Chapter 9. Identifiers (naming) in distributed systems

9.1. Introduction

Identification systems (often called naming systems) are at the heart of all computer system design. We have chosen to use the neutral term identifier rather than name because we want to be able to distinguish between various kinds of entities used to designate or refer to objects at all levels of the architecture and normal usage of the word name gives it a more specific connotation (see below). In spite of the importance of identifiers, no general unified treatment of them exists, either across computer science or within subareas such as programming, operating, or database management systems. These two statements are particularly true of distributed computing. A useful introduction to identification issues associated with operating systems, with applicability to distributed systems, is given by [Saltzer 78a]. Other introductions to the subject specifically aimed at distributed systems are contained in [Abraham 80, ISO 79, Livesey 79, McQuillan 78a-b, Pouzin 78, Shoch 78, Zimmerman 80]. It is premature (at least for ourselves) to attempt a definitive discussion of identification. Instead, we briefly review some general properties, design goals and issues associated with the topic, and then present some concrete examples in distributed systems. The central point we wish to make is that identification is not a topic to be treated lightly, in an ad hoc manner, or in isolation from other distributed system needs such as protection, error control, and resource management, although this often has been the case.

An *identifier* is a string of symbols, usually bits or characters, used to designate or refer to an object. These objects and their identifiers may be visible at the interfaces between layers of the Model, between modules of the same layer, or may be embedded and used strictly within a module. We are primarily concerned in this chapter with identifiers used at interfaces between layers. In particular, we focus on identifiers used with the objects defined by the DOS layer and their relationship with and use of lower level identifiers. The important use of identifiers, such as sequence numbers, between modules of the same layer for error control, synchronization, and resource management is discussed in many of the other chapters. The use of identifiers for protection is discussed in Chapter 10 and briefly later in this chapter. Some general characteristics of identification systems are:

- Identifiers are used for a wide variety of purposes, referencing, locating, scheduling, allocating, error controlling, synchronizing, and sharing of objects or resources (information units, communication channels, transactions, processes, files, etc.). One can use an identifier to refer to an object's value(s) or to refer to the object itself. Identifiers are used in either sense when constructing more complex objects out of simpler component objects. Associated with an identifier, implicitly or explicitly, is the *type* of the designated object [Chapter 2].

- Identifiers exist in different forms at all levels of a system architecture. [Shoch 78] has made a useful informal distinction between three important classes of identifiers widely used in distributed systems, *names, addresses, and routes*:

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

Names, addresses, and routes can occur at all levels of the architecture. Names used within the interprocess communication layer have often been called terms such as *ports, or logical or generic addresses* as distinct from *physical or specific addresses.* Where useful, we also follow this convention. If an identified object is to be manipulated or accessed, the identifier must be *mapped* using an appropriate *mapping function* and *context* into another identifier and ultimately the object as shown in figure 9-1. In general, an identifier is mapped in one *context* into another identifier that is subsequently evaluated in yet another context. Identifiers used at different levels referring to the same object must be *bound* together, statically or dynamically. Commonly one maps names into addresses and addresses into routes, although one could also map a name to a route and a route to an address, etc.

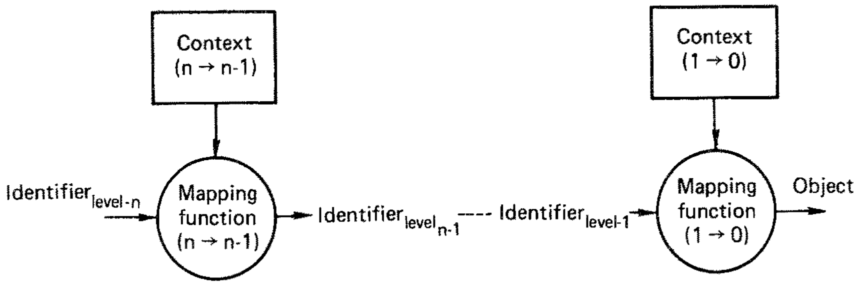


Figure 9-1: Mapping of identifiers through *n* levels to the represented object

- An identification system consists of the symbol alphabet(s) out of which one or more levels of identifiers can be formed, the structure or rules for forming identifiers, and the mapping/context mechanisms used to bind and *resolve* identifiers at one level into identifiers at lower levels. If objects can be relocated, shared, created, or destroyed, then mechanisms are needed for updating the appropriate contexts.

- Identification, protection, error control, diagnostic, and resource management mechanisms often interact. One has to be aware of how design goals or choices in one of these areas affect those in the others. The introduction of ideas such as data types associated with identifiers allow error control and user convenience also to be tied to the identification scheme.

- Choice of an identification scheme can affect whether or not it is possible, easy, or efficient to achieve the goals listed in the next section.

- There are many possible ways to designate the object(s) desired: by an explicit name or address (object x or object at x), by content (object(s) with value(s) y or values > expression), by source (all "my" files), broadcast identifier (all objects of a class), group identifier (all participants in subject V), by route (the object found by following path Z), by relationship to a given identifier (all previous sequence numbers), etc.

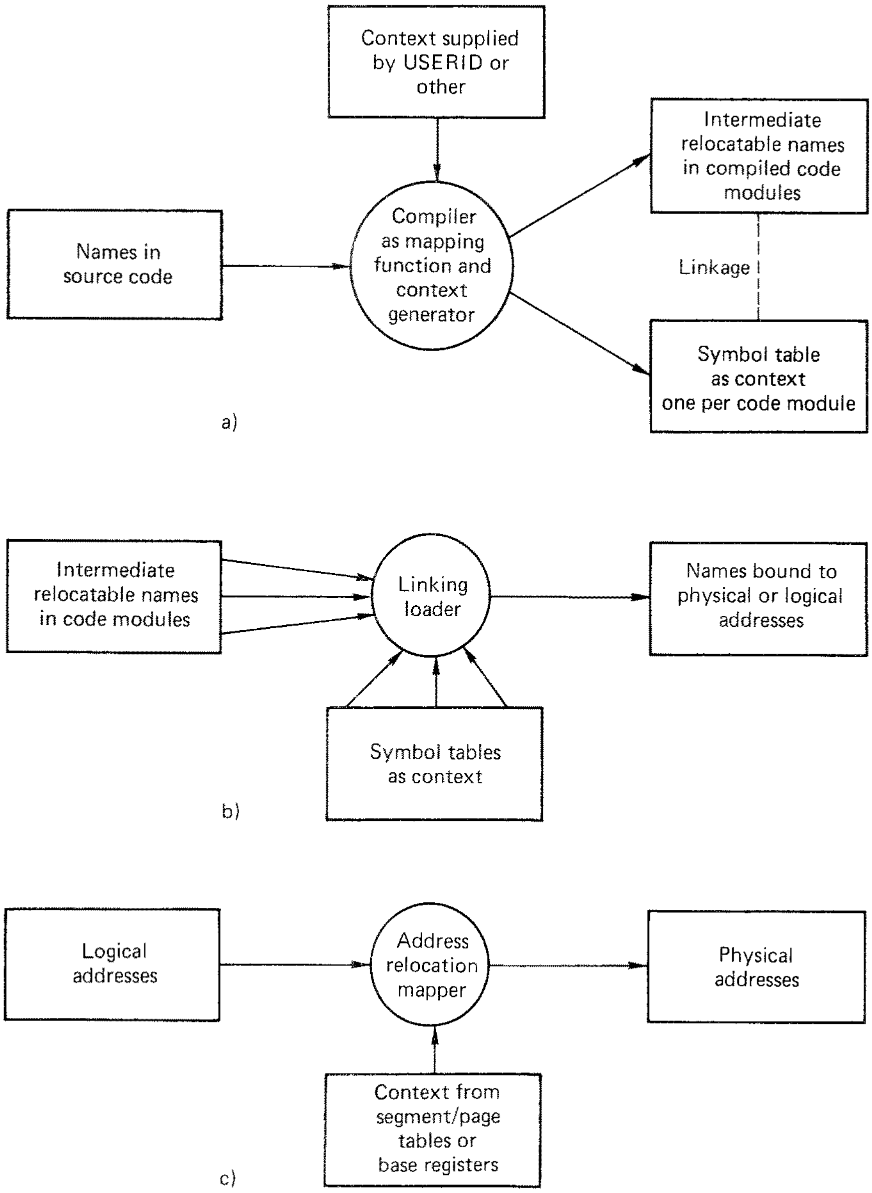

Context supplied by USERID or other
Names in source code
Compiler as mapping function and context generator
Intermediate relocatable names in compiled code modules
Linkage
Symbol table as context one per code module
a)
Intermediate relocatable names in code modules
Linking loader
Names bound to physical or logical addresses
Symbol tables as context
b)
Logical addresses
Address relocation mapper
Physical addresses
Context from segment/page tables or base registers
c)

Figure 9-2: Example of levels of identifier mapping in a programming/ operating system

As an illustration of the above points consider the familiar programming/operating system environment shown in figure 9-2. The programming language allows programmers to choose names from a very large symbolic name space for objects such as other programs or data structures in a form convenient for people. These names are translated into intermediate machine-oriented name forms that at load time are mapped with the aid of context (symbol tables) to locations in an address space. If static relocation is used, the intermediate names are bound to addresses in the processor's physical main memory and relocation at runtime is not possible. Runtime relocation is desirable to improve system utilization and to provide for virtual memory in a multiprogramming system. To meet these needs, another level of naming, mapping, binding can be introduced by having the loader bind the intermediate names to identifiers in a one dimensional logical-address space. These logical-addresses are then *dynamically bound* at runtime to physical-addresses with the mapping and context supplied by base registers or page tables. Even with this additional level of naming, problems still exist with respect to ease of sharing of program and data objects, and allowing them to grow dynamically without address conflicts. These problems can be resolved by adding yet another level of identification and context as in the Multics segmentation mechanism [Watson 70].

Below the level that a main memory address is produced, additional information, routing, address mapping, and control mechanisms may be involved to improve performance through use of one or more levels of memory caching. At higher levels of naming, the naming mechanisms of programming, file or information management systems, and those of address systems may be kept separate or be unified with significant usage and implementation implications [Saltzer 78a]. Protection and resource allocation mechanisms are usually tied to these mechanisms.

The important point of the above example is that by adding additional levels of identifiers and associated mapping and context, we are able to improve our ability to relocate, extend, or share objects. Or stated another way, gains are achieved if names do not have to be bound to addresses or routes until they are used, i.e. *dynamic binding*. Only recently have the needs for dynamic name binding and supporting mechanisms begun to be explored for use in distributed systems [McQuillan 78b].

The problems introduced to the design of identification systems by distribution result from the heterogeneous nature of component systems, each with its own possible local identification system for memory addressing, files, processes, and other objects; and problems introduced in maintaining distributed context and mapping information, compounded by delays and errors in message transmission and local system or network crashes. The questions that we want to address are:

- What type of identifiers are required at each of the levels of the distributed system architecture model to meet the goals below,
- Where should context be maintained and mapping performed between different levels of identifiers, and
- How should we deal with heterogeneity.

One set of answers to these questions is given in following sections.

9.2. Identifier goals and implications

There are many possible identification system goals for a distributed system, several being identical to those for nondistributed systems. Some of the more important ones and their implementation implications are:

- Efficient support of transaction as well as session oriented services and applications. In a *transaction* oriented system, the application has the identifiers of one or more resources on which it desires to perform a given operation and receive a reply, with no implication that these resources or the operation may ever be used again. In a *session* oriented system, on the other hand, it is assumed at the start that a stream of operations may be requested over time against an identified set of resources. Efficient support of transaction oriented applications and services implies that we want to minimize, at all levels of the Model, the delay and number of messages that must be exchanged before and after the actual message is sent to perform the desired function. With respect to an identification system, this means that we want to minimize the number of messages that must first be exchanged to initialize or map identifiers held at one level into identifiers directly usable by another level.

  This identifier mapping usually involves message exchanges and allocation and deallocation of identifier space and context. Minimizing this overhead requires large enough unique identifier spaces such that directly usable identifiers can be permanently allocated and permanent context can exist either self-contained within the identifier or elsewhere. Supporting efficient stream applications implies that we want to perform identifier mapping at most once before or during the first access.

- Support at least two levels of identifiers, one convenient for people and one convenient for machines. There should be a clean separation of mechanism for these two levels of identifiers. For each form of DOS identifier a related goal is that it have the same general structure for all types of resources. Other properties desired in these two forms of identifiers are somewhat different. A machine-oriented identifier should be a bit pattern easily manipulated and stored by machines and be directly useful with protection, resource management, and other mechanisms. A human-oriented identifier, on the other hand, is generally a human readable character string with mnemonic value. At some point in time and layer of the Model, human-oriented identifiers must be mapped into machine-oriented ones.

- Allow unique identifiers to be generated in a distributed manner. If a central unique identifier generator is required, complicating efficiency and reliability issues are introduced that do not occur if identifiers can be generated in a distributed manner. Structured identifiers are required to support this goal, although the structure does not have to be visible.

- Provide a system viewed as a global space of identified objects rather than one viewed as a space of identified host computers containing locally identified objects. Similarly, the identification mechanism should be independent of the physical connectivity or topology of the system. That is, the boundaries between physical components and their connection as a network, while technologically and administratively real, should be invisible, to the extent possible, and be recognized as logically artificial. If these boundaries are represented in identifiers, it should be within a unified identifier structure, such as a hierarchical form. (We do not intend that users cannot discover or influence an object's location.)

- Support relocation of objects. The implication here is that there be at least two levels of identifiers, a name and an address, and that the binding between them be dynamic. Also implied are mechanisms for updating the appropriate context for the required mapping when an object is moved.
- Allow multiple logically equivalent *generic* servers for particular resource types. The implication here is that a single identifier at one level can be dynamically bound to more than one address at a lower level. When mapping takes place one object is chosen according to some set of criteria associated with resource management, error control, or other constraints.
- Support use of multiple copies of the same object. If the value(s) of the object is only going to be read or interrogated, one set of constraints are imposed. If the value(s) can be written or modified, tougher constraints are imposed to achieve consistency between the contents of the distributed multiple copies and an ordering of reads and writes applied against the objects [Chapter 13]. Multiple copies are expected to be required frequently in distributed systems to achieve performance, reliability or robustness goals. More than one level of identifier and dynamic binding is implied.
- Allow multiple local user defined identifiers for the same object. What is implied here is that there be unique global identifiers for objects and at least one level of local identifier. Mechanism to bind local and global identifiers is required. The needed local context must be bound in turn to an object or human user, for example. An identification system must be able to switch context as appropriate.
- Support two or more resources sharing or including as components single instances of other objects, without identifier conflicts. This desire implies the need for local identification domains for each object and context switching mechanisms as above. For efficiency, the appropriate level of object granularity must be chosen. Granularity at the level of processes or servers, as in the Model, seems an appropriate level given current hardware architecture. Implicit in the last two goals is the need for a two dimensional identifier space consisting of a pair of identifiers *identifier-of-context, identifier-of-object.*
- Allow many different objects to share the same identifier. Such a need is useful to support broadcast, or group identifiers for conferencing or other applications. At least two levels of identifiers and one-to-many mapping are required.
- Minimize the number of independent identification systems needed across and within architecture levels. This requires recognition of commonality of need.
- All of the above goals may be desirable or be supported by different mechanisms at each layer of the Model, or be supported by mechanisms spanning layers as described later.

The sections below outline issues and mechanisms that can be used to create identification systems that meet one or more of the above goals. Whether an efficient identification system(s) can meet all the above goals for a distributed system with existing hardware/firmware components is an open question. Some of the goals are potentially contradictory such as the goal to achieve a transaction orientation and goals requiring multiple-levels of mapping.

9.3. Unique machine-oriented identifiers

It must be possible at some level of the Model to uniquely identify every object in the global system. On this foundation one can build higher level identification constructs to support the goals above. The question is how to create a *global* unique identifier space in a heterogeneous environment, where each *local* identification domain may use one or more different schemes to create identifiers unique within itself. Global identifiers are resolved within a global, possibly distributed, context; local identifiers are resolved within local contexts. One strategy for creating a global identifier space, called hierarchical concatenation in [Pouzin 78], is to create a unique identifier for each identification domain, usually a host computer, then to concatenate to the unique identifier for the domain the identifiers used within that domain. The RSEXEC system used this approach [Thomas 73]. This is one form of the pair (identifier-of-context, identifier-of-object-within-context). This particular scheme has the disadvantage that the form and length of identifiers used within heterogeneous domains may vary so widely that it may be awkward or inefficient for applications to be prepared to deal with them. It can also have the disadvantage that host computer boundaries are explicitly visible.

The alternative is to develop a standard, uniform global identifier form and space for all resources and some way to partition this global space among the local domains, which then map their local identifiers to these global identifiers. Binding of local identifiers to global identifiers can be permanent or temporary. This mechanism immediately introduces a level of indirection with some of the advantages desired: local identifiers can be chosen as most appropriate for their local language or operating system environment, several local identifiers can be mapped to the same global identifier or vice versa, and the global identifier can be reassigned if objects are relocated.

In order to outline the issues associated with creation of a uniform, unique, machine-oriented, global, identifier space, we now describe a scheme usable with the Model of Chapter 2. We call these entities *object-identifiers*. In review, the Model has the following properties:

- Processes are the fundamental communicating objects.
- All objects or resources are managed by server processes.
- Resources are created, manipulated, and destroyed by customer processes sending request messages to server processes and in turn receiving replies.

These properties are reflected in the identification scheme.

We envision the standard form for DOS object-identifiers as being defined by the Service-Support sublayer of the DOS [Section 2.4.4, Watson 80b, Fletcher 80] as a special data type. Object-identifiers can be unstructured or structured. In order to allow distributed generation, requires structured object-identifiers. It seems natural to allow each (possibly generic) server to generate object-identifiers for the objects it serves. This is allowed if object-identifiers have a structure with two components *server-identifier, unique-local-identifier.* To provide uniform identifiers, each of these components has a standard structure, which could be as simple as specifying a maximum variable length. We now discuss how each of these identifier components can be obtained, first the unique-local-identifier.

Each object type is represented at the interface by a data structure (usually abstract). A given server implements this structure as some real structure. A particular object exists as an instance of one of these real structures located at a particular place. This object, in the general heterogeneous case, will have a *local-implementation-identifier* unique within the local domain of the server. It is the responsibility of the server to create unique-local-identifiers according to the standard form, and to bind them to its local-implementation-identifiers: The idea is shown in figure 9-3. The server generates unique-local-identifiers in any convenient way, subject only to the constraints of the standard, such as a maximum length. Because different implementations of servers for the same type of object may desire to use different unique-local-identifier forms, it is best if holders of a global identifier do not attach meaning to them.

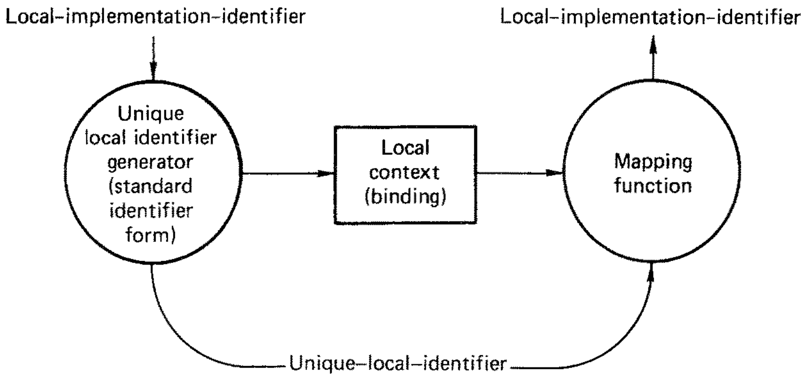

Local-implementation-identifier
Local-implementation-identifier
Unique local identifier generator (standard identifier form)
Local context (binding)
Mapping function
Unique-local-identifier

Figure 9-3: Generating and mapping unique-local-identifiers from/to local-implementation-identifiers

If there are several copies of an object, each instance has a unique local-implementation-identifier within the context of its server and location. When an object is relocated it gets a new local-implementation-identifier, and its implementation representation may change. However, in either of the two cases above, the multiple copies of an object or the relocated object, the object can still have the same object-identifier if the same server is involved. If the object can be relocated across servers, then additional constraints are imposed on identifier form and binding. Now we discuss how the server-identifier is obtained.

The IPC layer has to have identifiers for all processes in order to perform its message delivery service. We will call this IPC name an *address*. An address is bound to an object by routing context. Other terms have been used for this level of name such as *port*, *socket*, and *mail box*. Addresses are discussed further later. It is useful for processes to have more than one address, enabling parallel error and flow controlled conversations between two processes, partitioning of services, or other purposes (figure 9-4). It is also useful to associate one identifier (logical-address) with several physical processes (process local-implementation-identifiers) so that services can be relocated, or forms of generic services can be created where any of several processes can provide the same logically equivalent service. Figure 9-5 illustrates the above. The IPC or higher levels must map this name to a specific server's. This mapping can take place at the origin to a specific hierarchical address or be distributed [McQuillan 78b].

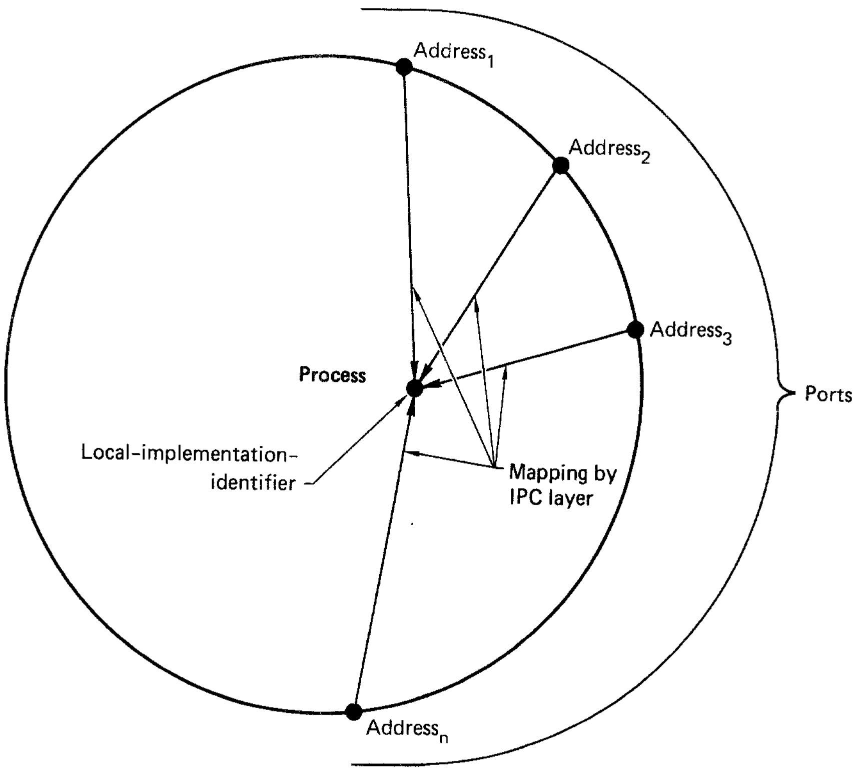


Figure 9-4: A process with multiple addresses or ports

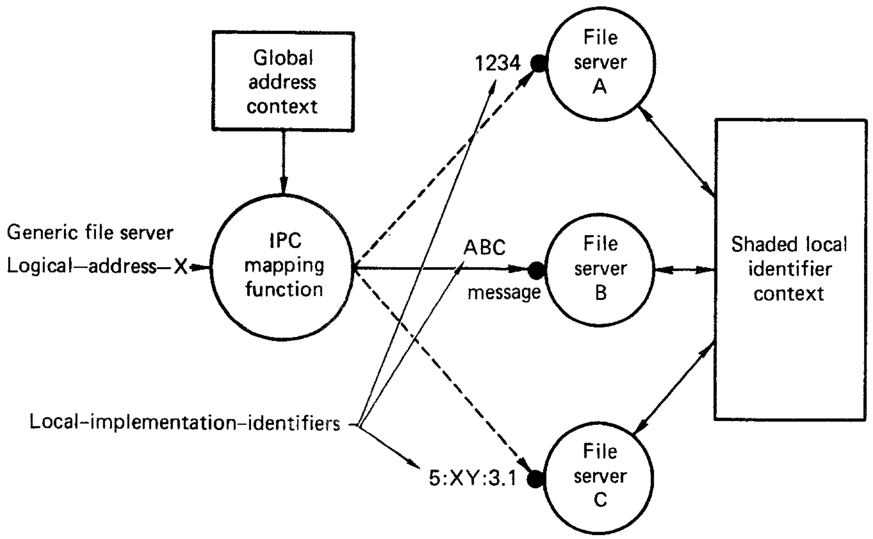


Figure 9-5: Generic file servers, IPC mapping routes address X to one instance (ABC)

Given origin and destination addresses and a message, one of the IPC layer services is to find a path or *route* from origin address to destination address along which to move the message as shown in figure 9-6. One or more routes (for robustness preferably the latter) may exist. The IPC layer maps addresses to routes. The last step in the routing maps an address to a process's local-implementation-identifier. Processes obtain their addresses from a process server. A process server either obtains addresses from the IPC service, if they are unstructured, or it may be able to construct them, if they are structured according to known rules. All process servers must form addresses according to global system rules. A process server binds addresses to process local-implementation-identifiers by setting routing tables as appropriate. If a service is relocated, the process server can rebind the address to a new process local-implementation-identifier and make appropriate changes to other routing tables. The mechanism to set routing tables would involve a routing table server, and is intimately tied to strategies for routing [Chapter 6, McQuillan 77,78a], and is a topic outside this section.

If an address is bound to many generic server processes simultaneously, the IPC routing algorithm will deliver messages to one, based on appropriate internal criteria, such as the "closest." If the generic server supports sessions, then future messages of the session must be directed to the instance first reached. Such servers would return a specific address as part of their request/reply protocol.

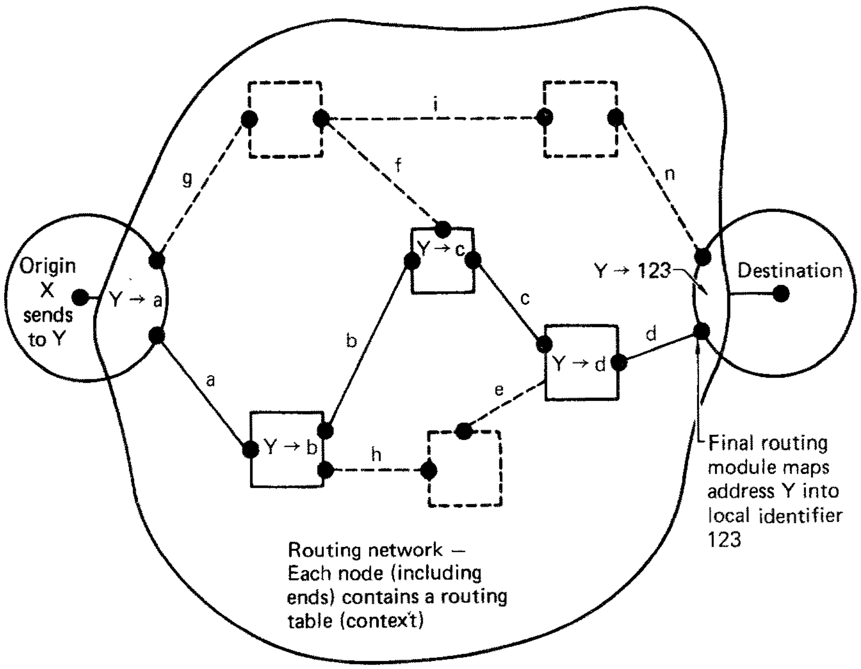

Origin
X
sends
to Y
Y → a
g
a
i
f
n
Y → c
b
c
Y → 123
Destination
d
Y → d
e
Y → b
h
Final routing module maps address Y into local identifier 123
Routing network – Each node (including ends) contains a routing table (context)

Figure 9-6: Router maps address *Y* to route *abcd* to local-implementation-identifier for object

It seems reasonable to use an address of a server as the server-identifier in the object-identifiers needed for its objects. Complete object-identifier mapping is shown in figure 9-7. The identifier form that results *address, unique-local-identifier* either provides or supports many of the identification goals listed earlier:

- For transaction oriented applications, the address needed to send a request can be directly obtained from the object-identifier or can be known in some a priori way. Dynamic binding supported by caching mechanisms may be required also.
- It is machine-oriented and can be used easily to create human-oriented identification mechanisms at higher levels.
- Identifiers can be generated in a distributed way.

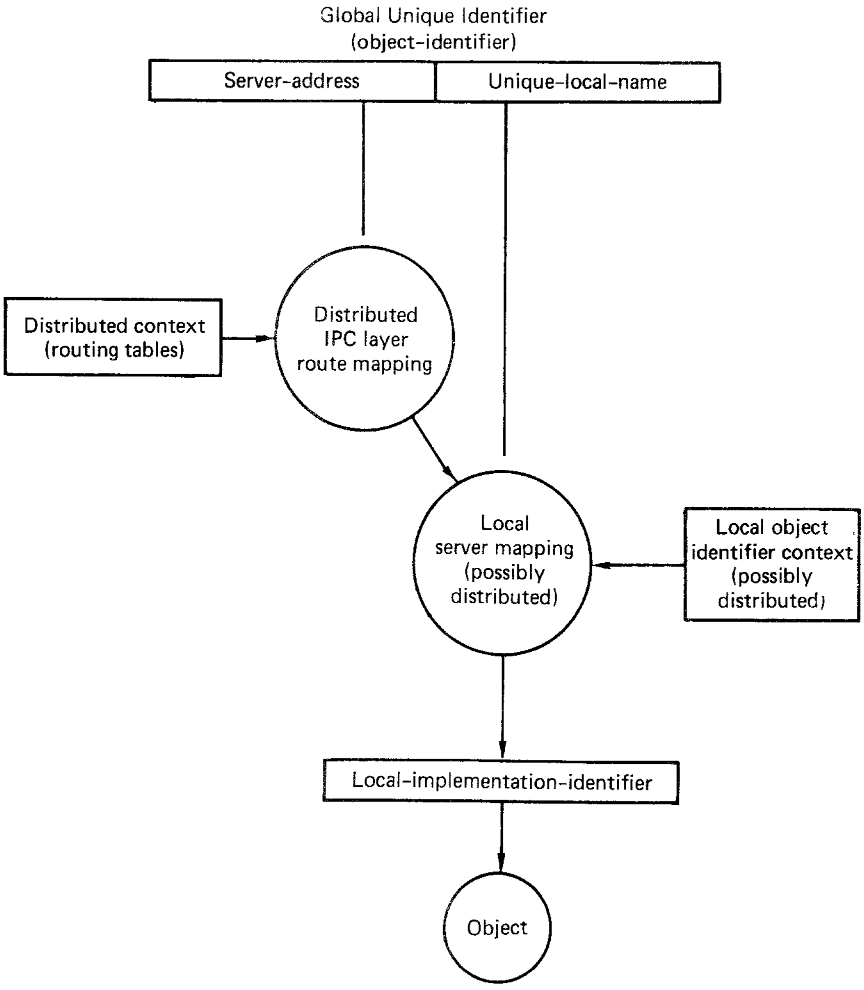

Global Unique Identifier
(object-identifier)
Server-address
Unique-local-name
Distributed context (routing tables)
Distributed IPC layer route mapping
Local server mapping (possibly distributed)
Local object identifier context (possibly distributed)
Local-implementation-identifier
Object

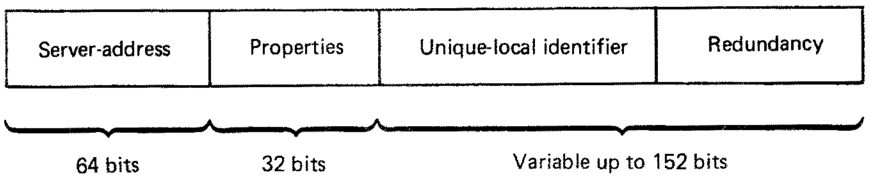
Figure 9-7: Mapping of unique global name (object identifier)

- A global space of uniquely named objects is created.
- With appropriate constraints on unique-local-identifiers and use of server logical-address, it directly supports object relocation and provides for generic servers and multiple copies. (Context in the IPC layer and in the servers may have to be changed.)
- If a new type of resource is constructed using existing ones as components, there is no identifier conflict and the inclusion of server addresses in the component identifiers can be directly useful to the new server.

The remaining goals can be achieved with a higher level identification mechanism such as that described below. It should be pointed out that processes have in effect two forms of identifiers associated with them, object-identifiers used as parameters in requests for services applied to them (the address part being the address of a process server) and addresses used to send them messages in regard to services performed by them. Let us now see how this identifier form can be used with protection mechanisms.

Two basic access control mechanisms are capability and access list control [Jones 78a, Chapter 10]. A *capability*, in the most general sense, is just a trusted identifier (object-identifier) with additional redundancy for protection. Possession of a capability is sole proof of right of access, with the modes of access associated with that capability [Dennis 66, Denning 76, Fabry 74]. An *access list* control mechanism requires, in addition to the object identifier, another trusted identifier representing the accessing *principal*, the entity with which access rights are associated. This trusted identifier might be a password, address, or other identifier form. Both mechanisms share in common the need to trust that an identifier, the capability in the former, and the trusted identifier in the latter, cannot be forged or stolen and then be used. Encryption or other use of redundancy and assumptions about what components (i.e. local operating systems, physical security and communications) can be trusted form the basis for deciding which approach to use and how to create a workable system [Watson 78]. Secure passing of capabilities in a DOS is discussed in [Donnelley 80, Nessett 80]. Authenticating that humans or processes are who they claim to be or are acting on behalf of whom they claim are topics not fully understood in distributed systems. Some of the issues are covered in [Section 9.3, Kent 77, Needham 78, Popek 78] and the last references above. The object-identifier in a system being designed at the Lawrence Livermore National Laboratory (LLNL) is a capability and has the form shown in figure 9-8. It consists of the following fields:

- The *address* is the address of the server that manages the resource. Often a process uses the address in one of the capabilities in a request message to determine where to send the request.
- The *properties* are a set of standard bits and fields that indicate to a customer process the resource type, access mode, resource lifetime, security level, etc. It is state information included in the capability to reduce messages to the server.
- The unique-local-identifier is used by the server to identify and locate the specific resource designated, and possibly for other server-dependent purposes.
- The redundancy, if present, guards the unique-local-identifier part of the capability against forgery. Encryption can also be used for this purpose [Chaum 78, Needham 79a,b].



Server address	– Address of the server of this named resource
Properties	– Fields containing resource type, access mode, security level, etc.
Unique-local-name	– Server dependent name encoding
Redundancy	– Protection against forgery (optional)

Figure 9-8: LLL standard capability form as example machine oriented identifier

An important need in all identification systems, whether global or local, is to be able to create unique identifiers in the face of crashes that can cause loss of identifier generator state information. One common mechanism is to use a clock guaranteed to continue operating across failures and that will not recycle during the period within which the needed identifiers must be unique [Garlick 77]. In exchange for the simplicity of this mechanism one may require rather long identifiers, depending on the granularity of clock interval needed. An alternative mechanism that can yield shorter identifiers, but increased complexity, can be built using two or more levels of storage or more than one storage device. One approach is to structure the identifier in a hierarchical fashion with one field for each level. One field normally is associated with main memory. A counter is kept on each level containing the highest value of the associated field assigned. The current values of these fields are cached in main memory. When a lower level field is about to cycle or the associated device crashes, the next higher level counter is incremented and lower levels are reset. Using the stable storage concept of Chapter 11, two levels of storage are sufficient for a safe efficient generator mechanism.

9.4. Human-oriented names

Higher-level naming contexts are required to meet the needs of human users for their own local mnemonic names, and to assist them in organizing, relating, and sharing objects. Higher-level names also can be used to provide for multiple copies and to support object relocation [Livesey 79]. Higher-level naming conventions and context can be embedded explicitly within particular applications or be supported by special name servers. Mapping from a high-level name to a machine-oriented identifier can be through explicit context of (high-level, machine-oriented) identifier pairs or implicitly by having some algorithmic or other relationship between the structures of the identifiers. An example of a higher level naming mechanism embedded in a distributed application is the mechanism used in the ARPANET mail service, where the name mapping takes place in several steps. For example, when a mail item is sent to the name DWATSON@BBN, the address of a mail server on a BBN computer is obtained from a table. The structured mail message is sent to this address. This service in turn will forward the message to an equivalent service on the specific BBN machine identified BBNB where the name DWATSON is mapped to the standard name of the file where my mail is stored.

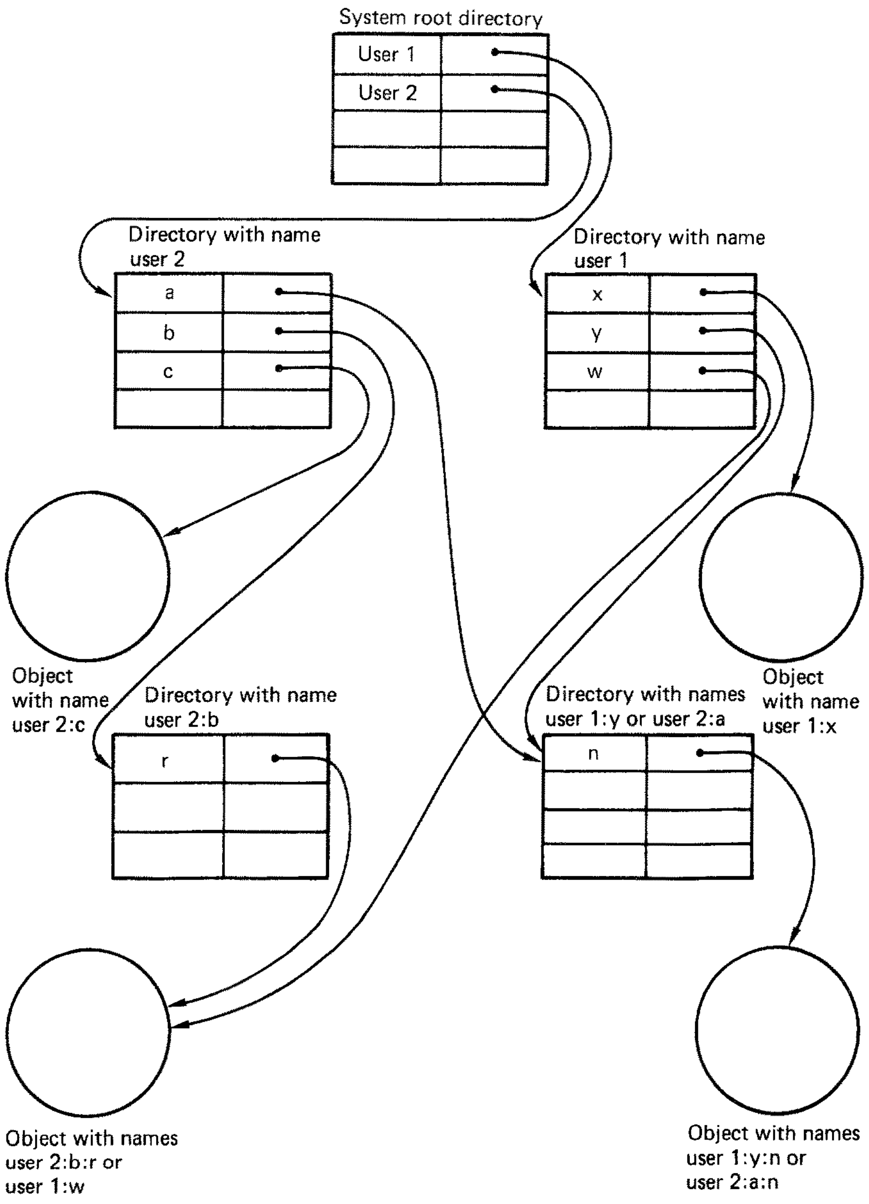

System root directory
User 1
User 2
Directory with name user 2
a
b
c
Directory with name user 1
x
y
w
Object with name user 2:c
Directory with name user 2:b
r
Directory with names user 1:y or user 2:a
n
Object with name user 1:x
Object with names user 2:b:r or user 1:w
Object with names user 1:y:n or user 2:a:n

Figure 9-9: Naming graph with shared multiply named objects

One view of database management systems is that their main function is to provide a general purpose higher-level naming mechanism. Although they usually store object representations and values, the items stored could just be the object-identifiers described above. The NSW uses database management techniques to provide human-oriented names [Millstein 77]. The design of database management systems allowing servers and databases to be distributed is a special domain in its own right beyond the scope of this section [Chapter 13, Bernstein 79].

Another mechanism used to create higher-level names is a *naming graph* implemented with *directories or catalogs.* These latter, in current operating systems, generally only contain names for files, but they can just as well be generalized to contain names for any resource. Capabilities, object-identifiers, or other form of identifier for any type of resource can be placed in a directory and given a mnemonic name. Since capabilities to directories can themselves be placed in directories, the directories constitute a naming graph as shown in figure 9-9.

The leaves of the naming graph are names of resources other than directories; the non-leaf nodes are directories. The *path-name* of a resource is the sequence of branches (directory entries) traversed from the root to a node. A given resource may be reachable by several paths thus allowing users to share resources, each being able to specify his own path name. Directories could be extended to maintain machine-oriented names for multiple copies of a resource [Reed 77]. Directories are managed by directory servers. Directory servers can be simple or complex. For example, in its simplest form, a directory server would have primitives for storing pairs (mnemonic name, machine-oriented-name) and retrieving machine-oriented names given a mnemonic name from named directories. Utility or other applications programs would be responsible for tracing down a naming graph according to a pathname to obtain the machine-oriented names at the leaves. It would also be possible to provide this latter service in a directory server.

One could also envision extending the directory service to provide message forwarding to processes identified by path-names. [Livesey 79, Reed 77] have proposed incorporating this service within the IPC layer. Saltzer gives a detailed discussion of the design issues associated with naming graph mechanisms [Saltzer 78a]. [Donnelley 80] discusses the relationship of directory and protection mechanisms.

Additional implementation and efficiency issues are introduced when there are multiple directory servers and the individual directories and other resources in a particular naming graph can be distributed [Thomas 78]. Tracing a path-name through a distributed naming graph may require exchanges of many messages. [Thomas 78] proposes mechanisms that attempt to create optimum location for directories, files and processes. This is just one example of the general issue of optimal, global location of resources when a particular object contains many other objects as components [Saltzer 78b]. The solution is likely to involve caching strategies, either explicitly under user control or automatically handled by servers or the IPC service. Until this problem area is better understood, at least in the case of directories, we believe that one should proceed slowly by only providing a simple directory server and leaving object and directory location and object-identifier caching as a user visible task.

Mapping of human-oriented names to object-identifiers is shown in figure 9-10. Separating machine-oriented identification (the object-identifier) and human-oriented naming allows:

- Servers, other than human-oriented name servers, to only have to deal with object-identifiers, and

- Several forms of application dependent and general purpose higher-level name services to be provided.

Ease of use or efficiency might be achieved for some applications and services if some servers other than name servers also implemented higher-level naming context, although this is not logically required [Fletcher 80].

9.5. Addresses and routing

The concepts of addressing and routing are introduced here in the context of the IPC layer, but they can exist at higher levels as well, using related mechanisms. Two of the services offered by the IPC layer are to provide a global identifier space of unique process addresses and to provide whatever mechanism is required to route information from one address to another. Addressing and routing are intimately related because routing is the level of identifier mapping that allows actual access to processes. There are two main design issues, the size of the address space and the structure of an address. The choice of an address structure has a significant impact on how addresses are mapped into routes, the size of the maps required at a particular node, whether the mapping can be distributed, and ease of network extension or reconfiguration [Shoch 78a, McQuillan 78a, Sunshine 77a,b]. Having a large enough address space is important in achieving a transaction oriented design as discussed later.

There are two forms of addresses in common use, single level or *flat*, and *hierarchical*. (Another form of address which explicitly specifies the route, a *source* address, has also been proposed as

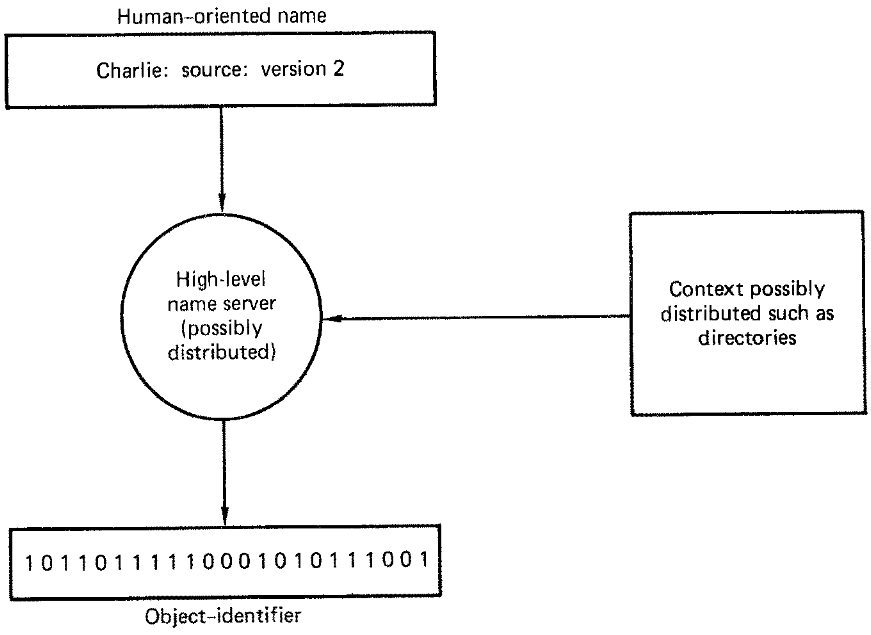

Human-oriented name
Charlie: source: version 2
High-level name server (possibly distributed)
Context possibly distributed such as directories
1 0 1 1 0 1 1 1 1 1 0 0 0 1 0 1 0 1 1 1 0 0 1
Object-identifier

Figure 9-10: A high-level name server maps human-oriented names into a machine-oriented name

being useful in internetworking and in network extensibility [Cerf 78b, Sunshine 77a,b].) With flat addresses, every routing node must maintain context to map all possible addresses. Therefore, potentially large routing tables or contexts must be provided at every routing node. If processes can be relocated without restriction or generic servers are constructed of component processes that can be located anywhere in the network environment, then flat addressing is required. Flat addresses may also be mapped into hierarchical addresses at the origin. The amount of context required at the origin can be reduced by a variety of techniques such as caching and broadcast searching when the cache fails [McQuillan 78b]. Higher level system design is often simplified if the ease of relocation offered by flat addresses is provided. Flat addresses are particularly attractive in environments supporting an efficient broadcast service. Flat addresses can be generated in a distributed manner by partitioning the address space.

Hierarchical addresses, because of information prebound in their structure, require less address-to-route context to be stored at each routing node. For this reason hierarchical addresses are normally used in mesh network environments. A hierarchical address consists of concantenated fields defining subtrees (i.e. network, cluster, host, process). That is, the address of a process reflects the hierarchical logical geometry of the network. Hierarchical addressing is illustrated in figure 9-11 and works as follows. The further away the destination is, in terms of the chosen hierarchy, the sooner the left-to-right parse is stopped. A node's routing algorithm maps an address field's value into the name of one of the node's outbound links or indicates that the next field should be treated with a further map. The decision process is repeated until successful or unsuccessful completion is indicated. This means that every routing node need not store information about every potential individual destination. Every branch down the hierarchical address tree could contain a different number of levels and a different fan-out at each level, i.e., fields within the address are not necessarily uniform in number or size in different subtrees. Hierarchical addressing can cause routing inefficiencies if a partition of the logical tree is geographically dispersed, because routing decisions may route all traffic for that partition to the same place even though the next decision may route packets within the partition to widely separated points [Shoch 78]. Within the hierarchical framework, one can provide for generic services and process relocation. A portion of the address space, characterized by a standard value for the leftmost bits, is set aside for this purpose. The routing tables in each node of the network then point to the nearest "representative" of a generic service or must be updated when an address is rebound to another process. Other IPC implementation choices can provide for logical addresses by using two levels of global addresses within the IPC level, flat and hierarchical, and map from flat addresses to hierarchical addresses at the origin. Various hybrid combinations are possible as discussed in [McQuillan 78b]. Routing is of course simplified with interconnection technologies such as broadcast bus, ring, etc. Routing is a special topic in its own right and is discussed further in [Section 6.3].

The above discussion introduced addressing and routing in the context of the IPC layer of the Model. In general it is important to restate that these functions can exist at any architecture level. An excellent example is reported by [Postel 79] where routing takes place at the application level for an internetwork electronic mail service. Postel discusses the identifier interactions and mapping between architecture levels. Routing can take place within the DOS layer, for example, when multiple processes are cooperating to provide a single service and the service has one logical-address. Name mapping in Naming Graphs (previous section) is another example.

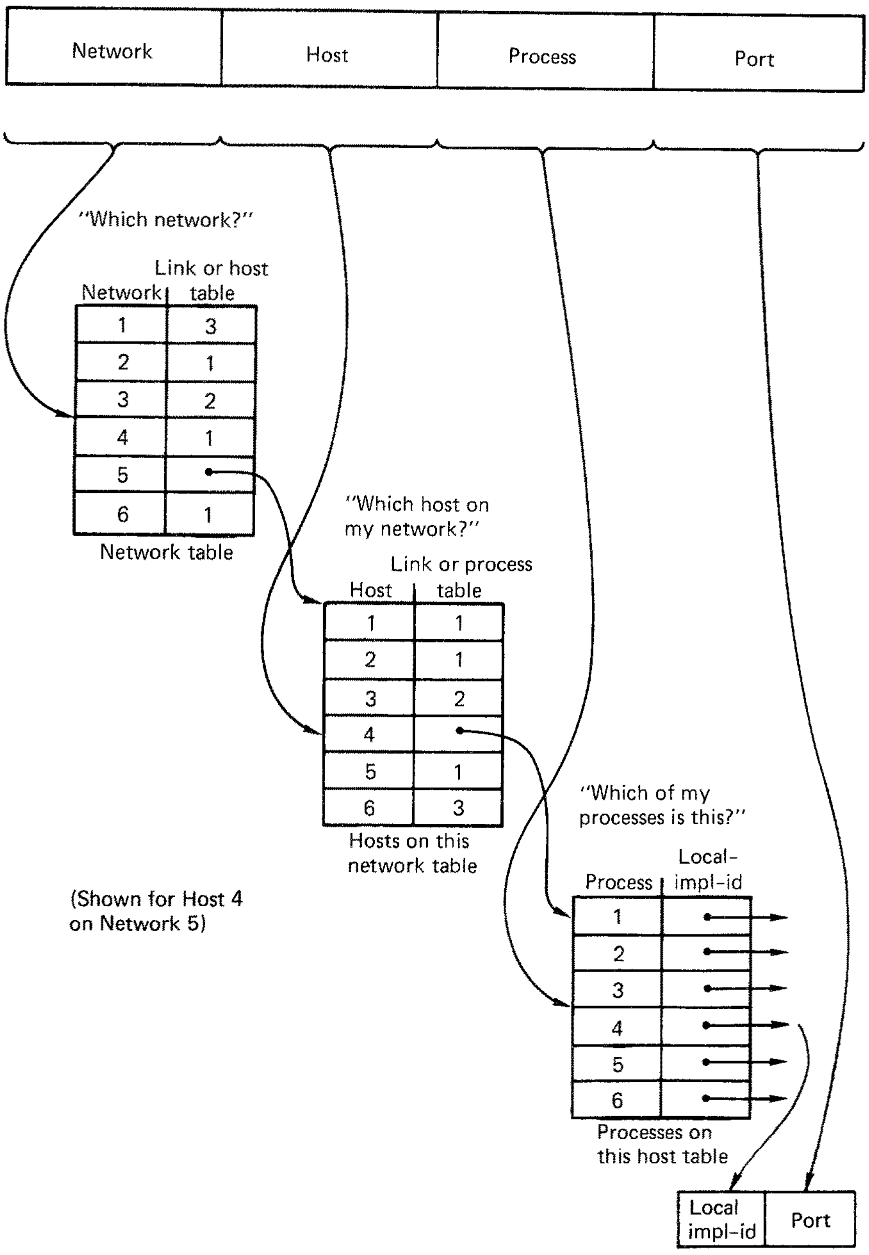


Figure 9-11: Relationship of hierarchical address and routing table structure

Another topic needing attention is the size of the address space. The network address space should be large enough that every process can have several addresses and none of the addresses has to be reused, even after the process is destroyed (assuming reasonable lifetime for the system). This feature is important as one element in achieving a transaction oriented system, because a process does not first have to send a message to a well known logger or connection establishment address, present a higher level name, and then be allocated a logical channel, socket, or other reusable address before entering a data transfer phase. There are also error control and security problems associated with using small reassignable addresses, in that old packets from a previous use of a given address may be in existence, or on recovery from a crash an address may be reassigned while one end of a previous association using that address still is sending to it.

The argument against large addresses, on the order of 64 bits, is that they increase the size of packet headers and thus may increase memory resources at intermediate and end nodes for packet and association state storage and increase the overhead on transmission bandwidth. With the rapid decrease in memory cost we believe that the first point is insignificant. The second point appears more serious, but a closer look at networking experience shows that there are two critical resources, the overall communication bandwidth available, and the time and resources required to create or process a packet and place it on or take it off the communication media. The latter often is the main limitation on throughput. Charging algorithms of public networks clearly recognize this situation by charging on a per packet basis. There is a relatively small overall incremental cost to sending additional bits, header or data, in a packet.

We believe the gain in flexibility for the system architecture, through use of larger address fields, are worth the incremental cost of increasing header size. This argument is particularly true for networks using high bandwidth links and is weaker when low speed links are used [Section 6.6]. In general, we agree with [Clark 78] that tradeoffs between increasing header size and complicating an application or protocol design or implementation favor using increased header information.

The pros and cons of fixed versus variable length addresses and headers revolve around tradeoffs between ease of extensibility and ease of implementation and packet processing. There is considerable controversy in this area. We believe that eventually variable length addressing will probably prevail, but as this is written the supporters of fixed length addresses tend to prevail.

One final topic needs discussion, the use of different address spaces at each level of IPC protocol. Each layer of IPC protocol defines a space of unique addresses for the entities that it connects with its abstract channels. Each pair of addresses (source, sink) at every level define a logical information channel (such channels could support a range of services depending on the protocol (i.e. datagram, virtual circuits, etc.). Providing for addressing, and thus channels, at each level of protocol is important because it allows:

- Channels at higher levels to be multiplexed on a single channel at a lower level for efficiency or other reasons; each higher level channel must have its own name,
- A single channel at a higher level to be multiplexed across many lower level channels for increased reliability or performance; the date elements for each higher level channel must be named because such splitting can cause out-of-sequence delivery, and
- Different protocols at a given level to use the same lower level protocol and be multiplexed on the same lower level channels.

For example, as mentioned in Section 9.7, a carriers X.25 virtual circuit (VC) tariff structure might make it more economical to set up a VC between a pair of hosts and then multiplex many end-to-end associations (simultaneously or sequentically) on that VC, than to use a separate VC for each association [Hertwick 78]. Many VC's in turn are multiplexed on a single link as shown in figure 9-12. Identifier multiplexing can also be generalized across the architecture at the DOS and Application levels as well.

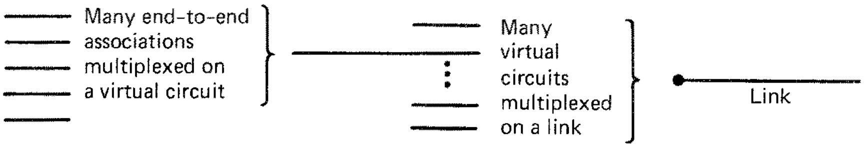

Many end-to-end associations multiplexed on a virtual circuit
Many virtual circuits multiplexed on a link
Link

Figure 9-12: Example showing levels of protocol addressing multiplexed on next lower level channel

It is also possible to multiplex protocols at one level on those at another without providing a separate address space in each [Cohen 79, Boggs 80]. [Cohen 79] points out that, while it is possible to merge layers in an implementation, the protocol and channel multiplexing flexibility inherent in a layered architecture is lost.

9.6. Conclusion

We have outlined characteristics and design goals for identification in a distributed operating system. We then presented an example identification system that met many of these goals, elaborating design issues as we proceeded. This identification scheme consisted of:

- Human-oriented name servers that provided for mnemonic names, sharing and organization of objects, and
- Unique, global machine-oriented identifiers structured as unique server addresses (supplied by the IPC service) and unique identifiers local to servers.

The human-oriented name service could be embedded in application processes or be provided by general purpose name servers using database management or directory techniques. The objects stored by these servers are the unique, global, machine-oriented identifiers (object-identifiers), for any type of resource. Other servers need only concern themselves with object-identifiers. We showed how binding would be handled between identifiers from heterogeneous identification domains and how naming at the Application, DOS Kernel/IPC layers interact.

We also discussed how an identifier generator could create unique identifiers in the face of system crashes and interactions between addressing and routing. It was pointed out that addressing and routing can occur at any level of the architecture.

We believe that the area of identification system design is an area with many open questions. One important need is to unify the many existing concepts and mechanisms. Another is to develop appropriate lower level hardware/firmware to support the desirable dynamic binding in an efficient manner.