1988

NAMER: A Distributed name server for a connected UNIX Environment

Thomas Tatakis

Rochester Institute of Technology
School of Computer Science and Technology

NAMER

A Distributed Name Server for a Connected UNIX[tm] Environment

by
Thomas J. Tatakis
December 20, 1988

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Peter Lutz 12-20-88
Date

James Heliotis 12-20-88
Date

Peter Anderson 20 Dec 88
Date

TABLE OF CONTENTS

ABSTRACT

A naming or identification scheme is crucial to the design of any computer system. One must be able to access and perform operations on resources and data in order to complete a given task. In a distributed system the problem is compounded by the geographical separation of resources. Moreover, the underlying network of computers should be transparent to the users. A user should simply be able to request that an operation be performed and the system should locate the necessary resources. This function of locating a named resource is performed by a special service called a name service. This thesis investigates and develops a name service that provides such capability.

ACKNOWLEDGEMENTS

I would first like to thank Dr. Peter Luts for his help in determining a valid thesis topic. I thank both Dr. Peter Luts and Dr. Jim Heliotis for their time and help during the proposal development and topic definition phase. Last, but certainly not least, thanks Julie for putting up with the long hours, days, and weeks apart during my graduate career.

CHAPTER 1

Introduction and Background

1.1. Problem Statement

One of the major issues involved in distributed computing is that of naming resources. For example, consider a system of personal computers, networked together on a local area network, where each machine is dedicated to a single user at any given time. Clearly, it is expensive to have a printer connected to each personal computer, dedicated to that machine as well as the user. The alternative is to have one, or a few, printers connected to the network so that any user can request the use of a printer when one is needed. A problem that arises in this situation is that each user must know to what machine the printers are connected; information the user should not be required to know explicitly. A user should be able to simply request that a file be printed, and the system should locate the printer and perform the desired operation. Of course this *print service* is only one example. Others include a *mail service*, *file service* and many other such services that need not be connected to every machine in a network. When a user requests that a function be performed, the system should locate the *service* that implements that function and perform the requested operation. This function of locating a named resource is performed by a special kind of service called a *name service*. This thesis investigates and develops a name service that provides this capability.

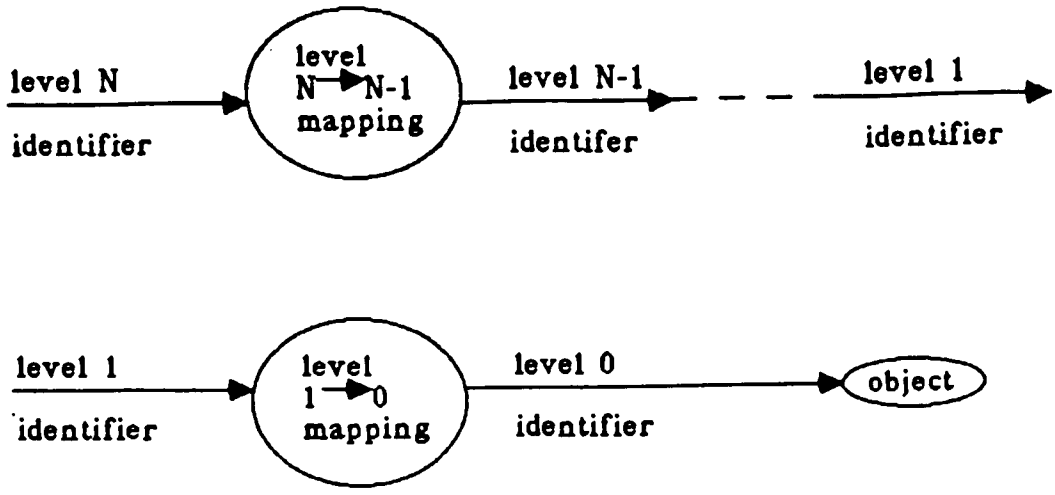


Figure 1: Mapping through system levels.

A naming system, or identification system, is crucial to the design of any computer system. One must be able to access and perform operations on resources and data in order to perform a desired function. Even an operation as trivial as reading input from a terminal and echoing it back must, at a minimum, be able to access the terminal and perform read and write operations on the terminal. In order to access such resources, a name for those resources must exist. A name can be a string of characters, a number, or some unique bit pattern; the specific representation depends on how the name is to be interpreted. We, as humans, tend to favor meaningful character strings, while computers prefer more compact bit strings.

All (distributed) operating systems support some kinds of data objects, such as files and I/O devices. *Naming* or *identification* can be viewed as the problem of assigning names to these objects and providing a means of mapping from the name to the object it represents [4], [18]. See Figure 1. In general, every computer system is divided into different levels. A high level programming language, the operating system, and the message delivery or communication sys-

tem are examples of some of the layers found in a typical system. Names or identifiers exist, although usually in different forms, in all levels of a system. For example, in a high level programming language names usually exist as character strings while in a operating system, an address may be used to represent the same identifier. The job of the naming system then, is to interpret these names and perform a mapping from one layer to another and ultimately to the object itself. Although outside the immediate area of concern, the telephone system provides a good example[1].

When we wish to place a call to another person, two things must happen. First we must find the number, then we dial the number. The second step, actually placing the call and making the connection is certainly more mechanical and predictable. In fact, we care little about the steps taken to make the connection. We simply rely on the communication system to take care of these details. The first step, however, can create a problem. Given a person's name, we must find his or her telephone number. If we already know the number, this is a trivial step. If we do not know it, we must consult the telephone directory. If the person lives locally, we 'present' the name to the white pages and search until we find it. Once the name is found, we perform the mapping function by moving across the page to the corresponding number. If the person does not live locally, we call the information operator in the desired city. Once reached, we begin an initial dialogue indicating our intent and then 'present' the name. The operator then returns the correct number (if any) to be used in the dialing step.

In short, the telephone directory provides a way of naming an object, the owner of the telephone, and a way of mapping from the name to the object. The telephone system prefers numbers, but its clients prefer strings of characters.

---

[1] The telephone system is not a completely correct model. It is shown later to break down under the transparency issue.

1.2. Naming: Theory and Concepts

In the most general terms, naming in a distributed system consists of the symbol alphabet out of which one or more levels of identifiers can be derived, the rules for constructing identifiers, and a mapping mechanism to map identifiers from one level to another [18]. An *identifier* in a distributed system is a string of symbols from the alphabet. These identifiers can be of many different forms: strings of bits, strings of numbers, and strings of characters are examples of different forms of identifiers. The form of the identifier depends on the level of the system, or the *context* in which the name is to be interpreted. In the telephone system example above, identifiers are strings of characters in the user context and numbers in the telephone system context. A distinction has been made between three important classes of identifiers [14]: A *name* indicates what we seek, an *address* indicates where it is, and a *route* tells us how to get there. A name is a client readable string of alphabet symbols. Not all users, however, may recognize the name. An address, on the other hand, is a data structure whose format can be recognized as an address by all elements in the computing system (i.e. system wide unique). Finally, a route is the necessary information needed to forward a piece of information to a specified address. So, to get from a name to the object it specifies, the identification system must: 1) map the name to an address, 2) map the address to a route, 3) send the request to the specified address over the route. The last two steps, of course, are performed by the communications system.

1.3. Goals of the Identification System

There are many possible goals for an identification scheme in a distributed computing system, each of which may depend on entirely different issues such as network considerations (e.g. broadcast or point-to-point) or the kinds of objects being named. Below are some of the goals that should be met by any identification scheme in a distributed system [11], [14], [18].

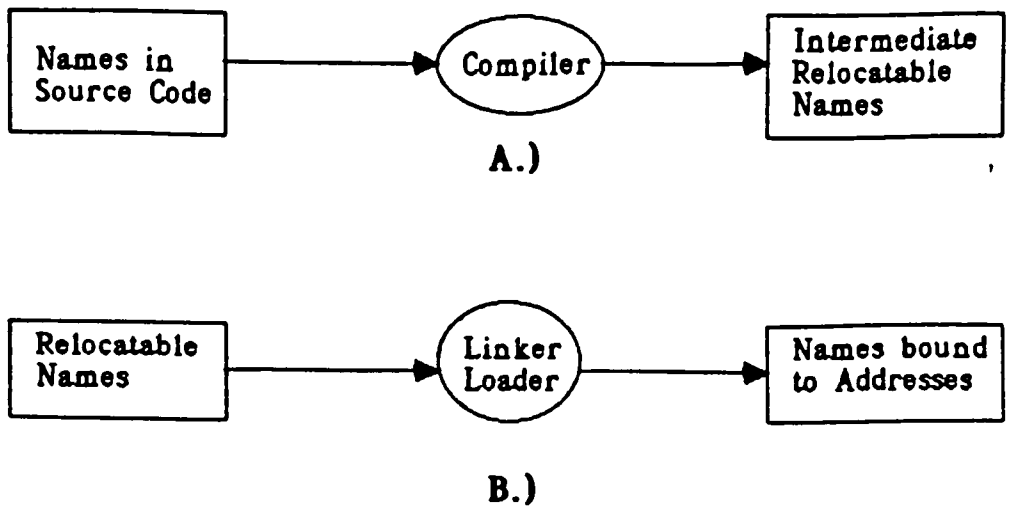


Figure 2. The programming language/operating system mappings.

Perhaps the most important goal to be met by the identification system is the support of *dynamic binding*. That is, do not bind a name to the address of the object it represents until that object is used. In this way, improvements can be made in the ability to relocate, extend, or share objects. Consider the familiar programming/operating system example in figure 2. At the highest level, the programming language allows a user to create names for objects, such as procedures, data structures, or other programs, that are useful to people. A compiler then translates these names into some intermediate machine oriented form. These are then loaded into memory over some logical address space much larger than physical memory. Finally, these logical addresses are dynamically bound, at runtime, to physical addresses in the context of the base registers and page tables. This allows portions of physical memory to be copied out and re-loaded at some other location in memory. If this were not the case, clearly, the concepts of page swapping and multiprogramming would be all but impossible to realise. Consider a situa-

tion in which programs are stored as absolute rather than relocatable images (i.e. address binding occurred at translation time). In this case, each time a program is executed, it must be loaded in exactly the same place in physical memory. Moreover, if two programs, **P1** and **P2**, have address spaces that overlap in any way, then both **P1** and **P2** will not be able to run simultaneously. This is certainly not an efficient way to implement multiprogramming.

The point of the above is that by postponing the binding between name and address, the ability to relocate, and share objects is increased. That is, dynamic binding enhances object relocatability or migration and facilitates the sharing of objects. This dynamic binding implies that at least two levels of identifiers are needed (recall figure 1.). For this second goal, there should be at least a level of identifier that is convenient for people, and a machine oriented identifier easily manipulated by a computer. The human oriented identifier should be a meaningful character string. That is, some word or phrase that helps indicate what kind of object is being identified. The machine oriented identifier is best represented as a more compact bit string. A string of 8 bits, 16 bits, or some multiple of the word size of the machines involved will work best since instructions to manipulate items of this size are usually the most predominant and efficient of those in the instruction set. For example, the higher level name (i.e. path name) of a file in UNIX is of little interest to the file server; only the inode number is required to locate the correct file.

A third goal of the identification system is to provide a global system view rather than the view of a network of computers each with its own objects. One of the key concepts involved in designing a distributed system is that of transparency [16]. That is, the fact that there is a network of multiple processors should be invisible to the user. This implies that the identifiers used to name objects must not be required to contain, at the user level, the location of the machine where the object is stored. This also enhances object migration since if an object moves, the user need not change how the object is to be referenced. The same high level name locates the

migrated object. For example, if a user wishes to delete a file from his directory, entering

delete(myfile)

should be sufficient regardless of where the file is stored or where the file server is located. Unfortunately, the boundaries of the network are real and must exist at some level of the identification system. If this were not the case, we would never be able to find anything. If these network boundaries are to be incorporated into the identifiers, it should be within some identifier structure such as a hierarchical form. It is not intended that a user cannot determine or force an object's location. For example, a user may wish to run his graphics program on machine **A** rather than machine **B** because machine **A** has better floating point capability.

A related goal to the above is that of allowing unique identifiers to be generated in a distributed manner. A single, central name generator compromises the reliability of the system. If the name generator crashes, names can no longer be formed unless some provision is made to start a new name generator. The overhead incurred in keeping track of the necessary information to start a new name generator can become costly. In order to support distributed generation of unique identifiers, however, a structured identifier scheme is needed. As mentioned above, this structure is not required to be visible to the user.

A fifth goal is to support a group of logically equivalent managers for a particular resource type. This implies that a single identifier can be bound to more than one identifier (address) at a lower level. The mapping is based on some set of attributes for that particular resource type. For example, suppose there is a group of printers of the same class (e.g. same speed and quality). A request for a printout can then be sent to any one of these printers based on current usage. If printer **P** in the group has a large number of pending requests, then the next request can be sent to printer **Q**.

The above list of goals is certainly not a comprehensive list. There are many other goals that may or may not be necessary to achieve. Many of these can be simple extensions to sup-

port a given user community (e.g. aliasing). Still others may depend on network considerations (e.g. allowing many objects to share the same identifier to support broadcasting). The above list does provide a good base from which to build, and encompasses issues that all identification systems in distributed computing environments should address. Also note that each of these goals can be achieved at different identifier levels or across levels. For example, providing a global view is best solved by providing a means for the user to specify objects as if there was only one system. Allowing for human and machine oriented identifiers, however, must be handled by at least two different levels.

CHAPTER 2

A NAME SERVICE DESIGN

Before designing any type of system, it is necessary to determine, and define the problem that needs to be solved. Hopefully, this was accomplished in the preceding sections. Next, a suitable system design model on which to base decisions must be developed. The design model should achieve as many of the requirements and goals of the specification as possible. From the model then, the system can be decomposed into successive layers until a final solution is reached. Indeed, this is simply the structured programming methodology and is how the naming system is developed in this paper. The model is followed by a system design, software design, and finally an implementation in following sections.

This next section of the paper, however, presents a model and discusses how the goals presented in section 1.3. are achieved. In addition, some fundamental questions about a name service (e.g. what it names, how it names) are discussed. This is followed by a system level design based on the goals outlined previously and the following considerations.

2.1. A Naming System Model

Recall that one of the key concepts in designing a distributed system is that of transparency (section 1.3.). That is, the user of such a system should not be required to know where objects maintained by the system are located. In addition, the structure of the underlying network on which the distributed system is 'running' should also be transparent to the user. The only knowledge a user needs is the fact that the system will indeed perform a desired operation on a specific object regardless of where the object and user are located or how the system locates the object.

Given this requirement, the telephone system example in section 1.1. has a severe limitation: the user must know if the person to whom the call is to be placed lives locally or in another city. For example, if the intended receiver of the call lives locally, then only the local exchange and extension need be dialed. If the person to be called does not live locally, then a 1 must be dialed, followed by an area code, and finally by the local exchange and extension. This is not necessarily a design deficiency, but simply a result of the hierarchical name in which the telephone numbers exist. One way around this non-transparency of location would be to implement a flat name space similar to the way social security numbers are used. That is, assign every phone a unique 9 or 10 digit number. Then no matter where the caller and callee are currently located, only the assigned number need be dialed; no area code or other qualifying numbers need be used as a prefix. This implies that every phone be directly reachable from every other phone as opposed to being switched from station to station as each 'field' of the number is parsed in the current implementation. The maintenance problems of this scheme should be clear. When a new number is introduced, then every other phone in the network must be made aware of this new available connection. In addition, a central name authority must be used to insure that new numbers are not duplicates; a contradiction to the distributed name generation requirement.

Another problem within the phone system is that if the owner of the telephone moves (outside of the current area code), that person must be referenced by a new phone number. That is, the same identifier can not be used to locate the original object and migration transparency is lost. This problem results from the fact that phone numbers are assigned to phone lines rather than the owner of the phone.

A third limitation, related to the location transparency issue is that there is no notion of forwarding. For example, each time a call is placed, a specific person (or phone) is the intended receiver of the call. If we place a (local) call to Tom Brown and that particular phone is out of

order, we usually do not wish to speak to another Tom Brown. There is no forwarding or searching for an equivalent object. In contrast, within a distributed computing environment it may not always be necessary to contact a specific entity to perform a task. It may be acceptable to contact any 'service' that is capable of performing a desired function (more detail is given in the following section). Consequently, in a situation in which the first choice is not capable of performing the task, the request may be forwarded to some other object that is functional.

The above paragraphs discuss the limitations in the telephone system as a model for a distributed computing environment. First, the telephone system is not fully transparent. Although the underlying network is not visible to the caller, the caller must still know where the person to be called lives. Second, each time the same number is dialed, the same phone at the receiving end is contacted making migration non-transparent. Third, there is no mechanism, nor should there be, to forward a call if the initial receiver is incapable of accepting the call. In addition, if a call is placed to a phone that is already in use, the second call is never completed. As mentioned earlier, the latter two limitations are not necessarily deficiencies within the phone system, but simply inherent in the way the telephone system was designed. Within a distributed computing environment however, the restrictions in the telephone system are not acceptable; it simply does not meet the intended goals. Clearly then, a more robust naming model is needed. In the next section I present such a model called an *Information Booth*.

2.1.1. Naming: An Information Booth

Consider a shopping mall containing 100 stores of similar type (e.g. all sporting goods stores, or all stores within some geographical area). Suppose that there are 100 of these malls connected by a 'network' of roads. Outside each mall is an information booth that knows the location of each store within its associated mall. In addition, a given information booth may know the location of one or more other information booths. When a shopper wishes to purchase

an item, say, a baseball bat, he first visits the local information booth and requests to shop at a sporting goods store. The information booth locates a sporting goods store within its associated mall and directs the shopper to the store. It should be clear that any sporting goods store will do as long as the shopper does not care to compare prices. If the shopper wishes to shop at a specific store, he can request to be directed to any store he chooses (e.g. Where is the sporting goods store called Herman's). In addition, if all of the 'local' sporting goods stores are closed, or there is an overload of shoppers at the sporting goods stores within the local mall, the shopper can be directed to another information booth. This second information booth can then search for an open sporting goods store within its mall and direct the shopper appropriately.

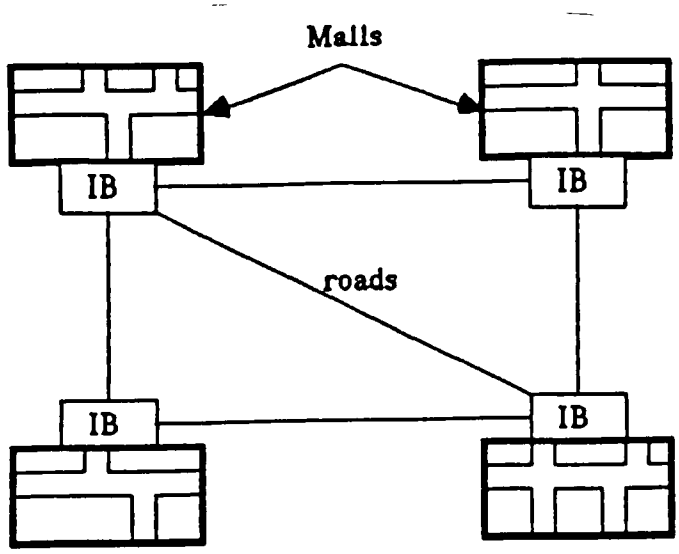


Figure 3: A Network Of Shopping Malls

Figure 3 shows a network of shopping malls. Note that it is actually the information booths (IB) that are connected by the roads, not the malls. Whenever a customer wishes to shop, he must first visit an information booth to locate an appropriate store. Only after the information booth locates this store does the shopper move to the store. Since the shopper is

directed to the correct store from the information booth, there is no need for the malls to be explicitly connected. In addition, the set of stores managed by one information booth may overlap a set of stores managed by another information booth.

2.1.2. Meeting The Goals

Above it was pointed out that the telephone directory was an inadequate model for a naming system within a distributed computing environment. Hence the need for the more robust model, the Information Booth. Before a design can be presented, however, it must be clear that each of the goals outlined in section 1.3. are satisfied. Only then will it be possible to develop a correct design, based on the model, that meets the intended goals.

First, the transparency issue. The shopper does not need to know, nor care, where a particular store is located or how it is found. The shopper is only concerned with knowing that there is indeed a store that can fulfill his request. However, if the shopper desires the use of a particular store, he may request any that he chooses. In addition, the shopper is not given directions to a store until he requests them (dynamic binding). The fourth goal discussed in section 1.3. is to allow identifiers (names) to be generated in a distributed manner. If a new mall is built, a new information booth can be added to the network making it available to the other information booths. The fifth goal is to allow a single identifier to be bound to more than one object. For example, if a particular information booth receives a sudden rush of requests for clothing stores, the requests can be spread among a number of stores, or even other information booths, to even the load. The last goal presented in section 1.3. is to support at least two levels of identifiers. The two levels of identifiers within the information booth model may not be immediately obvious, but they do exist. The first (higher) level of identifier is the identifier used by the shopper (e.g. sporting goods store). The lower level of identifier is the identifier to which 'sporting goods store' is mapped. This may consist of a hall number and a store number on that hall.

Unlike the telephone naming model, the information booth model satisfies all of the goals of a naming system for a distributed computing environment. While this list of goals, as mentioned earlier, is not exhaustive, it does provide a suitable foundation from which to build. Next, the model must be implemented for a particular systems need.

2.2. System Design: Implementing the Information Booth

Now that a suitable model has been developed, the first question that needs to be asked, and answered is: What will the naming system name? The answer to this question may seem trivial at first, but as we shall see, it may depend significantly on the design of other parts of

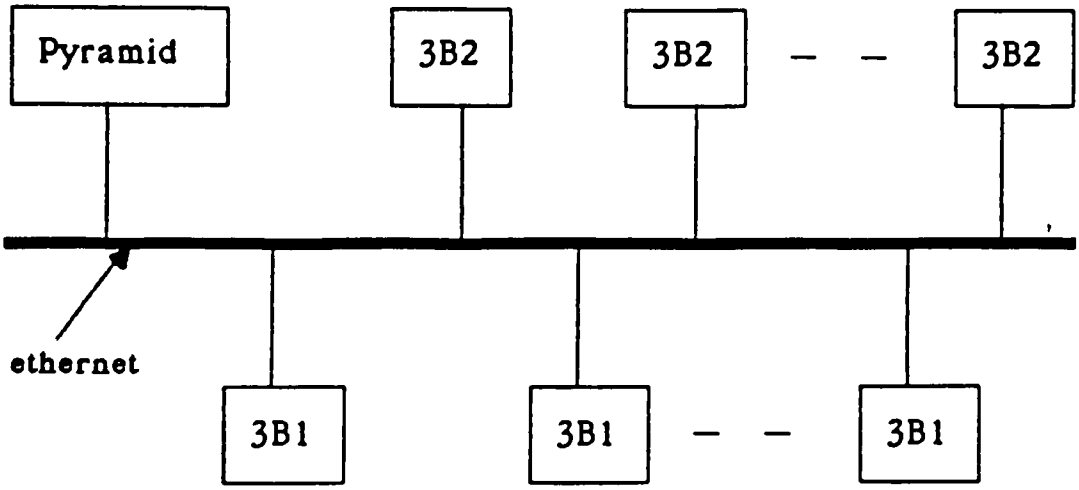


Figure 4: The Network.

the system. For example, in the system discussed in the previous section, the naming system named stores. The naming system researched and developed for this thesis is part of a distributed computing environment. As a result, the objects of the naming system will, of course, depend on the design and configuration of the particular distributed computing environment for which it is to be implemented.

2.2.1. The Underlying System Architecture

The hardware architecture consists of a network of machines each running some variant of the UNIX[1] operating system. In particular, it consists of a number of AT&T 3B1's, 3B2's and a Pyramid 90X mini computer connected over a single Ethernet. See Figure 4. Because each of these machines will already be running an operating system, many of the low level details of naming are already implemented. For example, the file system can already access a file given its proper name. Recall that given such an architecture, it does not make sense to have a printer, or print service connected to, and dedicated to each machine. Similarly, it is impractical to have other such services connected to or running on every machine. For example, a mail service, file service, or news service, as well as others can be implemented on only one, or a few machines in the network. Given this architecture and state of the system then, it makes sense for the name service to name services.

Suppose for example, a user requests the print service to send some file to a line printer by entering the command

>lpr myfile

The name resolution function would first map this string to a command offered by the print service. Next this request for the print service would be mapped to an appropriate print server. That is, the request is mapped to some node on the network and some process implementing the print service on that node. Note that this scenario demands that the print server 'knows' how to find the file myfile. Two ways this can be done is by having the print service on the same machine as the file service (or conventional UNIX file system), or by having the print service use the name service to locate the file service that manages myfile. The latter case is clearly more distributed, but in either case it is assumed that the print service will locate the correct file given only the name supplied by the user.

Note that in some situations it may be acceptable to map a user's request to any service capable of performing the desired function. This can be done only if the semantics of the operation are identical no matter which service implementing the request is contacted. For example, consider a distributed file system managed by multiple file servers. Each file server is capable of operating on the entire file structure. If a user enters a command to modify a file, any file server will suffice. That is, the same file will be modified regardless of which file server is chosen. However, it will be more efficient to choose a file server that is inactive or lightly loaded rather than to send the request to a heavily loaded server. In addition, allowing the name server to chose any appropriate service, the amount of work allotted to each service can be controlled. No one service will receive more work than another and because of the distribution, overall throughput and response time will be increased.

There are situations, however, in which allowing the name service to choose any server is not acceptable. Consider a situation in which a user requests to run a graphics program that requires a floating point processor. The name service must then choose a machine, or perhaps a process server, that includes the appropriate floating point capability. If an appropriate server is not chosen, the graphics program may be unacceptably slow, or worse, it may not run at all. Clearly then, to be consistent with the model, the name service must be capable of making mapping decisions in two different modes. The name server must **1)** be able to choose any server that is appropriate, and **2)** allow the user to specify an explicit object to be used. The details of how this is accomplished is discussed shortly.

2.2.2. The Name Space

Although the naming system will name services, and at this point only services, designing in a flat name space will make it difficult to enlarge the name space if it is ever desired. For this reason, I have implemented a hierarchical name space similar to the ones found in BIND[3] and

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

the Clearinghouse[11]. The name space is arranged as a tree based on some logical organizational structure. Each node in the tree is either a domain or an object. A *domain* may contain other domains as well as objects. An *object* is a leaf node that contains no other domains or objects. That is, an object is the entity that the name service is naming. To reference nodes, each domain and object is given a *label* that must be unique within its domain. The name of a node is the left to right concatenation of all the labels from the root node to the current node

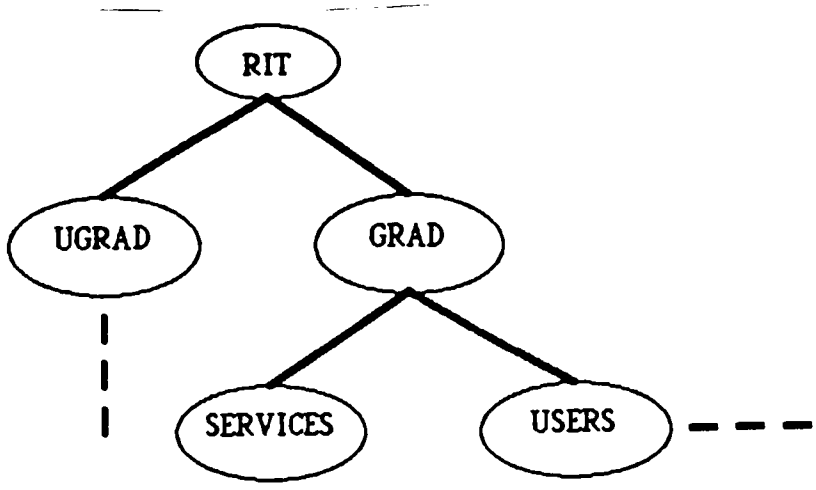


Figure 5: The Name Space.

separated by '/' (a forward slash). The name of a node then takes the form of a path name in UNIX. For example, the name of the services node within the grad domain is given by the *path name*

grad/services

The services node (as named above) may either be a domain or an object. If it has child nodes, then it must be a domain, otherwise it is an object. An example of the name space is shown in Figure 5. Note that RIT is the root and the two subtrees that extend from it are UGRAD and GRAD. This is an arbitrary structure and is based on the logical division found at RIT. The remainder of the name space discussion pertains to the GRAD subtree. It should be noted,

however, that the design can easily be extended to include the **UGRAD** subtree, or any additional subtrees as required. Within the **GRAD** subtree, it is only possible to name services at this time. As a result, all names managed by the name service begin with the string **RIT/GRAD/SERVICES**. At some other point in time, it may be necessary to extend the name space at this level, to include users, mail boxes, or other objects. All that would be required is to register the name with the name service. When the new object is referenced, the mapping function will traverse the hierarchy to locate the desired object. In addition, the hierarchy could also be extended downward to increase the number of levels beyond four.

2.2.3. Distributing the Name Space

Within the framework of the model, the names managed by the information booths are distributed throughout the network. This is accomplished by having multiple information booths each managing some subset of names over the name space. If a search for a name in one information booth is unsuccessful, the request can be directed to another information booth. If the search is again unsuccessful in the second information booth, the request can be directed to a third, fourth, and fifth until the name is found or it is determined that the name does not exist (i.e. an information booth does not manage the name nor know of an information booth that does). Clearly, some mechanism must be available in the design to support the notion of forwarding requests.

In the current definition of the name space there are two types of nodes, domains and objects. In order to support a distributed name space (i.e. distributing the name space over more than one name server), a new type of node is needed, a remote domain. Structurally, a *remote domain* is similar to an object, the only difference is its type specification. This remote domain allows the name service to forward a message to another instance of the name service, analogous to one information booth forwarding a shopper to another information booth. To implement this, a simple list of name service instances could be maintained. This would mean

an additional mapping step, however. That is, if the requested name can not be used or is not found in the portion of the name space managed by the current name service, then the list of other name services must be searched. In order to make mappings uniform, the remote domains are simply registered as other names (objects) in the data base. In this way, when the name space is being traversed, if a component of the path name is a remote domain, the request is forwarded to the name server specified by the remote domain. The search for the object is then continued at the second name server. The pattern of searching and forwarding is continued until the specified object is found, or it is determined that the object does not exist (i.e. a component of the path name can not be found). Another advantage to this scheme is that the requests can now not only be distributed over the objects in one name service, but over the entire 'network' of name services. Indeed,the distinction between an object and a remote domain could also be eliminated. There is no need for a given name service to know what type of service to which it is forwarding a request. It only need know that the service on the receiving can process the forwarded message in some (correct) way. The distinction is kept here, however, to conform to the above model.

2.2.4. Name Mapping

Mapping a request to a service is similar to the mapping used in the V-System IPC level mapping[4]. At the IPC level in V, names are mapped to a pair (logical host, local pid). Here, names are mapped to a pair **(logical host, port number)** where the port number is the port on the logical host where messages are to be sent to contact the service. That is, the command

rm myfile

would first be mapped to the file service which in turn is mapped to the pair shown above. For example

rm myfile→ file service→ (host, port)

is a valid mapping. The (host,port) pair is then used to complete the link between the user (client) and the service. That is, the request is forwarded to the server located on host and waiting for messages at port.

2.2.5. Meeting the Goals

For the sake of completeness, and of course correctness, the design developed from the model should be shown to meet the goals previously put forth. This is a rather strict requirement and one that does not work well in a software development environment. It is not always best, nor possible, to solve the entire problem and meet all of the goals at every stage of development. Some problems are best approached at the lower levels of the development (e.g. implementation). What must be maintained, however, is the ability to solve the original problems and the ability to meet the original goals at each stage. That is, restrictions can not be placed on the design that stifle lower levels from meeting system requirements. With this in mind, the goals met by the above design are discussed next.

The transparency of the system must certainly be maintained within the design. If it is not, it surely will not be maintained by the implementation. Transparency is maintained by the structure of the name space. No matter where the object or the user are located, the same (full) path name will map to the same (equivalent) object within the name space.

Allowing distributed generation of unique identifiers is also maintained by the design. When a new name is generated (i.e. a service is registered with the name service), all that is required is to insert the name of the new service in the existing name space. This newly registered service will then be available to any client that requests its use since the name space is traversed for each request. In fact, traversing the name space for each request is a necessary requirement for dynamic binding to be supported by the implementation.

Two different levels of identifiers are also present in the design. The human oriented identifier is simply the name of the service to which a request is made. The computer oriented

identifier is the pair <host,port> to which the human identifier is mapped.

As alluded to above, to maintain the concept of dynamic binding, the higher (human) level name can not be mapped to the object until a request is made and the name service is contacted. This requirement is met by the fact that the name service manages the identifiers of the objects. The user making the request only has knowledge of the human oriented identifier. Only when a command is entered, is the name space traversed and the human oriented identifier mapped to the object identifier.

2.2.6. Caching Requests

As with any system that involves the use of a computer network, one of the major issues is traffic intensity. Since sending messages over a network is usually slower than operating from the same machine, one must be concerned with the number of messages being sent over the network. On a network implemented by a single Ethernet as is the case for this thesis, sending large numbers of messages increases the probability of collisions and therefore increases the number of retransmissions. The network then becomes a bottleneck. Processes waiting to send messages wait even longer and throughput of the system drops. Clearly, some method of controlling the number of messages that are sent over the network must be found. Caching user requests allows the number of messages to be reduced.

Recall that each time a client requests access to a service, the name service must first be contacted to map the request to an appropriate server. If namer does not reside on the same machine as where the request originated, which will usually be the case, a message must be sent to namer and then forwarded to the server to which the request is mapped. If the requests are cached by the client, then every subsequent request can be sent by the client directly to the appropriate server reducing the number of messages from 2 to 1. This implies of course that upon completing a successful mapping, namer must send a message back to the client indicating the location of the mapped server. Figure 6 shows the message passing required for this caching

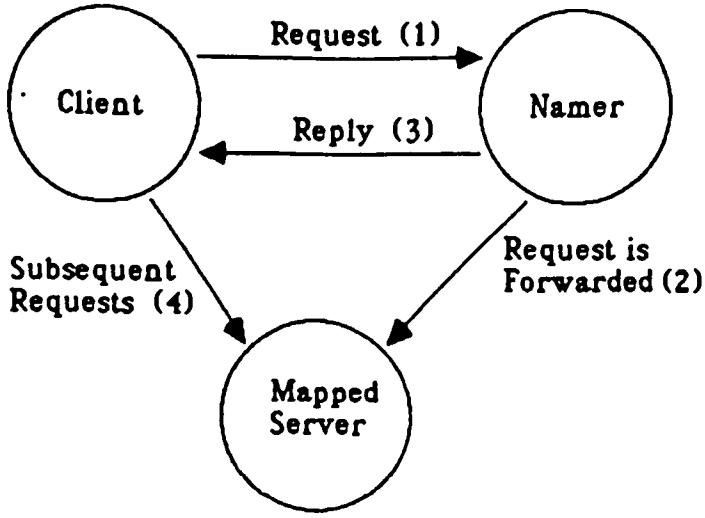


Figure 6: Message passing with a cache.

scheme. Note that initially caching requires 3 messages as opposed to 2 without caching. However, after two requests for the same service, the caching scheme begins to reduce the message count by half for each request.

Placing the cache on the server side was also considered. However, this would only reduce the actual mapping time. The number of messages would be the same as in the no cache case above. Since the mapping time is not as critical as compared to the network speed, placing the cache on the client side is more appropriate.

CHAPTER 3

SOFTWARE DESIGN

A top level design for the name service software is shown in Figure 7. The naming system is divided into three levels: the Application Level, the Function Level, and the Communications Level. The application level and the function level communicate through the user interface, and the function level and communications level communicate via the communications interface[1]. The entire naming system resides 'on top' of the UNIX operating system. As a result, no changes to any system code or configuration files are required to implement the naming system.

The naming system is also divided into two parts horizontally: the Client Side, (CS) and the Server Side (SS). This is similar to the design used in BIND [3]. Network services, including the name service, need not reside on all machines. Users working on machines where the name service does not reside may still require its use. The function of the Client Side is to take requests from the user and forward them to the Server Side for processing. In addition, to make all requests of the name service uniform, every request must first pass through the Client Side regardless of whether the name service resides on that machine or on a different machine. This implies that the Client Side be available on all machines in the network.

3.1. Client Side vs. Server Side

As mentioned above, the job of the client side is to take requests from the user and forward them to the server side where they can be processed. See Figure 8. The user sends a request via the user interface to the client side. The client side uses this request to build a message that can be sent to the server side. Once the message is assembled, the client side forwards

[1] It should be noted that the communications level here is built on a transport (or possibly higher) layer protocol [15]. Interprocess communications between machines, as well as intermachine communication is handled underneath the entire naming system.

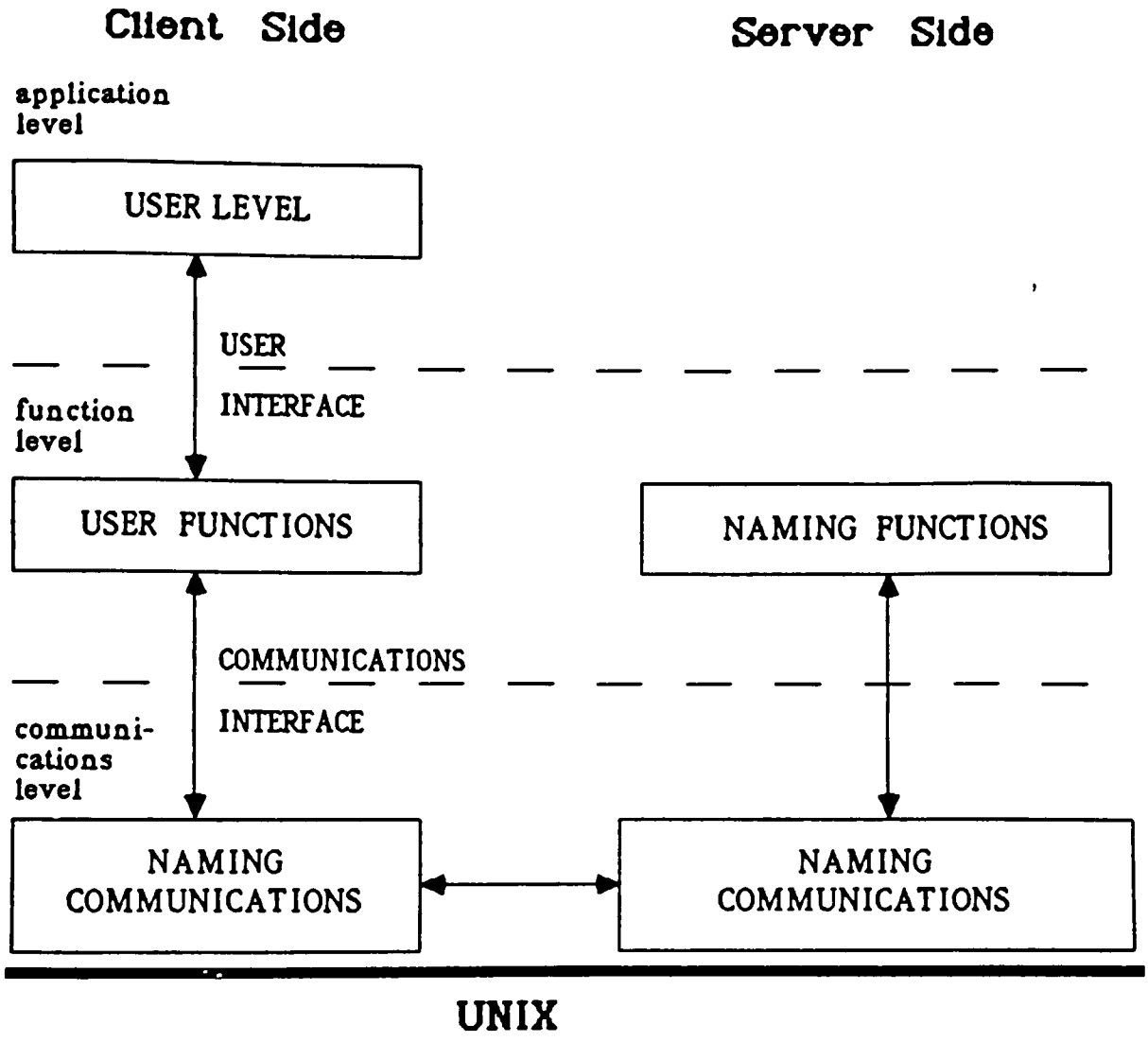


Figure 7: Name Server Structure.

the request to the server side through the communications level.

The server side is depicted in Figure 9. The function of the server side is to process user requests. It must first wait for a message to arrive through the communications level from the client side. Since there are a number of different requests available to the user, a decoding process must determine which request has been sent. Once the message is decoded, the proper

request can be processed. The processing of a request may require an update of the data base (e.g. registering a server), or merely examining the data base (e.g. locate the address of a service).

3.2. Module Design

The next section of the paper discusses the interfaces between each of the modules, or levels, of the name service. The communication interface and the user interface are discussed first since they do not change from the CS to the SS. Next the function level of both the CS and the SS are described. The implementation of this level clearly must be different on each side simply

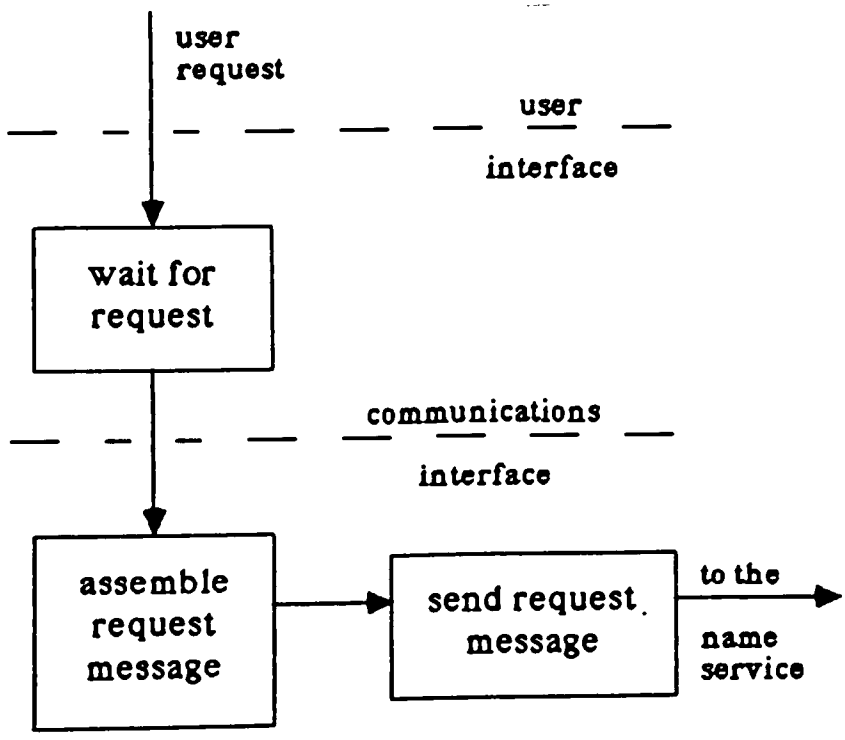


Figure 8: The Client Side.

to achieve the correct functionality. For this reason they are treated separately.

3.2.1. The Communications Interface

The communications level provides the necessary process communication between different parts of the name service. The interprocess communication mechanism that is used is Sockets [13]. Raw sockets are a very low level communications mechanism; to deal with them directly is a very complex task. As a result, I have designed an abstraction of sockets that better suits the needs of the name service. As with any abstraction, it limits the operations that may be performed on the object in exchange for ease of use. The abstraction, which makes up

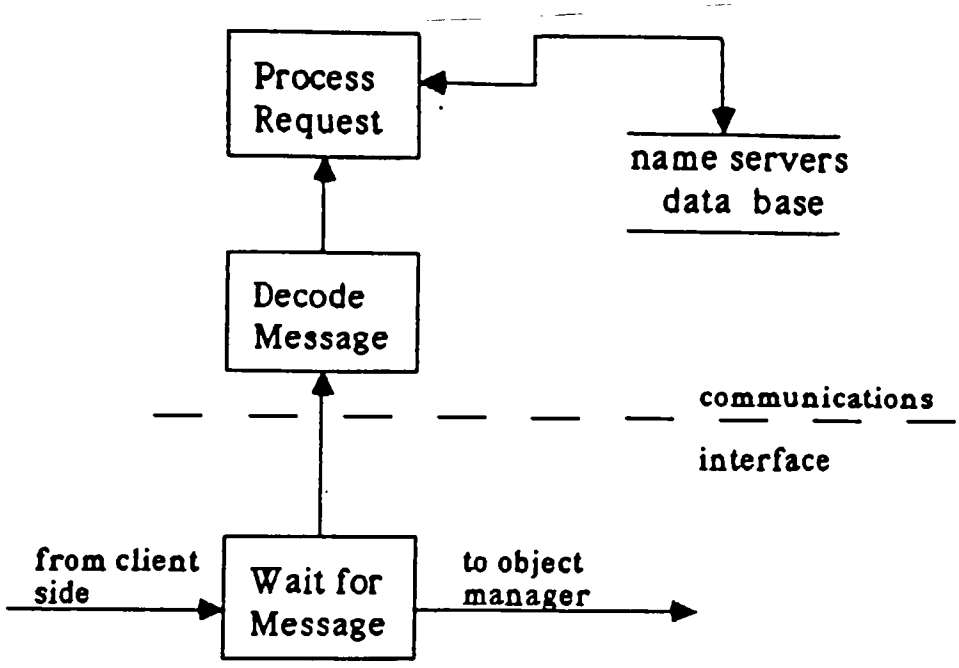


Figure 9: The Server Side.

the communications interface as well, consists of six parts. See Figure 10.

The first part of the interface initialises a socket for reading and binds a name to it so that other processes may send it messages. This is handled by a function called irdsock. When a process within the name service wishes to be available to receive messages from others, it simply executes this routine via the interface. Once executed, a socket is reserved for that process and other processes may begin to send it requests. For example, when the server side initializes upon booting, it calls irdsock to create a socket with a well known name (in this case a port number) so that client sides can send it subsequent requests.

The second part of the interface performs initialisation of writable sockets. That is, the function iwrsock creates a socket so that messages can be sent out to other processes. In this case, no name is bound to the socket when it is created since no processes other than its creator will refer to this socket. For example, when a client side receives a request for access to the name service from a user, it must first create a socket by calling iwrsock() to initialize a socket to write a message.

Once sockets are created by a process, they can send and receive messages on the appropriate sockets. These are the functions of the next two portions of the interface. Sendrel sends a message from a socket created by a iwrsock call to a socket created by an irdsock call. Nmaccept retrieves a message from a socket created by an irdsock call. When the use of a socket is no longer required, a mechanism must be available to give the socket back to the system so it may be allocated to other processes. The function closesock returns an open socket to the system. The socket is closed regardless of whether it was opened for reading or writing and once closed, it may no longer be used for communication. The final section of the interface, mkmsg, simply allows higher levels to construct meaningful messages to be sent.

3.2.2. The User Interface

Figure 11 shows the current state of the user interface. Recall that it is desired to have interaction with the name service be as similar as possible to that of a conventional UNIX

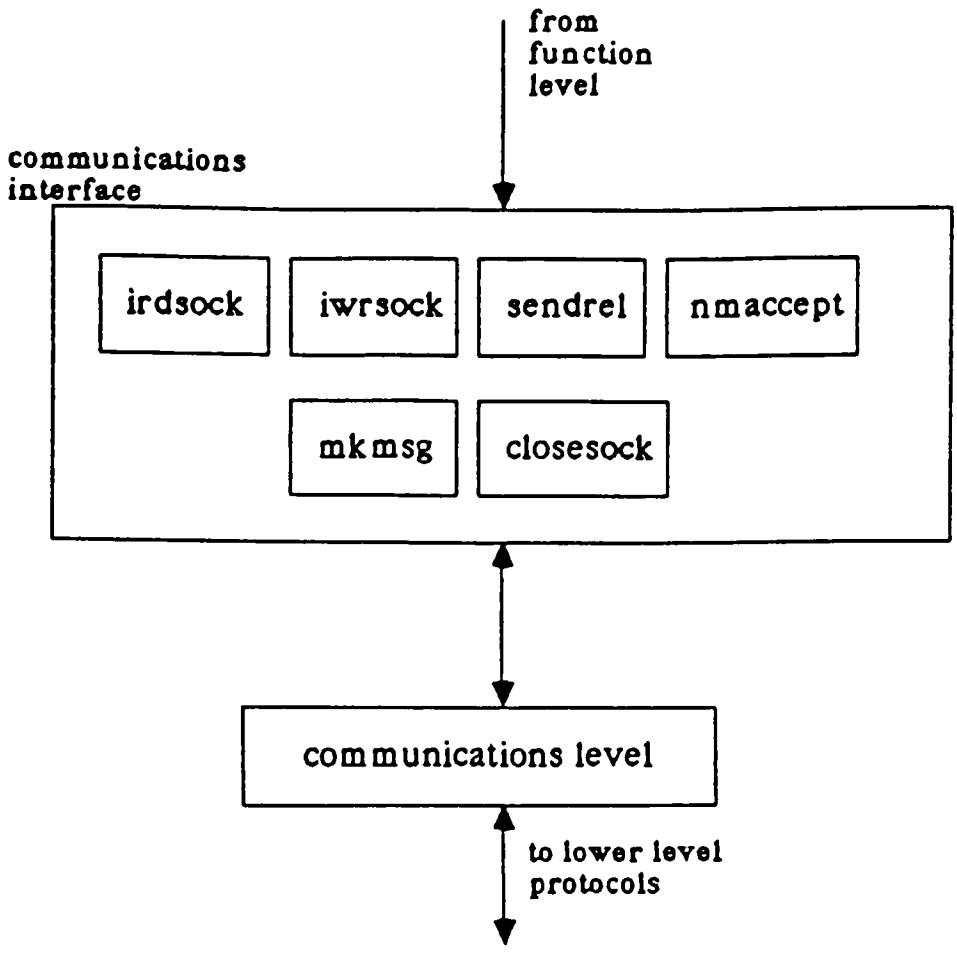


Figure 10: The Communications Interface.

environment. This interface can be expanded as needed when new functions are implemented without modification to any of the lower levels of the CS. Each of the commands is contained in its own executable file. However, the user functions are written in shell script. This is to facilitate modifications and additions to this portion of the interface. That is, when a new command is added, or an existing command modified, no compilation of the name service software will be required. The user function part of the user interface is the only portion of the name server that is not written in C.

Once the administration functions are in place, they will not change very frequently. For this reason they are written in C rather than shell script. When they are modified, or a new function is added, a recompilation of the name server will be required. Since this will not occur very often, they will be implemented in C to improve performance.

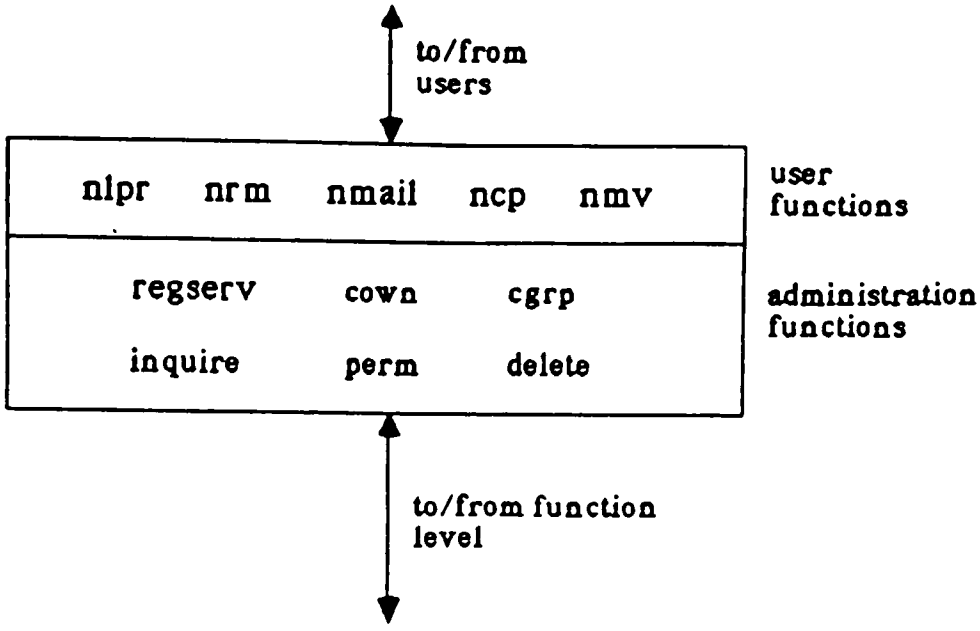


Figure 11: The user interface.

3.3. The CS Function Level

The function of the client side is to wait for messages from the user and then prepare a message and send it to the server side for processing. The function level must be prepared to accept two types of requests: user function and administration function requests. For this reason, the CS function level itself is divided into two parts.

To process user function requests, there is a single C function. All user functions require the same treatment so to implement each one separately would be of little benefit. That is, each user function is of the same form: <cmd> <args>. The CS can prepare a message for each in the same way so there is no need to divide the implementation any further.

Administration functions, however, may each require different function and expect different results from the server side. As a result, each administration function is implemented as a distinct function.

3.4. The SS Function Level

The SS function level is the heart of the name server. It is here that all of the name mapping, data base updates and queries takes place. It is the SS function level that is running on a single, or perhaps a few, nodes in the network and ultimately handles all user requests. Because of its complexity, an explanation of the algorithm used in processing a request is the best way to discuss the design. Figure 12 shows a visual table of contents (VTOC) of the SS function level software. The first action the name server takes is to initialize its data structures (as any piece of software). Of course the primary function of the SS is to wait for user requests and process them when they arrive. In order to prepare to receive requests, the SS must open a socket from which to read. The function of irdsock is to open a socket from which to read and attach the name server's well known address (i.e. port number) to enable client processes to send the SS subsequent requests. Providing the name service successfully opens the socket, the next action taken by the name server is to simply wait for requests. This is done in nmaccept: waiting, and accepting messages as they arrive.

Recall that there are two types of requests; user function requests and administration function requests. Once a request arrives, a decoding process starts by determining which type of request must be processed. When the message type is determined, the remainder of the decoding and processing is passed along to the appropriate routines. If a user requests has arrived, the command must be mapped to a node in the network where the server that implements the command is located.[2] Note that there may be more than one service that implements this command. For example, there may be two print servers each providing a printing service at different locations within the network. Of course, in this situation, a user may desire to specify a particular server to be used. This is done by including a command option (e.g. -epson) when the request is made by the user. If the user does not know the correct option, some form

[2]It may be interesting to experiment with implementing the remainder of the decoding and processing from this point as separate processes to give more time for the name server to accept and begin processing new requests.

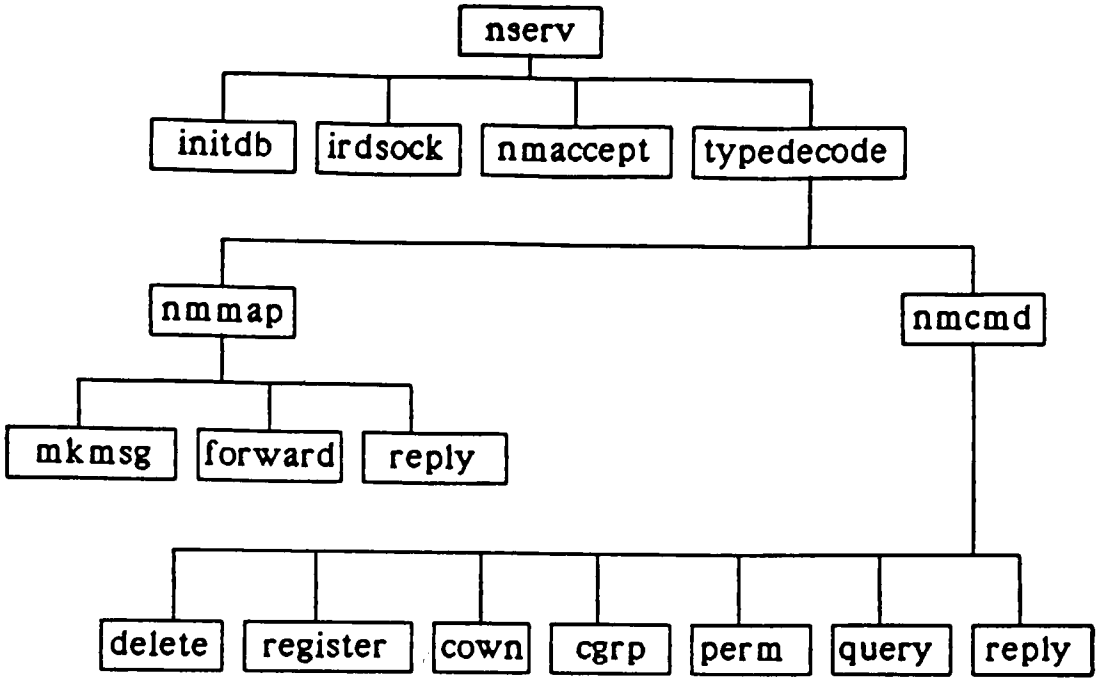


Figure 12: The SS function level.

of a help command may be performed to provide the information. If the user does not use the option to specify a particular server, the SS chooses one in a round robin fashion.

Once successfully mapped to the correct server, the command is then forwarded to that server and a reply sent back to the user. From this point on, any operations, or communication necessary between the server and user is strictly between the user an the server implementing his request. That is, once a command is forwarded and a reply sent, the name server has no knowledge of any actions or communication needed between the user and the server implementing the command. It is up to the server implementing the command to set up communication with the user making the request.

If an administration function is accepted, the name server must take a different action. In this situation, the remainder of the decoding process simply consists of determining which administration function to perform. Once this is determined, the appropriate function is called to perform the task. Depending on the function, an update of the data base may be required (e.g. registering a service), or simply an inspection of the registered services (e.g. query). In either case, however, the name space, at some point, must be traversed.[3] Finally, a reply is sent back to the user indicating the success or failure of the request.

3.5. Functional Specification

The name service will provide various functions to the user of the system. The functions themselves are divided into two categories: 1) user functions, and 2) administration functions. The *user functions* are those commands that are available on a traditional UNIX system. That is, they are the commands that, when invoked, contact the name service in order to locate the correct service implementing that command. The *administration functions* are those functions that modify, or simply query, the name server's data base. The next section lists and describes each of the functions found in the name service. A more complete and detailed description of the available functions can be found in the User's Reference Manual in appendix A and the Programmer's Reference Manual in appendix B.

3.5.1. Functions Performed

The administration functions are the basis for all name service operations. They allow a user (most likely a system administrator) to modify and query the name server's data base. The administration functions are listed below with a description of each.

Administration Functions:

1.) register	-register a service with the name service.
2.) cown	-change the owner of a service.
3.) cgrp	-change the group of a service.

[3]Utility routines such as the ones used in traversing the name space are not shown in the VTOC for clarity.

4.) perm -change the permissions on a service.
5.) delete -delete a service (un-register).
7.) inquire -request info. about registered services.

3.5.2. Limitations and Restrictions

Although a number of the administration functions deal with protecting items managed by the name service, there are severe limitations to the protection mechanisms. Assuming the correct identity of the users, the mechanisms are sufficient to provide adequate protection against an unintentional unauthorised user. A problem arises when a malicious user impersonates another. That is, no authentication of users will take place within the naming system. It is assumed that all users are friendly and will not intentionally corrupt the system in any way. However, consideration for this ability is incorporated into the design.

One of the major considerations when designing a distributed system is that of fault tolerance. Fault tolerance is the ability of a system to recover gracefully, without user knowledge, from both software and hardware failures. Recovering from such failures is beyond the scope of this thesis and will not be dealt with in the implementation. As with authentication fault tolerant considerations is not overlooked in the design.

Network wide data base consistency is also not maintained within namer. That is, although it is possible to register a service with more than one instance of namer, keeping each registration consistent with every other entry is not supported. This may cause discrepancies in which users have access to which services. If the execute permission on some registered service is changed, then each registration entry must be explicitly updated. If they are not, a user may be able to access the service one instant and not the next. This is not an acceptable situation, especially for highly protected or classified services (e.g. boot service). For a small local area network such as the network on which namer is currently implemented this does not present a major problem. The system administrator can keep track of with which instance of namer each service is registered. This does imply, however, that the registration and some other

administration functions should only be available to the system administrator.

Consistency in the user cache is also not maintained. If the privilege to use some service is revoked at some point in time for some group of users, the entries are not deleted (modified) in the user's cache. This also may allow unwanted users access to privileged services. A simple solution to this problem is to locate the user's cache within the login environment. In this way, whenever a user logs in, the cache will be cleared.

3.5.3. User Interaction

The normal user of the name service will be unaware of its existence. Commands will be entered, as on a conventional UNIX system, within some shell or command interpreter. The commands destined for the name service will simply be forwarded to, and executed on the (possibly remote) service. For example, if a user enters the command

```
> lpr myfile
```

the command will be mapped to the print service which will handle the execution of the lpr command to print the file myfile. If a user enters an illegal command, or an illegal request is made, a message will be presented to the user indicating that an error has occurred. These messages are dependent on the type of illegal request and therefore can not be defined until the commands and the services implementing the commands are designed and implemented. Shown below is a short list of possible user functions:

User Functions:

1.) lpr -locates the print service to print a file.
2.) mail -locates the mail service.
3.) rm -locates the file service to delete a file.
4.) cp -locates the file service to copy a file.
5.) mv -locates the file service to rename a file.

As can be seen, these are traditional UNIX commands. It is assumed that these functions are now handled by special services located somewhere in the network. However this list is not intended to represent a complete list of commands that may be implemented. It is simply a group of selected commands that may be implemented for demonstration purposes. In addition, the list is part of the user interface and can easily be expanded as new services and commands are implemented in the future. It is intended, however, that user interaction be as similar as possible to that of a conventional UNIX environment.

3.5.4. System Files

Presently, there are two files maintained by the name service. The first contains the registration data base. That is, it will contain every service currently registered with an instance of namer. Each instance of the name service will maintain a file of this type called register.LOG. As with the active data base, consistency will not be maintained among the different LOG files. Each line in the file will represent one registered service. The function of this file is to aid in the recovery of software, and possibly hardware failures in the future. If an instance of the name service crashes, the new name service can initialise itself by reading the registered services from this file. Each time a service is registered with an instance of namer, it is appended to the LOG file. When the service is deleted from the name space, its corresponding entry is removed from the file as well.

The second file maintained by the system is the user cache. Each user will have a private copy of this file. The file, called .ucache, keeps a record of what services the user has contacted. Each time the user requests access to a service, the cache is first scanned for an entry

corresponding to the request. If an entry is found the name service is not contacted to perform a mapping. Rather, the SS is bypassed an a message is sent directly to the intended service. As a result, each entry in the user cache must, at a minimum, contain the name of the service, its host, and the port number at which the service may be contacted.

Placing the cache in a file is intended to be a temporary solution. Since each user request for the name service is self contained, a way was needed to preserve the mapping results from one call to the next. Keeping these results in a file is the most straight forward approach, although perhaps not the most efficient in terms of speed. A better solution was alluded to above; place the cache in the login environment. In this way, not only are some of the consistency problems solved, but the cache will be located in main memory rather than a on secondary storage device making the cache lookup operation much faster.

CHAPTER 4

NAMER IMPLEMENTATION

4.1. Namer Communications

Because namer relies on clients and servers to exchange messages the first portion of the system to be implemented was the communications level. As mentioned previously, the communications is based on sockets and an abstraction of sockets was designed in order to simplify the interface to the communications facilities found in UNIX. Contained in this abstraction are operations to initialize readable sockets and writable sockets, to send and receive messages on these sockets, to create messages, and to return unwanted sockets to the operating system. A detailed description of each of the software routines is given in Appendix B in the Programmer's Reference Manual. However, the types of messages implemented for namer does warrant some discussion.

The name service itself must be able to exchange messages with both its clients and other servers (including other instances of namer). Three type of messages were designed and implemented to support the necessary communications; 1) a *control message*, 2) a *user message*, and 3) an *acknowledgement message* type. These three message types were designed to make it easy for namer to distinguish between types of requests. A control message is used to send administration requests to the server side. When the server side receives a message, the first part of the decoding process is to determine the type of message that was sent. If it is a control message, a field within the message indicates exactly which administration function to perform. The specific routine to handle the command is then called to perform the operation.

The user message is used to send user requests (i.e. user functions) to the server side. If the server side receives a user message, the mapping function is immediately executed to map

the user request to an appropriate server. If the mapping is successful the request is then forwarded to the mapped service. The message forwarded to the mapped service is the same message that was initially sent to namer.

The acknowledgment message is simply used to send an acknowledgement back to the user. Some type of reply must be sent for both the user functions and administration functions for confirmation that the request has indeed been carried out. If the request was successfully completed, the positive integer 1 is sent as the reply. If the request failed, a negative integer is sent as the reply of which its value is determined by the type of error. In addition, when a successful mapping has occurred, the user message replies include the address of the service to which the request was mapped for caching purposes.

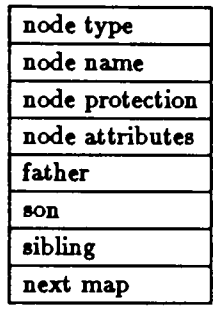
At first, the three message types described above seem to meet the communication requirements. This was not the case however. The administration function inquire requires that information be sent back to the user. As a result, a fourth message type was created for this purpose called an *information message*. When the server side receives an inquire request, it sends all the necessary information back to the user in a information message. The server side terminates the sending of information by sending an acknowledgement message. That is, when the server side finishes sending the information, the positive integer 1 is sent back to the user in an acknowledgement message to indicate that no more data will be sent and the inquire command is complete.

In order for the communications and the clients of namer to work correctly, all communications should be performed through this abstraction. This includes clients and services that will be developed in the future. In fact, as mentioned above, when the namer successfully completes a mapping, the same user message (with the destination fields in the header modified) is forwarded to the mapped service. The service must be able to recognize this message structure correctly in order to complete the request. Once the mapped service receives this message, it

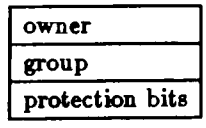
may process it in any way that it requires. In addition, in order to keep the communications within namer consistent, services that require communications with the original client (e.g. the user) should adhere to this abstraction as well.

4.2. Name Space and Mapping

The name space designed for namer is a hierarchical structure with some root node. As a result, a tree structure was chosen to represent the name space in the software implementation. Every node in the tree has the same basic structure and is shown in below:

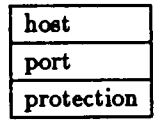


The field node type indicates the type of node. It may be either a Domain, an Object, or a Remote Domain. The name is simply the label given to the node when it is inserted (registered) into the tree. The remaining fields are all pointers to various items. The protection field points to the protection information associated with the node. The structure of the protection information is shown below:



The protection structure contains the owner and the group of the node in addition to the permissions each user is allowed. The next three fields in the node are pointers to allow the extension of the tree. The son points to the next lower level of the naming hierarchy from the

current level. Sibling points to a node on the same level, and father points to the parent node (domain). The father pointer was created to facilitate deletions and traversals of the tree. In addition, the root node is its own father. Next map is the pointer that allows namer to distribute the incoming requests. This implies that only in a domain does this field have meaning. If no specific service is specified by the user, this pointer is used to locate the next service that is to be used. After the request is forwarded, this pointer is updated to the next usable service within this domain. A round-robin algorithm is used to decide which service is mapped to next using this pointer. Finally, the attributes field points to the attributes of the registered service. Similarly to the next map field, this field only has meaning in nodes of the type object or remote domain. In fact, this pointer is set to zero in the domain nodes. The structure of the attributes is shown below:



The host field is the name of the host where the service resides. Port is the port number on host where the service may be contacted. The protection field is a pointer to the protection information associated with the registered service. The structure of this information is identical to that described above.

When a user request arrives at the server side, the name to be mapped is in the form

rit/grad/services/some_service_name

The name space must be traversed using this name in order to locate a service. The name is parsed from left to right. Each string between the "/"'s is stripped from the name and is used to search the next lower level of the name space tree. If the string is found, the next component of the name is stripped and the mapping proceeds. At each level, the protection values are

checked to insure that the user has sufficient privilege to traverse this portion of the name space. When the last component of the name is located, the next map pointer is used to find the next available service. At this point, the permissions on the registered service are checked to confirm that the client does indeed have the privilege to use the service. If the permission check is successful, the request is forwarded to the mapped service. If at any time a component of the name is not found, the mapping function is terminated and an error condition is returned to the client.

CHAPTER 5

EXPERIENCE AND CONCLUSIONS

My interest in this thesis began about a year ago while researching a paper on naming and protection in distributed systems. Shortly after the paper, I began work on a small prototype of a namer service to insure the feasibility of the thesis. During the development of the prototype, the need to split the name service into two sides became evident. This developed into the design of the client side (CS), and the server side (SS). Also, the prototype obviated the need for some type of standard communications to be used by the CS and SS to exchange messages. Finally, the functions and the information needed by the name service to perform those functions became evident from the prototype. This developed into the structure of the commands, the contents of each of the message types, and the contents of each node within the data base. This final section of the paper discusses my experience in designing and implementing these major components of the naming system.

5.1. The Communications

As mentioned previously, since correct operation of the name service relies on efficient exchange of messages, the communications level of the name server was one of the first components to be designed and implemented. The communications within the prototype involved the exchange of short and simple messages. Because of their simplicity, the problems of heterogeneous environment were not completely encountered. It was not until the name service was operating that the problems of byte alignment surfaced.

The problem of byte alignment arose between the Pyramid and the AT&T machines. On the Pyramid, integers are put on a word boundary (multiple of 4). On the AT&T machines, however, integers are not necessarily on a word boundary. Integers on the PC's may appear on

a half word boundary (multiple of 2). As a result, passing messages between these two machines failed at some points. One solution to this problem is to insert dummy bytes before sending the message. This could be done at run time or compilation time using the conditional compilation directives offered by C. Neither of these solutions were acceptable, however, mainly because of the need for the software to have explicit knowledge of the machine on which it is running. The added execution time to insert dummy bytes at run time is also a disadvantage not to mention the extra overhead of sending the dummy bytes over the network. The solution that was chosen was to restructure each of the message types so that all integers start on a word boundary. First, adding a new machine to the network will not require new code to be written or recompilation as in the solution above. The software does not depend on which machine it is running so new machines can be added while the name service is executing. Second, no problems are encountered in the code since the software in no way depends on the order of the fields within the each message. One requirement that was added to insure correctness in the future was to make every structure used in building messages to be a multiple of 4 bytes long. In this way, if the messages are restructured or if nested structures are introduced, byte alignment will be maintained.

Another issue that arose while developing the communications software was what type of underlying protocol to use; data gram or virtual circuit. All of the messages that are sent are relatively small (under 1K bytes). This suggests the use of data grams. However, the underlying data gram protocol is unreliable. The name service must have reliable communication in order to work correctly. To make data grams reliable, an additional protocol would have to be developed between each client and the name service. Each instance of the server side would then have to maintain a list of each client in order to correctly sequence messages and to recognize duplicate transmissions. This, and the added problem of initially setting up the additional protocol indicated that data grams were not the best solution.

The virtual circuit protocol is reliable. As a result, no further code must be written to insure correct message passage. A client simply makes a connection with the server side and sends its request. The underlying protocol insures that the message will arrive correctly, or present an error condition to the sender.

5.2. The Name Service

The development of namer began assuming that there would be only one instance of the server side running at any given time. Extending the name service to support multiple SS's running concurrently would be done at a later time. This decision was made mainly because all of the issues involved in developing this extension were not clearly understood. During the course of implementation, however, allowing multiple server sides turned out to be a minimal modification. All that was needed was another node type; hence the creation of the remote domain. When the mapping function encounters a node of this type, it simply forwards the request to the new server side.

Another issue that came up during the course of implementation were the definitions of domain and object. Recall that the main difference between the two is that a domain node has child nodes while an object node does not have child nodes. In order to keep to the strict definitions, no functions or commands were developed that allow the explicit creation of domains without child nodes. Rather, when an object is added, the domains implied by the path to the object are added as needed. This scheme maintains the definitions of domain and object as objects are created. However, what about deleting objects? In order to keep to the definitions, when the last object in a domain is deleted, the parent domain must also be deleted. But this domain may be the only child node of its parent domain. The next higher domain must then be deleted. Indeed, each level must be checked backwards up the tree until a non-empty domain is found. The protection problems made this very impractical to implement. For example, while traversing the tree downward, only execute privilege is required. Deleting a node, however,

requires write permission; but only on the domains that will be empty after deleting its descendents. If while traversing the tree backwards, the permissions do not allow domain to be deleted, what happens to all those nodes that have already been deleted? Are they discarded or is the entire deletion process aborted and the deleted nodes re-inserted into the tree? Because of these problems, and the fact that having empty domains exist in the name space does not present any major problems, the deleting of the empty parent domains is not implemented. When a delete operation is performed, only the specified object is deleted.

5.3. Alternative Designs

Currently there are operations available in the communications interface to open sockets for reading and to open sockets for writing. This requires that the client side software, if it expects a reply, to open two different sockets; one to send the request and another to receive the reply. This, in turn, implies that two connections must be established in order for the CS and SS to complete a request. Making two connections, rather than one, adds some unnecessary overhead within the communications. To reduce this overhead, an additional operation allowing a socket to be opened for both read and write operations could be developed. This would involve only one connection being established rather than two.

The basic design of the name service, I believe, is correct given the system requirements and specifications. Improvements may be available, however, within some of the data structures. These may seem like minor modifications at first, but if the messages can be reduced in size, or the size of the data base can be reduced, as the naming system becomes larger as a whole the adjustments will have a larger effect. In addition, some modifications may make it easier or harder to increase the functionality of the name service in the future.

The first adjustment can be made in the data structures that maintain the data base. Within the protection bits, only 9 of the 16 bits are used. The remaining bits could be used to convey additional information. The **node type** field could be eliminated by assigning one of the

protection bits to each type. This is similar to the way in which UNIX distinguishes between types of files. Similar reductions can be found in the messages. One way to reduce the message size is to redesign the communications to handle variable size messages. Currently, every message type has a fixed size. Unfortunately, the size must be fixed large enough to handle any size message. The size of the messages can depend on command options, arguments and the like. If the size of the commands is usually smaller than the size of the actual message sent over the network, efficiency is lost due to increased overhead. By allowing variable sized messages, the average message length sent over the network could be reduced, minimizing overhead and increasing efficiency.

5.4. Future Projects

During the development of this thesis, the ideas for many additional projects arose. Some were discussed in section 3.5.2. The first is to provide some authentication mechanism. Currently it is assumed that all users are friendly. There is no mechanism for the server side to insure that the client is who he claims to be. If a user modifies a message to indicate that he is another user (e.g.root) before it is sent to the SS, the server side must assume that the user ID is correct. Clearly, this can cause major security problems and in practice is an unacceptable situation. A malicious user claiming to be a privileged user such as root could compromise the entire system.

The second issue discussed was that of fault tolerance. If a part of the name service software crashes, there are no mechanisms to recover from the failure. Hardware crashes can also create problems. If a node crashes on which an instance of namer is running, users of that instance will not be able to locate any services. Essentially the users will be partitioned from the rest of the network.

The third issue discussed in section 3.5.2. was that of data base consistency. Some mechanism must be available to ensure consistency in the name space over the different

instances of the server side. In addition, in the presence of software and hardware failures, the data base may become inconsistent or contaminated in some way. A recovery system should be developed so when these server sides are restarted the data base entries are checked for consistency and integrity.

Another project that would be useful is to incorporate the name service with the shell or login process. This would allow paths to be set to distinguish between commands that require the use of namer and those that do not. As mentioned earlier, the user cache could then be moved to the shell where it would reside in primary memory rather than on secondary storage.

The above list of enhancements is certainly not complete. As the system evolves, more and more projects will no doubt develop. The projects listed above will, I am sure, provide sufficient work for quite some time. This thesis is, hopefully, only the beginning in the evolution of a distributed system. Namer provides a foundation to introduce the transparency required for such a system. It is hoped that the development of this thesis is continued through the projects mentioned above and any others that develop in order to create a reliable, transparent, and efficient distributed system.

REFERENCES

[1] Berglund, E. J., "An Introduction to the V-System", *IEEE Micro, August 1986.*

[2] Birrell, A. D., *et al.*, "Grapevine: An Exercise in Distributed Computing", *Communications of the ACM*, vol. 25, no. 4, April 1982.

[3] Bloom, J. M., "Experiences Implementing BIND, A Distributed Name Server for the DARPA Internet", *USENIX, Conference Proceedings,* Summer, 1986.

[4] Cheriton, D. R., Mann, T. P., "Uniform Access to Distributed Name Interpretation in the V-System", *IEEE International Conference on Distributed Computing Systems*, 1984.

[5] Comer, D. E., "A Model of Name Resolution in Distributed Systems", *IEEE 6th International Conference on Distributed Computing Systems Proceedings, 1986.*

[6] Comer, D. E., and Murtagh, T. P., "The Tilde File Naming Scheme", *IEEE 6th International Conference on Distributed Computing Systems Proceedings, 1986.*

[7] Curtis, R., Wittie, L., "Global Naming in Distributed Systems", *IEEE Software*, July 1984.

[8] Curtis, R., Wittie, L., "Naming in Distributed Language Systems", *IEEE International Conference on Distributed Computing Systems*, 1984.

[9] Lampson, B. W., "Designing a Global Name Service", *ACM Symposium on Principles of Distributed Computing*, 1986.

[10] Lau, F. C., and Manning, E. G., "Cluster Based Naming for Reliable Distributed Systems", *IEEE International Conference on Distributed Computing Systems*, 1984.

[11] Oppen, D. C., Dalal, Y. K., *The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment,* Xerox Technical Report OPD-T8103, Palo Alto, California, October 1981.

[12] Peterson J., Silberschatz, A., *Operating Systems Concepts*, Addison-Wesley, 1983.

[13] Sechrest, S., "An Introductory 4.3BSD Interprocess Communication Tutorial", *UNIX Programmers Manual Supplementary Documents*, volume 1, April 1986.

[14] Shoch, J. F., "Inter-Network Naming, Addressing, and Routing", *Proceedings of the 17th IEEE Computer Society International Conference (COMPCON),* September 1978, pp. 72-79.

[15] Tanenbaum, A. S., *Computer Networks, Prentice-Hall, Englewood Cliffs N. J., 1981.*

[16] Tanenbaum, A. S., Van Renesse, R., "Distributed Operating Systems", *ACM Computing Surveys,* vol. 17, no. 4, December 1985.

[17] Terry, D. B., "Structure-Free Name Management for Evolving Distributed Environments", *IEEE 6th International Conference on Distributed Computing Systems Proceedings*, 1986.

[18] Watson, R. W., "Identifiers (naming) in Distributed Systems", Chapter 9 in *Distributed Systems-- Architecture and Implementation*, Springer-Verlag, New York, 1981.

[19] Welch, B., and Ousterhout, J., "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System", *IEEE 6th International Conference on Distributed Computing Systems Proceedings*, 1986.

APPENDIX A

NAMER: A User's Reference Manual (URM)

Introduction

The *User's Reference Manual* (URM) describes the commands available to the users of the name service (namer). Other documents that contain additional information are:

1) The *Programmer's Reference Manual* (PRM) describes each of the commands, name service functions, data formats, and error conditions needed for programmers.

The commands described in this document are in alphabetical order simply for organizational purposes. Each command is described as follows:

NAME:
Gives the name of the command as it is to be entered followed by a secondary name. The secondary name is an English phrase describing the name.

SYNOPSIS:
Describes how the command is to be used. Some conventions used under the SYNOPSIS are:

1) Boldface strings are to be interpreted exactly as they appear.
2) Square brackets, [], indicate an optional command argument.
3) An argument beginning with an '=' (equal sign) is a command option. These options may sometimes appear in the same position as other arguments such as file names. It is therefore not recommended to have file names or other arguments that begin with equal signs.

DESCRIPTION:
Describes how to use the command and gives a brief summary of how the command works.

EXAMPLES:
Shows some simple examples of command use.

RETURN VALUES:
Discusses the values set when the program (command) terminates.

DIAGNOSTICS:
Discusses the error messages that may be produced during the execution of the command.

SEE ALSO:
Points to related information.

The Current Context

The name service maintains names in a hierarchical name space. At the top of the hierarchy is the root. The root is indicated by a leading '/' (forward slash) on a name and all (absolute) names begin with a '/'. In this scheme names can become quite long. As a result, a *current context* is maintained that is analogous to the current directory under UNIX. Whenever relative names are specified (i.e. names that do not begin with a leading '/'), the name is prefixed with the current context before the name space is traversed. If an absolute name is specified, the name is not prefixed with the current context.

To refer to objects or domains within the name space, some *pathname* for that entity must be provided. An absolute path name begins with '/' (the root). To refer to a specific domain within the hierarchy then, all domains names from the root domain leading to the domain in question are concatenated to the root and each is separated by a '/'. If an object within the domain is to be referenced, the object name is then concatenated to the *path* of domain names, again separated by a '/'. This string of names separated by slashes is the *pathname* of an object. If the pathname of an object does not begin a leading '/', it is a *relative* pathname and is always considered relative to the current context. That is, the relative name is prefixed by the current context before the name space is traversed.

Getting Started

Namer is a distributed name service for a connected UNIX environment. Before namer can be used for naming objects (services) within a network, it must be configured correctly.

Configuration

The first thing that must be done is to assign a host computer on which namer is to run. Next, a port number on that host must be assigned from which namer will accept messages to process. The assigned host and port will be used by all clients to contact the name service. Currently, the host is assigned 'ma', and the port number assigned is 41004. Each of these definitions are maintained in the file ncomm.h. In order to change the host and/or port definitions, the values of NHOST and NPNUM, respectively, must be redefined to reflect the chosen values.

Source Compilation

Once namer has been configured as above for the installation, the source code must be compiled. This, of course, assumes that any user functions and services implementing those functions are correctly written and installed to use the name service. For the compilation to work correctly, follow the steps below:

1) Create a subdirectory (within the directory containing the source) called OBJ. When the source is compiled, all of the object files are put in this subdirectory.

2) Create another subdirectory called BIN. All executable files are inserted in this directory. Note that in order to execute these files then, either your path shell variable must be set properly, or you must explicitly state the file you wish to execute.

3) Type the command

```
make namer
```

This compiles all of the source needed for the server side. That is, it creates the executable for the name service.

4) Next, the client side must be compiled. Since the client side must be available on all machines within the network that require the name service, the source for the client side must be compiled for each machine. To do this, simply install the client side software on each machine, create the directories **OBJ** and **BIN**, and enter the following command:

```
make nclient
```

5) Finally, the administration functions must be compiled. For the administration functions, it may not be necessary to have each available on all machines. For example, the registration and delete functions may only be necessary on a single machine (e.g. the system administrator's console), while the inquire command may be desired on all machines. That is, it may be unwise to allow every machine (user) access to register and delete services but allowing every user to determine what services are indeed available is certainly expected.

So, to complete the correct installation of the administration functions, it must first be decided on which machines each command will be available. Once this is determined, the following commands will create their respective executables:

```
make regserv
make rreg
make delete
make inquire
make cgrp
make cown
make perm
```

Pwc and cext are simply shell scripts and therefore do not need to be compiled.

The above list of commands may seem lengthy to type in; particularly if the commands or name service changes frequently (which is not recommended). In order to alleviate some of the typing, the following commands are also available:

1) make unamit
   -Create every executable for both the CS and the SS.

2) make nclient
   -Create the client side; including the administration functions.

3) make admin
   -Create only the administration functions.

Namer Execution

Once all of the files have been correctly compiled, namer must begin execution to accept and process requests. To do this, simply type the command:

namer

Name Space Initialization

When namer is first executed, the name space is initially empty. The name space remains empty until a service (object) is registered. That is, no initial domain hierarchy is inserted into the name space before a service is registered. As a result, the current context must be correctly established before a service is registered. To set up the current context, use the ccxt command. Remember that this command must be executed within the current shell. So to correctly set the current context enter the command

. ccxt pathname

The '.' command executes the script ccxt within the current shell. In order to avoid forgetting typing the '.' each time the current context is updated, creating an alias such as

alias ccxt='. ccxt'

is highly recommended.

As soon as namer has completed its initialization, a message is printed indicating the port number on which it expects to receive messages. This is the same port number that was assigned to namer during configuration. It is printed simply for confirmation. When this message appears, namer is ready to accept requests from the commands described in the remainder of this manual.

NAME

ccxt – change current context

SYNOPSIS

ccxt pathname

DESCRIPTION

ccxt changes the current context of the user to *pathname*. The current context is used for prefixing relative object and domain names during name mapping within the name service. As a result, *pathname* should be a string that begins with a '/' (forward slash) since all absolute names begin with a '/'.

ccxt changes the environment variable **ctxt** and would therefore be ineffective if executed as a normal command. It must therefore be executed within the current shell using the '.' command available in UNIX.

SEE ALSO

pwc(1)

NAME

cgrp – change group

SYNOPSIS

cgrp group object
cgrp group domain

DESCRIPTION

The group of the specified *object* or *domain* is set to the value according to *group*. The *group* must be specified as a decimal group ID as found in the password file.

Only the owner of the specified object or domain may change the group.

SEE ALSO

cown(1), perm(1),
cown(2) in the *Programmer's Reference Manual.*

NAME

cown – change owner

SYNOPSIS

cown owner object
cown owner domain

DESCRIPTION

The owner of the specified *object* or *domain* is set to the value according to *owner*. The *owner* must be specified as a decimal user ID as found in the password file.

Only the owner of the specified object or domain may change the owner.

SEE ALSO

cgrp(1), perm(1),
cgrp(2) in the *Programmer's Reference Manual.*

NAME

delete – delete an object or domain

SYNOPSIS

delete [=i] object
delete [=i] domain

DESCRIPTION

Delete removes an *object* or *domain* from the the name server's data base. The name to be removed must have been inserted into the data base by the **regserv** command. A *domain* can not be deleted if it contains other domains or objects

If the =i option is specified, the user is asked for confirmation before the specified object or domain is deleted.

The user must have write privileges in order for the delete operation to work correctly.

SEE ALSO

regserv(1), perm(1),
delete(2) in the *Programmer's Reference Manual.*

NAME

inquire – query the name server

SYNOPSIS

inquire object
inquire domain

DESCRIPTION

The *inquire* command retrieves information from the name server's data base.

If an *object* is specified, the name server searches its data base for the particular entry. If the object is found, the attributes associated for that object are displayed. If the object is not found, an error message is presented to the user.

If a *domain* is specified, the name server searches its data base for the particular entry. If the domain is found, then every name within the domain is returned whether it is an object, a sub-domain, or a remote domain. If the name is an object, or a remote domain its attributes are displayed as above. In the case of the remote domain, however, the name is suffixed with a '>' (a greater than sign) to indicated remote domain. If a sub-domain is displayed, it is suffixed with a '/' (forward slash). Since there are no attributes associated with a domain, none are displayed.

The user must have read permission on the specified object or domain.

SEE ALSO

regserv(1), rreg(1),
inquire(2) in the *Programmer's Reference Manual.*

NAME

perm – change permissions

SYNOPSIS

perm permissions object
perm permissions domain

DESCRIPTION

perm changes the permissions on an object or domain according to the argument *permissions*. The *permissions* must be specified as an octal number between 000 and 777 inclusive. The octal digits correspond (from left to right) to 'owner', 'group', and 'others' respectively. Within each octal digit, the bits correspond (from left to right) to read permission, write permission, and execute permission respectively. A 1 indicates permission, a 0 indicates that the associated permission is denied. For example, the permission below indicates that the owner has read, write, and execute privileges; the group has read and execute privileges; and others have only execute permission:

Owner	Group	Other	
rwx	rwx	rwx	
111	101	001	= 751 octal

Only the owner of the specified object or domain may change the permissions.

EXAMPLES

To set the access rights on an object called *lpr*, within the current context to the permission specified above, the user would type:

perm 751 lpr

SEE ALSO

cown(1), cgrp(1),
perm(2) in the *Programmer's Reference Manual.*

NAME

pwc – print working (current) context

SYNOPSIS

pwc

DESCRIPTION

Print current context displays the current context, as set by the *ccxt* command, to the user.

SEE ALSO

ccxt(1).

NAME

regserv – register a service with the name service

SYNOPSIS

regserv service host port

DESCRIPTION

regserv registers a service (an object) with the name service.

The *service* is the name of the service to be registered. It may either be an absolute name (i.e. begin with '/'), or a relative name. If a relative name is specified, the name is prefixed with the current context before the registration process begins. If an absolute name is specified, the name is taken literally and is not prefixed by the current context.

The *host* is the name of the host computer where the service resides.

The *port* is the port number on *host* where the service is waiting for requests.

EXAMPLES

To register a service named **man** on a host named **tinker** that is waiting for requests at port **1234**, the user would type:

regserv man tinker 1234

SEE ALSO

ccxt(1), pwc(1),
regserv(2), in the *Programmer's Reference Manual.*

NAME

<user command> – any user command

SYNOPSIS

<command> [=N] <arg>

DESCRIPTION

The user commands are those commands that are implemented by the services registered with the name service. Commands like, *lpr, mv, cd,* etc., are some examples. In any case, the commands are entered in the same way: the command name followed by its arguments. Those destined to use the name service are then forwarded to the name service, and finally to the service implementing the command. To the user, this forwarding and use of the name service is completely transparent.

The =N option is used to specify a particular object within a domain.

SEE ALSO

regserv(1),
Adding Clients(5) in the *Programmer's Reference Manual.*
Adding Servers(6) in the *Programmer's Reference Manual,*

APPENDIX B

NAMER: A Programmer's Reference Manual (PRM)

Introduction

The *Programmer's Reference Manual* (PRM) describes the commands available to the programmers of the name service (namer). Other documents that contain additional information are:

1) The *User's Reference Manual* (URM) describes each of the commands, available to the users of the naming system **NAMER**.

The functions described in this document are in alphabetical order simply for organizational purposes. Each function is described as follows:

NAME:
Gives the name of the function as it is to be entered followed by a secondary name. The secondary name is an English phrase describing the name.

SYNOPSIS:
Describes how the function is to be used. Some conventions used under the SYNOPSIS are:

1) Boldface strings are to be interpreted exactly as they appear.

DESCRIPTION:
Describes how to use the function and gives a brief summary of how the function works.

EXAMPLES:
Shows some simple examples of function use.

RETURN VALUES:
Discusses the values set when the program (command) terminates.

DIAGNOSTICS:
Discusses the error messages that may be produced during the execution of the function.

SEE ALSO:
Points to related information.

The Current Context

The name service maintains names in a hierarchical name space. At the top of the hierarchy is the root. The root is indicated by a leading '/' (forward slash) on a name and all (absolute) names begin with a '/'. In this scheme names can become quite long. As a result, a *current context* is maintained that is analogous to the current directory under UNIX. Whenever relative names are specified (i.e. names that do not begin with a leading '/'), the name is prefixed with the current context before the name space is traversed. If an absolute name is specified, the name is not prefixed with the current context.

To refer to objects or domains within the name space, some *pathname* for that entity must be provided. An absolute path name begins with '/' (the root). To refer to a specific domain within the hierarchy then, all domains names from the root domain leading to the domain in question are concatenated to the root and each is separated by a '/'. If an object within the domain is to be referenced, the object name is then concatenated to the *path* of domain names, again separated by a '/'. This string of names separated by slashes is the *pathname* of an object. If the pathname of an object does not begin a leading '/', it is a *relative* pathname and is always considered relative to the current context. That is, the relative name is prefixed by the current context before the name space is traversed.

NAME

intro – introduction to functions and error conditions

SYNOPSIS

#include nmerr.h

DESCRIPTION

This section describes each of the error conditions that may arise during the execution of a name service function. Each of these error conditions are represented by returning a negative integer from the function that caused the error.

These error conditions can also result from a user function.

-1 NOPEN_SOCK Can't open socket

The system call that is used during socket initialisation to reserve a socket for the name service failed. The socket may already be reserved.

-2 NBIND_PORT Can't bind socket

The socket can not be bound to the specified port. Perhaps the socket is already being used.

-3 NNAME_SOCK Can't get socket name

The system UNIX system call used to retreive the name of a socket (i.e. the port number it was bound to) failed. The socket descriptor passed to this system call may be incorrect.

-4 BADREAD Can't read from socket

The system call to read information from a socket failed. The socket may no longer be bound to a port or the socket descriptor used may be corrupted.

-5 UKN_HOST Unknown host

The host specified during a socket read or write is not known. The portinfo structure may be corrupted.

-6 NSPAC_SOCK Insufficient memory

During socket initialisation memory must be reservd for the socket name (port), its associated buffer and other information. This error indicates that there is insufficient memory to allocate.

-7 NOLISTEN Can't listen on socket

In order to accept messages, the program must *listen* on the socket for incoming messages. The socket descriptor may be corrupted so the read operation failed.

-8 NOCLOSE Can't close socket

The system call used to close a socket after use failed. The portinfo structure or the socket descriptor may be corrupted.

-9 NOCONNECT Can't connect stream sockets

An attempt to connect stream sockets failed. The socket on the other end of the connection may not exist. It may have been already closed or it may not be accepting connections.

-10 BADMSGTYPE Bad message type to process

The message received by the SS is corrupted; the message type can not be recognised. The

message sent may not have been constructed by a *mkcmsg()* or *mknmsg()* call.

-11 NOACCEPT Can't accpet messages
The system call to read the message from a socket after a *listen()* failed. The socket descriptor may be corrupted.

-12 BADWRITE Can't write to socket
An erro occured while trying to write a message from one socket to another. The socket descriptors may be corupted.

If one of the above error conditions arise, there is a serious problem. As a programmer or user, you have very little control over these conditions. If the problem persists, contact the systems programmer for help.

-13 BADSERVCMD Illegal name service command
An attempt to execute a name service control command that has not been implemented. Try some different commands to complete your task; or implement the one you would like.

-14 SERVNOTFOUND Service not found
The service specified is not registered with the name service. Check your spelling. Use the *inquire* command to check the data base. Check your current context.

-15 NOPRIV Insufficient privilege
The permissions on the object or domain will not allow the operation to be performed. Use the *perm* command to change the permissions to the appropriate settings.

-16 LEVNOTFOUND Name level (domain) not found
During the mapping process, a domain specified in the path name was not found in the data base tree.

-17 ILLEGNAME Illegal name
The specified name is not a legal path name. All domains within a path name must be separated by a '/'.

-18 TRUNCNAME Name truncated
The name specified was longer than NAMELEN (see file ncomm.h). As a result, the name was truncated in order to be inserted into the data base.

-19 SPACENOTEMPTY Name space not empty
An attempt to delete a domain from the name space can not be completed because the specified domain contains child nodes.

-20 NMLEVEXIST Name level already exists
An attempt to add an already existing name to the name space can not be performed. Change the name of the object specified.

-21 NOBJECTS No objects in domain
The path name does not contain an object, only domains. Objects must be specicifed in the path name for the operation to be completed correctly.

-22 PATHNOTDOMAIN Path name is not contain a domain
Part of the specified path name (except the last component) is not a domain so name space

mapping can not proceed.

-23 NOTOBJ Name specified is not an object

The name specified must be an object in order for the command to work correctly

-24 NOOBJINDOM No objects in domain

There are no objects in the specified domain. Mapping can not work correctly in this case.

-25 REQFRWRD Request forwarded

The object or domain specified is no registered with this instance of the server side. The request has been forwarded to the next instance of namer. This is more of an informational message rather than an error condition. If after searching the subsequent SS's, the error **SEVNOTFOUND** is returned.

-26 UKNODETYPE Unknown node type

During the mapping process, a node was encountered of unknown type. This is a serious problem. The node may not have been inserted into the data base via the *regserv()* or *rreg()* commands.

NAME

cgrp – change group

SYNOPSIS

```
#include ncomm.h

int cgrp(service,group,aopt,opt)
char *service;
char *group;
char *aopt[MAXCOPT];
int opt;
```

DESCRIPTION

Cgrp() changes the group of an object or domain. *Service* is the name of the object or domain within the current context of which the group is to be changed. This may be a relative or full path name. *Group* is the new group to which *service* is to be changed. *Aopt* and *opt* are both used for command options. *Aopt* is used for options that contain arguments; currently however there are no options of this type implemented. *Opt* is a bit string indicating what options (without arguments) are turned on. Again, there are no options of this type currently implemented.

The user must be the current owner of the specified *service* in order for the change group function to work correctly.

RETURN VALUES

Cgrp returns one (1) if it was successful in changing the group; otherwise, a value less than zero is returned. The description of the error values (i.e. values less than zero) can be found in the introduction of the *Programmer's Reference Manual.*

DIAGNOSTICS

The diagnostics are all self explanatory. They are the result of an error condition arising when the name services' data base was being accessed. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

cgrp(1) in the *User's Reference Manual.*

NAME

cown – change owner

SYNOPSIS

```
#include ncomm.h

int cown(service,owner,aopt,opt)
char *service;
char *owner;
char *aopt[MAXCOPT];
int opt;
```

DESCRIPTION

Cown() changes the owner of an object or domain. *Service* is the name of the object or domain within the current context of which the owner is to be changed. This may be a relative or full path name. *Owner* is the new owner to which *service* is to be changed. *Aopt* and *opt* are both used for command options. *Aopt* is used for options that contain arguments; currently however there are no options of this type implemented. *Opt* is a bit string indicating what options (without arguments) are turned on. Again, there are no options of this type currently implemented.

The user must be the current owner of the specified *service* in order for the change owner function to work correctly.

RETURN VALUES

Cown returns one (1) if it was successful in changing the owner; otherwise, a value less than zero is returned. The description of the error values (i.e. values less than zero) can be found in the introduction of the *Programmer's Reference Manual.*

DIAGNOSTICS

The diagnostics are all self explanatory. They are the result of an error condition arising when the name services' data base was being accessed. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

cown(1) in the *User's Reference Manual.*

NAME

delete – delete an object or domain

SYNOPSIS

```
#include ncomm.h

int delete(service,aopt,opt)
char *service;
char *aopt[MAXCOPT];
int opt;
```

DESCRITPTION

Delete removes an object or domain from the name servers' data base. That is, it is the opposite of the *regserv()* or *rreg()* functions. *Service* is the name of the object or domain that is to be removed. This may be a relative or an absolute name.

Aopt and *opt* are both used for options to the *delete* command. *Aopt* is used for options that contain arguments. Currently, however, there are no options of this type implemented. *Opt* is a bit string indicating what options (without arguments) are turned on. Currently, the only options implemented for the *delete* command is the interactive option. Setting the leat significant (right most) bit sets the interactive option. When this option is set, the user will be prompted for verification before the name specified is deleted.

The user must have write permission on the name that is to be deleted.

RETURN VALUES

Delete returns one (1) if it was successful in deleting the service; otherwise, a value less than zero is returned. The description of the error values (i.e. values less than zero) can be found in the introduction of the *Programmer's Reference Manual.*

DIAGNOSTICS

The diagnostics are all self explanatory. They are the result of an error condition arising when the name services' data base was being accessed. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

delete(1) in the *User's Reference Manual.*

NAME

inquire – query the name server

SYNOPSIS

```
#include ncomm.h

int inquire(name,aopt,opt)
char *name;
char *aopt[MAXCOPT];
int opt;
```

DESCRIPTION

The *inquire* command retrieves information from the name server's data base. The *name* is the name of the object or domain of which information is requested. This may either be a relative or an absolute name.

If an *object* is specified, the name server searches its data base for the particular entry. If the object is found, the attributes associated for that object are displayed. If the object is not found, an error message is presented to the user.

If a *domain* is specified, the name server searches its data base for the particular entry. If the domain is found, then every name within the domain is returned whether it is an object, a sub-domain, or a remote domain. If the name is an object, or a remote domain its attributes are displayed as above. In the case of the remote domain, however, the name is suffixed with a '>' (a greater than sign) to indicated remote domain. If a sub-domain is displayed, it is suffixed with a '/' (forward slash). Since there are no attributes associated with a domain, none are displayed.

The user must have read permission on the specified object or domain.

Aopt and *opt* are both used for options. *Aopt* is used for options that contain arguments. Currently, however, there are no options of this type implemented. *Opt* is a bit string indicating what option (without arguments) are turned. Again, there are no options of this type currently implemented.

RETURN VALUES

Inquire returns one (1) if it was successful in retrieving information; otherwise, a value less than zero is returned. The description of the error values (i.e. values less than zero) can be found in the introduction of the *Programmer's Reference Manual.*

DIAGNOSTICS

The diagnostics are all self explanatory. They are the result of an error condition arising when the name services' data base was being accessed. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

inquire(1) in the *User's Reference Manual.*

NAME

perm – change permissions

SYNOPSIS

```
#include ncomm.h

int cown(service,perms,aopt,opt)
char *service;
char *perms;
char *aopt[MAXCOPT];
int opt;
```

DESCRIPTION

Perm() changes the owner of an object or domain. *Service* is the name of the object or domain within the current context of which the owner is to be changed. This may be a relative or full path name. *Perms* is the new perms to which *service* is to be changed.

Aopt and *opt* are both used for command options. *Aopt* is used for options that contain arguments; currently however there are no options of this type implemented. *Opt* is a bit string indicating what options (without arguments) are turned on. Again, there are no options of this type currently implemented.

The user must be the current owner of the specified *service* in order for the change permission function to work correctly.

RETURN VALUES

Perm returns one (1) if it was successful in changing the permissions; otherwise, a value less than zero is returned. The description of the error values (i.e. values less than zero) can be found in the introduction of the *Programmer's Reference Manual.*

DIAGNOSTICS

The diagnostics are all self explanatory. They are the result of an error condition arising when the name services' data base was being accessed. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

perm(1) in the *User's Reference Manual.*

NAME

regserv – register a service with the name service

SYNOPSIS

```
#include ncomm.h

int regserv(service,host,port,aopt,opt)
char *service;
char *host;
int port;
char *aopt[MAXCOPT];
int opt;
```

DESCRIPTION

Regserv registers a service (an object) with the name service.

The *service* is the name of the service to be registered. It may either be an absolute name (i.e. begin with '/'), or a relative name. If a relative name is specified, the name is prefixed with the current context before the registration process begins. If an absolute name is specified, the name is taken literally and is not prefixed by the current context.

The *host* is the name of the host computer where the service resides.

The *port* is the port number on *host* where the service is waiting for requests.

Aopt and *opt* are both used for options. *Aopt* is used for options that contain arguments. Currently, however, there are no options of this type implemented. *Opt* is a bit string indicating what option (without arguments) are turned. Again, there are no options of this type currently implemented.

EXAMPLES

To register a service named **man** on a host named **tinker** that is waiting for requests at port **1234**, you would use

```
regserv("man","tinker",1234,' ',' ');
```

in your program.

RETURN VALUES

Regserv returns one (1) if it was successful in registering the specified service; otherwise, a value less than zero is returned. The description of the error values (i.e. values less than zero) can be found in the introduction of the *Programmer's Reference Manual.*

DIAGNOSTICS

The diagnostics are all self explanatory. They are the result of an error condition arising when the name services' data base was being accessed. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

rreg(2),
ccxt(1), pwc(1), regserv(1), in the *User's Reference Manual.*

NAME

rreg – register remote domain

SYNOPSIS

```
#include ncomm.h
int rreg(domain,host,port,aopt,opt)
char *domain;
char *host;
int port;
char *aopt[MAXCOPT];
int opt;
```

DESCRIPTION

Rreg registers a remote domain with the name service. A remote domain indicates that the server side (SS) that manages the names under the specified domain is located elsewhere in the network.

The *domain* is the domain name that is managed by another instance of the SS. It may be specified as an absolute name (i.e. begin with '/'), or as a relative name. If a relative name is given, the domain is prefixed with the current context before the registration process begins. If an absolute name is given, the name is taken literally and is not prefixed with the current context.

The *host* is the host computer where the SS managing the specified domain resides.

The *port* is the port number on *host* where the SS managing the specified domain may be contacted.

Aopt and *opt* are both used for options. *Aopt* is used for options that contain arguments. Currently, however, there are no options of this type implemented. *Opt* is a bit string indicating what option (without arguments) are turned. Again, there are no options of this type currently implemented.

EXAMPLES

If all print services are managed by an instance of the SS on a host named tinker at port 1234; and one wishes to register this with another instance of the SS that is used by the users, one may use:

```
rreg("print","tinker",1234,' ',' ');
```

within a program.

RETURN VALUES

Rreg returns one (1) if it was successful in registering the specified remote domain; otherwise, a value less than zero is returned. The description of the error values (i.e. values less than zero) can be found in the introduction of the *Programmer's Reference Manual.*

DIAGNOSTICS

The diagnostics are all self explanatory. They are the result of an error condition arising when the name services' data base was being accessed. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

regserv(2),
rreg(1), regserv(1), ccxt(1), pwc(1), in the *User's Reference Manual.*

NAME

itoa – convert integer to ascii string

SYNOPSIS

```
int itoa(num,str)
int num;
char *str;
```

DESCRIPTION

Itoa converts an integer to an ascii string.

Num is the integer value that is to be converted to a string.

Str is the location where the converted integer is stored. It is assumed that sufficient space has been reserved for the converted string.

SEE ALSO

reverse(3)

NAME

nmcpy — copy a name structure to a string

SYNOPSIS

```
#include ndb.h

int nmcpy(to,from)
char *to;
struct nmspace *from;
```

DESCRIPTION

Nmcpy() copies a name space structure into a single string in order to send it as a message across a network.

From is the name space structure to be copied.

To is the string to which the structure is to be copied. It is assumed that sufficient space has been reserved for the structure in the string.

SEE ALSO

nmput(3)

NAME

nmput – print name structure

SYNOPSIS

```
int nmput(str,hd)
char *str;
int *hd;
```

DESCRIPTION

Nmput prints a *nameatt* structure contained in the single string *str. Str must have been constructed by nmcpy()* prior to being used in *nmput()*.

Hd indicates whether or not to print a header above the information within the string. If *hd* is set then a header is printed, otherwise only the contents of the string is printed. This option is turned off during each call of *nmput()*. This is to allow the routine to be called multiple times to produce a table with only one heading.

SEE ALSO

nmcpy(3)

NAME

pmsg – print contents of a message

SYNOPSIS

```
pmsg(message)
char *message;
```

DESCRIPTION

Pmsg() prints the contents of the *message* to standard output. It first checks for the type of message in the header and then prints the appropriate fields.

RETURN VALUES

Pmsg returns one (1) if it was successful in printing the message; otherwise, a value less than zero is returned and are described below:

BADMSGTYPE Bad message type.
The type of message can not be determined. It may not have been created via a *mknmsg()* or *mkcmsg()* call.

DIAGNOSTICS

The diagnostics are all self explanatory. They are the result of an error condition arising when the name services' data base was being accessed. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

putnmspace(3), nmput(3)

NAME

putnmspace – print the current name space

SYNOPSIS

```
#include ndb.h

int putnmspace(ndb)
struct nmspace *ndb;
```

DESCRIPTION

Putnmspace prints an pre-order traversal of the name space. While traversing the name space tree, if the encountered node is an object or a remote domain, its attributes are also printed. If a domain is encountered, a message indicating the node has no attributes is printed indicating a domain.

RETURN VALUES

Putnmspace returns one (1) upon completion.

DIAGNOSTICS

The diagnostics are all self explanatory. They are the result of an error condition arising when the name services' data base was being accessed. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

pmsg(3)

NAME

reverse – reverse the bytes in a string

SYNOPSIS

```
reverse(str)
char *str;
```

DESCRIPTION

Reverse reverses the order of bytes in an ascii string.

Str is the string to be reversed. It is also used to return the reversed string.

SEE ALSO

itoa(3)

NAME

closesock – return a socket to the system

SYNOPSIS

```
#include ncomm.h

int closesock(thisport)
struct portinfo *thisport;
```

DESCRIPTION

Closesock() returns a socket and name to the system. The socket must have been created by a *irdsock()* or *iwrsock()* call in order for the function to work correctly.

Thisport is a structure containing the necessary information to close the socket. It is built by the *irdsock()* and *iwrsock* routines.

RETURN VALUES

Closesock() returns a 1 if completed successfully, otherwise the following error condition is returned:

NOCLOSE The socket was unable to be closed correctly. The system call used to close the socket failed: the structure containing the socket information may be corrupted.

DIAGNOSTICS

The diagnostics are all self explanatory. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

irdsock(4), iwrsock(4)

NAME

irdsock – initialize a socket to read from

SYNOPSIS

```
#include ncomm.h

int irdsock(thisport,pnum)
struct portinfo *thisport;
int pnum;
```

DESCRIPTION

Irdsock() initializes a socket to read from. This routine should be used if any communications is to be done with or within the name service. The socket created is a STREAM socket (SOCK_STREAM) and is in the internet domain (AF_INET). After the socket is initialized, it is assigned a name (a port number in this case) as indicated by *pnum*.

Thisport is a structure containing information about the created socket including the port number assigned to the socket, and a 1K buffer used to hold messages received on the socket.

Pnum is the port number (name) that is assigned to the created socket. Other programs may use this name to send messages to the created socket. It should be noted that this number is only a request; it does not guarantee the port will be reserved for the created socket. If the port can not be assigned to the socket for any reason (e.g. it is already in use), an error message is printed to standard error and the routine returns an error code.

If it is not critical to assign a specific port number, a value of zero (0) may be passed to pnum and the system call *bind()* will pick an available port number. In any case, if a port number is successfully reserved for the socket, it is returned within the structure *thisport*.

RETURN VALUES

If the socket is successfully initialized and a port number assigned, *irdsock()* returns a 1. Otherwise, it returns one of the following error conditions:

NOPEN_SOCK The socket can not be opened. It may already be in use.

NBIND_PORT A port number can not be bound (assigned) to the opened socket. The requested port number may already be in use.

NNAME_SOCK The assigned port number can not be verified. The structure containing the socket information may be corrupted in some way.

NOLISTEN The socket/port can not be made ready to accept messages. The system call *listen()* failed.

DIAGNOSTICS

The diagnostics are all self explanatory. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

iwrsock(4)

NAME

iwrsock – initialise a socket to write to

SYNOPSIS

```
#include ncomm.h

int iwrsock(sock,host,port)
struct portinfo *sock;
char *host;
int port;
```

DESCRIPTION

Iwrsock() initialises a socket to which to write. This routine should be used if any communications is to be done with or within the name service. The socket created is a STREAM (SOCK_STREAM) socket in the internet (AF_INET) domain. Unlike there is no name (port number assigned) assigned to the created socket since no other progams will send messages to this socket. That is, the socket is created for write only. In addition, the created socket can only be used to send messages to the It may not be used to send information to more than one desitination, it is strictly point to point.

Sock is a structure containing information about the created socket including a 1K buffer used to hold messages for sending.

Host is the name of the host to which the messages will be sent.

Port is the port number on *host* to which messages will be sent.

RETURN VALUES

If the socket is successfully initialised *iwrsock()* returns a 1. Otherwise, it returns one of the following error conditions:

NOPEN_SOCK The socket can not be opened. It may already be in use.

UKN_HOST The host to which messages are to be sent is not known. Check spelling and /etc/hosts.

DIAGNOSTICS

The diagnostics are all self explanatory. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

irdsock(4), sendrel(4)

NAME

mkcmsg – build a control message

SYNOPSIS

```
#include ncomm.h

int mkcmsg(msg1,service,host,pnum,cmd,arg,aopt,opt)
struct cmsg *msg1;
char *service;
char *host;
int pnum;
u_short cmd;
char *arg;
char *aopt[MAXCOPT];
int opr;
```

DESCRIPTION

Mkcmsg() builds a naming control message to be sent to the name service. All administration functions that must contact the name service create messages using this function.

Msg1 is the message that the function builds.

Service is the name of the service to which the control command applies. This may be either a full name or a relative name. In addition, this may be a domain specification or an object. It should be viewed as simply the name to which the command applies.

Host is the name of the host on which the service resides. This is typically used if a new service is being registered.

Pnum is the port number where the specified service may be contacted. Again, this is used for service registration.

Cmd is the control command the SS is to perform on the specified service.

Arg is any argument that may be associated with the control command. This is used for commands such as *cown, cgrp,* and *perm* to specify the new owner, new group, and permissions respectively.

Aopt is an array that contains the arguments to options that require options.

Opt specifies command options that do not require options. Currently the only option of this type available is for the delete command and is the verify option. To turn this option on, the least significant bit must be set.

RETURN VALUES

Upon completion, the function returns a 1.

DIAGNOSTICS

The diagnostics are all self explanatory. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

mknmsg(4)

NAME

mknmsg – build a message to send be sent within the name service

SYNOPSIS

```
#include ncomm.h

int mknmsg(msg1,acount,avect,aopt,opt)
struct cmsg *msg1;
int acount;
char *avect[];
char *aopt[MAXNOPT];
int opr;
```

DESCRIPTION

Mknmsg() builds a message to be sent within the name service. This function is used by the user function to create messages.

Msg1 is the message that is created.

Acount indicates how many arguments are present in the command line. Since this function is typically called by user functions invoked on the command line, the structures a 'argc' and 'argv' have been maintained.

Avect contains the arguments from the command line. The first entry in this array is assumed to be the user function that is to be executed by the remote service. All subsequent entries in this array are assumed to be user command arguments. These may be user command options as well (if required by the command).

Aopt is an array that contains the arguments to options that require options.

Opt specifies command options that do not require options. Currently, the only option implemented is to specify a specific object within a domain.

RETURN VALUES

Upon completion, the function returns a 1.

DIAGNOSTICS

The diagnostics are all self explanatory. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

mkcmsg(4),
user(1) in the *User's Reference Manual.*

NAME

nmaccept – read a message from a socket

SYNOPSIS

```
#include ncomm.h
#include nmerr.h

int nmaccept(port,len)
struct portinfo *port;
int len;
```

DESCRIPTION

Nmaccept() reads a message from a socket specified in *port*. The message is inserted into the buffer associated with the specified socket.

Port contains the socket and port information necessary to read a message including a 1K buffer to store the message. This structure must have been previously been built by an *irdsock()* call for the read operation to work correctly.

Len is the maximum length of the message that is allowed to be read. This should be the constant MSGLEN as specified in the file ncomm.h. Under no circumstances should it be greater than this constant since this is the buffer size.

RETURN VALUES

If a message is successfully read, the number of bytes read is returned (a positive number), otherwise one of the following error codes is returned:

NOACCEPT Can't accept messages on the specified socket. The structure *port* may be corrupted.

BADREAD The read operation failed. The buffer size may be insufficient or the structure *port* may have been corrupted.

DIAGNOSTICS

The diagnostics are all self explanatory. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

irdsock(4), iwrsock(4)

NAME

nmack – send an acknowledgement to the user

SYNOPSIS

```
#include ncomm.h

int nmack(msg,ackmsg,val)
char *msg;
char *ackmsg;
int val;
```

DESCRIPTION

Nmack() sends an acknowledgement back to the issuer of the command. This may either be a success ack, or a failure ack.

Msg is the message to which the SS is replying.

Ackmsg is an acknowlegement message to be sent along with the ack. This is normally used in error conditions if a message needs to be printed as a result of an exceptional condition.

Val is the value of the ack. That is, it indicates whether or not the operation was a success or a failure. A failure is indicated by assigning *val* one of the error values described in the introduction of the *Programmer's Reference Manual.*

RETURN VALUES

Upon successful completion, a 1 is returned; otherwise the error condition that resulted from the sending of the message is returned (see sendrel(4)).

DIAGNOSTICS

The diagnostics are all self explanatory. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

sendrel(4), nmreply(4), nmforward(4)

NAME

nmforward – forward a message to another server

SYNOPSIS

```
#include ncomm.h

int nmforward(msg,serv)
char *msg;
char *serv;
```

DESCRIPTION

Nmforward() forwards a message to another server. This function is used both for sending to other instances of the SS and to services registered with the name service.

Msg is the message to forward. This is (although it doesn't have to be) the original message sent to the SS. However, the destination host and destination port entries in the message header are updated so that the message is sent to its (next) correct destination.

Serv is the service to which the message is being forwarded.

RETURN VALUES

Upon successful completion, a 1 is returned; otherwise the error condition that resulted from the sending of the message is returned (see sendrel(4)).

DIAGNOSTICS

The diagnostics are all self explanatory. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

sendrel(4), nmack(4), nmreply(4)

NAME

nmreply – send an reply to the user

SYNOPSIS

```
#include ncomm.h

int nmreply(msg,rmsg)
char *msg;
char *rmsg;
```

DESCRIPTION

Nmreply() sends a reply back to the issuer of the command. This function is used in response to queries to send information back to the user.

Msg is the message to which the SS is replying.

Rmsg is the information to be sent back to the user.

RETURN VALUES

Upon successful completion, a 1 is returned; otherwise the error condition that resulted from the sending of the message is returned (see sendrel(4)).

DIAGNOSTICS

The diagnostics are all self explanatory. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

sendrel(4), nmack(4), nmforward(4)

NAME

sendrel – send a message to the name server

SYNOPSIS

```
#include ncomm.h
#include ndb.h

int sendrel(msg)
char *msg;
```

DESCRIPTION

Sendrel() performs the actual sending of a message from one socket to another within the network. The function first checks the header of the message to determine its type. This ensures that the size of the message being sent is the same as the message contents so no extra bytes are sent over the network.

Msg is the message to be sent.

RETURN VALUES

If the message is sent correctly, a 1 is returned, otherwise one of the following error messages is returned:

NOCONNECT Unable to connect to the remote socket. The remote socket mayu already have been closed.

BADWRITE The write of the message failed. The meaaage contents may be corrupted in some way.

BADMSGTYPE The message type to be sent was unrecognised. The message may be corrupted.

DIAGNOSTICS

The diagnostics are all self explanatory. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

nmack(4), nmreply(4), nmforward(4)

NAME

AddClients – add a new user function

SYNOPSIS

#include ncomm.h

DESCRIPTION

When new commands and/or services are implemented, it will be necessary to create the programs that request the name service to map the command to its appropriate service. These are the client programs.

The function that talks to the name service is called **nclient()**. This function is called by all of the user interface *user functions* routines. So, to create a new user function, only the user interface that calls nclient() must be created (as long as the intended service knows how to implement this new command). In addition, **nclient()** can be called from the command line. That is, only a shell script (one line at that) need be written.

EXAMPLES

The function **nclient()** expects 3 types of arguments:

1) The command to perform.
2) Arguments of the command (may be more than 1).
3) The domain name of the service implementing the command.

So, to contact the name service to map a command **mv** within the domain **file** with arguments **F1, F2, F3** one could type

nclient file mv F1 F2 F3

Note that the domain is the first argument.

DIAGNOSTICS

The diagnostics are all self explanatory. Further explanations can be found in the introduction of the *Programmer's Reference Manual.*

SEE ALSO

AddingServers(6)

NAME

AddServers – add a new service

SYNOPSIS

#include ncomm.h
#include nmerr.h

DESCRIPTION

When new services that need to be registered with the name service are implemented, they must be written to recognise and receive messages from the name service. This section of the manual gives an example of how such a service may be written.

EXAMPLES

Below is an example of the C code necessary to implement a service:

```
#include <stdio.h>
#include "ncomm.h"
#include "nmerr.h"


main()
 {
  struct portinfo myport;       /* port information for created sockets */
  int    irdsock();             /* routine inits a read socket */


/************************************************************************/
/*  Create a socket from which the name server can read from.       */
/************************************************************************/
  if ((irdsock(&myport,0)) < 0)
   {
    fprintf(stderr,"service==> Error initialising port.\n");
    exit(1);
   }
  printf("The services' port number is %d\n",myport.portnum);


/******************************************************/
/* Loop forever to read and process messages.      */
/******************************************************/
  while (1)
   {
    nmaccept(&myport,1024);
    processmessage(myport.msgbuf);
   }

  closesock(&myport);
```

```
}
```

The first thing the code does is initialise a socket to read from. This allows other programs to send it subsequent messages. Next it prints out the port number assigned to the service as verification. Finally, it simply waits for messages and when one arrives, it processes it (i.e. performs the command).

Note that before this service can be used in conjunction with the name service, it must be registered with the name service. This can be done in one of two ways:

1.) If you wish to register each service at the terminal (i.e. type in the register command for each service) then after the port number is printed out, typing the following command will register the service:

regserv service_name host port

2) If typing in the above command seems a bit tedious, you nay add code to the above service to register itself each time the service is started. Simply add the following line immediately after the print statement in the above code:

```
regserv("service_name","host_name",myport.portnum,' ',' ');
```

Now each time the service is started it registers itself. The last two arguments are the null string. Of course it may be a good idea to include some error checking to confirm that the service is indeed registered.

SEE ALSO

AddingClients(5)