context. The factors may be cardinal numerical values, ordinal values, binary (true/false) values, based on a table of values, or related to other previously stored information. The finite state representation of how these factors affect acceptance and deletion must also be defined. They will result in a diagram such as Figure 4-1. Both of these were done in the electronic mail implementation and are presented in Chapter 5 with the state diagram in Figure 5-6. For now, the nature of those factors will be considered further.

The relationships among the user, the recipients and the rest of the group are likely to fit into some sort of ordinal arrangement of the participants. A simpler

representation of information about the recipients is a count of the number of recipients without regard to the relative importance of them. In addition, if different applications have different effects, this will best be represented as a relative relationship among the applications. One is most important, has the most effect, another has the second most, and so on down to the least effective. It may be that these can be reduced to binary relationships by recognizing only two categories, those people or applications that have more effect and those that have less. In the simplest case, all participants and all applications are of equal importance. In this case, a count of the number of recipients may still be a factor.

The next three factors, time of usage, number of uses, and frequency of usage within a period, will all standardly have cardinal values, although the latter may have several possible values for different periods. It may be that approximations are made for each of these. Time of usage may simply be categorized into one of several periods, e. g. prior to some time, during a time period, or after a particular time. Number of uses may be used as a value up to some limit. This is what was done in the electronic mail system, where the limit was three. Finally, frequency of usage within a period may be recorded only for one fixed period (5 minutes or one hour or one day, but not all three), and again there may be a limit. In addition, there may be an upper or lower limit on the frequency; e.g., if the frequency is more than five per time unit, how much more may be unimportant.

Mutation and the relationship between a description and the final choice of a name are probably the most difficult factors to which to assign values for computation. One might attempt to assign relative numerical values, but the basis would have to be some heuristics. For this some of the techniques developed in the Artificial Intelligence community for recording the relationships between words and concepts should probably be employed. Unfortunately, more is needed than simply to record relationships. In addition an assignment of relative importance to various of those

relationships is needed and one needs the capability for adding new, yet unknown relationships and understanding how they fit into the previously existing schemes. In an operating system environment where efficiency of operation is critical, these sorts of activities are likely to add much complexity to the computation and therefore reduce efficiency.

The effects of previous choices may be evaluated in different ways. For instance, if at least the first three characters are the same as another previously accepted name, it might be that the boolean value True will be chosen for this factor, or False if fewer than three characters match. One might provide an absolute value of the number of characters that match with a lower limit, so that at least two must match before this factor comes into play. Congruence is more difficult, and probably involves a dictionary in order to provide recognition of opposites. As with the semantic relationships discussed above, if such operations for acceptance and deletion are included efficiency will probably be greatly reduced.

The final factor is sharing in other contexts. This may be given relative values based on how many people know the name in another context and the state of the entry in that other context, or it may simply be a binary value of whether the name is known to all and accepted in another context. Although this sounds like a straightforward computation, in fact there is a complication because the time and circumstances of the computation will be unpredictable and may be variable at different sites. For example, if the shared context is implemented and exists as a single object (whether or not there is replication), its state will be consistent at all times. This was not the case in the mail system. Multiple closely related versions existed, one for each sender or recipient. The updates on them were done independently. In a situation such as that, the state of the world may be different at the time of each update and therefore the results of using external information vary over time. In the mail system, that was acceptable because distributed information was not used in the

process of defining names. The user's expectation is very important in such a situation, since the users believe that they are communicating and reaching agreements with each other. The naming facility is unacceptable if routinely users believe that they have reached an agreement, only to discover that there are differences of opinion on this.

4.5 A Sample of Choices

This section presents a selection of factors that might be used for human interaction. These choices provide an example that might appear in the implementation of a user interface. Therefore such values as times and number of repetitions are chosen to fall within common human understanding. In another situation different choices might be made.[10]

Of primary importance is the number of uses. Because of Carroll's observations that small numbers of uses in fairly quick succession are most common in human conversation, the number four is used. The period for humans should be on the order of one day. This would require keeping a minimum of four timestamps for usage. An assumption is made here that all participants have equal status within the group, and that as with the electronic mail system, each participant has a private copy of the context, the set being kept in approximate synchrony. This means that as each participant sees four instances of a name within one day, the name becomes accepted for that participant. Since this is application independent, neither the factor of application nor time of usage is included. Of the remaining four factors,

---

[10]The only test of such choices in this research effort was made in the implementation of the electronic mail system. The choice there was kept especially simple, but implemented so that others could be substituted easily if the occasion arose. Due to limited use of the software, little was learned about this aspect of the implementation and it was felt that alternative decision making mechanisms could not have been tested well enough to be of value.

three are not included here because of the complexity of including them. These are mutation, relationship between a description and the final choice of a name, and sharing in other contexts. The final factor, previous choices, can be included in a limited form. For example, given a name with a particular extension, the choice of the same name with an extension chosen from a limited set of choices might be accepted after one use, if the first name were already accepted. In order to implement decision-making based on this set of factors, the only additional information beyond names and states that is needed is timestamping.

There are two further issues related to what happens if there are not four uses within one day. In humans' minds, a name will slowly lose ground, be forgotten by degrees over time. As it is losing ground further uses will revive it. Forgetting seems to happen more slowly than accepting a name. Therefore the proposal here is that the acceptance function work in eight hour intervals, but the final deletion step be an additional 24 hours. The final issue is how a name can begin to fade once accepted. Here perhaps a one week period might reflect reality. Thus the state diagram might be drawn as in Figure 4-3. It will be noted that no distinction is made between unknown and deleted. Again, this may be a simplification of reality for the sake of efficiency. It must be remembered that the choices made here were to demonstrate an example.

As mentioned, in addition to recognizing which factors are important for both acceptance and deletion, the implementor must determine the various possible states of a context entry and which factors will affect which transitions between states. Feasibility would dictate a simple set of states and transitions. This, in turn, probably means that in any implementation only a small number of factors can be considered. Not only must programming be done, but the computation must be done, and for many of the factors, historical information may need to be stored, such as the identification of all previous users of the name or the times of previous

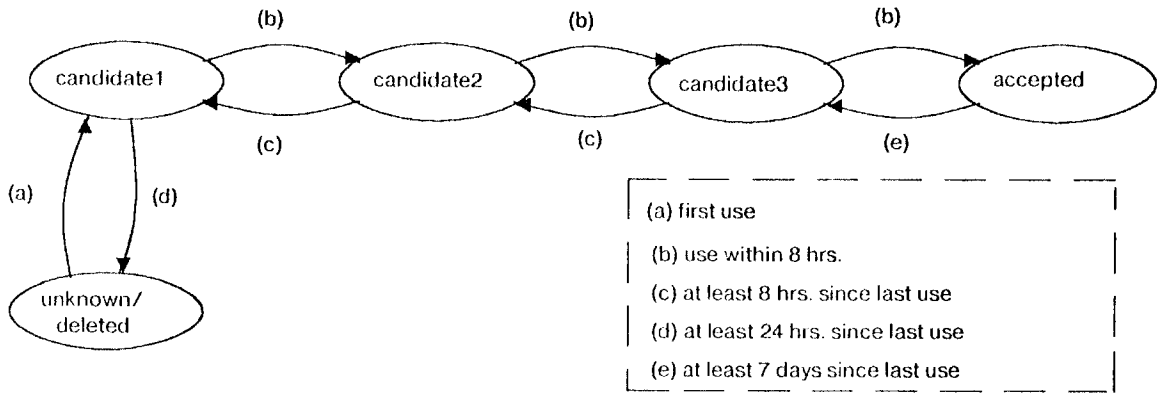

Figure 4-3: A state diagram for acceptance and deletion

uses. It is clear that if naming is too inefficient, it will not be useful to potential users. Therefore in addition to the goals of providing a naming facility efficiency must always be considered.

4.6 The Merging Problem

In addition to determining the states of entries in a context based on use of names and other related information, there is one further situation that may determine the states of the entries in a context. Consider the situation in which a context is created by merging two previously existing contexts. The operation that achieves this merging is another parameter to the context type generator. It determines the detailed nature of the type of such a context, although it will be used at most once in the lifetime at the creation time of a context.

The problem can be separated into two subproblems, the solution of one of which is manageable and the other is open ended. The simpler of the two is merging two contexts of the same type, that are parameterized by the same operations. In this case, although there are many decisions to be made, the problem is tractable. Unfortunately, if the contexts are parameterized by different implementations of the acceptance, deletion and merging operations, there is no basis of agreement from which to begin in general. If such a merge is to occur, a special procedure must be created for each particular pair of context types for which it is needed. In those cases the same issues must be addressed as will be discussed below for two contexts of the same type, although the final choices will be designed for the particular pair of context types.

A number of issues must be faced by the implementor of the merging operation is the determination of entries in the new context and the state of each. There are several factors that may be taken into consideration. First, the two contexts may be considered on equal standing or one may be considered more important than the other. With this knowledge, each entry in each context will be considered. For each name translation, consideration must be given to its current state, whether the name, the object, or the full entry exist in the other context, and the relationship between the original contexts. As mentioned earlier, in some cases additional information such as timestamps of uses is saved for the acceptance and deletion procedures. That information may also need to be merged or at least be used as part of the merging operation, although this adds complexity. In the case of merging, a table can be drawn up, as for example in Figure 4-2 based on the possible states of names. In the case of that figure the two contexts were considered of equal importance. In addition, the groups of participants will simply be joined into one.

There is a further problem of the creation of the environment in any new aggregate formed by using the new context as a current context. There are a number of

possibilities here. If the participant and owner of the aggregate was not a participant in either of the original contexts, then probably the environment should default to whatever it would for a new aggregate. If the participant was a participant in one of the contexts, then perhaps the environment should be that of that earlier shared context. Finally, if the participant shared in both of the original contexts, perhaps both should be reflected in the new environment. It is not clear in this latter case exactly how the environments should be merged. More importantly, it should be remembered that the environment is a reflection and representation of the individuality of each participant. As such, the recommendation here is that it should not be created automatically by the same mechanism for all users sharing the new context. Rather, nothing should be done other than any defaulting that the individual may have specified, thus leaving the management of the environment the responsibility solely of the individual.

The discussion of merging to this point has not considered what problems might arise from multiple copies of one or both contexts in a merge operation. If all copies of each context are in synchrony there is no problem. Consider a situation in which the copies of one context are not synchronized. Merging occurs by merging the local copies of two contexts forming a third local context. The question that must be addressed is what happens if a context entry is in one state in one copy of the context and in a different state in another. The merging tables presented in this report have a feature important to this discussion; an entry that exists in any state in one context cannot become unknown through the merging procedure. This means that entries cannot disappear. In addition, entries do not move farther from acceptance through use. Now the merging of local copies can be reconsidered. If an entry is accepted in one local copy and only a candidate in another, the result after the merge may be different in the new local copies, but that is an acceptable condition. In the worst case, if the two local copies being merged are not up to date

and an entry is unknown in both, but known in local copies elsewhere, the mechanism for proposing names can be used to bring the newly created local copy up to date. If an assumption is made that an entry exists, but it does not, the human recourse is to explore further by asking for further explanation or definition from the source. A similar procedure can be used in the world of contexts and aggregates, as it might be without a preceding merge operation. This analysis of merging contexts consisting of unsynchronized copies of the contexts leads to the conclusion that such a merge operation poses no new problems. The problems are only those of adding names and merging contexts composed of synchronized copies.

4.7 Summary and Review

This section concludes the presentation of the model proposed here as a framework for a naming facility. As such, the section will briefly review the problem addressed in the research and those concepts defined. In addition, a summary of the framework itself is presented, prior to a discussion of how the model addresses the posed problem.

Names are defined in this work as objects with an equality operation that stand for other objects. The purpose of a name is either to provide access to the object to which it is assigned, if that is possible, or to act as a place holder for the object. The equality operation tests for the equality of two names, not equality of two objects named by different names. The goal of the research is to explore the possibility of designing a naming facility that supports that definition of names, provides sharing and communication within federations of and by means of those names, and is implementable. The equality operation on names is needed in order to implement access of named objects through a naming facility. A federation is a loose coalition that may not be active at all times and that allows for both cooperation and individuality among the participants. Before proceeding with a review of accessing

objects and providing sharing and communication, feasibility of implementation can be dismissed for now. The purpose of Chapters 5 and 6 is to investigate implementations in two particular domains.

The model proposed as a framework for a naming facility presents the user of the naming facility with a collection of objects of a single type, **aggregate**, as the sole interface to the naming facility. An aggregate provides its owner with a private view of a shared namespace, known as a **context**. The shared context is known as the **current context** and provides the main focus for name resolution. In addition, each aggregate has an **environment**, a private list of partially ordered alternative contexts to be used in the individual's case if a name cannot be resolved in the current context. The type **context** is also newly defined in this research. A context also consists of two types of information, the translations from names into objects and some means of identifying the participants sharing the particular context. The translations can be in one a series of possible states ranging from just proposed as a candidate to fully accepted as a legitimate name to deleted and therefore not accepted as a name for a particular object. Further, those factors relevant to each context in order to move name translations from one state to another or enter them into one initially must be considered. This information may take the form of procedures for accepting and deleting context entries as well as merging contexts to form a new context with predefined translations. The definitions of aggregate and context incorporate exactly the definition of names presented in Chapter 2, therefore supporting that definition in the naming facility framework. An investigation of sharing and communication in the face of federation was based on human naming and provided a set of eight observations considered here to be subgoals. It is worth reflecting on each separately in order to explore how the framework supports them.

1. **Communication:** There are two uses for communication. The first is to share the use of names, to transfer names among users. The other is to

transfer information used to manage shared namespaces or contexts. For both of these the federation assumed as a system model provides the basis for communication on common ground. What the medium of communication is, need not be specified here and will vary from one system to another. The important fact is that contexts and aggregates are designed in such a way that names and information passed through that medium of communication can be incorporated into the contexts and aggregates. Furthermore, the participants sharing a context must believe that they have reached some form of agreement. Negotiation using the medium of communication will take place prior to the creation of a local copy of a shared copy, so that all the participants agree upon the various details of specification of a context, such as addition and deletion factors and procedures and a merging procedure.

2. **Individuality:** The environment part of the aggregate allows the individual to make use of personal experiences. The environment provides for potential name translation in cases in which the current context of an aggregate cannot translate a name. This allows the user to fall back on other experiences that he or she thinks may help in such situations.

3. **Multiplicity of names:** There are two means by which contexts provide for a multiplicity of names. First, a context contains relations between names and named objects. The existence of one relation within a context does not preclude the existence of any other relation between either the name or the object and any other name or object. Second, the fact that an individual or set of individuals are participants in one context bears no relation to whether any of those individuals participate together or separately in any other contexts containing a possibly different set of relations between names and objects. Therefore the full flexibility of multiplicity of naming is available through the naming facility.

4. **Locality of Naming:** Independent contexts provide locality of naming. The framework imposes no relationship between names in different contexts or between the contexts or aggregates themselves. Therefore, the naming within one context is completely local to that context.

5. **Flexibility of usage:** The definition of a name includes only a requirement of an equality operation. The naming facility also must

have some means of associating a name with an object and transmitting names between users sharing names. Other than these, there are no limitations on the nature of names, allowing for a large degree of flexibility in the choices of names defined by participants cooperating in sharing a context.

6. **Manifest nature of names:** The users of names are also the participants sharing responsibility for defining those names and managing the namespaces or contexts containing the names. Therefore, the users are free to select names that manifest whatever degree of meaning they jointly choose.

7. **Usability of names:** Humans, in the course of normal communication with each other, use names and switch namespaces often without a conscious thought given to it. In involving a computer facility in such activities, some actions and choices must be made more explicit because the recipient or medium of transport of the names is providing some interpretation, but does not have the capability of a human mind. The naming facility modeled here provides a simple means of involving a computer facility in such naming. Namespaces or contexts are local. Identification of contexts and aggregates themselves is based on that same local naming with the addition of identification of other users sharing the namespaces. In addition, the translations better reflect human name definition procedures allowing for different procedures in different situations and different sets of states reflecting patterns of usage. In addition, as will be seen in the next two chapters, the proposing of names and state changes for name translations can be made automatic.

8. **Unification:** There are no restrictions on the types of objects based on names. Names are not typed and a name can be assigned to several objects of different types. This allows for generic naming as described in Chapter 2 which is considered an advantage of this naming facility model. It is in sharp contrast with implementations of strong typing that depend on compile time type checking, because at times prior to execution, types may not be known since the relations may not be known or there may be several. In fact, even at execution time, if typing is inherent in the supporting system, adequate preparation must be made for handling type information.

This concludes the presentation of the model proposed to be a framework for a naming facility. The next two chapters discuss implementation designs in order to support the goal of implementability and simultaneously highlight advantages of using such a naming facility in those domains.

Chapter Five

Implementation of Naming in an Electronic Mail System

5.1 Introduction

Chapter 2 defined the goals of this research on a naming facility in a federated system. Chapters 3 and 4 proposed a model to be used as a framework for implementing a naming facility and as such is an approximation to the way in which humans manage and use names. The implementation discussed in this chapter is an approximation to the approximation. The model is simplified yet further in the implementation. In order to describe the model used and the design choices made in the implementation, electronic mail systems and their naming problems must first be considered in Section 5.2. Section 5.3 then will present the implementation decisions that were made for this work. Finally, in Section 5.4 review what can be learned from the implementation.

5.2 Electronic mail

Most electronic mail systems allow people to communicate with each other using a federated computer system to compose, send, receive, and read mail. One of the distinguishing features of mail is that the sender and recipient need not be present simultaneously in order for the communication to succeed. In fact, in most mail transport facilities, if the mail is travelling from one host computer to another, the two computers need not be in direct communication at the time of the composition and sending (from the viewpoint of the sender) or receiving (from the viewpoint of

the recipient) and reading[11]. In spite of that, at a bare minimum the sender must be able to identify the recipient to the computer system. There are further identifications without which the mail system is barely usable. First, there should be a facility for identifying the sender, in order that the recipient understand from whom the message came. Further, it would also be beneficial if the recipient could in turn become sender and respond to the sender, preferably using the same name used by original sender for self identification.

Figures 5-1 and 5-2 present an example that will be used in the remainder of the chapter. They are two forms of the same message, the first is taken from the implementation to be described here, while the second, containing only network addresses, is more like what the user is likely to see currently. The improvement in the former over the latter lies in the names and name management possible in the former. These examples will be discussed further below, including a discussion of choice of names for mail recipients, aggregates, and aggregate names.

Before considering an alternative for naming in an electronic mail system, it is valuable to consider a representative sampling of naming in other mail systems. This discussion is based on the five attributes of names listed in Chapter 2: assignment, resolution, scope of use, uniqueness/ambiguity, and meaningfulness. Consider for a moment the name "Brown.INP@MIT-MULTICS.ARPA" from Figure 5-2. It is a hierarchically structured name for a mailbox; the local name is "Brown" in the project "INP", on the host "MIT-MULTICS" (probably a Multics at MIT), supported by ARPA. The meanings of most of the components are probably irrelevant to most of the other recipients and the sender. The identity of the individual is important and "Alex who is interested in mail" may be more

---

[11] In a store-and-forward network, it is possible for the two never to operational simultaneously if there are intermediate forwarders

To: Sandy, Alex
Cc: Chris <cbosgd!hasmed!qusavs!ukm!ecg>
From: Randy
Subject: improvements
Aggregate: mail

The following features have been added to the mail program....

Figure 5-1: Message with shared nicknames

To: smith@MIT-CLEANSER.ARPA, Brown.INP@MIT-MULTICS.ARPA
Cc: cbosgd!hasmed!qusavs!ukm!ecg
From: rsmith@MIT-NEWCLEANSER.ARPA
Subject: improvements

The following features....

Figure 5-2: Message with mailbox addresses for names

appropriate for that. The assignment was made mostly be external authorities, although "Brown" may have been a personal choice. Although the name may appear in the message as it is delivered to a recipient, in fact it will be translated by various lower levels of protocols such as SMTP [38], if it is used on the Arpanet. The name was selected with the idea that it would be universal in scope, and globally unique. Ambiguous names might allow for sending a message to several mailboxes for a single user, or for naming a group, such as a mailing list. As will be

seen, other approaches support somewhat different decisions for those characteristics of naming listed above.

The Arpanet approach described in RFC 822 [9] (e.g. "MIT-MULTICS.ARPA") is that host names are the important part of the naming scheme and that they fall into a global hierarchy. In fact, RFC 822 specifies nothing about user names within a host. The structure and management of those user names is left completely to the local system, and may vary from one system to another. For example, Unix [40, 57] provides a flat namespace (e.g. "smith") with aliasing, both shared by the whole system and private to the individual. Multics [37] provides a two-level hierarchy of users within projects (e.g. "Brown.INP") and some aliasing. Finally TOPS-20 [12], provides a hierarchy similar to Multics, but of any depth, based on the directory structure of the system. The meaning of the components of a user name on TOPS-20 is simply that each component is a subdirectory of the directory name to its left, unless there is none, in which case it is a top level directory.

The UUCP approach [35] (e.g. "cbosgd!hasmed!qusavs!ukm!ecg") on Unix is similar to the Arpanet approach in lack of concern about local naming except that the scheme for naming hosts is different. Again the host name plays an important role with user name locally managed, but the namespace is neither global nor is it necessarily hierarchical. Rather a host name is a route from the sender's host to the recipient's host. The limitations on the number of routes is based on the topology of the network and explicit interconnection capabilities at individual sites. In addition, there is nothing that limits a name (route) to a single object (host). A route from host A to host B may also identify the route from host C to host D and there would be no problem of conflict, although there might be other problems, such as discovering or understanding a name of a host. Returning to the characteristics listed earlier, most of such naming is meaningless to both the sender and the recipient. The structure is that of a directed graph. The names are chosen in a

distributed fashion. For each node, someone responsible for it chooses exactly one name. Use of a particular name for a particular location must be completely local, although names need not be unique. In many cases, there are several routes between two nodes, each providing a legitimate name with no means of testing for identity.

The other three mail systems to be mentioned here include the user's name in their schemes. Grapevine provides a hierarchical, two-layer scheme. Users are named within registries. These user names are assigned within the Grapevine system. Registries identify administrative domains, that may also reflect organizational or geographic distribution. The Grapevine approach is to provide a global hierarchy. An example of a Grapevine style name is "Smith.PA", where "Smith" is the user's name and "PA" is the name of the registry, representing Palo Alto. In this case the name of the registry is geographical and must be included as part of the name in Grapevine. This means that a user of the name must realize the Smith works within the Palo Alto region, which may be not only irrelevant, but not a known fact. Grapevine does allow a name to refer to a list, thus providing a mailing list capability, allowing for uniqueness or ambiguity, although name assignment is managed by an administrator of the registry where a name will be assigned.

The IFIP Working Group 6.5 standard [18, 59] proposes that users be named and that their names consist of a collection of components that provide what appears to be a hierarchy to users of the names. An interesting aspect of this structure is that the ordering of the components is of no import. Therefore, the namespace may look like different hierarchies to different users of the namespace. The names, in fact, form a global lattice. All share the same set of names, although multiple names can exist for any recipient. In this case, a full set of components must be examined at each node which in turn will resolve that part that it understands.

Finally, the Cocos project [11] and the related research by Kerr [20] propose that each mail recipient be identifiable by a set of attributes. No host name is needed. The attribute names are not nested. Again the namespace is global. In both the Cocos project and the proposals of the IFIP WG 6.5, the idea is that the component names be names that are meaningful to users, although the components are chosen and resolved by outside authorities. In the IFIP proposal, each component is chosen by a separate authority, while in Cocos the complete set of attributes is predetermined and built into the system. In both, the complete schemes are universal, although in the IFIP proposal a name need not be unique. In none of the above projects are names selected by the users, or even in most cases by those being named. In addition, in most cases the users of the names have not been considered, and therefore names in cases other than these last two are probably not very meaningful. All of these approaches to mail provide for names for mail senders and recipients although none provides the sorts of naming set as goals in the earlier chapters of this work.

At this point it is valuable to reconsider the assumptions and goals of this research in relationship to a mail system. First, in terms of mail delivery, federation must be assumed. Even if the user community uses only a single computer, mail allows for a separation of sender and recipient that matches the definition of federation. When it comes to managing the namespace used for identifier mail recipients, only the UUCP approach of source routing[12] allows for local names, but in this case they cannot be shared because a name is location dependent. There is an additional problem in UUCP; when two hosts attempt to communicate each one must have the correct authorization. The sending host must allow sending to that particular receiving host and the receiving one to receive from the particular sending host.

---

[12] See Sunshine [50] and Saltzer et al. [43] for a more detailed discussion of source routing in general.

Thus the common technique of generating a return address hop by hop during the original traversal of a message may produce an invalid address. Grapevine and the IFIP WG 6.5 standard and the newer Arpanet standard [31, 32] all propose distributing the naming authority, although the responsibility still does not lie with the users of the names to define the names that they will use as discussed in earlier chapters of this report.

The purpose of a mail system is to support communication. That communication involves both sharing information, such as who the other recipients of a mail item are, as well as jointly determining the names that will be used. In communication outside a computer system, people communicating will jointly decide on names, as in conversation. They should also be able to determine the names they use jointly when a computer system provides the medium of communication. People may have many interactions with each other and may interact on different bases in different situations. In addition, the same name may be chosen for different people under different conditions. As a result multiple names are important. As mentioned before, people do not use globally unique names for each other. If, by chance the names are globally unique, they probably are not very useful.[13] Certainly in the case of a mail system, the flexibility of using various sorts of names would enhance such a system for the human users. In addition, whatever mechanisms are built to support a naming facility must be easy for humans to use. Although the goal of unification was not achieved in the implementation of the mail system, it could and probably should have been. The naming scheme is used only for naming people. It should

---

[13]Consider telephone numbers. With their full country and area codes they may be unique, but it is not clear what they are naming. They certainly are not really naming people. They are not naming telephones, because a telephone can move and can be assigned a different number. They are not naming locations, because numbers can move. They appear to name a particular location or set of locations at a particular time, with the additional information that such a name is not likely to change very often. A feature such as forwarding (known as "call-forwarding") allows a phone number to be used indirectly on a temporary basis, blurring the meaning even further.

also have been used for naming at least aggregates and contexts as well. A separate mechanism with less flexibility was provided for aggregates and contexts, simply a flat namespace where each such name is interpreted relative to the user's private namespace. If an operating system with a library of subsystems rather than particular subsystem were being built; the idea is that users could use the same naming facility to name people in the mail system as, for example, people in a calendar system, and any other system in which naming people was of use as well as unifying naming people with naming other objects.

The remainder of this chapter will discuss the implementation of the mail system naming facility in addition to a discussion of conclusions in the last section of the chapter.

5.3 The Implementation

This section describes the actual implementation, beginning with the model of contexts and aggregates and the user environment. That is followed a discussion of the operations provided at all three levels, contexts, aggregates, and the user interface. Finally, a review is presented of those decisions that were made in order to design the implementation.

Before discussing what confronts the user of the mail system, a brief overview of those decisions about data structures and the possible choices discussed in Chapters 3 and 4 are presented here. In addition, the organization of the management of the information is discussed. The discussion then turns to what the users sees in the mail system and how it can be used.

Both contexts and aggregates have exactly that information discussed in Chapter 3 and diagrams of them would be identical to Figures 3-1 and 3-2 on pages 60 and

67 respectively, except that contexts do not have separate lists of users and reservation of names not assigned to objects is not possible in the mail system. The entries in a context are more limited than the general form of contexts and aggregates. Specifically, both the names and objects are strings. Therefore, contexts and aggregates themselves are not named in this way. Instead, each user has a private list of contexts and aggregates and their names. The names of contexts are not universal or global. A name for a context or aggregate is translated by the individual using one of those private lists of contexts and aggregates. As for joint management, mail is used for negotiation. When a mail item arrives with a name in the aggregate field that is unknown, a new aggregate by that name, containing a new context by that name is created. If a new aggregate is created, but a context by that name already existed locally, then the existing context is used as the current context for the new aggregate. The final aspect of joint management is proposing and selecting names. Name translation pairs can be in on of five possible states. This is discussed in more detail below.

The representations of the objects needed for this implementation are simple. Names and addresses are simply strings. A context is an unordered set of pairs of strings. Searching is linear because it is assumed that contexts will remain small. The lists of aggregates and contexts for each user are lists of pairs consisting of names and aggregates or contexts respectively. An aggregate has two components. The current context is a pointer into the context list and the environment is a list of unordered sets of pointers into the context list.

Due to the pre-existing software used in this implementation, the management of the naming information was implemented as a separate process. Therefore, sending a message involves passing the message header to the separate process for possible name translation and sending it back to the user mail process for verification prior to passing it to the Unix sendmail process [57]. When mail is read, before it is

displayed for the user the header is passed to the recipient's name managing process for translation. Figure 5-3 depicts these activities and the three processes involved.

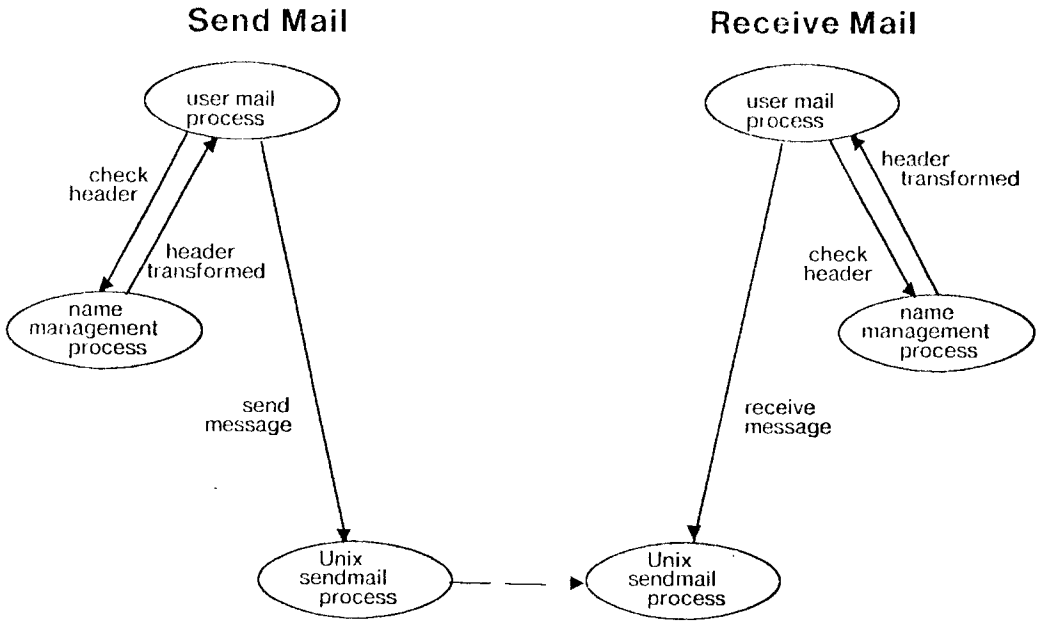

Figure 5-3: Processes in the mail system

The user of the mail system has a small collection of new objects to manage. When a user enters the mail system, he or she is provided initially with a single basic aggregate, named "basic_a" containing a current context named "basic_c" and an undefined environment. Each user of the mail system has his or her own private version of basic_a and basic_c. These are not shared. In addition, each user has two lists, one of named aggregates and one of named contexts in which he or she is a participant. In order to describe the use of contexts and aggregates in the mail system, Figure 5-1 will be reconsidered. In addition, the operations of listing aggregates and listing the contents of the "mail" aggregate as in Figures 5-4 and

mail
basic_a

Figure 5-4: The list of aggregates

Current context:

A Sandy	smith@MIT-CLEANSER.ARPA
A Alex	Brown.INP@MIT-MULTICS.ARPA
A Randy	rsmith@MIT-NEWCLEANSER.ARPA
C Chris	cbosgd!hasmed!qusavs!ukm!ecg

Environment:

Figure 5-5: Displaying an aggregate

5-5 will help in this discussion. The assumption is that the message in Figure 5-1 is at least the third message sent among the group. but that Chris is new to the group. There are a number of points to note about using the system. Figure 5-5 is Randy's "mail" aggregate; no environment has been specified.

- Contexts do not contain separate lists of participants because the names in a context are not only the objects being named, but also the participants.

- Since an aggregate is a namespace, each outgoing and incoming message will have a newly defined field attached to it, as allowed under the Internet specification [31, 32]. The field's name is "Aggregate" and it will name the private aggregate containing the shared context to be used for the envelope of that message, in this case "mail".

- When Chris first uses the mail system, a private aggregate "basic_a" will be created. Later when Chris first reads the message from Randy, another aggregate will be created named "mail". In addition, a new context named "mail" will be created and it will be the current context of the new aggregate. If, for some reason, a context named "mail" already existed, that context would have been chosen as the current context of the new aggregate.

- A message may be sent without the aggregate field specified. This will occur either if the sender specifies no aggregate field or if the sender specifies use of the "basic_a" aggregate. In either case, the sender's "basic_a" will be used for any translation needed.

- Names specified in "<>"'s will not be translated. The combination of a name in "<>"'s and a preceding phrase, as in the "Cc:" field of Figure 5-1 allows for adding new names and addresses to the current context of the specified aggregate. This will be discussed further below.

- A message may arrive without an aggregate field specified. There are two possible causes for this. Either the sender used his or her "basic_a" aggregate, or the sender was not using a facility that supported specifying aggregates. In either case, the recipient's "basic_a" aggregate will be used when reading the message.

- Finally, there is a facility allowing assignment of an aggregate to a message after arrival, so that on succeeding readings of the message, its names will be translated with respect to the assigned aggregate. This is especially useful for messages coming from senders not using this mail system.

In the implementation two decisions were based on the fact that this is a mail system. The first has to do with the nature of the names and objects supported and the second with the transport of names and proposed translations. The names that are used for people are strings. In addition, since names are translated into network addresses which in the Internet specification also consist of strings, the objects are represented as strings as well. The second decision is that the only means of transporting names within the federated computer facility is the mail messages

themselves. The reason that this is possible is that in the Internet specifications, each field that represents a person can have multiple parts, an initial phrase, an address, and a comment. Since the comment part often has unpredictable information in it and the initial phrase, if present, generally has only a name, this fact is being used. It is not foolproof, but no problems have been reported and any could be easily corrected by the user. Normally, such a field in messages generated with this mail facility contains a phrase that is the shared name in the current context and a net address. In the most common case, the sender specifies a name and the mail system appends the net address before sending the message. Figure 5-1 contains examples of both. At the receiver's site, when the message is read, the address is stripped off and the recipient sees only the name. This hides the awkward and user-unfriendly network address in the user interface.

There are several ways in which this can vary. First, the sender may be using a name that has not previously been used in that aggregate. If the name exists in the environment, its translation is taken from there and proposed as a candidate in the current context. If this is a completely new name translation pair, the sender must include both name and address, which is then proposed in the current context. At the receiving end, if the name translation pair has been accepted, the recipient sees only the name. Otherwise the recipient will see both. This last case reflects a situation in which the name has not yet been accepted, therefore the translation is provided as well as the name as might be done in direct conversation. If the name is completely new to the recipient, it is proposed in the current context. If it already exists, its usage is reflected in the current context as appropriate. Thus users can propose both new aggregates and new names within existing aggregates to be shared with other users. In the message in Figure 5-1, Randy is proposing a new name to the participants in the mail aggregate. To Chris, the new participant, the aggregate itself and all its entries are new. The aggregate displayed in Figure 5-5 is Randy's,

with only one candidate entry for Chris (indicated by "C" as opposed to "A" for the other entries). In Chris's case, all the entries would be candidates. The only variation from this pattern is use of the basic aggregate, which does not escape the owner's domain.

The mail system provides two approaches to managing the names and objects to the mail user. One is to create aggregates and enter names manually. For this, specific operations are provided listed in Appendix B. These operations allow for creation of aggregates and contexts and adding, deleting, and modifying the state of entries. The other approach is automatic, allowing names to be entered with usage as was suggested in the example discussed in this chapter. When a message is sent or read, an aggregate is chosen by the mail system. If there is no aggregate field, the basic aggregate is chosen, and otherwise the specified aggregate is chosen. If a name-address pair is found that does not exist in the current context, it is made a candidate. When a message is sent, if a name is found that exists only in the environment of the currently active aggregate, that name-address pair is proposed as a candidate to the current context. The implementation allows for both approaches and the user can intermingle the two.

For this implementation a simple scheme for accepting names has been chosen. A name can be in one of five states, candidate1, candidate2, accepted, deleted, and unknown, see Figure 5-6. This is simplified from Figures 4-1 and 4-3. The solid lines indicate transitions that can occur automatically; the dashed line transitions can only be achieved manually. Unknown implies that there is no such entry. When a name is first proposed it is in the first candidate state. Upon another use of that name with that object (net address), it moves to the second candidate state. The third use puts it into the accepted state, where it remains unless it is manually deleted. It is only when a current context entry is in the accepted state that the address is not displayed when the name is displayed. Thus the only factor discussed

in Chapter 4 for acceptance is the number of uses. In addition, in order to allow for cleaning up a context, an expunge operation is included as well, which removes all deleted entries from the context, making them unknown again. A name-address pair can go from either the deleted or unknown state into the first candidate state.

The hooks are available in the implementations of contexts and aggregates for merging, although this was not put into the prototype of the user interface. When two contexts are merged, the states of all the entries in them are determined as in Figure 5-7. This is a simplification of Figure 4-2. Because the aggregates and contexts are being used by only one application and in a very stylized way, the acceptance and merging procedures can be included in them directly and need not parameterize them by these procedures.[14]

There are three levels of operations provided to support naming as described above. The topmost level is the user interface to the mail system. This is supported by operations on aggregates, which in turn in some cases (except for operations on the environment) are supported by operations on contexts.[15] The functions and operations are all listed in Appendix B. It should be noted here that several operations have been included that should not be accessible to users, because this is

---

[14] There was a problem in Clu. For reliability every change was to be saved onto disk. In Clu there were two choices. This could be done by converting all the information into a file losing type information and requiring conversion code within the procedures. The other alternative was to use a function called gc_dump to copy the object with its type information into a file. For efficiency the choice was the latter, but the context cluster needed to be parameterized by procedures for acceptance, deletion, and merging. Such objects can be created and were originally, although it was discovered later that due to implementation limitations, procedures cannot be gc_dumped.

[15] This implementation was embedded in a pre-existing mail system written by Mark Rosenstein at MIT. It is written in Mock Lisp, the extension language of Gosling's Emacs [14] and runs on a Vax 11/750 running BSD 4.2 Unix [57]. Mock Lisp is not a rich enough language to achieve what was needed, so contexts, aggregates, and an interface are written in Clu [30] and run in a separate process. Only the user's interface within the mail system and the operations defining the context and aggregate clusters are considered here.

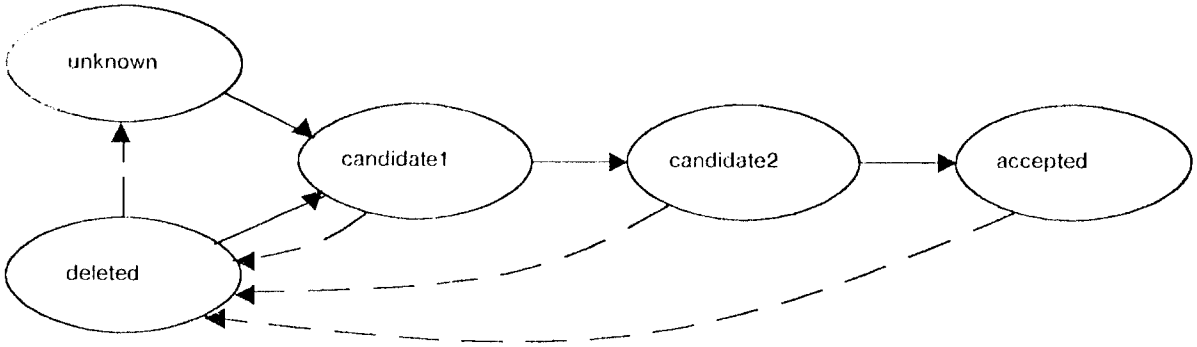

Figure 5-6: Possible states and transitions for entries a context

	u	d	c1	c2	a
u	u	d	c1	c1	c1
d	d	d	c1	c1	c1
c1	c1	c1	c1	c1	c2
c2	c1	c1	c1	c2	c2
a	c1	c1	c2	c2	a

u	unknown
d	deleted
c1	candidate1
c2	candidate2
a	accepted

Figure 5-7: State table for merging two contexts

only a prototype and the users of it are sophisticated programmers and Emacs users. These are expunge-context and change-status. This allows the user more direct

access than generally recommended to names and contexts, side-stepping the aggregate mechanism. The changes to the operations listed in Appendix B.1 are due to three factors. The two legitimate ones are the addition of the Aggregate field to messages and the need to translate names both when sending and displaying messages. The third cause for changes to the mail system was incomplete support for multiple processes in Mock Lisp. Those operations are indicated as such.

This section has described a simplified version of the model, that was used in the implementation of the ideas in a mail system. Users have private copies of shared current contexts and aggregates. Contexts can only contain names for user mailboxes, representing the users to be named in a shared context and also the participants in the sharing of that context. Each mail item carries with it the name of the aggregate and the names and addresses of all addressees as well as the sender. In general, the sender and recipient need not see or use those addresses. In addition simplified acceptance and merging procedures were used and no deletion occurs automatically.

The next section discusses conclusions that can be drawn from the experience with the mail system.

5.4 Lessons from the Mail System

The mail system was a further simplification of the model that was presented in earlier chapters. In turn the ideas presented in those earlier chapters were a model of human naming and communication. In spite of these simplifications, there are lessons to be learned from the mail system. Three are important enough to highlight here. First, even with the simplification of some of the mechanisms such as acceptance and deletion, a model can still be provided that is useful to users and reflects patterns comfortable to them. Second, the limitations placed on the mail

system by using only the mail system itself as the medium of communicating new names reflects human patterns. Although computers could provide much more sophisticated mechanisms for support and update of shared names, those might be disconcerting at best to the human users. Third, only the mail senders and recipients have been included in the aggregate and context mechanism. Research into conversation-based mail [8] is progressing in grouping and managing messages on a similar basis to that suggested here for name management. Each of these three points will be discussed in further detail.

First, consider the use of a single, simple acceptance procedure and no deletion procedure. Carroll's studies [7] have shown that one facet of accepting names is repeated usage. For simplicity it has been assumed here that it makes no difference who reuses them from an individual's point of view. In fact, carrying this further, the assumption is made that reviewing them by looking at a message repeatedly will have the same effect as reuse for the individual. In addition, three possible states on the road to acceptance have been assumed as mentioned earlier and depicted in the state diagram, Figure 5-6, reduced from the four suggested in Chapter 4. When a new name and address pair arrives in a message and the recipient reads the message, the name and address pair is added to the current context in the candidate1 state. Upon each successive reading or use of the name in an outgoing message, the context entry moves to the next state in the state diagram until it becomes accepted. Until the time when it is accepted, when it is displayed to the user its translation is displayed as well. Once the user has seen the name with its associated net address three times, it is assumed that the user will know to which address the name refers. This procedure reflects part of what humans do in jointly choosing names. Another part, not included in this implementation, is a mechanism for allowing names to mutate during the acceptance procedure as discussed in Chapter 4. This was determined to be too complex to include in the implementation.

The second lesson to be learned from the implementation deals with limiting the potential uses of the computer facility. Consider briefly a situation in which three people are discussing a particular subject. One day one of them is unavailable and the other two continue, defining new names in the conversation. The third will probably never be brought fully up to date about what went on. Suppose the two defined a new name "zibble", the name for a new concept that they are proposing. The third one will not realize that anything went on until the new name is used. In this mail system, if the name is not yet defined in the third person's copy of the current context, then when it arrives in a message, it will be added as a candidate and its translation will be included until it has been used enough in the local copy of the context. Thus, the third person will be brought up to date on any names that continue to be used and were defined during any absence. In such a mail system, the computer system could easily provide complete recall even of those events in which someone did not participate. Thus while one person was not participating his or her private view of the context could be changing. This was done by Comer and Peterson [8] with respect to messages, but it would be disconcerting at the least to discover that one's working namespace had changed while one was not actively viewing the changes. Although computers could provide a more automated form of name management, it would have the problems of not reflecting humans' patterns of naming.

The third lesson is that some of the goals of this research are applicable to other domains than naming. The goal in this work has been to analyze and address problems of naming. In doing so one conclusion has been that communication, cooperation and sharing play in important role in the functions and uses of names. The work of Comer and Peterson [8] is one of the most recent steps in the area of conversation based mail. They propose that not only should messages be tagged with the conversation of which they are a part, but in addition, each message carries

a reflection of the state of the sender at the time that the message was sent. Thus, each message reflects both the conversation and those messages in the conversation that were read by the sender prior to sending the message. In a different approach from that of this research, Comer and Peterson are presenting some of the same [illegible]s that have been presented here. They identify a conversation on the bases both of the group of participants and the topic of interest. Such a conversation consists of a set of messages identified on those bases, and each message is identifiable only locally within the conversation of which it is a part. In addition, the idea that there is something unique about the state of each participant is also important. In this case, the state of the person is reflected in the list of messages previously read. It is the idea of the context from which a sender is sending that is new and unique in Comer and Peterson's work and which, indeed, ties it more closely to that of this report. Comer and Peterson choose to provide a standard globally unique naming scheme. This work is progressing in Peterson's doctoral research. In an ideal mail based conversation, everything would be based on the conversation itself, both those aspects that are shared as well as those that are unique to an individual participant. Such a system would incorporate both the ideas of this research and those of Comer and Peterson.

Chapter Six

Design of a Naming Facility for a Programming Support Environment

6.1 Introduction

By considering electronic mail, much was learned about a naming facility. In order to understand naming facilities better, the requirements and a design for such a facility in a programming support environment will also be explored. Programming in anything but the smallest project is a social activity requiring cooperation and coordination among a group of people working toward a single goal, each with a separate but complementary set of tasks. A programming support environment may provide many functions for all involved in a programming effort. Certain naming facilities can help to improve even the simplest functions. It is the supporting naming facilities that will be explored in this chapter. This study will begin with an examination of the problem and brief summary of related work in this area. The chapter follows a structure similar to the previous chapter discussing the electronic mail system. The chapter will begin with an overview of what is needed in a programming support environment, followed by a presentation of the extended model used in this domain, a discussion of the operations needed, a proposal for a possible representation for the data structure and some concluding remarks comparing this version of the model with the previous one.

6.2 The Programming Support Environment

A programming support environment is many different things for different people at different times, but one can say that it supports people in their programming

efforts. In particular, it is especially useful when the programmer has a number of tasks related to a programming effort and must coordinate the work with others working on the same or related projects. The tools of a programming support environment may include editors, compilers, interpreters, linkers, loaders, testing facilities, debuggers, documentation facilities, product and revision announcement facilities, etc. Exactly which tools are needed and in what form is not the topic of this research. For a number of such programming support systems, see the "Software Engineering Symposium on Practical Software Development Environments" [47] in addition to the earlier work by Tichy [55, 56], Schmidt [45], Kay [53], Dolatta and Mashey [13] (for more on the Programmer's Workbench see also Bianchi and Wood [4]), Weinreb and Moon [58], and Lancaster [27] as examples.

One important problem to be solved in a programming support environment is how to distinguish an object from among a set. Although commonly not addressed in programming support environments, the problem of identification and distinguishing among objects can be separated into several problems, as was done in earlier chapters in this research. One part of the larger problem is naming. It implies possibly joint decisions about the names that will be assigned to objects and the contexts in which they will be recognizable. There is an additional part of the problem that plays an especially important role in programming support environments. That is the issue of selection of an object based on information about the object that has not been pre-selected as a name.

A brief example will help to explicate the distinction being made here. Consider a procedure named "integrate". The name is chosen as a name and assigned to the

procedure.[16] In addition, suppose it is the intention of the programmer that this procedure be in Clu, although a first version might be sketched out in a pseudo-Clu invented by the programmer for this purpose. The programmer might also identify the procedure with the label "language: Clu". This name will be available whether or not the sketch is converted to Clu that can be compiled. Suppose that the programmer requests that a compiled version of the "integrate" procedure be installed in a public library, but a compiled version does not exist. A friendly programming support environment may search out the object named "integrate" and "language: Clu", interpret the latter and attempt to compile the code, although the fact that the object is identified as being in Clu does not guarantee that it is. Therefore, the installation request may fail, because a name for the object was not correctly meaningful. The installation procedure would in fact use the compiler not only to compile, but also to identify an object that can be compiled and therefore matches the language specification for Clu. Selection of objects in Clu cannot be done on the basis of names assigned to those objects, but require some additional functionality from the selection mechanism. On the other hand, the naming function remains important and bears separate investigation because its functionality is universal.

Lancaster provides an approach different from the other researchers in this area. Her work is described here briefly, because her approach is similar to the approach taken in this research and is not readily available in the literature. The problem domain is that of selecting an implementation from among a set of implementations for a particular specification. In order to achieve this and support a collection of goals similar to the observations about human naming first enumerated here in

---

[16] It is probably chosen because it is meaningful to potential users of it and therefore is more easily remembered, although a name such as "x27" might be chosen simply as an identifier. To the user of the procedure it is no less or more usable depending on which names was chosen.

Chapter 1, she proposes a library. She recognizes that the names must be shared but does not discuss shared management of the names. She proposes what she has identified as a naming scheme to address many of the problems inherent in selection in a programming support environment. Her library is used to identify implementations by means of sets of attributes. Each attribute consists of a name and a value, which may define relationships between objects. The library does not actually contain objects, but rather points to objects outside the library. The library is separate from a general filing scheme that would contain all implementations, as well as all other related objects such as specifications, compiled versions of the implementation, and, in fact, the implementations themselves. For all objects identified in the library there are required and optional attributes. The set of all these attributes or subsets of them can be used to identify implementations and select individual ones.

Where this research parts ways with hers is in the definition of naming as opposed to other activities. A clear distinction was made in earlier chapters of this work between information recorded to be used as a name and other information that has more to do with the state of the object used as part of a computation that may result in selection. There may be situations in which these two appear to be similar, but the support mechanisms to use the two are dissimilar. The naming facility is a service that can easily and valuably cross application boundaries whereas the computation/selection requires simultaneously more complex and more application specific service. It is not unreasonable to join the two in a particular situation if naming is not to be unified across application boundaries, as was done by Lancaster.

This work concentrates on the naming support as distinct from other forms of selection that is needed for a programming support environment, especially recognizing that programming efforts must be done in conjunction with other people. In general the sharing of name management and name resolution is left to

two mechanisms, the library and the file system. File systems present a problem in a programming support environment. They do not provide the support for shared and cooperative naming, the flexibility for the individual, nor the flexibility in structure that humans use in their everyday activities. This was discussed earlier in Chapter 2.

As mentioned earlier, Lancaster provides an example of a library facility. A library can provide a number of functions: cataloguing, modularizing the namespace, allowing for overlap in choices of names, selecting among multiple implementations and multiple versions, locking, recording dependencies, providing consistency based on them, etc. Much of this functionality is not naming.

In addition, there is another area of naming in a programming support environment, the names embedded in the objects created within the programming support environment. The problem here is that not only must programmers cooperate in their naming, but also there must be provision for both the programmer and user to bind names to objects. The situation is the following. The programmer must use names in some cases bound to objects and in other cases not bound during the programming effort. Those names not bound during programming must be bound at later times. The Known Segment Table in Multics mentioned earlier is one mechanism for achieving this. Binding may occur in several stages. For example, some binding may arise from compiling source code. Further binding may occur when compiled code is linked, loaded or executed. In each case, the new bindings are the result of merging those already known and some found through the bindings of the client or user requesting that the activity occur. Thus, in each case a merge occurs of what was provided as a partially defined template for a namespace and bindings found through the client or user's namespace. As will be seen below, this merge is the same kind of merge discussed in Chapter 4.

A programming support environment has even more need for more complex names than those provided in the electronic mail system implementation. In the mail situation names consisting only of strings sufficed. A richer naming facility would allow for attributes, each of which has a name and a value. This approach has been used in a number of places, such as Lancaster [27], Oppen and Dalal [36], Dawes et al. [11] and Kerr [20]. In addition, much work has been done in this direction in the Artificial Intelligence community. The approach that will be taken here will follow more closely the work of the four papers mentioned above. Such an extension would have enhanced the mail system, but did not appear to be as important as in the case of the programming support environment. The structure implied here is simply a means of organizing the meanings of names, as was discussed in Chapter 2 when meaningfulness and structure were addressed as part of understanding the nature of names.

In order to achieve the desired functionality, two facilities will be designed. Both are based on the framework previously proposed in this work. The first is a library naming facility to aid in cataloguing, sharing and cooperating in naming and the second is templates and the associated operations to make them useful.

6.3 The Model

The model for naming in a programming support environment consists of aggregates and contexts, expanded from that model used earlier in Chapter 5. In addition, certain contexts and aggregates will be used in stylized ways in order to achieve the desired effect. Therefore the modifications to the basic mechanisms will be discussed first, then how they will be used, followed by a discussion of the operations needed to achieve the goals. No changes are proposed here for aggregates, so the discussion will be limited to contexts, followed by discussions of two new terms, **library contexts** and **template aggregates.**

One of the ways in which humans identify the context within which they want to resolve names is by the other participants involved. The electronic mail system was anomalous in that the objects being named were also the participants in a shared context. Therefore, these two facets of the context were combined, simplifying contexts. In most cases, the named objects will be distinct from participants in sharing. Thus, in the programming support environment, a shared context must also have associated with it a separate set of participants. Certain participants may have different effects on the shared context from the other participants. For example, it may be that a librarian for a program library is the only one allowed to create new names in the library, while other participants can only call on the library to resolve names. This interaction between the set of participants and the acceptance and deletion procedures will recur later in this discussion.

A second modification of the context model is that names may be chosen without knowing into which object they will be mapped. This is needed in order to provide for such situations as the recursive function, or including a call to a procedure that has yet to be written. The name must be included in the source code. In fact, as long as the code is not actually invoked, many compilers will allow it to be compiled, in order to begin the process of testing and debugging with incomplete code.

The third change from the previous model, as has been discussed, is a meaningful structure consisting of names as pairs of attribute or name and value. This last change allows for names that can manifest more meaning, better reflecting human naming.

There is a special use for both of the types of contexts and aggregates. The special use of the context is as a **library context**. There are three requirements or restrictions placed on a library context.

- A library context will contain only attributes from a pre-specified set.

For simplicity, since this work is not research into programming support environments, a superset of Lancaster's standard attributes will be assumed. Others such as Schmidt [45] propose a slightly different set. Since, in a general programming support environment, namable objects may be other things besides implementations, such as specifications or shared sets of definitions (in Clu a set of **equates**), the set of standard attributes will be enlarged. It will also be expanded to provide each object a name that is unique within the library context.

- An object can exist in at most one library context. As previously discussed, a name in a context may label another name allowing for indirection and control of binding time between the name and the object. On the other hand, a name may also label the object directly. A restriction on library contexts is that an object in the programming support environment will exist in at most one library context and in that context will have exactly one unique name, although it may have other non-unique names, for example OwnedBy or RelatedSpecification.

- A library context must be able to store names that are not yet assigned to objects. The understanding is that before one needs to access the object using the name, the object will have been created. The problem is exhibited in its simplest form when one writes a recursive function. One must be able to name the function before it is fully defined.

The use of library contexts will be in conjunction with unrestricted contexts. The unrestricted contexts will provide the full flexibility of naming discussed in previous chapters with one minor difference. Names or attributes can be translated only into other names in other contexts. Those may or may not be names in library contexts. These additional contexts will allow for private work or work by subgroups of a larger group. For example, a subgroup may want to use a new experimental set of objects not yet released for general use. It is worth noting here that there may be objects in no library context, but only in non-library contexts. An example of one such object is the list of errors due to running a compilation. Such an object probably does not belong in a publicly used library, but only in a private context. The additional contexts will be needed to meet the goals of the full richness of

naming spelled out in earlier chapters, that are also beneficial for a programming support environment.

The model presented thus far is somewhat over restricted. It would not allow objects to migrate from one library to another. But in a distributed computing facility, one may discover that an object should be relocated for convenience or efficiency. If an object is moved to another library, all those references to the object in the original library will be left dangling unless a forward pointer is added to the library entry. Therefore, by allowing such "tombstones" pointing to another library, more than one library entry is permitted for some objects.

The special use of the aggregate in the programming support environment is as a **template** aggregate. In the model here each object will consist of the actual object, such as a procedure, and a template aggregate. The template aggregate is not special in form, although, most likely it contains some names not yet assigned to particular objects, but reserved for future use. Providing a namespace for an object that is separate from the namespace in which the object was created is not a new idea. This is done regularly and was elucidated by Saltzer in his general discussion on naming [42].

The template aggregate provides a special case of the merging problem discussed in Section 4.6. Not only must the object's and the user's contexts be merged, but in this special case an environment must be created as well from the two aggregates. Exactly how this is to be done must be specified by the creator of the particular template. It may differ for each template. The specification may depend on whether or not both current contexts affect the resulting current context; if both do, how conflicts are resolved; if not, does the unused one simply become part of the environment, and how conflicts in the rules of the two environments are resolved.

An understanding of the enhanced model for contexts and stylized uses for contexts as library contexts and aggregates as templates and a discussion of the operations needed to support them is now possible. That will be followed by a presentation of a possible representation.

6.4 The Operations

An understanding of the objects and their uses is only part of the description needed in a design of an implementation. In addition, a list of operations is needed. The model for contexts has been expanded from the mail system; the resulting operations on both contexts and aggregates are listed in Appendix C.1. For completeness those operations include arguments for state modification of entries. It should be noted here that although in the operations, names are represented as strings, they should in fact be logical combinations of strings, allowing the client to name an object by a set of names. An implementation of this would be embedded in the implementations of the appropriate operations. New operations are also needed in the programming support environment to implement library contexts and template aggregates.

The library serves a number of functions in a programming support environment. In addition to the cataloguing, sharing and joint management that have an effect on naming, a library may also record and manage relationships among catalogued objects as well as provide support for other forms of selection among sets of objects. This research is considering only the naming functions and therefore will discuss only the operations needed for library contexts.

Library contexts provide a shared context for all the participants in perhaps a particular project. The library context will be the sole repository for the "official" versions of all objects of interest to the project as a whole. Entries in a library will be

restricted so that each type of object will have a fixed set of names. For example, a procedure object might have, in addition to its name, the name of the author, the name of its specification, the names of other implementations of the specification, the names of related documentation, the names of other procedures on which this one depends, etc. Different types of objects will have different sets of names.

For simplicity, each object in a library should be contained in no more than one library context, although there is no way to enforce this, since libraries are independent of each other. The problem is that most names have manifest meanings and as such may become inapplicable or incorrect. An added complication is that the fact of an object's containment in a library is not an attribute of the object. Therefore, when the object is modified or its names change, this will be recorded only where specified. Keeping names in more than one library in synchrony would be difficult at best and might be impossible if one could not locate all of them. Therefore, for the purposes of this work it will be assumed that an object is in, at most, one library and that whenever an object is added to or modified within a library some of its names may change. There are several issues relevant to library contexts that can be addressed separately.

Creation and updating of names in a library must be considered. When a new object is entered into a library, a set of names will be specified for it based on its type, as mentioned earlier. Some of these will be defined at the time of creation, others only later. Some may be optional. Since this is not research into programming support environments, although the facility must be here to support it, those choices are left to others in the field of programming support environments. In addition, there are situations in which only a label is chosen, for example, if the object does not exist, but the name is needed or should be reserved. The standard context operations are listed in Appendix C.1. The additional procedures needed for library contexts are listed in Appendix C.2.

Another important issue in considering library contexts is moving objects from one library to another for convenience or necessity. The fact that names can be mapped into other names in other contexts will be used in order to avoid dangling references and help previous users of the object being moved; indirect names will replace direct references. As previously mentioned, if an object is contained in two or more libraries, the names may become obsolete. There are two possible approaches to this. The first is to assume that all such information about an indirect reference may be obsolete. The second is to include an operation on libraries that causes them to trace all such indirect references and update all names for each indirect reference. The operation needed to support the latter is also in Appendix C.2

Finally, with respect to library contexts, it should be pointed out that all library context operations can be implemented out of the standard context operations. For example, consider move_library_reference. It will mean creating a new reference in the new library using add_name. If the new label needs to be unique in the new context, some further checking in the new library may be needed before the object is moved. Once the name has been selected and the new reference created in the new library, the old reference can be modified to reflect an indirect reference.

Three special operations are needed for template aggregates beyond those for aggregates listed in Appendix C.1. They are listed in Appendix C.3. The first operation is a replacement for the create operation of aggregates. It is needed because a template aggregate is created by creating an aggregate and then simply wrapping it in the template aggregate type. The second procedure is the merging operation that will be used when a template is to be merged with a client's aggregate. Finally, an aspect of a template that must be considered is whether all users of the object share a single current context or whether each will have a private copy. The last operation, share_current_context allows for selecting this option.

6.5 Design of an Implementation

In order to validate the proposal for a more complex implementation in this chapter, a representation is described in this section. An implementation would follow directly from it. Since library contexts and template aggregates are quite similar to contexts and aggregates their implementations are not discussed in detail. Furthermore, since aggregates here are the same as in the electronic mail system, they are not reconsidered.

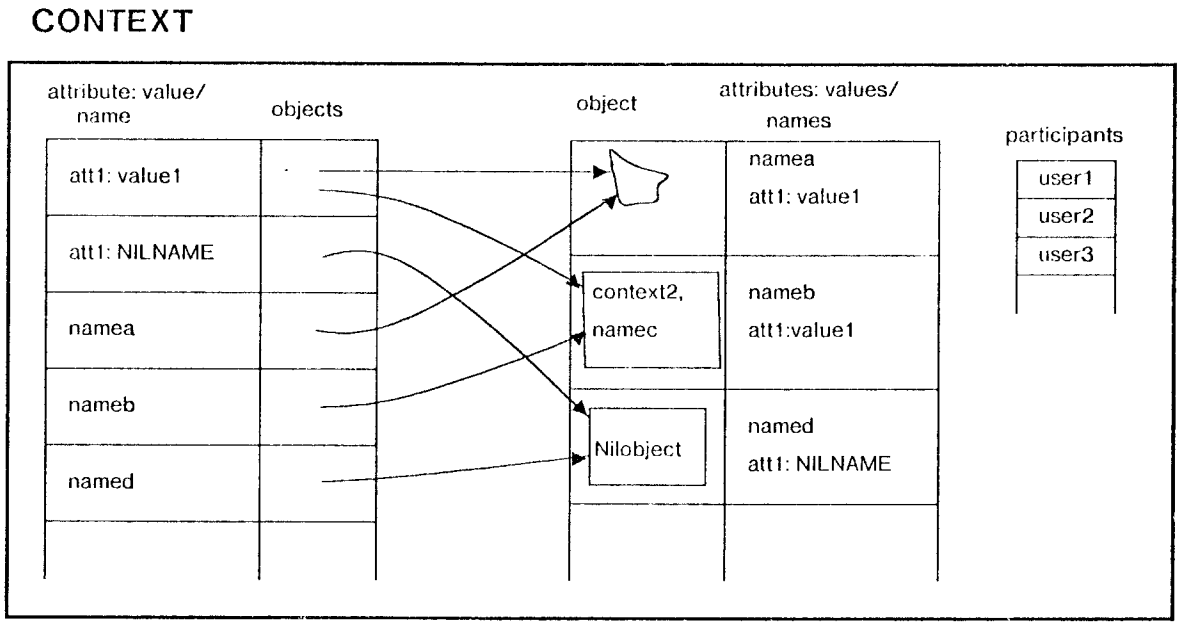

Figure 6-1:A representation of a context

The representation of a context proposed here is as follows and is depicted in Figure 6-1. A context consists of three sets, two of which are discussed here together and the third later. The first is a set of names. A name may be a pair or a single entity,

and each name is associated with a set of objects. The second set in the context is the set of objects. An entry in this set consists of an object or an indirect reference to the object in another context and a list of all names associated with it. Although this means that information will be duplicated within a context, it will allow for more efficient operation than otherwise. The set of names should be organized to optimize searches on average. This whole arrangement will allow for two sorts of fast access. The first is searching for all objects having a certain name. The second is finding all the names for a particular object. The tradeoff is that modification requires access to both sets. In those cases where a name is applicable, but not yet defined, Lancaster's approach of using **Nil** is proposed. In cases where a name is not applicable, the object is not in the set of objects to which the name can be applied. There is one further consideration: what to do in the set of objects about names that have been selected for objects that do not currently exist. Dummy objects are proposed to solve this problem. A dummy object is a place holder. In the set of names, the dummy object appears no different from any other object. In the set of objects, the dummy object has something in common with **Nil** as proposed by Lancaster; there is no object there, although there may be a set of names, rather than just one. The two reasons that one might want such an unassigned name are, first, that one may want to reserve a name and, second, that one may want to assign a collection of names to such a dummy object, later being able to attach that whole set of names to a real object. Thus there will now be **NilName** (which is the **Nil** that Lancaster proposed) and **NilObject**.

The third set associated with a context is the set of participants. How the participants are identified is not addressed here fully. As mentioned earlier, it may be a problem of authentication. The context is not expected to be an authentication service. Rather an authentication service is assumed to be accessible to the context and user. There are two possible approaches to using an authentication service.

First, the user can make a request of the authentication service to produce an unforgeable object that the context will believe, to be passed to the context either directly by the authentication service or by the user. Second, the context can request that the authentication service authenticate a particular requestor of the context.[17]

Before leaving this section library contexts and template aggregates must be reconsidered briefly. First, library contexts contain a little information above and beyond a standard context. A library context also has a record of those required and optional names that have been identified in it for specific types of objects to be named in it. Not all types need to have such specifications, and names not included in those lists can also be attached to objects of any type. This facility of pre-specifying attribute names allows objects of certain types to have names that fall into certain patterns. For example, it may be that part of entering a source code object into a library must be an indication of the language of the source code. An optional name might be the author of the code, assuming that it is known. The only additional information associated with template aggregates is whether or not the current context resulting from a merge is to be shared by all current users of the associated object. These pieces of related information in library contexts and template aggregates must be considered in their representations.

---

[17]It should be noted that authentication need not depend on globally unique identification. In fact, at best, it can depend on mostly unique identifiers. Encryption keys provide a good example of the fact that an absolute guarantee of uniqueness and unforgeability are impossible. It is all a matter of degree; cost and degree of the guarantee are closely linked.

6.6 Comparisons and Conclusions

Since the model presented in this chapter is an expansion of that of Chapter 5, the differences must be examined as a means of recommending in each area which choice is more general. In some cases, the simpler version may be more appropriate to the general case, with certain exceptions needed for particular applications. In other cases, the more complex version may be more appropriate, with the understanding that there are situations that do not need such full functionality.

This chapter contains a proposal for a second area in which the naming framework can beneficially be applied. There are a number of ways in which the framework was modified from the previous proposal. Each of those will be examined individually, considering whether each is of general applicability or not.

- **Names without bindings:** The programming support environment needed to allow for names to be chosen as place holders for objects that were not currently known to exist. For instance, that would permit naming of procedures that were to be written later. Although the issue did not arise in the electronic mail system, it might have been useful there as well. An example is a name that represents a role, for example "chair of the committee." There may be a time when there is no person in that role, but the role still exists.

- **Participants:** The reason that a separate list of participants was not necessary in the mail system was that the set of recipients was the set of participants. A set of participants must be a part of every context, although as occurred in the mail system the implementation of contexts could be simplified because the entries in the context and the set of participants were identical.

- **Restricting an object to being in only one context:** It would appear that such a limitation exists for those objects in library contexts. In fact, such a restriction was suggested only among library contexts in order to simplify implementation and synchronization of information, although as suggested, there is no means of enforcing it. Such a restriction would certainly be detrimental to a mail recipient naming scheme as well as many other facilities and is unnecessary. Therefore it is not

recommended as a general feature of contexts. It should be noted here that restricting an object to being in no more than one library context is a separate issue from whether or not the library context itself consists of multiple copies. Multiple copies can be synchronized to any desirable degree.

- **Access control:** Access control is related to naming in that it may be used to restrict the privileges of certain participants in a context. It may depend on authentication. In a library facility access control may be used to allow only the librarian special privileges. Access control was not discussed in the electronic mail system, although it could well be a useful part of such a system. The advantage of including access control and authorization is that one can leave objects completely accessible if one wants, while having the opportunity to control access when it is needed. Therefore, an access control mechanism is recommended, although it is external to a naming facility.

Thus the choices here are to allow for flexibility, permitting the implementer or user the choice of whether names should have bindings initially, whether objects can be entered into one or more than one context, and what the access control ought to be. In addition, the set of participants should be distinct from the set of objects named in a context.

Chapter Seven

Conclusion

7.1 Reflection of the Ideas

In this research, a name is defined to be an object that can be associated with another object and has an equality operation defined on it. The most common use of a name is as a handle for an object. A name used thus provides access to the object. A second use for a name is as a place holder for an object. The reason that place holders are important is for use as a substitute for the object itself. Substitution may be needed either if the object is to be shared and cannot exist in more than one place at one time or if the named object does not exist at the time. The problem being addressed in this research is the design of a computer naming facility achieving the following goals. First, names must provide access to named objects as well as be usable as place holders for the objects named by them. Second, it must be possible to share those names across computer boundaries. Third, it must be possible to communicate using names. There are two forms that this communication takes. One is the transmission of the names themselves and the other is transmission of information in the names because the names are meaningful to be to the user and recipient of the name. Finally, an implementation must be feasible.

Computer naming, as described in this research, reflects a social process. The social process is assigning and using names privately or in limited groups and sharing the responsibility for that assignment, modification, and deassignment. The process of naming, when done cooperatively, involves entities that can operate independently as well as in cooperation with each other. As such, these entities form a federation

in which each brings some individuality to the joint effort and within the cooperation retains a certain degree of autonomy. Human naming has provided this research with both goals and examples on which to base solutions for two reasons. First, humans function as an amorphous set of federations that form and reform unpredictably and when needed, using naming as part of the interaction within the federations. Also, computer systems are built, in the end, to support humans in their activities. Therefore, this research set out to investigate the sort of naming that humans do jointly. In order to understand the problem better, various parts of the problem can be considered separately before looking at a solution.

Characteristics

A number of characteristics of names can be identified. First, there are three roles related to names and naming, the assigner of a name, the resolver of a name, and the user of a name. The assigner determines which name should be associated with which object. The resolver performs name resolution or translation. The user of a name can only use names that the assigner has chosen. If resolution is needed, then the resolver must also be able to do its job for the user. The user will use a name either to access the named object or as a place holder for the object. Beyond these three characteristics of names, one can also consider the degrees of uniqueness and meaningfulness of a name. If a name is unique within the domain of a resolver, it will be resolvable to no more than one object. The more meaningful a name is, the more information the name itself carries from name user to name receiver. Meaningfulness may be manifested in the form of structure of names.

Observations

Returning to the analysis of the research problem, a set of observations can be made about how humans name the objects in their worlds. Humans use names to a great extent to communicate with each other. Part of that social process of communicating also involves each participant in that process bringing an

individuality into it. In the formation and reformation of cooperating groups, names are frequently reused in different contexts and at different times to have different meaning. In addition, a particular object may have more than one name at any given time reflecting either different meanings and characteristics or different perspectives. Both in order to achieve such multiplicity and because the size of a universal namespace is unmanageable, small, local namespaces are used. In addition, there are several more aspects of usage of names. Humans use a number of approaches to naming and generally do not restrict a particular approach to a particular type of object. As mentioned earlier, names often have meanings that are conveyed between user and recipient when names themselves are shared. One final point about human naming is that it appears to take little or no effort to choose, share and use names both privately and cooperatively in a group.

Cooperation

Cooperation and joint management of names form the final part of the examination of the problem of naming. This involves first recognizing that a name passes through a number of stages from the time it is proposed until it is accepted as a name for a particular object. There also may be a range of stages as a names falls into disuse and is slowly forgotten or is more explicitly replaced. Many factors can be identified as potentially playing a role in these activities. A small number appear to be both important and practical to implement in a computer system. The number of uses of a name in association with an object is probably the single most important factor. Frequency of use may also be quite important. Finally, the fact that a name bears a similarity to another previously selected name and that similarity has a manifest meaning may make the later choice more readily acceptable. In current file systems, an example of this is accepting a file name with an extension of "bin" as the result of a compilation with the primary component being the same as the primary name of a file containing source code. This is a restricted and stylized use

of information about previous choices, but for efficiency it is probably better to limit this factor to such a simple form.

The model

To address the problem of creating a naming facility, this research proposes a model consisting of a set of objects for each client of the system. The objects are known as aggregates. Each aggregate provides a private view to the client of a possibly shared namespace. An aggregate is composed of two parts, the shared namespace, known as the current context, and the environment, that part of the aggregate that personalizes it for this particular client. The current context contains the names shared by the group, while the environment identifies a set of other mappings between names and objects which the individual client may wish to use as proposals for the current context. The environment consists of a partially ordered set of other namespaces in which this client is also a participant. Both the current context and the environment are based on a simpler form of object, also proposed as part of this research, the context. A context also has two parts, a mapping from names to objects and a list of participants. The model supports acceptance and deletion of names in stages based on usage and jointly by the participants sharing responsibility for the context. No particular structure is placed on either the organization of contexts or the internal structure of names within contexts. Instead both of these are left to the discretion of the participants in the sharing. The context provides the basic mechanism for name translation and shared management of namespaces.

The implementation designs

The discussions of implementations demonstrate both the feasibility and usefulness of the mechanisms. A brief summary of how the problems and issues of Chapter 2 are reflected in the domains of electronic mail and a programming support environment and how the designs in those domains address the issues will serve here as a review of Chapters 5 and 6. In both domains, activity occurs in cooperation

among varying groups of participants communicating and cooperating only when such joint activities are needed. Federation is the norm and is assumed in both of the implementation designs. Furthermore, names as used in the implementation designs fall under the definition that they only be required to have an equality operation and can be used either for access or as a place holder. In both domains, names are chosen to be strings. In addition, in the electronic mail implementation, since the objects named can only be strings, the untranslate operation is also guaranteed to be available. In the programming support environment, it is only possible the untranslate if an equality operation exists for the objects named in a context.

Five attributes can be used to describe a set of names: the assigners, the resolvers, the users, the degree of uniqueness, and the degree of meaningfulness. In both domains, the assigners and users of the names are the same pool of participants, although the programming support environment allows for some participants such as a librarian to have special privileges in terms of defining names. In both examples, the resolver of a name is always a specified aggregate that the programmer or user can select. As for uniqueness, in the electronic mail implementation, no restrictions were placed on the number of assignments either of a name or to an object. Some such limitations might be useful in the programming support environment, although the proposed mechanism does not enforce any. Finally, in considering attributes of names, since the assigners and users are generally people and the names are strings in which humans can easily discern meaning, the degree of meaningfulness is to whatever extent the human participants desire and choose.

In terms of the goals of the naming facility, the first was to support the definition of names; this is done in the two domains as discussed in the paragraph above. The second goal required support for sharing and communication of and by use of those

names. The mechanisms of contexts and aggregates including the joint management facilities provide for sharing both the names themselves and responsibility for managing them. This functionality is maintained from the model to the implementation. Communication is supported both by the representation of the names as string, allowing for information to be shared in the names themselves, as well as in the electronic mail system using the mail itself as the medium for passing names around. The programming support environment did not propose a particular medium of communication, because in an implementation that will depend on the characteristics of a supporting distributed system. The third and final goal was that the model be implementable. That is demonstrated through the implementation of the electronic mail system and the implementable design for the programming support system.

This section has presented a review of the problem addressed in the research reported here, followed with a brief summary of the general proposal for a solution and brief return to the two domains for application of the model. There must be two further parts to such a review. A research project such as this cannot be considered in isolation. There will be parts of the project or related issues that have not been investigated fully or satisfactorily. In general such unfinished business leads to suggestions for alternative or further work that would enhance the project. The other side of this coin is a review of those areas in which the research was successful and has made useful contributions. The following two sections will address these to sides of such a review.

7.2 Lessons and Future Research

With a topic as broad as naming, the research possibilities are endless, especially when one attempts to walk the narrow line between facilities that are efficient enough to be useful and those that more and more accurately mirror direct

interpersonal communication. In attempting to do so in this research many parts of the problem could not be treated fully. The following is a list of such issues in increasing order of generality. Each affords opportunities for identifying both possible weak points in the research as well as possible areas for further research.

1. Consideration of the implementation in the electronic mail system leads to a number of possible improvements.

   - The choice of a simple but little used mail system meant that few users were found for it. An implementation in a more widely used and better supported environment would be beneficial. This would allow studies along the lines of Carroll's, in order to observe the patterns that humans choose, given the freedom to choose.

   - A further enhancement would be to extend the namable objects in the mail system beyond the recipients. The other namable objects in such an environment would be messages, aggregates, and contexts.

   - One might extend contexts to reflect a combination of the ideas of this research and those of Comer and Peterson [8] as well. This research has explored those ideas only within the domain of naming. Such an extension would allow a deeper study of the social aspects of naming.

   - Finally, a more challenging implementation would be a broader subsystem or system, such as the programming support environment or a whole operating system. This would require that clients use only aggregates for all naming, being unable to step outside such a system. It would provide a more controlled environment in which to study patterns of usage.

2. Chapter 4 explored the idea of how the determination of a state of a context entry is made. Much further work can and should be done to examine these issues further. In order to learn more, either surveys could be done or systems could be built as previously suggested, that would allow for testing of different factors, with means of measuring user satisfaction with various factors. The latter would only test previously recognized factors, while the former might shed light on new factors as well.

3. In the discussion of a programming support environment, it became clear that the question of how selection is done, once naming has taken place is an important problem for some applications, closely related to naming. Although selection has not been studied here, there may be aspects of selection that are common across application boundaries. Some of the factors that may come into play are who used the objects in question most recently, when, the types of the objects, and how the objects were last used. Other factors may be important as well, as can be seen in the literature on programming support environments. Further work in this area would certainly be beneficial.

4. An interesting problem for which an adequate solution was not proposed in this research is initialization. There are two parts to this problem. The first issue is how such a system will start at the very beginning. The question of how the first context will be shared must be addressed. A second part of initialization is how any individual will be initialized when joining a pre-existing community. This problem was considered in the discussion of the mail system, but further work is needed on it also.

5. This research suggests that globally unique names are neither useful nor in fact implementable in general, with the expansion of the various electronically linked computational facilities. Yet many researchers, architects, designers, and builders of such distributed systems continue to propose naming mechanisms based on an assumption of the existence and use of globally unique names. This research suggests that humans do not need them and that they also are not needed in computer systems, at least not globally unique names. Of course, local uniqueness is possible and, in fact, necessary. Further thought, research and experimentation is needed in the area of globally unique names.

6. The proposal for the relationships among contexts in this research is that those relationships be unconstrained. If one considers human naming, there are many example of namespaces that form unconstrained networks. On the other hand, when people are making an effort to organize and catalogue objects, they will often use a hierarchical structure. If the problem is very complex, they may use several hierarchies with pointers from one to another. Consider briefly genealogies, a method of organizing familial information. A genealogy is generally viewed as a hierarchy with a root either in the past and

branching chronologically or the reverse reflecting the ancestry of an individual. Thus, although the flexibility of an unconstrained network is useful in many cases, a tool for hierarchical structuring may also be beneficial. Further research into this is needed. One way to study this problem is to use one of the existing non-hierarchical file systems to set up experiments and observer human behavior.

7. The proposals of this research are aimed at solving naming problems for small enough groups of users to permit reaching agreement and being able to share responsibility for management of namespaces. This may break down if the community grows large. Name management for large groups has not been considered but needs further work because those large loosely coupled communities are growing in frequency of occurrence.

8. Finally, the most open ended question in this area, the nature of names themselves, their development and relationship to the objects being named as well as the users of the names, can well afford further study. This research has examined names and naming carefully enough to identify various factors about which there has been much confusion in the past, but the concepts of names and naming are still far from being well-defined.

Although the items in the list above cannot be listed in order of importance, some deserve special attention. In looking toward computational facilities of the future, there are two aspects of naming that need the most thought and attention. They both are the result of the proliferation of personal computers with communications capabilities and the hardware networks for that communication. It is of paramount importance that the naming needs for very large communities of communicators be studied. Currently most developments are completely disorganized and achieved on a local and ad hoc basis. In addition, as the user community extends beyond the community of programmers and sophisticated users who have learned to manage in alien environments, it becomes more important to support environments more comfortable to humans. Several of the items listed above are aimed at that. The other issues raised above are also useful, although they are not as important as these two.

7.3 Contributions

This work will conclude with a review of the major contributions of this research. The research is a synthesis; it has pulled together ideas from several areas, ideas that in many cases have been recognized as useful in particular situations, but have not been recognized as part of a larger problem.

One contribution of this research is the recognition that a computer naming facility should support cooperation, communication, and sharing of names. Sharing objects or information has long been recognized as important, but sharing and cooperating in managing names for those objects is less frequently recognized as a goal for a naming facility. This research proposes that communication and sharing of names as well as objects must be part of the goals of a naming facility. The benefit of this contribution is in achieving greater functionality through less restrictive and more flexible naming.

A second contribution is the recognition that a computer naming facility should not support non-naming functions, such as selection, although naming facilities may have done this traditionally. Selection, involving means of distinguishing objects from each other by other mechanisms than naming, such as performing computations on the objects or various properties of the objects, is not and should not be considered naming. Separate facilities are needed for such necessary functions. In addition, names cannot generally be used to test for identity. Whether two objects are in fact the same object is dependent on various factors such as the types of the objects and the application using the objects. These should not and cannot be known to the naming facility. Finally, in a related problem, naming cannot be the only solution to authentication. Naming may be part of the solution, but more information that is not susceptible to any significant degree of masquerading or other forms of subversion of authentication procedures is needed

to perform authentication. Thus, this research proposes a further modification of the functionality defined as naming. This latter set of modifications allows the researcher, architect, designer, and programmer to recognize and separate functions and thereby reflect desired policies in a system more clearly and accurately.

The presentation in this research of a model for a single, unified naming facility providing local naming contributes a new idea to computer supported naming. As mentioned earlier, several universal name servers have been proposed or built, but they are remote services, not useful for naming small, local objects frequently. Addressing naming problems across application boundaries not only provides a savings in terms of efficiency by not repeating work, but in addition, allows for greater functionality that a collection of separate naming facilities. The reason for this is that it is difficult or impossible to use naming to reflect relationships across the boundaries of separate naming facilities.

An important contribution is the development of a method for joint management of shared contexts. The method includes a representation of degrees of acceptance of a name as a series of states. There are a few file systems, such as TOPS-20 [12] that provide a much simplified version of this as a convenience to the user. In that file system, the deletion procedure occurs in two stages, deletion and expunge. Deletion is reversible for a limited period of time, while expunging is not reversible. This mechanism allows users to change their minds about deletion. The mechanism proposed in this research reflects the negotiation and shared use of names, so that as a name's usage increases, it is more likely to become generally accepted and as it falls into disuse, it becomes more difficult to remember and use. This reflects the contribution of a new concept to naming.

The final contribution is the recognition that naming is a social process of communication. For this reason, the naming facility must distinguish the individual

from the group, in order to support the needs and contributions of both. That has been done in two separate ways. The group's needs and contributions are reflected in the concept of the context that contains those names upon which the group has reached agreement. In addition, the identities of the participants are recognized as an important aspect of the context. The individual is given recognition in the aggregate, which provides a private view of the shared context, as well as the individual's additional source of influence on the shared context. Thus these separate concepts reflect the different needs and influences of the group and the individual, allowing the group to communicate using shared and jointly defined names, while providing a private view and set of influences brought by each participant in that communication and sharing. The recognition of this last idea of naming as a social process is of benefit to all members of the computer community. It expands the functionality achievable by those involved in creating systems. That in itself is of benefit to clients of those systems as well. But it also extends the style and means of interaction through naming toward what would be possible among those clients outside the computational facility. The idea of communicating, cooperating, and sharing responsibility for names and name management with exactly those clients sharing a common interest is the most important contribution of this work to the future development of loosely coupled distributed computer systems.

References

1. G. T. Almes, A. P. Black, E. D. Lazowska, J. D. Noe. The Eden System: A technical Review. Tech. Rep. 83-10-05, Dept. of Computer Science, University of Washington, Seattle, Washington, October, 1982.

2. Apple Computer Inc. *Macintosh.* Apple Computer Inc., Cupertino, Califfornie, 1984. Reorder Apple #M1500. This is the introductory manual for the system.

3. J. H. Benjamin, M. L. Hess, R. A. Weingarten, W. R. Wheeler. Interconnecting SNA networks. *IBM Systems Journal 22*, 4 (1983), 344-366.

4. M. H. Bianchi and J. L. Wood. A User's Viewpoint on the Programmer's Workbench. Proc. 2nd International Conference on Software Engineering, October, 1976, pp. 193-199.

5. A. Birrell, R. Levin, R. Needham, M. Schroeder. Grapevine: an Exercise in Distributed Computing. *Comm. ACM 25*, 4 (April 1982), 260-274. Also presented at the 8th Symposium on Operating Systems Principles, Asilomar Conference Grounds, Pacific Grove, CA, sponsored by SIGOPS and ACM, December 1981

6. J. M. Carroll. Creating Names for Personal Files in an Interactive Computer Environment. IBM Research Report RC 8356, IBM, July, 1980.

7. J. M. Carroll. Naming and Describing in Social Communications. *Language and Speech 23*, 4 (1980), 307-322.

8. D. E. Comer and L. L. Peterson. Conversation-Based Mail: An Overview. Tilde Report CSD-TR 465, Dept. of Computer Science, Purdue University, March, 1984. Revised September, 1984.

9. D. H. Crocker. Standard for the Format of Arpa Internet Text Messages. NIC/RFC 822, University of Delaware, August, 1982.

10. R. J. Cypser. *The Systems Programming Series. Vol. : Communications Architecture for Distributed Systems.* Addison-Wesley Publ. Co., Reading, MA and Menlo Park, CA, 1978.

11. N. W. Dawes, et al. The Design and Service Impact of Cocos, an Electronic Office System. International Symposium on Computer Message Systems, IFIP TC-6, Ottawa, Canada, April, 1981.

12. Digital Equipment Corporation. *DECSYSTEM-20 User's Guide.* Digital Equipment Corporation, Maynard, Massachusetts, 1978. Order No. AA-4179B-TM. Updates have been made since this version was published.

13. J. A. Dolatta and J. R. Mashey. An Introduction to the Programmer's Workbench. Proc. 2nd International Conference on Software Engineering, October, 1976, pp. 164-168.

14. J. Gosling. *Unix Emacs.* Carnegie Mellon University, Pittsburgh, PA, 1982. This is the version in the public domain.

15. K. Harrenstien, V. White, E. Feinler. Hostnames Server. NIC/RFC 811, Network Information Center, SRI International, March, 1982.

16. IBM. *IBM Virtual Machine/System Product: Remote Spooling Communications Subsystem Networking General Information.* IBM, . No. 6H24-5004-3.

17. IBM. *IBM Virtual Machine/System Product: Remote Spooling Communications Subsystem Networking Program Reference and Operations Manual.* IBM, . No. 5H24-5005-2.

18. IFIP WG6.5. European SEG Meeting Report on Names, Directories and Lists. N 77, IFIP WG6.5, Systems Environment Group European Section, October, 1982. Bonn, October, 1982 and Rome, January-February, 1983

19. W. H. Jessop, J. D. Noe, D. M. Jacobson, J. Baer, C. Pu. An Introduction to the Eden Transactional File System. Tech. Rep. 82-02-05, Department of Computer Science, University of Washington, Seattle, Washington, February, 1982.

20. I. H. Kerr. Interconnection of Electronic Mail Systems - a Proposal of Naming, Addressing and Routing. International Symposium on Computer Message Systems, IFIP TC-6, Ottawa, Canada, April, 1981.

21. R. M. Krauss and S. Weinheimer. Changes in referential phrases as a function of frequency of usage in social interaction: A preliminary study. *Pschonomic Science 1* (1964), 113-114.

22. R. M. Krauss and S. Weinheimer. Concurrent feedback, confirmation, and the encoding of referents in verbal communication. *Journal of Personality and Social Psychology 4* (1966), 343-346.

23. R. M. Krauss, C. M. Garlock, P. D. Bricker, L. E. McMahon. The role of audible and visible back-channel responses in interpersonal communication. *Journal of Personality and Social Psychology 7* (1977), 523-529.

24. R. M. Krauss and S. Glucksberg. Social and nonsocial speech. *Scientific American 236* (1977), 100-105.

25. B. W. Lampson and R. F. Sproull. An open operating system for a single-user machine. Proc. 7th Symposium on Operating Systems Principles, ACM SIGOPS, Asilomar Conference Grounds, Pacific Grove, CA, December, 1979, pp. 98-105.

26. B. Lampson. Panel Discussion at SIGPLAN '83 Symposium on Programming Languages Issues in Software Systems. *SIGPLAN Notices 19*, 8 (August 1984), 51-60. Moderator/Editor: L. A. Rowe

27. J. N. Lancaster. Naming in a Programming Support Environment. MIT/LCS/TR 312, Massachusetts Institute of Technology, August, 1983. Also MS thesis.

28. K. A. Lantz and J. I. Edighoffer. Towards a Universal Directory System. Department of Computer Science, Stanford University, Palo Alto, Calif., unpublished paper.

29. B. Lindsay. Object Naming and Catalog Management for a Distributed Database Manager. Proc. 2nd International Conference on Distributed Computing Systems, Paris, France, April, 1981. Also Available as IBM Research Report RJ2914, San Jose, Calif., August, 1980.

30. B. Liskov et al. Clu Reference Manual. MIT/LCS/TR 225, Massachusetts Institute Technology, October, 1979.

31. P. Mockapetris. Domain Names - Concepts and Facilities. NIC/RFC 882, Network Working Group, USC ISI, November, 1983.

32. P. V. Mockapetris. The Domain Name System. Computer Message Services, IFIPWG6.5, Nottingham, England, May, 1984, pp. 59-70. Also Proc. IFIP6.5 Working Conference

33. R. M. Needham and A. D. Birrell. The CAP Filing System. Sixth Symposium on Operating Systems Principles, Special Interest Group on Operating Systems of the ACM, ACM, November, 1977, pp. 11-16.

34. R. M. Needham. The CAP project - an interim evaluation. Sixth Symposium on Operating Systems Principles, Special Interest Group on Operating Systems of the ACM, ACM, November, 1977, pp. 17-22.

35. D. A. Nowitz. Uucp Implementation Description. October, 1978

36. D. C. Oppen and Y. K. Dalal. The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment. OPD T8103, Xerox Office Products Division, Systems Development Dept., October, 1981.

37. E. I. Organick. *The Multics Experience: An Examination of Its Structure.* M.I.T. Press, Cambridge, Mass, 1972.

38. J. B. Postel. Simple Mail Transfer Protocol. RFC 821, Network Information Center, August, 1982. The author is at USC ISI, Marina del Rey, CA.

39. W. V. O. Quine. *Word and Object.* Technology Press of Massachusetts Institute Technology and John Wiley & Sons, New York, 1960.

40. D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Communications Of The ACM 17*, 7 (July 1974), 365-374.

41. R. Rom. Name Assignment in Computer Networks. TR 1080-310-1, SRI International, October, 1982.

42. J. H. Saltzer. Naming and Binding of Objects. In *Lecture Notes in Computer Science, Vol. 60,* Springer Verlag, New York, 1978, ch. 3, pp. 99-208.

43. J. H. Saltzer, D. P. Reed, and D. D. Clark. Source Routing for Campus Wide Internet Transport. Local Networks for Computer Communications, IFIP, IBM Research Laboratory, Zurich, Switzerland, August, 1980, pp. 1-23. Also Proc. IFIP Working Group 6.4 International Workshop on Local Networks

44. J. H. Saltzer. On the Naming and Binding of Network Destinations. International Symposium on Local Computer Networks, IFIF/T.C.6, April, 1982.

45. E. E. Schmidt. Controlling Large Software Development in a Distributed Environment. CSL 82-7, Xerox Corporation, December, 1982. Also Ph. D. Thesis for the Dept. of Computer Science, University of California, Berkeley.

46. J. F. Shoch. Internetwork Naming Addressing, and Routing. Proc. 17th IEEE Computer Society International Conference, IEEE, September, 1978, pp. 72-79. IEEE Cat. No. 78 CH 1388-8C.

47. , SIGSOFT and SIGPLAN. *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April, 1984.

48. J. E. Stoy and C. Strachey. OS6 - An experimental operating system for a small computer Part I: General principles and structure. *The Computer Journal 15*, 2 (May 1972), 117-124.

49. J. E. Stoy and C. Strachey. OS6 - An experimental operating system for a small computer Part 2: Input/output and filing system. *The Computer Journal 15*, 3 (August 1972), 195-203.

50. C. Sunshine. Source Routing in Computer Networks. *Computer Communications Review 1*, 7 (January 1977), 29-33.

51. L. Svobodova. A Reliable Object-Oriented Repository for a Distributed Computer System. Proceedings of the 8th Symposium on Operating Systems Principles, Special Interest Group on Operating Systems of the ACM, December, 1981, pp. 47-58. Also published as Operating Systems Review, Vol. 15, No. 5

52. D. P. Reed and L. Svobodova. Swallow: A Distributed Data Storage System for a Local Network. Proc. of the International Workshop on Local Networks, IFIP Working Group 6.4, Zurich, Switzerland, August, 1980.

53. L. Tesler. The Smalltalk Environment. *Byte 6*, 8 (August 1981), 90-147. This issue of Byte is devoted almost exclusively to the Smalltalk system.

54. J. C. Thomas and J. M. Carroll. Human Factors in Communication. *IBM Systems Journal 20*, 2 (1981), 237-263.

55. W. F. Tichy. Software Development Control Based on Module Interconnection. Proc. 4th Internation Conference on Software Engineering, ACM SIGSOFT, European Research Office, Gesellschaft fur Informatik, IEEE Computer Society, Munich, Germany, September, 1979, pp. 29-41.

56. W. F. Tichy. Software Development Control Based on System Structure Description. CMU-CS 80-120, Carnegie-Mellon University, January, 1980. Also Ph. D. Thesis

57. University of California. *Unix Manual.* 4.2 edition, Department of Computer Science, University of California, Berkeley, California, 1983.

58. D. Weinreb and D. Moon. *Lisp Machine Manual.* Fourth edition, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass., 1981.

59. J. E. White. A User-friendly Naming Convention for Use in Communication Networks. Computer Message Services, IFIPWG6.5, Nottingham, England, May, 1984, pp. 37-57. Proc.IFIP6.4 Working Conference hosted by Nottingham University and Plessey

60. M. V. Wilkes and R. M. Needham. *The Computer Science Library: Operating and Programming Systems. Vol. 6: The Cambirdge CAP Computer and Its Operating System.* North Holland, New York, 1979.

Appendix A

Operations in the General Model

The operations here are in a Clu-like [30] form, in which the name of the operation is followed by the names and types of all arguments, the keyword **returns**, and the types of the returned values. Although signals would also normally be included in a Clu specification, they have been omitted here for simplicity.

A.1 Operations on Contexts

Operations for managing contexts

```
create = proc returns (cvt)
merge_names = proc (context1, context2: cvt)
merge_participants = proc (context1, context2: cvt)
copy = proc (context1: cvt) returns (cvt)
display = proc (context1: cvt)
```

Operations for managing names in a context

```
translate = proc (context1: cvt, name: string) returns (set[any])
untranslate = proc (context1: cvt, object: any) returns (set[names])
add_name = proc (context1: cvt, name: string, object:any)
reserve_name = proc (context1: cvt, name: string)
assign_object_to_reserved_name = proc (context1: cvt,
 reserved_name: string, object: any)
delete_name = proc (context1: cvt, name: string)
delete_entry = proc (context1: cvt, name: string, object: any)
```

Operations on participants sharing a context

```
add_participant = proc (context1: cvt, participant_name: string)
delete_participant = proc (context1: cvt, participant_name: string)
get_participants = proc (context1: cvt) returns array[string]
```

A.2 Operations on Aggregates

Operations for managing aggregates

```
create = proc returns (cvt)
create_with_context = proc (context1: context) returns (cvt)
merge_current_contexts = proc (aggregate1, aggregate2: cvt)
copy_current_context = proc (aggregate1, aggregate2: cvt)
merge_environments = proc (aggregate1, aggregate2: cvt)
copy_environment = proc (aggregate1, aggregate2: cvt)
display = proc (aggregate1: cvt)
```

Operations for name management in the current context

```
translate = proc (aggregate1: cvt, name: string) returns (set[any])
untranslate = proc (aggregate1: cvt, object) returns (set[string]
add_name = proc (aggregate1: cvt, name: string, object: any)
reserve_name = proc (aggregate1: cvt, name: string)
assign_object_to_reserved_name = proc (aggregate1: cvt, reserved_name: string,
 object: any)
delete_name = proc (aggregate1: cvt, name: string)
delete_entry = proc (aggregate1: cvt, name: string, object: any)
get_current_context = proc (aggregate1: cvt) returns (context)
```

Operations for managing participant names

```
add_participant = proc (aggregate1: cvt, participant_name: string)
delete_participant = proc (aggregate1: cvt, participant_name: string)
get_participants = proc (aggregate1: cvt) returns (set[string])
```

Operations for managing the environment of an aggregate

```
insert_rule = proc (aggregate1: cvt, rule: int, context1: context)
append_rule = proc (aggregate1: cvt, context1: context)
add_to_rule = proc (aggregate1: cvt, rule: int, context: context)
move_context_to_rule = proc (aggregate1: cvt, context1: context)
delete_from_rule = proc (aggregate1:cvt, rule: int, context1: context)
delete_rule = proc (aggregate1: cvt, rule: int)
get_environment = proc (aggregate1: cvt) returns (array[set[context]])
```

Operation for setting working aggregate

```
set_working_aggregate = proc (aggregate_name: string)
```

Appendix B

Operations in the Mail Implementations

The operations in the user interface are functions in MockLisp [14]. Those functions listed in the user interface that are followed by an asterisk (*) are invoked directly by humans, whereas the others are only used indirectly. The operations supporting contexts and aggregates are in a Clu-like [30] form as in Appendix A. In this case, the signals have been included since they are in the code, and the text was taken directly from the code currently in use.

B.1 Functions in User Interface

New functions in the user interface

Name of function	Comment
list-aggregates*	lists names of all aggregates
list-contexts*	lists names of all contexts
display-aggregate*	displays an aggregate, defaults to basic_a
display-context*	displays a context, defaults to basic_c
display-environment*	displays an environment, defaults to basic_a
new-aggregate*	creates a new aggregate
set-current-context*	given an aggregate name, sets current context to named context
set-environment*	sets environment of one aggregate equal to the environment of a second
append-to-current-context*	appends the contents of a context to the current context
expunge-aggregate*	expunges all names deleted from current context
add-name*	adds a specific entry to current context
delete-entry*	deletes a specific entry from current context
delete-name*	deletes all entries with given name from current context
change-status*	changes state of an entry in the current context
expunge-context*	expunges all names deleted from context

move-context*	prompts for rule # of new location of context in environment
add-to-rule*	adds context to rule
delete-from-rule*	deletes context from rule
add-rule*	creates a new rule
add-aggregate*	adds an aggregate field to a message - this is the only new operation that modifies the .mailbox file
read-names	only used indirectly when reading a message to translate names
send-names	only used indirectly when sending a message to translate names
mail-help*	displays this information

Functions modified in the user interface to the mail system

Name of function	Comments
display-message	used in displaying a message
quit*	exit mailer
start-edit*	begins mailer in send mode, stand-alone
send-mail*	begins mailer in send mode from within emacs
init-mail	used both stand-alone and within emacs to initialize mail file
mail-mode	sets definitions for using emacs in mail mode
load-mail	loads mail from file into a large buffer
next-message-nd*	goes to next undeleted message
previous-message-nd*	goes to previous undeleted message
edit-mail*	enters buffer to create new message to send, from reading
forward-mail*	forwards the current message
reply*	replies to current message
send-message*	sends a message, forwarded message, or reply

B.2 Operations on Aggregates in the Mail System

Operations for aggregate management

```
create = proc (new_aname, new_ccname: string)
  returns (cvt)
create_with = proc (new_name: string, curcont: context)
```

```
  returns (cvt)
equal = proc (aggregate1, aggregate2: cvt) returns (bool)
merge_new_cc = proc (aggregate1, aggregate2: cvt, new_ccname: string)
copy = proc (new_aname, new_ccname: string, aggregate1:cvt)
  returns (aggregate)
append_to_current_context = proc (aggregate1: cvt, context1: context)
set_current_context = proc (aggregate1: cvt, current_context: context)
get_current_context = proc (aggregate1: cvt) returns (context)
get_my_name = proc (aggregate1: cvt) returns (string)
_gcd = proc (x: cvt, tab: gcd_tab) returns (int)
```

Operations for name management

```
translate = iter (aggregate1: cvt, label: string, add_data: int, cond:
    condtype) yields (string, int, bool) signals (no_such_name)
untranslate = iter (aggregate1: cvt, obj: string, add_data: int, cond:
    condtype) yields (string, int, bool) signals (no_such_name)
add_name = proc (aggregate1: cvt, new_name, transformation: string,
    add_data: int) returns (bool)
delete_name = proc (aggregate1: cvt, delname: string, del_data: int)
  returns (bool)
delete_entry = proc (aggregate1: cvt, delname, deltranslation: string,
    del_data: int) returns (bool)
entry_status = proc ( aggregate1: cvt, name1, obj1: string) returns
    (int)
force_state = proc (aggregate1: cvt, curr_name, curr_transl: string,
    curr_state: state)
```

Operations for environment management

```
append_to_environment = proc (aggregate1, aggregate2: cvt) signals
    (duplicate_id)
add_to_rule = proc (aggregate1: cvt, prior: int, label1: string,
    context1: context) signals (no_such_rule, already_used)
delete_from_rule = proc (aggregate1: cvt, label: string)
add_rule = proc (aggregate1: cvt, at_rule: int, label: string,
    context1: context) signals (out_of_bounds, already_used)
delete_rule = proc (aggregate1: cvt, del_rule: int) signals
    (out_of_bounds)
list_environment = proc (aggregate1: cvt) returns (as)
move_rule = proc (aggregate1: cvt, i, j: int) signals (out_of_bounds)
```

B.3 Operations on Contexts in the Mail System

Operations for context management

```
create = proc (cname: string)
equal = proc (context1, context2: cvt) returns (bool)
copy = proc (old_context: cvt, new_name: string) returns (cvt)
append = proc (context1: context, context2: cvt)
_gcd = proc (x: cvt, tab: gcd_tab) returns (int)
disp_list = iter (context1: cvt) yields (string)
get_name = proc (context1: cvt) returns (string)
merge = proc (context1, context2: cvt, new_name: string) returns (context)
get_ctext = proc (context1:cvt) returns (at)
get_my_name = proc (context1: context) returns (string)
expunge = proc (context1: cvt)
```

Operations for name management

```
accept = proc (context1: cvt, new_name, new_translation: string,
    add_data: int) returns (bool)
delete_name = proc (context1: context, delname: string, del_data: int)
    returns (bool)
delete = proc (context1: cvt, del_name, del_translation: string,
    del_data: int) returns (bool)
translate = iter (context1: cvt, label: string, add_data: int, cond: condtype)
  yields (string, int, bool)
untranslate = iter (context1:cvt, obj: string, add_data: int, cond: condtype)
  yields (string, int, bool)
names = iter (context1: cvt) yields (string, state)
force_state = proc (context1: cvt, curr_name, curr_transl: string,
    curr_state: state)
entry_status = proc (context1: cvt, name1, obj1: string) returns (int)
```

Appendix C

Operations in the Programming Support Environment

C.1 Operations on Contexts and Aggregates

Both contexts and aggregates are parameterized by procedures. This is not standard Clu syntax, but it has been done in the style of Clu syntax. The parameterization has been specified in two equates on the names of the clusters in order to simplify reading.

Operations on Contexts

Equate for context type

```
contexta = context[cmerge: proc (context1, context2: cvt) returns (cvt),
 acc, del: proc (context1: cvt, name: string, obj, state_data: any)]
```

All operations here are in the contexta cluster.

```
create = proc (merge_option: oneof[
      "context1 has priority, although context2 used also",
      "context2 has priority, although context1 used also",
      "only context1 used",
      programmer_supplied_proc: proc (context1, context2: cvt) returns (cvt)],
   acc, del: proc (context1: cvt, name: string, obj, state_date: any))
 returns (cvt)
equal = proc (context1, context2: cvt) returns (bool)
copy = proc (context1: cvt) returns (cvt)
display = iter (context1: cvt) yields (string)
merge = proc (context1, context2: cvt) returns (cvt)
translate = iter (context1: cvt, name: string, state_data: any) yields (any)
untranslate = iter (context1: cvt, obj, state_data: any) yields (string)
add_name = proc (context1: cvt, name: string, obj, state_data: any)
reserve_name = proc (context1: cvt, name: string, state_data: any)
```

```
add_reserved_name = proc (context1: cvt, previously_reserved_name, new_name:
 string, state_data: any)
assign_obj_to_reserved_name = proc (context1: cvt, reserved_name: string,
 obj, state_data: any)
delete_entry = proc (context1: cvt, name: string, obj, state_data: any)
delete_name = proc (context1: cvt, name: string, state_data: any)
expunge = proc (context1: cvt, state_data: any)
get_status = proc (context1: cvt, name: string, obj: any) returns (string)
add_participant = proc (context1: cvt, participant_name: string)
delete_participant = proc (context1: cvt, participant_name: string)
get_participants = proc (context1: cvt) returns (array[string])
```

Operations on Aggregates

Equate for aggregate type

```
aggregatea = aggregate[amerge: proc (agg1, agg2: cvt) returns (cvt), acc,
 del: proc (agg1: cvt, name: string, obj, state_data: any)]
```

All operations here are in the aggregatea cluster

```
create = proc (ccmerge_option: oneof[
      "context1 has priority, although context2 used also",
      "context2 has priority, although context1 used also",
      "only context1 used",
      "only context2 used",
      "context1 to new cc, context2 first rule in new environment",
      "context2 to new cc, context1 first rule in new environment",
      programmer_supplied_ccmerge: proc (agg1, agg2, agg3: cvt, state_data:
      any) returns (cvt)],
    envmerge_option: oneof[
      "env1 has priority, env2 in succeeding rules",
      "env2 has priority, env1 in succeeding rules",
      "env1 only",
      "env2 only",
      "merge two rule by rule",
      programmer_supplied_envmerge: proc (agg1, agg2, agg3: cvt) returns (cvt)],
    acc, del: (agg1: cvt, name: string, obj, state_data: any)) returns (cvt)
set_current_context_to = proc (agg1, agg2: cvt)
copy_current_context = proc (agg1, agg2: cvt)
```

```
merge_current_contexts = proc (agg1, agg2, agg3: cvt, state_data: any)
  returns (cvt)
copy_environment = proc (agg1, agg2: cvt)
append_env = proc (agg1, agg2: cvt)
merge_environments = proc (agg1, agg2, agg3: cvt, state_data: any)
  returns (cvt)
copy = proc (agg1: cvt) returns (cvt)
display = proc (agg1: cvt) yields (string)
translate = iter (agg1: cvt, name: string, state_data: any) yields (any)
untranslate = iter (agg1: cvt, obj, state_date: any) yields (any)
add_name = proc (agg1: cvt, name: string, obj, state_data: any)
reserve_name = proc (agg1, cvt, name: string, state_data: any)
add_reserved_name = proc (agg1: cvt, previously_reserved_name, new_name:
  string, state_data: any)
assign_obj_to_reserved_name = proc (agg1: cvt, reserved_name: string, obj,
  state_data: any)
delete_entry = proc (agg1: cvt, name: string, obj, state_data: any)
delete_name = proc (agg1: cvt, name: string, state_data: any)
expunge = proc (agg1: cvt, state_data: any)
get_status = proc (agg1: cvt, name_ string, obj: any) returns (string)
add_particioant = proc (agg1: cvt, participant_name: string)
delete_participant = proc (context1: cvt, participant_name: string)
get_participants = proc (agg1: cvt) returns (array[string])
add_rule = proc (agg1: cvt, rule: int, context1: contexta)
append_rule = proc (agg1: cvt, context1: contexta)
add_to_rule = proc (agg1: cvt, rule: int, context1: contexta)
move_rule = proc (agg1: cvt, old_rule, new_rule: int)
delete_from_rule = proc (agg1: cvt, rule: int, context1: contexta)
delete_rule = proc (agg1: cvt, rule: int)
get_environment = proc (agg1: cvt) returns (array[string])
```

C.2 Operations on Library Contexts

The library_context type (or type generator) will have all the context operations of Appendix C.1 as well as these few others. As with the context type generator, library_context is a types generator, also parameterized by the same procedures as context.

```
set_required_name = proc (library_context1: cvt, name: string, t: type)
set_optional_name = proc (library_context1: cvt, name: string, t: type)
move_library_reference = proc (old_library: cvt, old_name: string,
  object: any, new_library: cvt, new_name: string)
update_indirect_library_references = proc (library_context1: cvt)
```

C.3 Operations on Template Aggregates

These are the additional operations needed for template aggregates, beyond those listed for aggregates in Appendix C.1. There is one difference here. The standard create operation of aggregates will not be transferred to the template_aggregate type generator. Instead, a separate create operation has been included here, creating a template_aggregate from a pre-existing aggregate.

```
create = proc (aggregate1: aggregate) returns (cvt)
merge = proc (template_aggregate1: cvt, client_aggregate: aggregate) returns
  (aggregate)
shared_current_context = proc (template_aggregate1: cvt, "shared" |
  "not_shared")
```