Coherence in Naming in Distributed Computing Environments

Sanjay Radia* and Jan Pachl†

Software Portability Laboratory
Dept. of Computer Science, University of Waterloo

Abstract

Many different kinds of names (identifiers) are used in computer systems. Names are resolved (interpreted) in a context. A context is a function that maps names to entities. Multiple contexts allow the flexibility of giving different meanings to a name in different parts of the system; however, there are situations where it is desirable for the meaning of a name to be the same in different parts. This property is called coherence in naming.

Since the meaning of a name depends on the context selected, our analysis of coherence is based on the notion of closure mechanisms—implicit rules that select a context for resolving names. We define coherence and show how it is affected by various closure mechanisms. Then we present several approaches for dealing with the lack of coherence. Incoherence arises from selecting an incorrect context, and consequently, closure mechanisms are involved in the solutions.

1 Introduction

The subject of this paper is coherence in naming. *Naming* in computer systems deals with giving identifiers (names) to entities and using the identifiers to access the entities. Many different kinds of identifiers are used, spanning the range from those interpreted by hardware (such as memory addresses or network link numbers) to user-friendly names of objects and services at the application level. Since many entities at all levels in computer systems must be given identifiers, naming issues are pervasive. Designing a computer system or subsystem almost always includes designing a naming system, or at least choosing one from available alternatives.

*Present address: Sun Microsystems Laboratories Inc. 2550 Garcia Ave., Mtn. View, CA 94043.

†Present address: IBM Canada, Centre for Advanced Studies, 895 Don Mills Rd., North York, Ontario, M3C 1W3

This paper does not discuss a naming design for a particular computer system, nor does it give or analyse any algorithms for implementing naming schemes. We examine an aspect of naming—coherence—which is important in many naming designs, especially those for distributed systems. We develop the basic concepts on an abstract level, and then illustrate them with concrete examples.

To avoid awkward sentence constructions, we use the terms *name* and *identifier* interchangeably. Thus for us memory addresses, network addresses, process identifiers, file names, user names, and any other identifiers are all "names".

Coherence

In a distributed system, the same name often has different meanings in different parts of the system. However, there are situations where it is desirable for the meaning of a name to be the same in different parts. This property is called *coherence in naming*.

Names are resolved in a context. A context is a function that maps names to entities. Multiple contexts allow the flexibility of giving different meanings to a name in different parts of the system. Each subsystem in a distributed environment may have its own local context with name mappings for local entities. Confusion in the meaning of names can arise if there is interaction across context boundaries. Saltzer ([14], p. 188) lists several naming issues that have to be resolved when "two or more parallel and independently operating naming systems are asked to cooperate coherently with each other".

The need for coherence in naming was observed early in the development of distributed systems. However, it is not clear what exactly coherence in naming is and in what situations it is required. The problem of coherence in naming, a fundamental problem for distributed systems, deserves to be defined and examined on its own, independently of its manifestations in concrete systems.

0-8186-3770-6/93 $3.00 © 1993 IEEE

Global names

The requirement of coherence in naming led early distributed systems to provide a globally shared context. For example, Locus[17] and the V system[2] have a global context for file names, shared by all machines. However, even if it were possible to agree on a global context shared by all computers in the world, it would still be desirable and often necessary to use names relative to local contexts. This would lead to incoherence and the need to solve the coherence problem. Furthermore, we believe that several competing "global" contexts are likely to emerge. Autonomous systems, each with its own "global" context, will then need to connect to each other. Others have also made a case against a unique global name space [16, 13].

Thus the assumption of a single global context, although very convenient, is not realistic. As we argue later, the trend is towards multiple shared name spaces which provide names that are "global" in limited scopes. We must be prepared to support coherence without relying on names that are global in the entire system.

This Paper

In this paper we discuss coherence in naming, a fundamental problem for distributed systems, and examine how coherence in naming, or its restricted forms, can be provided in environments and situations that do not have global names.

Our analysis of coherence is based on the notion of closure mechanisms—implicit rules that select a context for resolving names—developed in [10]. Since coherence in naming concerns consistency in meaning of names and the meaning of a name depends on the context selected, understanding closure mechanisms is necessary for understanding coherence.

We define the property of coherence and identify several situations where coherence is desired. We show how different closure mechanisms affect coherence, and examine the degree of coherence in some common naming schemes used in distributed systems. We propose general mechanisms to support coherence and show how they fit in modern naming schemes for distributed systems.

Our work is based on experience with several distributed systems, in particular Waterloo Port[18] and networked versions of Unix.

2 Naming model

Our analysis of coherence is based on the formal naming model developed in [10]. Here we present a subset of the model.

We distinguish between active entities called *activities* and passive entities called *objects*. An activity performs computation on objects and communicates with other activities. An example of an activity is a Unix process, and an example of an object is a Unix file.

Entities (activities and objects) are referred to using names (identifiers); we say that entities are *denoted by* names.

Let $[D \rightarrow R]$ denote the set of total functions from the domain D to the range R. The naming model is described by the following sets:

Set of activities	A
Set of objects	O
Set of entities	$E = A \cup O \cup \{\perp_E\}$
Set of activity states	S_A
Set of object states	S_O
Set of states	$S = S_A \cup S_O \cup \{\perp_S\}$
Set of names	N

where $\perp_E$ is the undefined entity; $\perp_S$ is the undefined state; A, O are disjoint and $\perp_E \notin A \cup O$; and S_A, S_O are disjoint and $\perp_S \notin S_A \cup S_O$. Each entity has a state in S which may change over time. We denote the state of the entity e by $\sigma(e)$. Thus σ is a function from entities to states, $\sigma \in [E \rightarrow S]$, which determines the global state of the system.

A name may denote different entities in different contexts. Formally, a *context* is a function that maps names to entities; the set of contexts is defined as

$$C = [N \rightarrow E]$$

The state of an object can be a context; thus $C \subseteq S_O$. An object whose state is a context is called a *context object*. An example of a context object is a Unix file directory.

To *resolve* a name is to determine the entity denoted by the name. A name is always resolved in a context. In the context $c \in C$, the name $n \in N$ is resolved to the entity $c(n)$, the value of the function c at n. Resolutions are performed implicitly as part of operations on entities. For example, an open-file request involves the resolution of a file name; the result of the resolution is a file. When $e = c(n)$, we say that name n is *bound* to entity e in context c.

Compound names

A context object is an entity, and like any other entity it can be denoted by a name in a context. This leads to so-called compound names. A *compound name* is a nonempty sequence of names. Path names of files

in a tree structured file system[3] are an example of compound names.

The notion of resolving a name in a context extends to compound names. We denote by N^+ the set of all nonempty sequences of elements of N (names). For a compound name $n = n_1 \ldots n_k \in N^+$ of length $k \geq 2$ and a context c, define recursively

$$c(n_1 \ldots n_k) = \begin{cases} \sigma(c(n_1))(n_2 \ldots n_k) & \text{when } \sigma(c(n_1)) \in C \\ \perp_E & \text{otherwise} \end{cases}$$

Note that when a compound name of length $k \geq 2$ is resolved, the result depends on the state of the context objects along the resolution path.

The *naming graph* describes the state of context objects in a system. The naming graph is a directed graph with labels on edges. The nodes in the graph are the elements of $A \cup O$, and there is an edge labelled n from object $o \in O$ to entity $e \in A \cup O$ if o is a context object and $\sigma(o)(n) = e$. Resolving a compound name corresponds to traversing a directed path in the naming graph.

3 Closure Mechanisms

How is the context selected for resolving names? For a given name n, what context c should be used to yield the "correct" entity $c(n)$? An *implicit* context is needed whenever a name is resolved. An implicit context cannot be avoided, because whenever a context is specified explicitly by a name, another implicit context is needed to resolve that name; therefore one implicit (nameless) context is needed whenever a name is resolved.

Closure mechanisms are the rules that select a context from the possibly many contexts stored in the system. The term *closure* has been used in the semantics of programming languages to denote a (context,expression) pair[6]; names in the expression part are resolved in the context part and therefore the closure allows the value of the expression to be evaluated.

Recall that we distinguish between *activities* and *objects*. As illustrated in Figure 1, an activity can obtain a name

1. by generating the name internally within itself,
2. from another activity that sent the name in a message, or
3. from an object that contains the name.

A simple rule, commonly used in operating systems, is to make the resolution of a name depend on the activity performing the name resolution. For this, each

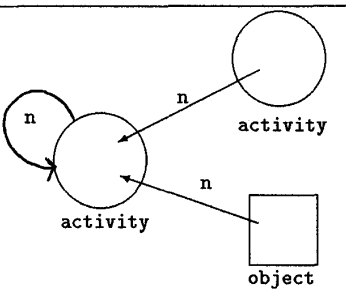


Figure 1: Three Sources of Names

activity is associated with a context and when an activity invokes the resolve operation, the activity's context is used. However, for more flexibility, it may be desirable to make the entity denoted by a name depend on how or where the name was obtained, rather than simply depend on the activity using the name.

In principle, the entity denoted by a name can be made to depend on any number of factors such as the activity using the name, the activity or object from which the name was obtained, the sequence of objects accessed to obtain the name, etc. The factors describe the circumstances in which the name occurs.

We use the notion of a resolution rule to model the dependence on various factors. A *resolution rule* defines a context to be used for resolving a name that occurs in a computation; the context is selected from the numerous contexts in the system. The function $\mathcal{R}()$, with a variable number of parameters, describes the resolution rule that selects a context: $\mathcal{R}(arguments) \in C$. The selected context is then applied to resolve the name: $\mathcal{R}(arguments)(name)$.

The arguments to $\mathcal{R}()$ describe the circumstances in which the name being resolved occurs. The set describing the circumstances is denoted by M, the *meta context*; hence

$$\mathcal{R} \in [M \rightarrow C]$$

The circumstances that affect name resolution in a particular naming scheme can be arbitrarily complex and therefore are difficult to define formally for all possible naming schemes. A resolution rule may incorporate the activity resolving the name and the entities from which the name was obtained. These can be described by rules of the form $\mathcal{R}(a), \mathcal{R}(a,o), \mathcal{R}(o), \mathcal{R}(o_1, o_2, \ldots)$, etc.

Operating systems usually make the resolution of a name depend on the activity a performing the resolution regardless of how or where the name was obtained. The resolution rule has the form $\mathcal{R}(a)$. Thus the system maintains a context $\mathcal{R}(a)$ for each activity a. (This does not mean that a *separate* context is

stored for each activity. Indeed, in the extreme case of a single global context only one context is stored, and is shared by all activities.)

It is often desirable to make the meaning of a name depend on the object o from which the name was obtained; the resolution rule has the form $\mathcal{R}(o)$. Again, this means that the system maintains a context $\mathcal{R}(o)$ for each object o. The resolution rule $\mathcal{R}(o)$ is useful for dealing with names embedded in objects. Many programming languages allow the meaning of a name to depend on the object (function or block) from which the name was obtained, whereas operating systems rarely do.

As we will see later, the rule $\mathcal{R}(sender)$ instead of $\mathcal{R}(receiver)$ is useful for activities that exchange names.

4 Coherence in Naming

Being able to give different meanings to a name in different situations and in different parts of a system is useful. Multiple contexts and closure mechanisms allow this flexibility. However, there are circumstances where it is desirable for the entity denoted by a name to be the same in different parts of the system.

In programming languages, names may denote different variables in different functions and procedures. However, it is often useful for a name to denote the same variable in different parts of the program. For example, a global name can be used to refer to a global variable from any part of a program. When a function is passed as a parameter, it is desirable to resolve the non-local variable names of the function in the context where the function was defined, instead of the context of the callee; the funarg mechanism was introduced in Lisp for this purpose. Similarly, call-by-name is preferable to call-by-text so that the parameter has the same meaning for the caller and callee.

In operating systems also, there are circumstances where the entity denoted by a name should be the same for different activities. We call this property *coherence in naming*. We identify the need for coherence in three circumstances. They correspond to three different sources of names during a computation: An activity can generate a name internally, obtain a name from another activity, or obtain a name from an object that contains the name.

1. *Coherence among activities for a name generated internally within each activity*
   This includes well-known common names used by activities. We also include in this category names obtained from a user; this is modelled by the user-interface activity generating the name. Coherence for these names is needed to allow the same name to be used for referring to an entity by different users and by a single user in different parts of the environment.

2. *Coherence among activities that exchange a name*
   Names are frequently exchanged between activities in computer systems: between parent and child activities, and between client and server activities. An activity sends a message containing a name denoting an entity to another activity which then uses the name to refer to the same entity.

   For example, systems such as Unix and Thoth execute a command by creating a new process and passing arguments to it; the arguments can be names of entities. Process identifiers are exchanged between client and server processes in the Waterloo Port system[18].

3. *Coherence among activities for a name obtained from an object*
   Names can be embedded in objects to build structured objects. For example, text formatting systems such as LaTeX[5] and Troff[8] allow file names to be embedded in files to build documents whose components are stored in several files. The C programming language[4] allows the use of embedded file names to build a program source from several files.

   The meaning of a structured object depends on the meanings of the embedded names, that is, on the objects denoted by the embedded names. When a structured object is shared among several activities, it is often desirable for the meaning of the structured object to be the same for each activity.

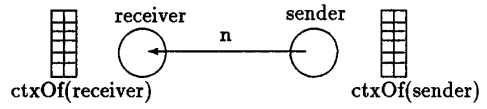
The degree of coherence in a naming scheme depends on the closure mechanism used. In particular, it depends on the resolution rule that selects a context for resolving a name. We now describe the relation between coherence and resolution rules in more detail.

Coherence and Resolution Rules

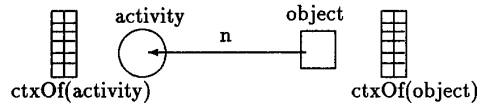
We examine coherence for the three sources of names identified earlier, and the related resolution rules.

1. When a name is generated internally within an activity, the context selected can depend only on the activity performing the resolution and therefore the resolution rule is $\mathcal{R}(activity)$. Only a global name—a name that denotes the same entity in the context of each activity—can be used as a common reference to a shared entity.

2. When an activity receives a name from another activity, the context selected can depend on the receiver and the sender (see Figure 2a):



(a) Context Selection for Names Exchanged in a Message



(b) Context Selection for Names Obtained from an Object

Figure 2: Coherence and Resolution Rules

- $\mathcal{R}(receiver)$
  When names are resolved in the context of the activity doing the resolution (the receiver), there is coherence only for global names.
- $\mathcal{R}(sender)$
  When names are resolved in the context of the sender, there is coherence between sender and receiver for all names sent by the sender.

It is also possible to conceive of more complex rules of the form $\mathcal{R}(receiver, sender)$. However, we have found no instances of, and no justification for, such rules.

3. When an activity obtains a name from an object, the context selected can depend on the activity and the object (see Figure 2b):

   - $\mathcal{R}(activity)$
     When names are resolved in the context of the activity doing the resolution, there is coherence only for global names.
   - $\mathcal{R}(object)$
     When names are resolved in the context associated with the object from which the names were obtained, there is coherence among all activities for the names embedded in the object.

   Again, more complex rules of the form $\mathcal{R}(activity, object)$ are possible, but their benefits are doubtful.

Thus, in all three cases, if the common resolution rule $\mathcal{R}(a)$ is used, where a name is resolved in the context of the activity a performing the resolution, then global names are essential for achieving coherence: Global names allow common reference to shared entities, they can be freely exchanged, and they can be embedded in objects. Consequently, if we wish to achieve coherence for non-global names, we have to look beyond the resolution rule $\mathcal{R}(a)$.

5 Coherence in Some Common Naming Schemes

We now examine the degree of coherence in some naming schemes commonly used in distributed computing environments. As we will see, coherence may be limited to a subset of activities and names. Also, a naming scheme may provide coherence in some circumstances, but not others.

Some important objects in distributed systems (for example, executable code for commands) are *replicated*. In terms of our naming model this means that several objects $o_1, \ldots, o_k \in O$ ("replicas of a replicated object") satisfy the property

$$\sigma(o_1) = \ldots = \sigma(o_k)$$

for every legal state σ of the system. In such a situation the coherence in naming as defined above is unnecessarily restrictive, and weak coherence is sufficient. *Weak coherence* for a name n means that n denotes replicas of the same replicated object in differents activities in the system.

We describe three approaches that have been used in common naming schemes. In two of the approaches an activity can access only a part of the naming graph, and hence refer to only a subset of the entities. The resolution rule is $\mathcal{R}(a)$ in all three approaches, where a is the activity performing the name resolution. The degree of coherence can be determined by comparing the contexts $\mathcal{R}(a)$ associated with different activities a.

5.1 The Single Naming Graph Approach

A common approach is to construct a single naming graph, shared by all activities. Often the naming graph is restricted to be a tree.

The context $\mathcal{R}(a)$ is based on either

- a distinguished node in the graph, called the *root*, from which all nodes in the graph can be reached, or
- a node in the graph depending on the location of the activity in the distributed environment.

In the former case, there is a high degree of coherence. In the latter case only activities in the same location have a high degree of coherence. It is worth noting that a shared naming tree does not imply that names are global; whether names are global depends on the relationship between the contexts $\mathcal{R}(a)$. The Newcastle Connection described below illustrates this.

Examples

Unix File Names

Unix, like many systems, uses a naming tree for naming files. The context $\mathcal{R}(p)$ of a Unix process p has two bindings: one for the root directory, and the other for the working directory. In a typical Unix system, $\mathcal{R}(p)(/)$ is the root of the tree for all processes p; consequently there is coherence for the set of compound names starting with '/'. The flexibility provided by the notion of a working directory is useful and the restriction on coherence is acceptable. However, in Unix, all processes need not have the same root and therefore, in general, there is coherence only among processes that have the same binding for the root directory.

Parent and child processes have a higher degree of coherence because a child inherits the context of its parent. A parent and a child have coherence for all names until one of them modifies its context. A parent can pass any file name as an argument to a child.

The V system[2] and distributed versions of Unix, such as Locus[17], combine subtrees in different parts of the distributed system to form a single naming tree. These systems follow the tradition of binding the root directory of each process to the root of the naming tree.

File Names in the Newcastle Connection

The Newcastle Connection[1] also creates a single naming tree from the individual naming trees of several machines. However, processes executing on different machines have different bindings for their root directory: typically $\mathcal{R}(p)(/)$ is the root of the machine on which p executes.

A single tree is created by attaching the naming tree of one machine to another, or by creating a new root node and attaching the trees of two or more machines. The Unix '..' notation is used to refer to nodes above a machine's root. Figure 3 shows a system with three machines.

Only processes that have the same binding for the root directory have coherence for names starting with '/'; typically, these are processes on the same machine. There is incoherence across machine boundaries: there is no common reference for shared files, names cannot

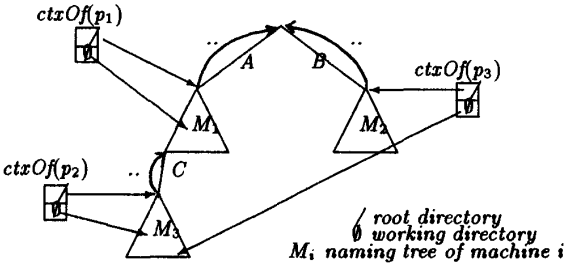


Figure 3: A Newcastle System with Three Machines

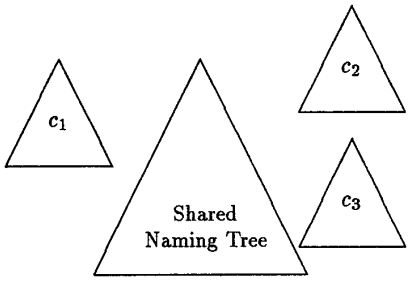
be freely exchanged, and a shared file with embedded names may have a different meaning on different machines. However, a simple rule can be used to map names across machines.

During remote execution, the root directory of the remote child is bound either to the root of the machine where the execution was invoked or to the root of the machine where the child executes. The former case provides coherence and names can be passed as parameters. The latter case does not provide coherence for parameters, but the program executes in the context of the remote machine and has the advantage of being able to access local objects on that machine.

5.2 The Shared Naming Graph Approach

In contrast with the previous approach, the system does not have a single shared naming graph for the entire system. Instead, numerous "client" subsystems share a naming graph, while maintaining their own private naming graphs. This is illustrated in Figure 4. Activities in a client subsystem have access to *both* the local naming graph of the client subsystem and the shared naming graph, but *not* to the naming graph of other client subsystems. Thus, the context of an activity depends on the client subsystem on which it executes. Only entities bound in the shared naming graph can be shared among client subsystems. This approach leads to more loosely-coupled distributed systems than the single naming graph approach.

The Waterloo Port System[18], Andrew[15], and OSF DCE[7] use this approach. For example, in Andrew, a system based on Unix, each client machine attaches the shared naming tree in the local naming tree under the node */vice*. The shared naming graph is the common subgraph in the graphs rooted at $\mathcal{R}(p)(/)$ for each process p. Only files in the shared naming graph have global names: these are names prefixed with */vice*. There is coherence among all processes with



c_i : Naming tree of client subsystem i

Figure 4: A Naming Graph Shared among Clients

respect to these global names, and activities within a client subsystem have coherence for local files named relative to the root of the local naming tree. Contrast this with the single naming tree of the Unix system where the entire tree is shared and there is a potential for coherence for *all* files.

There is also coherence for the names of replicated commands and libraries such as */bin, /usr/bin, /lib, /usr/lib* because each machine has bindings that map these names to either instances in the local naming tree or in the shared naming tree.

Incoherence arises when names are exchanged between activities on different client subsystems. For example, during a remote execution from a client subsystem to an execution server in another client subsystem, name conflicts can occur for files bound in the local naming tree of the client subsystem and the execution server. This problem can be avoided by ignoring all files in the execution server or all files in the client's home subsystem. Andrew uses the latter approach and therefore only entities in the shared naming graph can be passed as argument.

Incoherence can also arise when structured objects built with files containing embedded names are copied or moved between the shared naming tree and the local naming tree of a client.

In the OSF DCE environment, the shared naming tree (called the Global Directory Service) is attached in the local naming tree under "/...". DCE allows an additional local context called a cell which is accessed via the name "/.:". The cell is an organizational unit such as a division or department in an organization, or even the organization itself. Incoherence arises for names that are relative to the cell context. An organization can have several cells, but a machine is allowed to know of only one local cell. A single local context such as the cell is not going to be sufficient; it is useful to be able to use names relative to several local contexts such as those of the divisions, departments, and projects within an organization. This would add more non-global names and hence incoherence.

5.3 Extending the Naming Schemes

Often it is necessary to extend the naming schemes to support limited interactions between autonomous systems in a federated environment. Cross-links can be added to extend the naming graphs of the systems as illustrated in Figure 5.

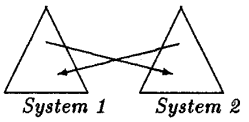


Figure 5: Cross Links between Autonomous Systems

The context of each activity is still based on its local system, but has been extended to allow access to the remote naming graph. There are no global names between systems unless they happen to use the same prefix name for a shared entity.

Incoherence arises when structured objects are shared across system boundaries and when names are exchanged. For example, during a remote execution from one system to another, name conflicts, similar to those in the shared naming graph approach discussed above, can occur.

Both the Newcastle Connection and the shared naming graph approach are special cases of extending machine contexts to access remote entities. The Newcastle Connection takes a systematic approach by creating a single naming tree. The Newcastle Connection is a distributed system that can be extended recursively because each extended system is still a Unix system with a single tree. The shared naming graph approach which uses a large shared naming tree avoids the need for attaching individual naming trees of other machines. But even such a system often needs to be connected to other similar autonomous systems with their own shared naming graphs; that leads to incoherence.

6 Dealing with Lack of Coherence

This section describes two general approaches to achieving coherence in naming schemes that do not have global names. For each of the approaches, we give specific solutions that use the approach; these solutions are used in our experimental extension of Wa-

terloo Port. The general approaches and the specific solutions can be adapted to many other systems.

I. Use an Appropriate Resolution Rule

We use resolution rules that select a context depending on the activity or object from which the name was obtained. This provides coherence for names exchanged between activities and for names embedded in objects.

Example 1. Partially Qualified Identifiers

Our first example is a naming scheme for communicating processes that uses partially qualified identifiers instead of fully qualified ones[10, 11]. In this scheme, process identifiers (pids) are qualified only as far as necessary. Pids have the form $p = (p.naddr, p.maddr, p.laddr)$. A process with local address l on machine m and network n has the following pids depending on the context of reference: $(\emptyset, \emptyset, \emptyset)$, $(\emptyset, \emptyset, l)$, $(\emptyset, m, l)$, and (n, m, l). The pid $(\emptyset, \emptyset, \emptyset)$ can be used by any process to refer to itself. Partially qualified pids have an advantage over the conventionally used fully qualified pids: when the address of a machine or a network is changed as part of relocation or reconfiguration, pids of local processes within the renamed machine or network remain valid and therefore the subsystem maintains its internal connections and does not have to be shut down.

A pid embedded in a message is valid in the context of the sender, but not necessarily in the context of the receiver. The resolution rule is $\mathcal{R}(sender)$; that is, use the context of the sender process that sent the embedded pid. The resolution rule is implemented by mapping the embedded pid. (Details of the implementation are in [10, 11].)

Example 2. Embedded File Names

Names can be embedded in objects to build structured objects. For example, programming languages such as C[4] and text formatters such as LaTeX[5] and Troff[8] allow embedded file names for referring to components stored in several files. An executable program may also be stored in several files. For example, a part of the program data may be stored in a separate file for convenience. The executable code for a multi-process application may be stored in several executable files with embedded names.

The following naming scheme, which is described in more detail in [10], provides coherence for names embedded in files. The context used to resolve such an embedded name depends on the file from which the name was obtained; the resolution rule is $\mathcal{R}(file)$. The context $\mathcal{R}(file)$ is determined using the Algol scope rules; instead of nested blocks, there are nested subtrees. A name embedded in a node n is resolved using a matching binding at the closest ancestor in the tree. The binding is found by searching up the tree, from node n to the root of the tree, for a directory node that has a binding matching the first component of the name. Figure 6 illustrates a subtree where the name a/p is embedded in node n within the scope of a binding at a node n'. The embedded name denotes node n'', which is determined by resolving a/p relative to node n'.

In this naming scheme, the name has the same meaning regardless of the process accessing the file and its site of execution. The subtree containing the structured object can be simultaneously attached in different parts of the distributed environment, and also relocated or copied without changing the meaning of the embedded names. Furthermore several structured objects (stored in subtrees) can be combined to form a larger structured object and a process can use several structured objects concurrently without name conflicts.

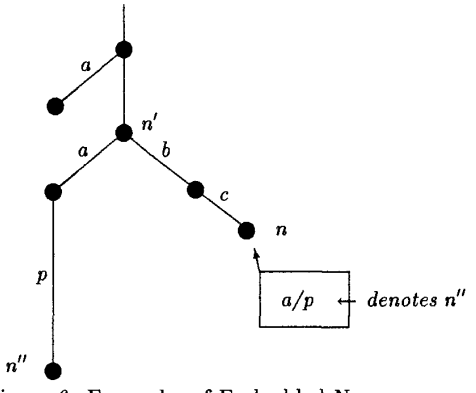


Figure 6: Examples of Embedded Names

II. Associate Appropriate Contexts with Activities that Exchange Names

In some situations it is possible to associate appropriate contexts with two communicating activities so that there is coherence for the subset of names that are exchanged. That is, for two activities a_1 and a_2, arrange that

$$\mathcal{R}(a_1)(n) = \mathcal{R}(a_2)(n)$$

for each $n \in N'$, where N' is some subset of N. Then, activities a_1 and a_2 have coherence for names in N'.

With this approach, the resolution rule is unchanged—names are resolved in the context of the

activity using the name. Instead, the contexts of a subset of activities that exchange names are arranged so that names can be exchanged. However, this is not a general solution because it may not be practical to change the contexts of two arbitrary activities that communicate.

This approach is difficult to build on top of most existing systems: The context of an activity is usually tied to its site of execution and it may not be possible to change it so as to resolve names in the context of a remote site. The approach can be used in the systems that provide a *per-process*, rather than a per-machine, view of naming. Two recent examples of systems with this view of naming are Plan 9[9] and our extension of Waterloo Port[10, 12]. Each process has its own individual root node to which the naming trees of subsystems known to the process are attached. The per-process view of naming decouples a process from the underlying context of its execution site: A process executing on a subsystem may use the context of another subsystem. In our extension of Waterloo Port, this yields a flexible naming environment which is used to construct a powerful remote execution facility. The remotely executing process can access files on both its local and its parent's machines. Thus, in spite of not having global names, the approach allows us to provide coherence for names passed as parameters from a parent process to its remote child. It can also be used to provide coherence in other applications where names are exchanged between processes. Sollins[16] also advocates such private views of naming.

7 Coherence: The Overall Picture

Sections 3 and 4 identified three sources of names: An activity can (1) generate a name internally (this includes obtaining the name from a human user), (2) receive the name from another activity, or (3) retrieve it from an object. The two approaches to achieving coherence described in section 6 apply to the second and third source. These approaches involve the use of special resolution rules. These rules cannot be used for names generated within an activity because the context selected can depend only on the activity doing the resolution.

An overall design of a naming scheme has to deal with the remaining source of names—the names generated within an activity and, in particular, those obtained from a human user. This section describes a general architecture for naming schemes for distributed systems that addresses the coherence problem for these names, and explains how the solutions of section 6 fit into this architecture.

For names generated internally, one requires consistent meaning of names across the activities where coherence is desired. However, it is not necessary to have a single global name space with all entities having global names. Instead, it is sufficient to share name spaces in a limited scope among activities that have a high degree of interaction. In recent naming schemes[10, 13, 9, 7], there is a trend towards having such shared name spaces rather than relying on a single global name space.

Such a shared name space should be attached by a *common name* to the contexts of activities in the scope.[1] There may be several shared name spaces. For example, the name space of home directories of different users in an organization may be attached under the name */users*, and the name space of services may be attached under */services*.

Some name spaces may be shared under a common name within a group in an organization, some in the entire organization itself, and some may be shared in even larger scopes that cross organization boundaries.

This approach would provide coherence for names generated by an activity with respect to a name space shared among activities under a common name. The solutions of section 6 would address coherence for other sources of names. The solutions would also provide coherence for names in non-shared name spaces, and for names resolved relative to local contexts within a shared name space.

Now what happens when scope boundaries are crossed? It may not be possible to use a common name to attach a name space. For example, consider two organization that attach the home directories of local users under the name */users*. When the first organization needs to refer to the home directories of users in the second organization, it may have to attach the home directories under the name */org2/users*. In such situations, one has to rely on humans to map names by adding the prefix */org2*. This is acceptable if mapping is required infrequently and the mapping rules are simple and intuitive. The mapping "solution" can be viewed as a closure mechanism used by humans to address incoherence. If the interaction across scope boundaries is high, then mapping names can becomes a hindrance and enlarging the scope may be necessary.

What about embedded names and those exchanged between activities in messages across scope boundaries? One cannot rely on human users since they do not generate the names. Here the solutions of section 6 would again prove useful. Continuing with the ex-

[1] Systems which allow a per-process view such as Plan 9 and the one described in [10] provide the flexibility of attaching name spaces directly to the context of an activity.

ample above, consider a user in the first organization who accesses a subtree in the second organization using a name with the prefix */org2/users*. The subtree may contain embedded names which would lead to incoherence (the names would surely not be prefixed by */org2/users*). Our solution for embedded names would restore coherence.

8 Conclusions

Coherence in naming is a fundamental problem for distributed systems. We have analysed the problem of coherence and examined the degree of coherence in some common naming schemes and proposed some solutions. Closure mechanisms are involved in the solutions.

Early distributed systems relied on global names to achieve coherence in naming. In recently designed systems, only a part of an application's name space is shared, rather than the entire name space. This has been in recognition of the difficulty and inappropriateness of having a single global name space, and the need to operate in federated environments. We expect this trend to continue and therefore stress the use of closure mechanisms and naming schemes that do not depend on a single global name space.

References

[1] D. R. Brownbridge, L. G. Marshall, and B. Randell. The Newcastle Connection – or UNIXes of the World Unite. *Software – Practice and Experience*, 12(12):1147–1162, Dec. 1982.

[2] D. R. Cheriton and T. Mann. A Decentralized Naming Facility. CS-86-1098, Stanford University, California, 1986.

[3] R. C. Daley and P. G. Neumann. A General Purpose File System for Secondary Storage. In *Proceedings of AFIPS 1965 Fall Joint Comput. Conference*, volume 27, pages 213–229, 1965.

[4] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall Inc, 1978.

[5] L. Lamport. LaTeX: *A Document Preparation System*. Addison-Wesley, 1986.

[6] P. J. Landin. A λ-Calculus Approach. In L. Fox, editor, *Advances in Programming and Non-Numerical Computation*, pages 97–141. Pergamon Press, Oxford, 1966. An earlier version of this paper appeared in The Computer Journal, 6 (1964).

[7] Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142. *OSF DCE Release 1.0 Developer's Kit, DCE Administration Guide, DCE Directory Service*, Feb. 1991.

[8] J. F. Ossanna. NROFF/TROFF User's Manual. TR 54, Bell Laboratories, Computer Science, Oct. 1976.

[9] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Proceedings of the Summer 1990 UKUUG Conference*, July 1990.

[10] S. Radia. *Names, Contexts, and Closure Mechanisms in Distributed Computing Environments*. PhD thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1989. Also ICR report UW/ICR 90-01.

[11] S. Radia and J. Pachl. Identifiers for End-Points in Dynamically Connected Systems, 1992. submitted for publication.

[12] S. Radia and J. Pachl. The Per-Process View of Naming and Remote Execution. In *Proceedings of 26th Annual Hawaii International Conference on System Sciences*, pages 377–386, 1993.

[13] H. C. Rao and L. L. Peterson. Accessing Files in an Internet: The Jade File System. Tech. report, University of Arizona, Tucson, AZ, 1991.

[14] J. H. Saltzer. Naming and Binding of Objects. In *Operating Systems: An Advanced Course*, volume 60, pages 99–208. Springer-Verlag, New York, 1978.

[15] M. Satyanarayanan, J. Howard, D. Nichols, R. Sidebotham, A. Spector, and M. West. The ITC Distributed File System: Principles and Design. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, Dec. 1985.

[16] K. R. Sollins and D. D. Clark. Distributed Name Management. In *Proceedings of the IFIP 6.5 Intl. Working Conference on Message Handling Systems*, Munich, Apr. 1987.

[17] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, Oct. 1983.

[18] Waterloo Microsystems Inc., 175 Columbia St. West, Waterloo, Ontario, Canada. *Waterloo Port Network Operating System*, 1984.