Naming Issues in the Design of Transparently Distributed Operating Systems

by

Robert J. Stroud

Ph.D. Thesis

The University of Newcastle upon Tyne

Computing Laboratory

July 1987

Abstract

Naming is of fundamental importance in the design of transparently distributed operating systems. A transparently distributed operating system should be functionally equivalent to the systems of which it is composed. In particular, the names of remote objects should be indistinguishable from the names of local objects.

In this thesis we explore the implication that this recursive notion of transparency has for the naming mechanisms provided by an operating system. In particular, we show that a recursive naming system is more readily extensible than a flat naming system by demonstrating that it is in precisely those areas in which a system is not recursive that transparency is hardest to achieve. However, this is not so much a problem of distribution so much as a problem of scale. A system which does not scale well internally will not extend well to a distributed system.

Building a distributed system out of existing systems involves joining the name spaces of the individual systems together. When combining name spaces it is important to preserve the identity of individual objects. Although unique identifiers may be used to distinguish objects within a single name space, we argue that it is difficult if not impossible in practice to guarantee the uniqueness of such identifiers between name spaces. Instead, we explore the possibility of using hierarchical identifiers, unique only within a localised context. However, we show that such identifiers cannot be used in an arbitrary naming graph without compromising the notion of identity and hence violating the semantics of the underlying system. The only alternative is to sacrifice a deterministic notion of identity by using random identifiers to approximate global uniqueness with a know probability of failure (which can be made arbitrarily small if the overall size of the system is known in advance).

Acknowledgements

I would like to thank my supervisor, Professor Brian Randell, who has given me the opportunity to work in this area and learn a great deal about transparency and distribution by experimenting with the Newcastle Connection.

I would also like to thank Professor Peter Lee who has acted as an unofficial supervisor for the last year and has read previous drafts of this thesis with painstaking care. The comments of both my supervisors have greatly improved the content and presentation of this thesis and I am grateful for the time and trouble they have taken with it.

I have also enjoyed working with my other colleagues in the Computing Laboratory, past and present, and have discussed many of the ideas in this thesis with them. It is not possible to mention everybody but I would like to single out Dr Lindsay Marshall who wrote the code for the Newcastle Connection practically single-handed. Without the Connection none of this work would have been possible or even conceivable. Lindsay and I have had many lively but fruitful discussions about most of the topics covered by this thesis.

Last but not least I would like to thank my family and friends who have been a great support to me and never lost faith me in me, even though I despaired of ever completing this thesis on many occasions. In particular, I would like to thank Laura Martin who gave me a reason for completing the thesis by agreeing to marry me!

This research has been sponsored by the UK Science and Engineering Research Council whose support I gratefully acknowledge.

Table of Contents

1. Introduction 1

1.1. Transparently Distributed Systems 1

1.2. Purpose of Thesis 4

2. Naming 7

2.1. First Principles 7

2.1.1. Naming and Identity 7

2.1.2. Binding, Bootstrapping, Contexts and Closures 10

2.1.3. Lifetime and Visibility 11

2.1.4. Naming Graphs 13

2.2. Some Naming Systems 17

2.2.1. A Database Naming System 18

2.2.2. A Capability Naming System 19

2.3. Naming in Unix 22

2.3.1. Inodes, Pathnames and Directories 22

2.3.2. Unix Pathnames and Contexts 24

2.3.3. The Unix Naming Tree and 27

2.3.4. Canonical Pathnames 30

2.3.5. An Alternative to 35

2.3.6. Symbolic Links 38

2.3.7. Other Unix Name Spaces 40

2.4. Conclusions 41

3. Joining Name Spaces Together 42

3.1. First Principles 43

3.2. Joining Name Spaces Together within a Unix System 46

3.2.1. Inodes and Devices 46

3.2.2. Joining Name Spaces Together with Mount 49

3.3. Non-Transparent Distributed Unix Systems 55

3.4. Transparent Distributed Unix Systems 57
3.4.1. The Newcastle Connection 58
3.4.2. NFS 62
3.4.3. RFS 64
3.4.4. LOCUS 65
3.5. Some Impediments to Transparency 69
3.5.1. Naming Graph Semantics 70
3.5.1.1. pwd 70
3.5.1.2. find 73
3.5.2. Low-Level Identifiers 75
3.5.3. Ownership and Authorisation 78
3.5.4. Remote Execution 80
3.5.5. Summary 82
3.6. Conclusions 82
4. Distributed Systems and Global Identifiers 84
4.1. Global Naming and Name Resolution 84
4.2. Allocating Unique Identifiers 87
4.3. Combining Name Spaces 89
4.3.1. Adding an Extra Level of Hierarchy 89
4.3.2. Heterogeneity 91
4.3.3. Dealing with Old Names 92
4.3.4. Merging Name Spaces 95
4.3.5. Random Identifiers 97
4.4. Reorganising Name Spaces 98
4.5. The Power of Indirection 99
4.6. Are Globally Unique Identifiers Realistic? 100
4.7. Conclusions 103
5. Recursive Transparency and the Newcastle Connection 105
5.1. Recursive Transparency 105

5.2. Connected Servers and the Newcastle Connection 107
5.2.1. Remote Execution 109
5.2.2. Network Heterogeneity 111
5.2.3. Name Space Management 112
5.2.4. Summary of Connected Servers 113
5.3. Connected Servers and Unix Pathnames 114
5.4. Multiple Servers 115
5.4.1. Access Rights and Ownership 117
5.4.2. Resource Allocation and Locking 121
5.4.3. Flattening the Recursion 123
5.5. Remote Execution 125
5.6. Other Distributed Unix Systems 130
5.7. Towards a Solution – DIY 132
5.8. Conclusion 133
6. An Abstract Approach to Recursive Transparency 135
6.1. Introduction 135
6.2. Name Resolution, Recursion and Transparency 137
6.3. Combining Perform and Resolve 142
6.4. Other Pathname Algorithms 147
6.5. Summary and Conclusions 152
7. Conclusions 155
7.1. Summary of Thesis 155
7.2. Contributions of Thesis 160
7.3. Future Work 164
7.4. Concluding Remarks 166
References 167

Chapter 1
Introduction

1.1. Transparently Distributed Systems

A distributed system is a group of computer systems which are able to work together and share resources via a network. Ideally, a distributed system should appear to be "a virtual uniprocessor" rather than a collection of individual machines [Tanenbaum85]. If this ideal is achieved in practice, the distribution is said to be **transparent** because users of the distributed system need not be aware of which component system executes their programs or stores their files. However, this is rather a strong requirement and may only be possible if the distributed system is designed and built from scratch with this objective in mind. Such distributed systems do exist: examples include Amoeba [Tanenbaum86], Accent [Rashid81] and the Stanford V kernel [Cheriton84a].

A more pragmatic way of building distributed systems is to augment existing software designed to run on stand-alone machines with the facilities necessary to access remote resources. This approach takes into account existing functionality and is therefore evolutionary rather than revolutionary. A distributed system built out of existing systems will be transparent if it is functionally equivalent to the systems of which it is composed. In other words, a transparently distributed system will appear to be a single system and will therefore act as a "virtual uniprocessor" in the sense discussed above.

A system may be characterised as an interface providing a set of objects and operations to client programs. If distribution is to be added transparently then the specification of this interface cannot be changed. For instance, it would not be possible to add an extra argument to an operation to indicate on which machine a remote object is to be found. Instead, the mechanism used to identify local objects

must be extended to include remote objects but without violating transparency by changing the form of identification. Typically, objects are identified by name where a name is a string of characters constructed in accordance with the syntactic rules of the system. Consequently, extending the identification mechanism involves finding a way to accommodate remote names as part of the local name space so that the names of remote objects are indistinguishable from the names of local objects, in form if not in content. For this reason, naming is of fundamental importance in the design and construction of a transparent distributed system.

In practice, there is more than one level within a given system at which an interface can be extended transparently to include distribution. At the highest level, distribution can be added to particular applications. For example, network architectures such as the DoD Arpanet [Cerf83] and the Xerox XNS architecture [Xerox81] include protocols for file transfer, remote terminal access, electronic mail and so on.

Adding distribution at the application level is the approach taken by international standardisation bodies in the move towards Open Systems Interconnection (OSI) [Zimmerman80] because it is particularly appropriate for heterogeneous networks composed of machines running different operating systems. In such an environment, it would not be possible to implement a single integrated system without making radical alterations to all the existing software, even assuming it was possible to find enough commonality between the various systems for a single integrated system to be achievable .

The problem with providing distribution in the form of specialised network applications is that such services tend not to be well integrated with the local system. For example, the command for copying files between machines may be quite different from the command for copying files within a single machine, even

though they perform basically the same function. Although it might be possible to incorporate the network file transfer protocol into the local system copy command, making it possible to access both local and remote files with a single command, this would only make copying files across the network transparent. Other operations on files (such as comparison) would not be affected so that the concept of a remote file would remain confined to the copy command.

This difficulty can be overcome by providing transparent distribution at a lower level of the system. For example, if the file system abstraction provided by the operating system is extended to include remote files then all the applications which use the file system will be able to benefit from the new facility immediately.

It is more appropriate to add distribution at a lower level of the system if the network is homogeneous and all the machines run the same operating system. This is also true for a less homogeneous network provided that the various systems on the network are sufficiently similar that a common abstraction such as a file system can be identified and transparently extended to include remote objects. However, the nature of the interface to be extended is also important when considering at which level of the system to add distribution. Clearly, it is easier to extend a simple interface rather than a complex one. Furthermore, since transparent distribution involves the recursive notion of constructing a distributed system which is functionally equivalent to the systems of which it is composed, it follows that systems whose structure is already recursive in some sense will be best suited to this method of constructing distributed systems. Perhaps for these reasons, the Unix operating system [Ritchie78] with its relatively simple system call interface and hierarchical contextual name space has formed the basis of many such distributed systems. Examples include the

Newcastle Connection [Brownbridge82], Locus [Walker83], NFS [Sandberg86] and RFS [Rifkin86].

1.2. Purpose of Thesis

In this thesis we will be mainly concerned with the evolutionary problems of building a distributed system out of existing systems rather than the revolutionary approach of building new distributed systems from scratch. Taking an evolutionary approach is obviously pragmatic because it protects investment in hardware, software and human expertise. However, it is harder to achieve transparency because certain design choices made during the construction of the original system may be inappropriate for a distributed system. Although the effort which must be expended in solving these problems might be viewed as misguided ingenuity (since given a clean slate and the opportunity to take a revolutionary approach, backwards compatibility would not be an issue), this view is short-sighted. A distributed system built from scratch may initially be self-contained but sooner or later it may be convenient to extend it or even to merge it with a similar distributed system constructed independently. Joining two transparent distributed systems should be no different from joining two conventional systems if the distributed systems are really transparent and so the same problems will arise, even in distributed systems which have been built from scratch rather than constructed by joining a set of existing systems together. A revolutionary design which ignores these issues will not scale properly.

It is usually taken for granted in the design of a distributed system that all objects will be ultimately identified by globally unique names, sometimes unique in time as well as space. We propose to question this received wisdom and argue for a more structured approach based on names which are unique only within some localised context and not necessarily unique across the entire system. We believe that the very concept of "global uniqueness" is alien to the distributed

nature of the sort of systems being considered and betrays centralised thinking, at least in the design stage if not the actual implementation. Although ultimately we may not have anything better to offer, we feel it is important to explore these issues in more detail than they have hitherto received.

Specifically, this thesis will consider the problem of joining systems together to form bigger systems and the implications this has for naming. Ideally a transparent distributed system is indistinguishable from the systems of which it is composed and consequently it should be possible to combine both single systems and distributed systems recursively and, in theory at least, indefinitely. However, if individual distributed systems are designed assuming unique identifiers, there is no guarantee that those identifiers will continue to be unique when two such systems are combined in this way. Since individual designers will have their own ideas about the construction of unique identifiers, such a clash is almost inevitable unless it is possible to impose a truly global (indeed universal) discipline which will ensure uniqueness not only within but also across all possible distributed systems. In a world containing many different distributed systems and vested interests this is not possible for political rather than technical reasons. Even if agreement was possible, we believe that there are serious management problems for really large systems which use globally unique identifiers. Consequently, this thesis will explore ways of structuring name spaces to overcome (or at least reduce) uniqueness problems and will examine algorithms for merging independently managed name spaces and sharing the names of objects between systems.

To summarise the structure of the rest of this thesis, chapter 2 will begin by examining naming issues in detail with specific reference to the naming mechanisms of Unix. Chapter 3 will consider the problems of joining names spaces together, both within a single system and between systems across a network to construct a distributed system. Again, Unix will be used to illustrate

these ideas and highlight some of the difficult areas in constructing a transparent distributed system. Both chapters will explore some of the weaknesses in the Unix naming mechanisms that cause problems and discuss some alternatives. In chapter 4, conventional distributed systems with names based on globally unique identifiers will be studied to see how they tackle the problem of joining systems together. Chapters 5 and 6 will then explore the idea of constructing distributed systems recursively using names which are only unique within a local context. Chapter 5 will examine the implications of recursion for the design of one particular implementation of transparent distribution for Unix, the Newcastle Connection, and highlight the basic issues. Then chapter 6 will analyse the problem in more abstract terms and develop a distributed naming architecture based on two operations, *perform* and *resolve*, which can be generalised to handle recursively constructed systems. Finally, chapter 7 will pull all the threads together and declare a final verdict on the relative merits of global versus local identifiers.

Chapter 2
Naming

Naming is of fundamental importance to the construction of transparently distributed systems and so this chapter explores some of the issues that arise in the design of a naming system. After introducing some general principles of naming, three systems will be discussed in particular: Aspect, Flex and Unix. Each takes a different approach to naming. Unix is of particular interest because it has a hierarchical naming structure and therefore the element of recursion necessary for the construction of transparently distributed systems is already built in. However, the Unix naming algorithms have various deficiencies and these will be analysed too. The concept of a **canonical pathname** will be proposed as a way of overcoming some of these problems.

2.1. First Principles

This section will establish some basic concepts of naming. The terminology used will be that established by Saltzer [Saltzer78]. For a more thorough treatment of naming issues, see the thesis by Brownbridge [Brownbridge84].

2.1.1. Naming and Identity

A fundamental property of an object is its identity. Even if two discrete objects are alike in every other way, they will retain their own identity. Consequently, **identity** may be defined as that which distinguishes one object from another [Copeland86].

Given a collection of identical objects, the only way in which it is possible to identify a particular object unambiguously is literally to point at it and say "this one here". However, this is not always practical, even in the real world, let alone in the abstract world represented by the internal state of a computer system.

Names are a way of tagging objects so that they can be identified more abstractly without such physical intervention.

In this sense a name is an abstraction of identity, making it possible to write algorithms which manipulate objects without having to include those objects as part of the algorithm. However, it does not follow that there is a one-to-one relationship between names and identities. A given object may have more than one name and two distinct objects may have the same name.

Of course, if the same name can denote more than one object at the same time, the naming system is ambiguous. Without a means of identifying objects explicitly (by pointing at them), names are the only substitute for the notion of identity and must therefore be unambiguous to prevent confusion. However, it is interesting to observe that in computer systems with an interactive graphical interface where you can indeed point at objects directly, names are no longer necessary to distinguish objects and cease to be so important. Only the icon representing an object on the screen matters. There is a one-to-one mapping between the image of an object on the screen and the identity of the object itself, and in this sense, the actual image (as opposed to some label attached to that image) is a name for the object that it represents.

For example, the electronic desktop implemented by the Xerox Viewpoint system [Xerox85] represents documents on the screen as icons which can be pointed at with a mouse. Although an icon may contain the name of the document it represents, it is quite possible for two icons representing distinct objects to have the same name because they remain physically distinct on the screen. In particular, when an icon is copied, thereby making a copy of the document it represents, the copy will have the same name but a different identity.

Similarly, the Flex capability-based system [Foster82] developed at RSRE displays values on the screen in boxes called *Cartouches*. The text inside the box

may indicate how the box was created, and in particular it may be a name which was looked up at some point in the past to get the value. However, it does not follow that the name in the box is still valid or denotes the same value, or even that the text has anything to do with where the object came from at all. It could describe the type of object denoted by the *Cartouche* or simply be an arbitrary label.

It could be argued that inside such a system there must be some value which identifies an object uniquely and that this is simply a name which is known to the system but not to its users. This is a reasonable point of view but so long as the system preserves the distinction between the identities of distinct objects (and if it did not, it would be broken), it is free to alter such a value as much as it chooses, so that if the value is indeed a name, it has a very transitory existence. Such a value is really a means of locating the object and could therefore be an address in memory or on disk. There is nothing to stop the system from rearranging the contents of its memory or disk and changing such values accordingly (for example, during garbage collection), providing the identities of the corresponding objects is preserved. The distinction between a name and an address is really only one of degree or perhaps level of abstraction. From a fixed viewpoint, names tend to be more permanent and more visible but less location dependent than addresses, but the same criteria could be used to describe the difference between virtual addresses and physical addresses in a paging system.

To summarise, a naming system is a mapping between names and identities. This is a recursive notion; identities may be represented internally by low-level identifiers which are themselves names in a lower level naming space. At each level of such a hierarchy, as names are mapped into identifiers they become less abstract and closer to physical storage locations. Adding extra levels of

indirection makes it possible for names to be location independent (i.e. transparent).

2.1.2. Binding, Bootstrapping, Contexts and Closures

It is not always practical to point at objects directly, especially in a non-interactive system. Nor is it feasible to embed objects directly in algorithms, especially when writing general purpose reusable code. Binding the identity of a specific object into an algorithm too tightly has a limiting effect on abstraction. Names are a way of abstracting over identities and the process of replacing a name with the identity of the object it denotes is called **name resolution.** Delaying the time at which the name is **resolved** makes the system more flexible but less efficient at run-time if names have to be looked up every time objects are needed.

More formally, a **binding** may be defined as an association between a name and an object (or rather its identity). This may be generalised to the notion of a **context** which is a list of such bindings. Contexts are an important structuring mechanism which allow a large name space to be subdivided into several smaller name spaces. In particular, the meaning of a name depends on the context in which it is resolved and although a given name can only have one meaning in a particular context, each context in which it appears may bind it to a different object and so give it a different meaning.

Contexts also have names relative to some other context which itself must be named (relative to yet another context and so on). This potentially infinite regression can only be prevented if there is at least one context which does not require a name but is always known to the system. Such a context can then be used as the basis for all names. However, because it cannot be named in the conventional way, the definition of this context must be established as part of the **bootstrapping** process by which the system is brought into existence initially.

Even if the bootstrap mechanism is able to name such contexts, these names will have no meaning to the system being bootstrapped. Thus, all names are ultimately relative to some point outside the system to which they belong.

When names are embedded in a program it is sometimes important that they denote a particular object, so that the program's behaviour is independent of the naming context in which it is executed. This may be achieved by making such names relative to an absolute context which cannot be moved and has a global definition known to all users of the system. Alternatively, a mechanism called a **closure** may be used to bind such names statically (when the program is defined) rather than dynamically (when it is executed). Although closures are a very powerful concept, they are usually only found in implementations of programming languages which encourage a functional style of programming because static binding makes it possible to treat functions as first class values unambiguously [Landin64]. A facility for defining names statically would also be useful in a general purpose operating system. However, without closures, the alternative mechanism of using an absolute name (which is usually all that is available) is not always adequate because it does not encourage modularity and is not recursive.

2.1.3. Lifetime and Visibility

Another important issue is the relationship between the concepts of lifetime and visibility. In any sane and self-consistent system an object will exist so long as there is a name for it or some other way of accessing it. The same name will not suddenly denote a different object or cease to denote any object at all. Consequently, there should be a strong connection between the existence of a name for an object and the lifetime of that object. Indeed, it is reasonable to argue that if there is no way of accessing an object, by name or any other means, then the object has effectively ceased to exist within the system. Certainly, its

existence can have no meaning or significance to a user of the system (although internally the system may retain some knowledge of the non-existent object so that the resources consumed by the object can be freed if necessary). However, it is also important to realise that not all names are directly accessible (i.e. visible) at any one time. For example, the names in a closure are not visible outside that closure, although the objects they refer to will continue to exist for as long as the closure itself exists.

Some systems allow the same object to have more than one name. In general, deleting a name for an object will only delete the object itself if it has no other names. So long as there is a name for an object somewhere in the system, that object will continue to exist. However, with other systems, although objects can still have several names, one name is distinguished as being the principal name and all the other names for the object are merely aliases which provide a convenient naming shorthand. For such a system, deleting the principal name for an object could delete both the object and all its aliases. Alternatively, the system might not allow an object to be deleted until all its aliases had also been deleted. These precautions are necessary in order to guarantee that the system cannot be left in an inconsistent state with dangling names pointing at objects which no longer exist.

If a large name space can be decomposed into smaller name spaces, it may be possible for part of the whole name space to become temporarily unavailable. This could occur if the name space was spread across a network or a collection of removable disks. When such a partitioning of the name space occurs, each side of the partition should remain self-consistent, regardless of whether it is otherwise active or passive, until such time as the whole name space is reunited. References from one partition to objects in the other partition must be treated with caution during this time. Such references cannot be resolved without the cooperation of both partitions and consequently the objects referred to will be temporarily

unavailable. Nor will it be possible to delete a reference to an object in another partition or an object which is referred to by another partition and still guarantee that the name space will be in a consistent state when the partitioning ends. Nevertheless, it should be possible to override this protection if an inopportune crash of part of the system leaves the name space in an inconsistent state. Similarly, there should be a mechanism for detaching part of the name space deliberately for backup purposes or in order to transport it elsewhere physically.

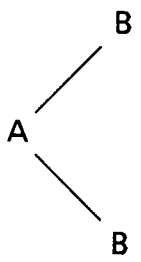
2.1.4. Naming Graphs

A naming system which only allows simple names relative to a single context is not very interesting. More powerful naming systems provide the concept of a **pathname**, a structured name involving several contexts. Pathnames start from a known context and are divided up into components. Each component names the context in which the next component is to be resolved with the last component naming the object referred to by the pathname as a whole. A good way to model such naming systems is with a naming graph.

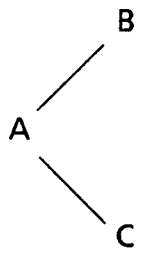
Informally, a graph is a collection of nodes, some of which are joined together by arcs. The nodes of the graph may be thought of as objects and the arcs as naming paths. Therefore, if a given node represents a context, the arcs leading from that node determine the name bindings in that context. However, there are several ways of labelling such a graph and interpreting it as a naming system. Other properties of the graph such as whether the arcs are directional or whether the graph is acyclic are also important and characterise the naming system too.

For example, if the nodes rather than the arcs are labelled then each node will only have one name, regardless of how many arcs lead to it. This means that in order for the labelled graph to be well-formed as a naming graph it must satisfy a local uniqueness property that ensures that all the arcs leading from a given node reach nodes with different names. (If a node representing a context has two or

more arcs leading from it to nodes with the same name, that name is ambiguous in this context.) An arbitrary labelled graph is not guaranteed to have this



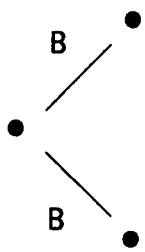
(a) ambiguous



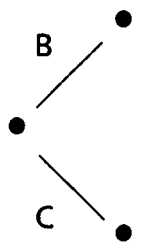
(b) unambiguous

property and therefore not all labelled graphs are valid naming graphs.

Alternatively, the arcs could be labelled rather than the nodes, allowing a given node to have many names, each name being the label on an arc leading to it from another node. Again, there would need to be a consistency property which



(a) ambiguous



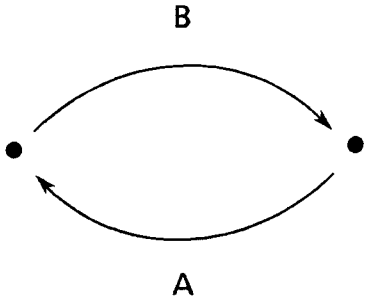
(b) unambiguous

ensured that two arcs leading from the same node did not have the same label or name since this would effectively give two bindings for the name in that context.

Quite apart from the actual form the graph labelling takes, there are also questions about the nature of the arcs and the connectivity of the graph. If all the arcs are bi-directional, every node can act as a context and name all the nodes

which can name it. In this case, it will be natural to label the nodes rather than the arcs.

On the other hand, if the arcs are uni-directional then there are various ways in which the graph can be connected. If every pair of interconnected nodes is joined by two arcs, one in each direction, then each node can name the other. For a



general purpose naming system, this degree of connectivity is very natural; there is no point in introducing anomalies such as one-way naming paths unnecessarily. However, if this kind of flexibility is needed, perhaps to restrict access to (or from) parts of the naming graph, then there is no reason why it should not be available.

Of course, having arcs leading from every node makes every node a naming context. Whilst preserving full connectivity between those nodes which do act as naming contexts, it is useful to recognise leaf nodes as a special case. A leaf node acts as a sink for naming arcs. In other words, whilst a leaf node has arcs leading to it and may therefore be named, it does not have arcs leading from it and hence may not act as a naming context. In most real systems, the leaf nodes would be the objects of interest and the other nodes that led to them in the graph would act purely as contexts. However, the model is more general than this, at least in theory if not in practice.

The opposite of a sink node is a source node which has arcs leading from it but no arcs leading to it. A source node may act as a naming context but may not be named from another context. Instead, it must be given an absolute name, independent of any context. This name may then be used as the starting point for a pathname.

It is convenient to allow other nodes in the graph to be given special names so that they too can act as a starting point for a pathname. However, whereas a source node can have no other name, any other point in the graph will always have at least one absolute pathname relative to some source node. Consequently, such points can be redefined and may therefore serve as a current context, making full absolute pathnames unnecessary. The difficulty with this is that the interpretation of pathnames relative to a redefinable current context depends on the dynamic definition of that context. If such pathnames are to be used in algorithms unambiguously then some sort of closure mechanism is needed to guarantee that the correct definition of the current context is used when they are resolved. However, pathnames relative to source nodes may be used unambiguously in any context because their interpretation only depends on the fixed location of the source node.

One final property which characterises a naming graph is its overall topology and in particular whether it contains cycles. A **cycle** is a closed sequence of nodes joined together by naming arcs, in other words, a loop in the naming structure that returns to its starting point. The presence of a cycle allows infinite pathnames even in a finite graph and makes it difficult to visit each node in the graph systematically. An **acyclic** graph contains no cycles and does not suffer from these problems. A **tree-structured** graph is a special case of an acyclic graph with the additional property that every node may be reached from the unique source node of the tree (its base) in exactly one way. Each node can only be

named from one other node (its parent) and consequently every node in the tree has a unique pathname from the base of the tree.

This discussion of naming graphs has described them as if they were static entities with a fixed structure. Of course, in any real system modelled by a naming graph the structure of the graph will change as objects (and names) are created and destroyed. However, such operations in the real system should be constrained so that invariant properties of the naming graph (such as whether or not it is tree-structured) are preserved. The system should also remain self-consistent in the sense that all references must lead to a valid object and that all objects which are not referenced from elsewhere in the system should be destroyed. This is the problem of garbage collection. The system must detect when the last reference to an object is deleted so that it can destroy the object itself. One solution is to count the references to each object. However, a cyclic graph will allow self-referential structures to exist in isolation, unreferenced by the rest of the graph. This is another reason why cyclic graphs are awkward to handle, making acyclic or tree-structured graphs more desirable. Nothing is gained by introducing cycles into a naming graph since the new names will go nowhere but it is difficult to give an algorithm for incrementally modifying an acyclic graph without introducing cycles, short of scanning the entire graph for a cycle every time a new arc is added.

2.2. Some Naming Systems

The Unix naming system provides an excellent example of a hierarchical naming graph and will be used to illustrate many of the ideas in the rest of this thesis. However, before considering Unix in detail, two other systems which take a very different approach to naming will be discussed briefly.

2.2.1. A Database Naming System

Aspect [Hall85] is an Integrated Project Support Environment (or IPSE) constructed from a relational database using the RM/T data model [Codd79]. In the RM/T data model every object is identified by a **surrogate**. This is a unique internal system identifier that need never be disclosed which will remain associated with the object throughout its lifetime.

Aspect implements naming with a special relation in the database called *known-as*. This associates a surrogate with an external name in the context of a name space. Name spaces are themselves objects with surrogates and may have their own external names in a further name space. However, objects do not have to be given names in a name space because their surrogate is sufficient to identify them and guarantee that they exist. The only restriction on the structure of a name space is the requirement that a given external name can only appear once within a single name space. This prevents ambiguity but still allows objects to have more than one name, possibly from within the same name space.

Aspect is able to interpret pathnames which pass through several name spaces in an obvious manner. Each user is given a default name space which can be used as a starting point for all other names.

Here is an example which shows how a simple naming graph would be represented in the database:

The pathname `Robert/test/A` identifies the object whose surrogate is s(1). This pathname is relative to a default name space whose surrogate is s(6). However,

surrogate	external-name	name-space
s(1)	A	s(3)
s(2)	B	s(3)
s(3)	test	s(5)
s(4)	doc	s(5)
s(5)	Robert	s(6)

s(6) does not have a name and therefore only appears in the *name-space* column of the *known-as* relation.

The Aspect *known-as* relation is a relational representation of a naming graph with multiple source nodes and unidirectional labelled arcs. The labels on the arcs correspond to external names. The nodes are labelled with surrogates.

Surrogates are a way of separating the problem of accessing an object from the problem of identifying an object. The *known-as* relation is a very flexible naming mechanism which makes it easy for objects to have more than one name. However, because objects are ultimately identified to the system by surrogates rather than names, it is not necessary to give every object an external name. Surrogates may also be stored in other relations allowing objects to be selected by their properties rather than their names. The RM/T data model will guarantee referential integrity by ensuring that all the surrogates stored in the data base refer to objects which actually exist. Surrogates act as keys to relations which define objects and this check effectively prevents them from being forged or used before the corresponding object is defined (or after it has been destroyed).

2.2.2. A Capability Naming System

The Flex system [Foster82] is a Programming Support Environment built on top of a capability machine. The Flex model of naming is equivalent in expressive

power to the Aspect *known-as* relation because surrogates and capabilities are effectively the same thing, but the two systems are very different in the way in which they implement naming. The Flex architecture supports closures which makes it possible to use higher-order functions (returning other functions as results) throughout the system interface. One consequence of this is that names are less important to a Flex user than they would be to the user of a conventional system. It is worth exploring the reasons why this is so.

Flex provides support for contexts in the form of *Dictionary* objects but its model of naming is actually more general than this. Whenever the Flex command interpreter *curt* is invoked, one of the arguments which must be supplied is a *find* function which will be used to resolve names. In theory, this allows an arbitrary naming scheme to be plugged into the system but in practice *curt* is always invoked automatically (either as part of logging in or from the editor) and so a default function is usually supplied. This default is a function to read dictionaries, bound into a closure with a list of default system dictionaries and private dictionaries. This version of the *find* function does not recognise pathnames so the naming system is not recursive and there is no need for dictionaries to contain the names of other dictionaries.

The Flex naming graph produced with this default naming scheme consists of several sub-graphs, one for each dictionary, with only one level of structure. In each graph, the source node corresponds to the dictionary and all the other nodes are sink nodes and correspond to dictionary entries. The same leaf node will appear in more than one graph if the corresponding object can be named from more than one dictionary. Consequently, names are represented by labelled arcs rather than labelled nodes.

Because Flex is a capability-based system, it can support structured files containing a mixture of uninterpreted text and low-level identifiers for objects

(i.e. capabilities). Whole files may be stored hierarchically in other files but need not be named because they can be identified by position alone or from a description in the surrounding text. Consequently, names are typically only used to denote large objects at the outermost level, perhaps representing workspaces for particular projects. When such a workspace is examined with the editor, the objects it contains are displayed on the screen as *Cartouches* (or icons) which can be selected with a mouse. A *Cartouche* represents a capability for an object rather than an unresolved name. In effect, Flex files are closures in which references to other files are represented as fully bound names in the form of capabilities.

In particular, the Flex separate compilation system, which is based on modules, works by including in the program text a capability for each module that a program fragment depends on, rather than just its name. This means that the program is unambiguous and may be compiled in any context without fear of picking up the wrong version of a library module by resolving its name in the wrong context. The name has already been resolved so that the program text is really a closure.

Similarly, a module is a closure containing references to its source code, object code and interface specification. Once a module has been created these values can be updated atomically without altering the capability for the module itself. This use of indirection means that the capabilities for modules embedded in program texts always refer to the latest version of those modules.

To summarise, Flex is able to dispense with names most of the time because it has an iconic interface which allows objects to be pointed at directly and because, being built on a capability machine, it can use capabilities to allow direct access to identifiers safely, without compromising the integrity of the machine. In effect, the capabilities for objects represented graphically on the screen by *Cartouches* are really names but the interactive interface and two-dimensional presentation

of information give the illusion that names are not necessary and this is certainly true in the conventional sense.

2.3. Naming in Unix

Aspect and Flex both rely on some form of globally unique identifier to implement their naming schemes. In Aspect the identifiers are called surrogates whereas in Flex they are called capabilities. Like any other naming system, Unix must also rely on a unique identifier internally to identify objects unambiguously but Unix differs by not requiring these identifiers to be globally unique and this makes it possible to combine Unix name spaces recursively. In the rest of this chapter we will consider the structure of a single Unix name space and in the next chapter we will show how Unix supports more than one name space and allows several name spaces to be combined more or less transparently.

Unix consists of an operating system kernel and a series of utilities. Although the basic support for the various Unix naming spaces is provided by the kernel, many of the utilities extend the naming facilities by adhering to a series of conventions. Most of this section is concerned with naming in the Unix file system but other forms of Unix naming are briefly discussed in section 2.3.7.

2.3.1. Inodes, Pathnames and Directories

The Unix file system supports a name space based on a tree-structured naming graph. For the purposes of this discussion, the file system contains two sorts of object: files and directories. Directories provide the naming contexts in which the pathnames used to identify objects are resolved. Files are the leaf nodes in which information belonging to users of the system is actually stored.

Internally, the Unix kernel represents all file system objects by **inodes** which contain information about the location of the object, its owner, access rights, creation date and so on. Inodes are identified by small consecutive integers called

inode numbers which are actually indexes into a table of inodes. An object's inode number is an abstraction of its identity and inodes are used internally as names by the kernel. However, inodes may not be accessed directly by the user of the file system because objects may only be named with pathnames. The kernel name resolution algorithm maps pathnames into inode numbers using the information contained in directories.

A pathname starts from a known directory and progressively traverses the naming graph via other directories until the object it names is reached. A Unix directory is a context containing a list of bindings between simple names and inode numbers. Name resolution proceeds by matching each component of the pathname against an entry in the appropriate directory to obtain the inode number of the next directory in the chain (or eventually, when the pathname is exhausted, the inode number of the object the pathname denotes).

In the more abstract terminology of section 2.1.4, the bindings in a Unix directory correspond to labelled naming arcs in the graph. Because the arcs rather than the nodes are labelled, it is possible for an object to have more than one name from the same context or to be reachable from more than one context.

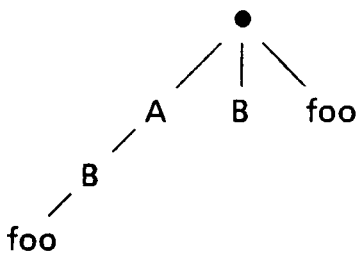
Every reference to an object from a directory is called a **link** in Unix terminology. Two directory entries denote the same object if they bind two names to the same inode number. (The names need not be the same.) Apart from their names, all the links to an object are equivalent and it is impossible to distinguish the first link to an object from subsequent links. Since Unix does not support anonymous objects, an object must always be created with a link (i.e. a name). When the last link to an object is deleted, the object is no longer accessible and can be destroyed by the kernel.

Although it is not possible to access inodes directly, it is possible to map pathnames into inode numbers outside the kernel and hence determine whether

two pathnames denote the same object. This is the only way of testing for identity which is unfortunate because making low-level identifiers visible in this way makes it difficult to join name spaces together transparently. These difficulties will be explored in chapter 3 and then again in chapter 6. However, there is no easy solution.

2.3.2. Unix Pathnames and Contexts

A Unix pathname consists of a series of simple names denoting the directories it passes through, separated by / to prevent ambiguity. Thus, if the object `foo` can be accessed as an entry in the directory `B` found in directory `A` then the pathname to reach `foo` unambiguously from the directory which contains `A` is the pathname `A/B/foo`. Notice that the directory from which the pathname starts



may itself contain an entry called `B` or even `foo` but that these entries do not necessarily refer to the same `B` and `foo` as the pathname `A/B/foo`. It should therefore be clear that pathnames are only meaningful when their starting point (i.e. their context) is known. Consequently there needs to be a way of naming contexts. This is the bootstrapping problem discussed in section 2.1.2. The base of the Unix naming tree has an address known to the bootstrap program (inode 2) and all other contexts are ultimately named relative to this point.

Unix pathnames may begin from either of two contexts. These contexts are defined individually for each process rather than globally for all processes, making all names not only context relative but also process relative. However, in

practice all processes share the same definition for one of these contexts, the root context, and therefore to all intents and purposes root-relative names are absolute names whose meaning does not depend on the process which uses them.

Strictly speaking, one naming context would suffice for all names but absolute names are somewhat unwieldy and too precise. The provision of a second naming context, the current directory context, makes it possible to use much shorter names to refer to objects relative to some local point in the tree without needing to know the absolute location of that point. This is a form of location transparency which makes names more abstract. However, the dynamic definition of the current directory context means that such names are only unambiguous from within a closure which binds them to a particular directory. Unfortunately, Unix does not provide such a mechanism (unlike Flex) and consequently names cannot be statically bound in programs or passed between contexts unambiguously (see section 2.1.2).

The syntax of pathnames makes it clear whether they begin from the root context or the current directory context. If the pathname begins with `/` it is relative to root; otherwise, it is relative to the current directory. Thus, the pathname `/A` names object `A` in the root directory whereas simply `A` names a different object `A` in the current directory (unless of course the current directory happens to coincide with the root directory which is perfectly possible). `/` denotes the root directory itself and therefore it would seem logical and consistent that the empty pathname should denote the current directory. Historically, this was indeed the case but nowadays the empty pathname is specifically excluded from the definition of the pathname syntax given in at least one of the (regrettably many) Unix standards documents, the System V Interface Definition, otherwise known as the SVID [AT&T85]. A similar argument can be applied to interpret malformed pathnames such as `A//B`. By arguing that there is a null name between the two slashes which by analogy names the context in which it is

interpreted, it becomes clear that `A//B` is the same as `A/B`, just as `A/` is the same as `A`. Again, these interpretations are illegal or at best undefined by the SVID although they are perfectly consistent.

Of course, there are occasions where it is necessary to name the current context explicitly and giving it an empty name is rather messy if not ambiguous to the casual observer. There is a distinction between applying a command to no arguments and applying it to an empty argument but it is too subtle a distinction, even by Unix standards of brevity and obscurity. Instead, there is a convention that every directory contains a entry for itself whose name is `.` (pronounced "dot"). This guarantees that the pathname `.` always names the current context. Similarly, pathnames such as `A/./B` may be simplified to `A/B` and the pathnames `A/.` and `./A` may be written more simply as just `A`.

Unix allows both the root context and the current directory to be redefined. Naturally the name of the new context in each case can only be given relative to the old context (or to the other context which remains unaffected) and consequently names will always be relative to some point in the naming tree. There is no concept of an absolute name because there is no fixed name for the base of the naming tree. (In this sense, the Unix naming graph does not have any source nodes.) Even though the root context normally corresponds to the base of the naming tree, it may be redefined by an individual process so that the definition of root is not even guaranteed to be consistent throughout the system.

Despite this, it is normal (and indeed prudent) to keep the root context fixed at the base of the naming tree so that to all intents and purposes it can be used as an absolute naming point. The correct operation of the Unix system depends on the existence of certain directories and files whose root-relative (and therefore supposedly absolute) names are embedded in various utilities and even the kernel itself. If root were to be moved to an arbitrary point in the naming tree without

ensuring that these root-relative names were still valid from the new location, then the Unix utilities which depended on their presence would not work correctly. Indeed, serious breaches of Unix security would be possible if root could be moved to an arbitrary location because it would be possible to substitute bogus versions of these system files. Consequently, the root directory may only be redefined by privileged users. However, there is no check to ensure that root is only moved to a position in the naming tree which provides the necessary system files and sub-directories. Unix really needs the concept of a root-directory type in the file system to control the positioning of root or even some alternative mechanism such as a closure for naming system objects.

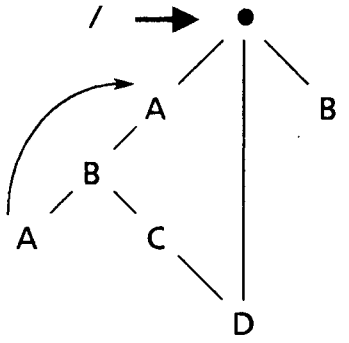
Even with these difficulties and potential problems, there are still occasions when the ability to redefine root is useful. The facility was originally introduced to allow several subsystems to co-exist within a single Unix system. However, although this might appear to be a recursive notion which generalises nicely to a transparent distributed Unix system, in fact the idea of root denoting a system context as well as an absolute naming point causes problems in a distributed environment as we shall see in the next chapter.

2.3.3. The Unix Naming Tree and . .

For the reasons discussed at the end of section 2.1.4, it is highly desirable that the Unix naming structure be a tree rather than an arbitrary graph. However, nothing discussed so far has been sufficient to guarantee this. Indeed, because Unix allows an object to have several different names (or links), any directory may refer to any other object, file or directory, and consequently it is theoretically possible to create circularities in the naming graph.

For example, suppose `/` contains a directory `A` which contains a directory `B`. If `B` contains a link to `/A` called `A`, then the pathname `/A/B/A` is the same as `/A`, and indeed the sequence `/A/B/A/B` etc. can be repeated indefinitely without

getting anywhere. On the other hand, there is no harm in providing a shorthand



notation for the directory `/A/B/C/D` by creating a link to it called `/D` since this does not create such circularities.

The difficulty is in deciding which links cause circularities and which do not. One approach might be to allow links down the tree but not up the tree, but this could not be made to work correctly for links between two separate branches of the tree. Unix sidesteps the problems of defining such an algorithm by simply not allowing links to be made to directories, thereby ensuring that each directory has only one name. This restricts the Unix naming graph to a lattice-like structure in which only the leaf nodes (i.e. the files rather than the directories) can have more than one name. Although such a graph is not necessarily a tree structure because it could have more than one starting point, it will be at most a forest of distinct trees with some leaf nodes in common; it is impossible for two trees in such a structure to share a branch node (i.e. directory) without that directory having two names or links (which is explicitly forbidden by construction). Indeed, since there are only two starting contexts for pathnames (root and current directory), those objects which can be named are restricted to the two trees whose starting nodes are these two contexts. All other parts of the graph are unreachable with simple directional pathnames. Imposing the restriction that the current directory context can always be named from the root context ensures that the current directory name tree is a subtree of the root name tree so that to all intents and

purposes the Unix naming graph is a single tree whose source is the root context (hence the name "root"). This restriction is easy to enforce; it suffices for the bootstrap process to bring the Unix system into existence in such a way that the first process (from which all other processes are descended) has its current context equal to its root context. All names thereafter will be relative to this original root context which will naturally correspond to the base of the tree.

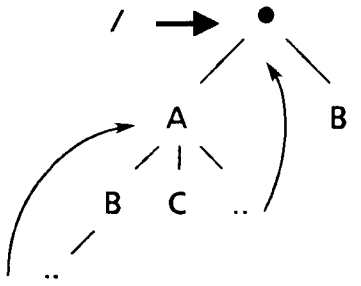
In fact, the Unix naming system is not quite as restricted as this description of a unidirectional tree might imply. The main disadvantage of the scheme just described is that names can only move down the tree. In particular, if the current context is repositioned outside its naming subtree, its new position must be described relative to the root context because there is no other way of naming other parts of the tree. Although this may not matter if the current context is not repositioned very often or if it is repositioned to somewhere completely unrelated to its current position so that a root relative name is more natural, this restriction also prevents nearby objects in sideways related parts of the tree (such as uncle and cousin nodes) from being named relative to the current context. The only way in which such nodes could be named would be by repositioning the current context at the common ancestor node (e.g. grandfather or great grandfather).

To overcome this difficulty, Unix directories always contain a second special name . . (pronounced "dotdot"). The . . entry in a directory refers to the unique parent of that directory and this allows movement up the naming tree, one step at a time, from the current context. Because the graph is tree-structured, no directory can have two parents and hence . . is defined unambiguously.

Apart from . and . . links cannot be created to directories. Recognising . and . . as special cases allows the Unix file system to use reference counting to implement its garbage collection. Every inode contains a link count and when the last link to the inode is deleted, the file it represents is also deleted. Thus,

although . and . . permit circular or at least redundant pathnames, they do so in a controlled manner.

For example, if the current context is positioned in /A/B then . . refers to /A and . . /C refers to /A/C. However, any sequence of the form A/. . /B in a pathname is clearly redundant and can be simplified to B.



Thus, it is possible to transform a pathname involving . and . . into a canonical form.

2.3.4. Canonical Pathnames

The concept of a **canonical pathname** is important and applicable to any tree-structured naming graph (so in particular to the Unix file system naming tree). Pathnames need not necessarily be in their simplest form (especially if they are machine generated) and it is useful to be able to translate an arbitrarily complex redundant pathname into the most direct route between the starting context for the name and the object it denotes. Furthermore, because the naming graph is tree-structured, it is possible to perform this translation without needing to know about names elsewhere in the graph.

Being able to reduce an arbitrary pathname to its simplest form statically (before it is resolved) rather than dynamically (as it is resolved) makes the name resolution process more efficient. This is particularly important for a transparently distributed system where a name might span several distinct

naming trees on systems linked only by a network. Clearly if name resolution involves sending messages across a network, and if the number of messages depends somehow on the complexity of the pathname, then it makes sense to minimise the number of messages sent by simplifying the name as much as possible before attempting to resolve it. We will be returning to this point in chapters 5 and 6.

Canonical pathnames also make it possible to compare pathnames by first reducing them to their canonical form and hence determining whether they are equal (i.e. denote the same object). However, the concept of a canonical pathname is only valid for a tree-structured graph and perhaps this is too restrictive. If the naming graph is not tree-structured, it may be possible to reach a given object by two equally acceptable paths of the same length, in which case no sensible definition of the canonical (i.e. most natural) path will be possible. This poses several questions. If more general acyclic graphs which allow objects to have more than one name are useful then is there an alternative algorithm which can determine whether two pathnames are equal and is this a useful thing to do in any case? Chapter 6 will consider this problem in more detail.

The canonical transformation for Unix pathnames seems relatively straightforward at first glance. Every occurrence of . can be omitted (except perhaps the first to prevent a null pathname) and every occurrence of . . preceded by a name (other than . or . .) can also be eliminated along with that name. This algorithm may be described by the following context-free transformations:

(a) ./X → X

(b) X/../Y → Y

After performing these simplifications repeatedly, eventually any remaining . . components of the pathname will move to the front of the path whilst the other name components move to the back. In other words, a canonical Unix pathname

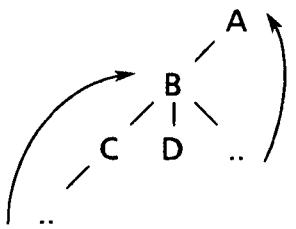
optionally goes up one or more levels using . . and then comes back down the tree again through a series of named nodes.

But this algorithm has overlooked two important points. Firstly, there is the question of what / . . means. In other words, is it possible to move upwards from the root context? As we discussed earlier in section 2.3.2, although it is usual for the root context / to correspond to the base of the naming tree, Unix allows / to be redefined so that it is really a relative name rather than an absolute name. Clearly, the base of the naming tree can have no parent directory and so by convention its . . entry has the same meaning as its . entry and points to the base directory itself. (It would be equally appropriate for it to have no . . entry at all.) If / corresponds to the base directory as it nearly always does for an individual Unix system then clearly / . . will be the same as /. However, as we will see in the next chapter, if a group of individual Unix systems have been grouped together in a larger naming tree to form a transparently distributed Unix system and if / still refers to the root of a particular system, there may be an arbitrary amount of naming structure between / and the base of the larger naming tree. Therefore, in general / . . should have no special meaning but should simply refer to the parent directory of root. Applying . . repeatedly to / will eventually reach the base of the tree.

This interpretation of / . . gives a system with an **open** root. In view of the fact that / is used to name important objects such as system directories and files upon which the correct execution of the rest of the Unix system depends, there is also a case for a **closed** root in which / . . is always defined as /, regardless of whether it actually corresponds to the base of the naming tree or not. This might seem perverse because it makes the part of the naming tree which is outside this / forever inaccessible but this is just what is required in order to create a self-contained subsystem (i.e. a Unix system within a Unix system). This is inwards recursion and the fact that it is possible on an unchanged Unix system bodes well

for the concept of outwards recursion, building bigger systems out of smaller systems rather than decomposing bigger systems into smaller systems. However, simply treating `/..` as a special case in the name resolution algorithm is an unpleasant compromise. Early Unix systems implemented an open root but more recently the trend has been towards a closed root. Unfortunately, although `/` should be just a naming context, it has acquired an extra significance as a way of identifying a Unix system.

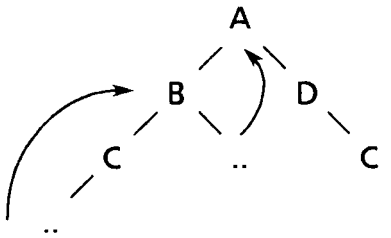
The second difficulty with this canonical transformation algorithm is that it is too simplistic and does not always result in a pathname in its simplest form. For example, suppose the current context is `/A/B/C`. Then the pathname `../../B/C`, although apparently in canonical form, may be simplified to the empty pathname (although this would more usually be written as `.`).



The named portion of the path simply retraces the steps made up the tree by the `..` portion. Similarly, `../../B/D` could better be expressed as `../D`. In this case, only part of the `..` sequence has been undone, namely the innermost `../B`. The redundant part of such pathnames is always centred around the highest point they reach in the naming tree. We will refer to this point as the **centre** of a pathname.

Eliminating this form of redundancy is much harder because it is context dependent. The simplification is only possible if the full pathname of the starting context is known. Notice that this really must be the full pathname from the base of the tree, not just the pathname from the root context, since otherwise it would

be impossible to apply the simplification to pathnames which entered the unknown region above root. Given this full pathname, the simplification consists of matching the tail of the candidate pathname against the corresponding tail of the full name of its starting context and eliminating matching entries and their corresponding `..` from the centre outwards until no further eliminations are possible. Elimination must be from the centre outwards to prevent errors in a case such as the path `../../D/C` relative to the context `/A/B/C`. Although the



trailing `C` may match, it occurs as part of a different subtree (from `D` rather than `B`) and so no simplification is possible.

Whereas the Unix kernel need normally only store the inode number of the current context to resolve pathnames, the canonical transformation algorithm requires knowledge of its full pathname from the base of the tree. Although this knowledge can be acquired incrementally as the context is changed, it still requires having to store an arbitrary amount of non-local information about a local context. Furthermore, the whole concept of a full pathname only works well with an absolute immovable closed root context corresponding to the base of the naming tree (in which case the full pathname is the same as the root-relative pathname). If it is possible to add naming structure above the root context and in particular to move the base of the naming tree further away from the root named by `/` then the value of the full pathname will change as the base recedes from root.

For example, if the full pathname was /X/Y/Z and then a directory containing the original base as subdirectory W was made the base of the tree, the new full pathname would be /W/X/Y/Z. Similar changes to the full pathname would be necessary if structure was deleted from the top of the tree, perhaps in splitting a recursively constructed system into subsystems.

All this complexity is caused by the fact that .. is essentially an anonymous name for the parent directory. A sequence of the form ../.. etc. may take a pathname arbitrarily far from its starting point so that an arbitrary amount of contextual information is required to apply the simplification. It is curious that X/.. may always be eliminated whereas the symmetrical case ../X cannot. The difference is simply that in the first case the X provides enough knowledge of the position relative to the unknown current context to cancel out the effects of .. whereas in the second case the .. occurs first and simply compounds the unknown. Pathnames should have the Markovian property that their meaning is independent of the history of their starting context but the existence of .. makes this impossible.

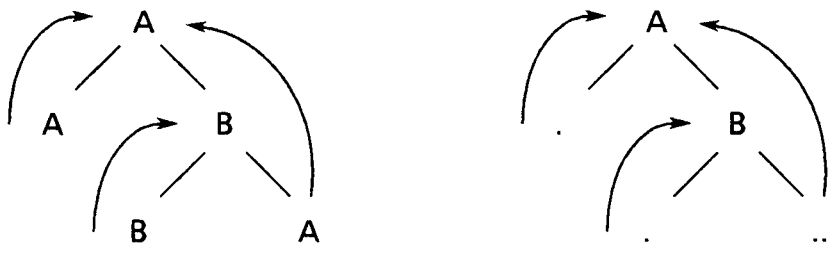
2.3.5. An Alternative to ..

There is actually a very simple alternative to .. which eliminates this complexity and makes it much simpler to construct canonical pathnames. By imposing a slight restriction on the choice of names in a given context, it is possible to give a rule for simplifying redundant pathnames which does not require an arbitrary amount of non-local knowledge to be stored but rather depends on a locality property that is valid at every point in the naming tree.

The basic idea behind the new algorithm is to avoid the problems caused by .. by always referring to directories explicitly by name. In a tree-structured naming graph, every directory has only one parent and hence only one name. This name can be used to replace the . entry in the directory itself and the .. entries in any

sub-directories it may have. Only the base of the naming tree is problematical because it is unique amongst directories in having no name (i.e. no entry other than `..` for it in another directory). Clearly it must be named, however arbitrarily. The name `root` or `base` spelled out would suffice,

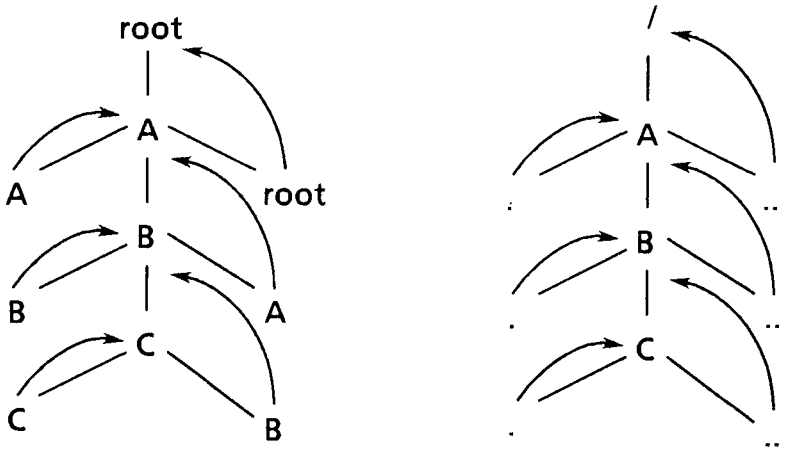
It is not sufficient to simply replace the `.` and `..` entries with the real names of the directories they denote. To prevent ambiguity we must ensure that if a directory is to be called `A` for example then no other entry in itself, its parent or any of its child directories has the same name. Indeed, all of these directories



(a) new form (b) old form

must already contain an entry called `A` if the tree is well-formed and fully connected. These entries would be `.` or `..` under the original Unix naming scheme (apart from the entry which defines `A` in its parent directory).

This restriction prevents names of the form `A/A/A/A` etc. which will always simplify to `A` of course but it does not go to the extreme of requiring that every directory have a unique name. In fact, names must only be unique within pairs of consecutive directories, allowing the parent and child of a given directory to use the same name for a different object. Thus, `A/B/C` and `C` are not necessarily the same but `A/B/A` is always the same as `A`. This observation is the essence of the new canonical naming algorithm. Any sequence of the form `X/Y/X` in a pathname may be replaced with simply `X`. After applying this transformation repeatedly until no further simplification is possible, the pathname will be in canonical form.



(a) new form (b) old form

There is one exceptional case, namely a pathname which loops back to its starting context. This will always simplify to `P/C` where `P` is the name of the parent context and `C` is the name of the starting context. Introducing the convention that all pathnames must begin with the name of their starting context eliminates this problem. Instead of simplifying to `P/C`, the pathname reduces to `C/P/C` which can further be simplified to `C`, effectively the null pathname. For example, consider the pathname `../../B/C` in the context `/A/B/C` which caused difficulties for the original algorithm. Under the new scheme, this would be written as `C/B/A/B/C` which simplifies via `C/B/C` to just `C`. Similarly, `../../B/D` may be written as `C/B/A/B/D` which just simplifies to `C/B/D`. This may be translated back into Unix notation as `./../D` or simply `../D`.

Unfortunately, this algorithm is only applicable to a tree-structured graph. A more general graph will allow there to be more than one path between two points on the graph, making the whole concept of a canonical pathname meaningless. Regrettably, the Unix naming graph is not a pure tree because files may have more than one name. Consequently, this algorithm does not work with general Unix pathnames but it can be used to simplify pathnames between directories and hence ensure that a given pathname for a file does not visit more directories

than necessary, even though an alternative pathname may also exist. This is particularly important for a distributed Unix system because it reduces the cost of resolving pathnames which span more than one system, as we shall see in chapter 6.

2.3.6. Symbolic Links

Some versions of Unix provide another naming feature which further complicates the idea of a canonical path. A **symbolic link** is a third type of object in the naming tree (besides directories and files) and contains a pathname. This is used to provide an indirection or aliassing feature. During name resolution, whenever a symbolic link is reached, the remainder of the pathname currently being resolved is interpreted in the context denoted by the value of the symbolic link. (In effect, name resolution continues after prefixing the contents of the link to the unresolved portion of the pathname.) This redirection may occur several times during name resolution but since a symbolic link may point to a directory (or even to itself) loops are possible. Consequently, the kernel limits the number of redirections that can be made during the resolution of a single pathname and assumes that if this number is exceeded there is a loop in the naming graph.

Symbolic links are a useful way of hiding the directory structure and may be used as a forwarding mechanism when a subtree in the name space is moved elsewhere. For example, suppose individual user directories are stored as subtrees of the `/user` directory which has to be moved to `/usr/fs` for some reason. If `/user` is made into a symbolic link to `/usr/fs`, old pathnames of the form `/user/robert` will still work.

Symbolic links effectively allow links between directories so that the Unix naming graph is no longer tree-structured. Without knowing which nodes in the graph are really symbolic links, it is impossible to reduce an arbitrary pathname to its simplest form statically. Of course, it may be reduced dynamically by

simulating the kernel name resolution algorithm and tracing it through the naming graph but the whole point of the canonical naming algorithm is to able to perform the transformation statically without requiring knowledge about names elsewhere in the graph.

A further difficulty is the semantics of `..` in the presence of symbolic links. If `/user` is a symbolic link to `/usr/fs`, does `/user/..` denote `/` or `/usr`? The answer will depend on whether `..` is interpreted statically ("where I am now") or dynamically ("how did I get here"). Because `..` is simply a special entry in a directory and Unix keeps no record of the path by which a given context was reached, its interpretation of `..` is static, even though a dynamic interpretation would work more naturally with symbolic links. Consequently, `/user/..` is interpreted as `/usr` rather than `/`. This can cause unexpected anomalies with pathnames of the form `../X` when the current context has been reached unknowingly via a symbolic link and breaks the canonical simplification of `X/../Y` to `Y`.

Symbolic links have another curious characteristic. Their value is a pathname and if this begins with a `/` it is interpreted relative to root as might be expected. However, if on the contrary the pathname contained in the symbolic link does not begin with a `/`, it is interpreted relative to the directory in which the link is found rather than the current directory. Thus, absolute symbolic links are in fact relative to a dynamic definition of root which may have changed since the link was created whereas relative symbolic links are in fact absolute because they are not affected by the definition of either root or the current directory at the time when the link is resolved! This distinction is particularly important in a transparent distributed system where processes from different systems may have different definitions of root and may therefore interpret the same symbolic link in different ways.

Symbolic links would be much more useful if they were implemented as true closures, defining the context in which they were to be resolved. This would make symbolic links not relative to root work sensibly. Their absolute semantics is counter-intuitive and makes them behave differently from an ordinary Unix link. Creating a link in another directory to a file in the current directory has a quite different effect from creating a symbolic link in the same directory to the same file! The problem is that the pathname value of the symbolic link is simply a string of characters which is not interpreted in any way until the symbolic link is resolved. Consequently, it is possible to create symbolic links to non-existent objects and symbolic links do not behave like true aliases in the sense of section 2.1.3 because they are not deleted when the object they reference is deleted. Although they were introduced to overcome some limitations of conventional Unix links, they have managed to muddle Unix naming semantics by confusing dynamic and static name resolution. If they behaved like real links or real aliases they would be tolerable but instead they are an unpleasant kludge.

2.3.7. Other Unix Name Spaces

The Unix file system naming space is based on hierarchical names, but the other name spaces supported by the Unix kernel are completely flat. Processes and users have unstructured names consisting of simple integers. There is no equivalent of a directory or a pathname. These names are globally unique rather than relative to some context. In effect, they are relative to some implicit system context. However, because the context is implicit, it is difficult to extend such names to a transparently distributed Unix system made up of individual systems. Each system will contribute its own processes and users but their names will no longer be globally unique nor will it be possible to distinguish names belonging to different systems. We will return to this problem in chapter 3 when we have discussed ways of joining name spaces together to build distributed systems.

2.4. Conclusions

We have discussed many naming concepts in theory and shown how they have been implemented in practice by three different systems: Aspect, Flex and Unix. Internally, all three systems use a unique identifier to identify objects but they differ in the structure of the naming graph they allow. Aspect and Flex are built on top of abstract machines which allow low-level identifiers to be manipulated as first-class objects. It is impossible to forge identifiers or use them inconsistently and consequently arbitrary naming structures can be created very easily. However, this simplicity and flexibility is offset by the hidden cost of implementing the underlying abstract machine. Unix takes a more pragmatic approach, restricting the naming graph to a tree structure and not allowing internal identifiers to be manipulated directly.

A tree-structured graph has the useful property that there is a unique shortest path between any two points on the graph. We have shown that this makes it possible to define the concept of a canonical pathname which can be used to simplify redundant pathnames automatically. However, several features of Unix such as the anonymous . . directory and the presence of links (especially symbolic links) make this concept less useful than it could be, although arguably a tree-structured naming graph is too restrictive in any case. Another problem area is the way in which Unix depends on root-relative pathnames to name system objects, confusing the concept of root as a naming context with the notion of a system. Many of these difficulties could be alleviated by the use of closure objects but the resulting system might look quite unlike Unix. However, it is possible to use closures to solve naming problems, as the Flex system described in section 2.2.2 demonstrates.

Chapter 3
Joining Name Spaces Together

Chapter 2 discussed name spaces as if they were self-contained entities existing in isolation of each other. In practice, when a group of systems is joined together by a network, each system will have its own name space and these name spaces must be merged in order to construct a transparent distributed system. If the individual name spaces are still distinguishable in the distributed system then transparency has not been achieved.

Joining name spaces together to build a distributed system is a useful way of sharing objects and other resources between systems across a network. It is a recursive mechanism for combining name spaces to build bigger name spaces. Recursion can also be used to decompose name spaces into smaller name spaces within a single system. This is a way of overcoming the management problems of scale by dividing up a large name space into smaller domains which can be administered independently. Thus, it is useful to consider mechanisms for joining name spaces together, both within a single system and between systems. Ideally, the same recursive mechanism should be applicable at both levels if the system has a uniform naming scheme.

If it is possible to merge name spaces transparently so that the composite name space is indistinguishable from the name spaces of which it is composed then it should be possible to merge the composite name space with other name spaces recursively. In this respect, a distributed name space designed from scratch should be no different from a distributed name space built by combining existing name spaces transparently. Thus, although this thesis is mainly concerned with the evolutionary problems of building a distributed system from

existing systems, it should also be applicable to the revolutionary problems of joining together distributed systems built from scratch.

In this chapter, we will discuss some mechanisms for joining name spaces together. One approach, taken by systems such as Aspect and Flex, is simply to assume the existence of globally unique identifiers. Obviously, if global uniqueness can be attained in practice, it will be possible to combine independent name spaces without conflict. However, the problem of managing a large flat space of identifiers without any structure will remain. We will consider these issues in more detail in the next chapter and explore whether global uniqueness really is attainable. In the meantime, we will concentrate on the mechanisms within Unix for joining name spaces together.

Unix does not rely on globally unique identifiers and it is therefore possible to combine Unix name spaces recursively within a single Unix system to form a larger name space. Although it is not quite transparent, this mechanism has been generalised to allow Unix systems to be combined across a network to form distributed systems. This chapter discusses some of the distributed Unix systems which have been built and analyses some problematical areas of the Unix semantics in greater detail. These problem areas are not specific to Unix but must be tackled by the designer of any transparent distributed system, evolutionary or revolutionary.

3.1. First Principles

The purpose of a naming system at any level is to map names into internal identifiers. Thus, a naming system actually involves two name spaces: external names visible to its clients and internal names known only to the system. Typically, the internal names come from a flat naming space and are closely related to the physical location of the object they identify. Conversely, the external names come from a highly structured name space and are location

transparent, allowing names and hence objects to be grouped together in a way which reflects the organisational needs of the user rather than those of the system.

When two name spaces are joined together they may be merged at either of these levels. If their internal name spaces are merged by extending the internal names to identify objects in one naming system or the other, their external name spaces need not be affected. Although the mapping from external name to internal identifier will have been changed to reflect the larger internal name space, this change has occurred internally. Externally, names will continue to be location independent and there will be no indication that two name spaces have been merged. In other words, the name given to an object need not depend on the system from which it originates.

Of course, because the result of merging two name spaces transparently is itself a name space, the external name spaces must be combined in some way. Otherwise, unless it is possible to use internal identifiers directly, there will be no way of referring to an object from another name space. The two naming graphs may either be joined in their entirety at some extreme point, thus preserving them intact within a larger graph, or else partially or even completely merged to share some sub-structure.

Once two name spaces have been joined it will be possible to name objects from either name space quite transparently. However, until the name spaces have been joined there will be no transparent way of naming an object from the other name space and consequently the actual join operation must use some non-transparent form of naming to indicate which parts of which name spaces are to be joined. This requires some external scheme for naming name spaces outside the naming system or perhaps the direct use of internal identifiers.

Merging name spaces involves resolving conflicts at both the internal and external level of the system. Internal system identifiers are usually assumed to be unique within the implicit context of a single naming system. When two such systems are combined, their internal identifiers are no longer sufficient to identify objects uniquely within the combined system. They must either be qualified with the identity of the naming system to which they refer or else be replaced with some other identifier that is unique in the larger scope of the combined naming system.

Merging external name spaces is not so difficult because they are usually already structured and therefore already have an explicit notion of context. Rather than resolving conflicts across the implicit context of an entire flat name space, conflicts need only be resolved locally within a limited context. Objects can be renamed or the problem can simply be avoided by keeping both contexts in the merged graph. It is always possible to combine two name spaces in their entirety by simply giving their source nodes names in a new context and making no further attempt to merge them. This approach may be used to combine two contexts at any level in the system

The result of joining two name spaces together is another name space. If this construction is truly transparent, it should be possible to apply it recursively, joining composite name spaces together to produce even bigger name spaces. This observation may be expressed in terms of the recursive structuring principle:

> "A composite system should be functionally equivalent to
> the systems of which it is composed."

Although full transparency is the ideal, it may not always be achievable in practice. However, it is possible to compromise. The composite name space may not attempt to hide the individual name spaces from which it is constructed but simply group them together loosely, providing limited support for names which

cross the internal boundaries between name spaces. The overall effect will be to give the illusion of a single naming space but peculiar restrictions on naming will expose discontinuities at the points where individual name spaces were joined together.

For example, the Unix mount mechanism (which will be discussed in section 3.2.2) joins name spaces together at a single point by making a leaf node in one name space refer to a source node in the other. Apart from this single name that crosses the name space boundary, there is no other way of creating a reference from one name space to the other. This restriction tends to highlight the boundary between the two name spaces because it does not apply to a single name space. If the name spaces had been joined together completely transparently, there would no such restriction.

3.2. Joining Name Spaces Together within a Unix System

The rather abstract principles of the previous section will be illustrated with a concrete example, the Unix file system. During the discussion of Unix naming in section 2.3 we deliberately described naming in terms of a single atomic name space. In practice, the Unix name space can be subdivided into smaller name spaces, even within a single system.

3.2.1. Inodes and Devices

Section 2.3.1 described how the various objects in the Unix file system are represented by inodes, with directories providing the association between names and inode numbers needed to resolve pathnames. Inode numbers are simply small integers but are not guaranteed to be unique across an entire Unix system. Instead the Unix name space is partitioned into subspaces called devices in which inode numbers are unique. Devices correspond to physical storage media such as removable disk packs or partitions of fixed disks. Because inode numbers are not

unique and because directory entries contain no device identifier, an object must be named from a directory on the same device on which it is stored. Directory entries (or links) referring to objects on other devices are prevented by the implementation. Although Unix provides a mechanism for joining the name spaces stored on individual devices into a single composite name space, the boundaries between the individual name spaces are still visible in the sense that it is not possible to create a link to an arbitrary object from anywhere else in the name hierarchy. Links are restricted to being within a single name space (or device) and cross-device links are not allowed. This means that name spaces can only be joined together at a single point in the naming graph and consequently there is a strong correlation between the global pathname to an object and the name space to which it belongs. Since name spaces are associated with physical devices, names are no longer location independent and a useful form of transparency has been lost.

Obviously, if Unix directory entries were to be extended to allow a device number, this could be used together with the inode number to identify any object on any device in the system uniquely and hence allow cross-device links. However, there are two important issues to be considered here. Firstly, there is a trade-off between the extra space taken up by the larger directory entries and the frequency with which cross-device links will be required. In other words, there is a trade-off between locality of reference and space-efficiency. Secondly, and more importantly, it is possible for an object referred to by a cross-device link to become temporarily unavailable, for example if the disk on which it resides were to be removed. The complications this causes for the naming algorithms were discussed in section 2.1.3.

Extending the size of the directory entry solves the problem at one level but is not a recursive solution to the more general problem. Unless the low-level device and inode number pairs are themselves globally unique across all possible Unix

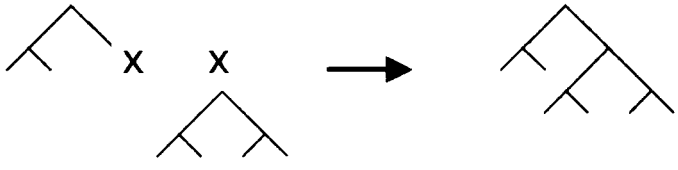
systems, it will still not be possible to join Unix systems together recursively and allow cross-system links for precisely the same reason that the non-uniqueness of inode numbers prevented cross-device links. Again, the introduction of unique system identifiers would appear to solve the problem, but only until it became necessary to introduce more structure and recursively compose systems of systems. The basic problem is that introducing a flat name space at any level restricts the growth of the system beyond that point unless the global uniqueness of names can be guaranteed between all existing and potential systems at that level. On the other hand, extensible sequences of locally unique identifiers are amenable to recursive construction techniques, providing it is feasible to use such complex names at the lowest level in the system.

Ultimately, all objects must be uniquely identified by the system which defines them, since otherwise there is no concept of identity and the distinction between objects becomes meaningless. The difficulty is in deciding what form that identifier should take and what constraints its choice of value should impose on the construction of other systems which may be merged with the local system. In particular, is it reasonable to hope for transparency at every level of a recursively constructed system or is there a balance point at which the cost of providing the extra level of transparency outweighs its benefits? Should we aim for the illusion of a single system with its own internally unique set of identifiers or a compromise solution in which internal system boundaries are visible because implementation constraints make the use of globally unique identifiers internally impossible? There is little point in providing functionality which will not be used or is not required. Often the best compromise is to make the common cases work well and the rare cases possible.

3.2.2. Joining Name Spaces Together with Mount

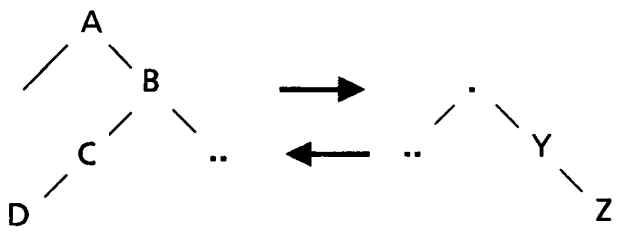
So far we have discussed the impact that joining name spaces together has on the low-level identifiers used internally by a system. If it is possible to hide the boundaries between name spaces internally by extending the range and uniqueness property of the internal identifiers to cover the composite name space then there is no reason why the high level pathnames seen by users of the system need be affected. However, in general this kind of internal modification is not possible and will be limited in scope in any case for the reasons discussed in the previous section. Consequently, there will still be a need to extend the pathname mechanism at the higher level as an additional way of joining name spaces together. Some systems make no attempt to do this transparently but simply make the full name of an object include a device name which is distinguished from the rest of the name by a separator. This is not extensible. In contrast, the Unix concept of mount is a genuinely recursive mechanism for joining name spaces together transparently (although it does not tackle the problem of cross-device links).

The idea of mount is to join together two disjoint file systems to form a continuous whole. As far as naming is concerned a file system is just a self-contained name space stored on a single device. Being tree structured it will have a unique directory at the base of the tree and being self-contained this directory's . . entry must point to itself. The mount operation consists of overlaying a leaf directory in a local file system with the base directory of another file system (i.e. mounting one system onto another).



In achieving this, several problems must be overcome. Pathnames through the directory which is overlaid (the mount point) must cross from one name space to another. Similarly, pathnames from within the mounted volume which pass through its base directory with . . will emerge at the mount point in the parent name space and continue up the parent naming tree. It is possible to mount further naming spaces, even onto volumes that are themselves mounted; in other words, the process is recursive.

An example will illustrate these points. Suppose a naming system containing the object Y/Z has been mounted at A/B. The base directory of the mounted naming system will now coincide with the mount point A/B and so Z will now be accessible as A/B/Y/Z.



On the other hand, any objects previously accessible from B (such as a subtree beginning /A/B/C/D) will be unreachable until the Y/Z naming system is unmounted. B will coincide exactly with the base directory of the mounted volume, even as far as the . and . . entries are concerned. In particular, . . from the base directory of the mounted volume will coincide with the original meaning of B/. . (i.e. A). This will clearly require some ingenuity in the kernel name resolution because in effect . . in the base directory has become a cross-device link, a concept that is supposedly forbidden because of the way in which directory entries are implemented in terms of inode numbers which are purely local to one device. The transition across the mount point must be handled from the inside going out (. .) as well as from the outside going in.

Mount is implemented by locking the inode for the mount point and the base directory of the mounted volume into memory. Every time the name resolution algorithm goes to fetch an inode from disk (in order to read the contents of the directory it refers to and resolve the next portion of the name) it checks the table of mount points first and indirects to the inode for the base directory of the mounted volume if necessary. This handles crossing the mount point from outside; crossing it from inside via . . is rather more complex. Essentially, there has to be a special case for a . . entry occurring in the base directory of a mounted system (this directory can be recognised by its inode number which always has the same value). When a pathname includes such a . . component, it is interpreted in the original inode for the mount point rather than the inode for the base of the mounted volume. In this way, the kernel ensures that . . correctly indicates the parent directory of the mount point rather than the parent of the mounted volume's base directory which is always itself. Notice that it is not possible to modify the . . entry in the base directory at mount time to make this unnecessary because directory entries describe files just in terms of inode numbers, unique only to a particular volume, and are therefore incapable of naming files on another volume. An unfortunate consequence of this is that any user-level program that reads directory entries and pays any attention to the inode value must be particularly careful at mount points because the information in the directory is not correct. Not only does an implementation detail make itself visible but it does so in a way which violates the transparent bridging of the gap between naming systems provided by mount.

One program that has to be aware of this subtlety is the algorithm used to determine the root relative pathname of the current directory, the *pwd* program. Although the pathname can be expressed from the current directory as a sequence of . . segments corresponding to the depth of the current directory in the tree, this is not very helpful! The *pwd* command must effectively reverse this

pathname and give a name to all the anonymous . . values. This involves recursively searching the parent directory for an entry whose inode corresponds to that of the current directory; the name of that entry is the "real" name of the directory (in the sense discussed in section 2.3.5). Since the directory search will not succeed at a mount point, further ingenuity is required. There is a *stat* system call which returns details about a file, effectively the contents of its inode entry, including the inode and device number. Naturally, the inode value returned by *stat* should normally match that found in the directory entry for the file but in the presence of a mount point they will differ since the indirection implied by the mount will take place.

It is worth observing that the *pwd* algorithm would be trivial with the alternative naming scheme proposed in section 2.3.5 that avoids the use of . . altogether, provided it was possible to recognise the directory entry for the parent directory (which under the new scheme would no longer have a uniform name such as . .). Since by convention . and . . are respectively the first and second entries in every directory, the same convention could be employed with the revised naming system. The *pwd* command would then simply have to reverse the chain formed by the second entry in every directory between the current context and the root directory, the arrival at root being recognised by the use of stat to match the prospective pathname against /. The disadvantage of this scheme would be that the locality condition which ensured no ambiguity would have to extend across the mount boundary, imposing restrictions on the names in a physically distinct naming graph. This is a pessimistic way of looking at it; a more optimistic view would be that the restrictions are imposed on the choice of site for the mount point, perhaps an improvement on the current implementation which allows a mount to occur at any point in the naming tree, possibly hiding a sub-tree.

We have discussed in some detail the mechanisms by which mount joins two name spaces together and ensures that the join (almost) doesn't show. The link from inside the mounted volume via . . to the parent of the mount point has to be implemented as a special case; in effect, . . is the single cross-device link allowed. However, it would be possible to generalise this mechanism by adding a special kind of inode for references outside the naming system. Mount would then be more like completing part of a jig-saw puzzle by matching up these unresolved references to the corresponding entries in the other system. Indeed, the result would be more symmetrical, combining two name spaces to form one and resolving some pairs of unbound references while leaving others still unbound in the composite name space. Such an algorithm could be applied recursively to generate bigger name spaces out of smaller ones providing the mechanism for binding references across name spaces was extensible. This would tend to rule out anything based on unique names for naming spaces; instead, references would have to be resolved using relative names for adjoining name spaces. The idea discussed in section 2.3.5 of eliminating . . and replacing it with a locally unique name would suffice for this purpose and would give a pleasing symmetry between the implementation of directories and larger naming spaces such as devices.

Cross-device links are effectively prevented in a standard Unix system by the way in which inodes and links between directory entries are implemented. However, there are other reasons for imposing such a restriction which must be overcome before it is possible to propose an honest alternative which improves on this situation. The fact that it is possible to mount and unmount name spaces means that parts of the naming tree may not always be present. A cross-device link is effectively unbound if the device which it refers to has not been mounted. Furthermore, an object cannot be deleted while there is a remote reference to it from some other name space. This effectively removes some of the autonomy from

each name space. The implementation of cross-device links requires synchronisation and cooperation between name spaces.

Given that it is acceptable for an object to be absent sometimes, these problems are soluble by using an extra level of indirection through a dummy inode that has no local name, thereby ensuring that the reference count on a remotely accessible object can never fall to zero so long as this shadow entry exists. Deleting the cross-device link would therefore involve deleting this special anonymous reference on the local disk, permitting the object to be reclaimed when necessary. However, at mount time there would remain the difficulty of matching up the two name spaces. If it were possible to mount a different volume at the same mount point then confusion could ensue. Without introducing unique identifiers over the space of all possible mountable volumes this is basically insoluble although the probability of confusion can be made arbitrarily small by using random numbers as "pseudo-unique" identifiers. Perhaps this is not actually a problem in practice since Unix makes no checks at mount time and devices tend to be mounted in the same place. Otherwise well-known pathnames would simply not work. Consequently, a simple interface signature based on matching unresolved names would probably suffice although with malice aforethought it would be possible to forge an interface and wreak havoc, assuming the physical opportunity to substitute one volume for another was available. Exchanging purely locally unique identifiers is all very well but could theoretically fail since identical sets of such "unique" identifiers could be generated independently on independent volumes.

Of course, the symbolic links discussed in section 2.3.6 also provide a solution to the problem of creating cross-device links. Indeed, this is probably why they were introduced in the first place. Because the value of the symbolic link is a pathname rather than an inode number, it can cross mount points and refer to objects on other devices. However, the existence of a symbolic link to an object

does not guarantee the existence of the object itself because symbolic links are not real links, unlike the genuine cross-device links discussed in the previous paragraph. Because a symbolic link does not affect the reference count of the object it denotes, that object may be deleted leaving the symbolic link behind as a dangling pointer into thin air. A symbolic link is not an alias in the sense of section 2.1.3.

In conclusion, mount provides a very elegant (if not quite perfect) way of joining name spaces together. It is therefore not surprising to discover that many of the attempts to construct a transparently distributed Unix system have been based on a generalisation of mount. However before examining such transparent distributed Unix systems in section 3.4, it is instructive to put this work into historical perspective by considering some of the earlier attempts to build a distributed Unix. The resulting systems were not transparent, and the problems this caused were one of the main motivations in the development of the idea of transparency.

3.3. Non-Transparent Distributed Unix Systems

Unix was developed in a research laboratory of Bell Telephone so it is perhaps not surprising that since its very early days, attempts have been made to join Unix systems together over networks to produce some sort of distributed system. The UUCP system [Nowitz78] was one such attempt and is still in use today, forming the basis of Usenet, a worldwide collection of about 2000 Unix machines which can transfer mail and news between each other.

UUCP usually operates over serial lines and other forms of wide area network. It supports a point to point network, with the software recognising names of the form `system!pathname`. It is possible to indicate a route by concatenating system names. For example, `A!B!C` would name a file `C` on machine `B` reached via machine `A`. The local machine (which is not named in the path) need only know

about system A. It does not necessarily know about machine B and might even know about a different machine B which is why it is necessary to designate a route through machine A explicitly. However, at least within Usenet, system names are supposed to be globally unique and it is therefore possible to rely on automatic route finding software and simply use a name of the form B!C unambiguously.

This kind of naming mechanism is far from transparent. It introduces a new form of name with an unconventional separator (! rather than /) and thereby distinguishes system names from file names quite explicitly. Special utilities are required for accessing remote files as opposed to local files because no attempt has been made to integrate the UUCP commands with their Unix equivalents. For example, the Unix *cp* command cannot be used for remote file transfer because it does not recognise UUCP pathnames. Instead, a special program called *uucp* must be used to perform the copy. But perhaps this lack of transparency and failure to integrate UUCP with Unix is reasonable considering that a loosely coupled wide area network is being used for communications. After all, the remote copy will take much longer than the local copy, the source or destination machine may not always be available, special forms of authorisation may be required and it may even be necessary to perform the actual transfer offline rather than on demand. To pretend that the two types of copy operation are the same by integrating them into one command might be misleading. This is the dilemma that the designer of a transparent system must face. The whole point of transparency is to mask the distinction between local and remote objects but if it is not always natural to do so, is some transparency better than none at all (or is transparency, like virginity, an all or nothing property)?

More recently, the University of California at Berkeley have implemented the DoD Arpanet protocols in their version of Unix. As well as providing the ARPA *telnet* and *ftp* protocols which were designed for remote terminal access and file

transfer between arbitrary operating systems, Berkeley have also provided more Unix specific application protocols for remote login and remote copying of files which take care of some of the authentication issues transparently. The Unix specific utilities are more usable between Unix systems than the general purpose ARPA utilities which have to be able to cope with heterogeneous systems with little in common. There is also a facility for remotely executing shell commands which makes it possible to pipe the output of a command on one machine into the input of a command on a different machine. However, like UUCP, the naming syntax for remote objects (in this case `system:pathname`) is only recognised by certain applications. It is not possible to use an arbitrary Unix command such as *diff* with remote files unless they are first copied to the local machine. Nor is it possible to move the current directory onto a remote machine.

Rewriting every application so that it understood the new naming syntax would not help to solve the current directory problem since the current directory is a property of the process rather than the application and is used internally by the Unix kernel to resolve pathnames. It would be necessary to duplicate the entire Unix name resolution algorithm in each application. A much better solution would be to move the recognition of the new style of name from the application into the Unix kernel itself, thereby making the ability to access remote objects common to all applications. This is the idea behind the various attempts to build a transparent distributed Unix system.

3.4. Transparent Distributed Unix Systems

Achieving transparency is not just a question of moving the resolution of remote names into the kernel. Indeed, names of the form `system:pathname` are not transparent at all. They are manifestly different from ordinary Unix pathnames and make it quite explicit that the object being referred to is remote rather than local. In order to achieve full transparency, it is necessary to find a

way of integrating remote names into the standard Unix pathname syntax. There are several issues to be considered and not surprisingly the various implementations of transparent distributed Unix systems have adopted different solutions. We will discuss some of the naming possibilities now with particular reference to four such distributed Unix implementations. For a more complete survey and comparison of distributed Unix systems see [Barak86, Brownbridge82,Vandome86,Wupit83].

3.4.1. The Newcastle Connection

The Newcastle Connection (or NC) [Brownbridge82] was designed solely from the naming viewpoint, the chief issue being how remote objects would be named in a distributed Unix system. It also reflects the recursive design philosophy prevalent at Newcastle [Randell83], which led to the formulation of a version of the recursive structuring principle described earlier in section 3.1 for distributed systems:

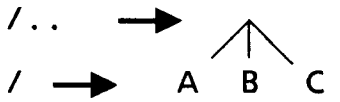
> "A distributed system should be functionally equivalent to the systems of which it is composed."

There is some justification for this approach in the fact that a stand-alone Unix system may be partitioned into subsystems by redefining the root directory as described in section 2.3.2. Such a partitioning is effectively inwards recursion; what the NC provides is outwards recursion. However, the difference is that whereas a closed subsystem is intended to have a closed root, the NC is intended to be used with an open root as we shall see shortly.

The NC ensures that the overall naming graph of the distributed Unix system remains tree-structured by grouping the root directories of individual systems together into what is sometimes referred to as a "super-root" directory. The most logical name for this "super-root" directory is `/..` because it is the parent

directory of all the individual root directories. This is why the NC requires an open root semantics.

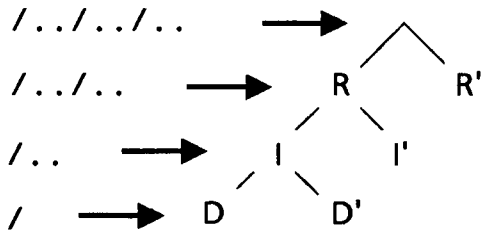
For example, consider three Unix systems named A, B and C, each with their own root directory. The naming graph for the distributed system which results from joining these three systems together with the NC would look like this:



From system A, files on system B can be reached via a pathname that begins /../B. This leads to the root directory for B. Thus the directory /user/robert on B would be named /../B/user/robert from A. Notice that the /.. directory contains named entries for each system's root directory but that local pathnames are not affected by the extra directories because the definition of / on each system remains unchanged.

Although some distributed Unix systems have mistakenly made this "super-root" into something special with its own naming syntax, in keeping with its recursive structuring principle the NC treats /.. as just another directory and the concept is therefore extensible. Instead of having a single directory above /, it might sometimes be more appropriate to add further levels of structure.

For example, an organisation might wish to group its departments D into institutions I and its institutions into regions R. With such a naming graph, a pathname of the form /../../../R/I/D would be required to reach an arbitrary remote system but in a particular case a shorter name might be possible. For example, /../D' would suffice for another department within the same institution and similarly /../../I' would suffice for another institution within the same region. It is always possible to construct the most general



pathname to a given point in the tree from the absolute name of that point relative to the base of the tree and this name may then be reduced to its simplest and most direct form with one of the canonical simplification algorithms discussed in section 2.3.5.

One problem with the / . . naming scheme is that it is necessary to impose a definite order on the naming hierarchy. In our example, departments are grouped within institutions, rather than vice-versa. An alternative structure might sometimes be more appropriate but it is not possible to allow two views of the same naming structure to co-exist simultaneously because Unix does not allow links between directories or across devices. Consequently, the naming tree is fixed and every system shares the same global naming tree. This problem is nothing new. The same ordering conflicts can arise in a hierarchical database and this is one reason for introducing the flat relational model. The Aspect model of naming described in section 2.2.1 allows greater flexibility because it does not impose a pre-ordained order on the naming graph.

Of course, it is not always necessary to extend the hierarchy outwards like this. In some cases the systems being united in the distributed naming tree might not be on an equal footing but instead exist in some kind of client/server relationship. It would then be natural for the server machine to recognise the client as a sub-directory but to the client the server would still be above the root (i.e. somewhere in / . . or higher). The important point is that whenever systems are joined together the directory structure which links them is shared. In effect,

the `/..` directory is replicated across all systems or alternatively exists in its own right on a separate system acting as a name server. However, although the NC expects individual system administrators to set up the extended naming tree, it does not implement replicated directories nor provide any other means of enforcing consistency so the tree structure is not guaranteed.

Directories above root may cause problems for Unix utilities such as *pwd* which assume that the current directory is always below root. This is discussed in more detail in section 3.5.1.

Equating whole systems with directories as the NC does is all very well but suffers from a lack of location transparency. If an object is named `/../A/foo` then it is manifestly located on system `A`. If it were to be moved to system `B`, its name would change to `/../B/foo`. This is not very desirable from an administrative point of view nor terribly friendly to the user since it highlights the distinction between the individual systems that go to make up the distributed system, making it less than fully transparent. Again, a more general linking mechanism could be used to hide the underlying physical structure, making it possible to group objects on semantic or functional grounds rather than purely by location. However, it is possible to manage without this facility simply because the individual systems do indeed remain distinct within the distributed whole and so it is natural that projects which might require such a grouping of objects remain confined to one system. But this is scarcely a justification and indeed rather makes a nonsense of the whole idea of distribution. The problem is really with Unix rather than the idea of transparent distribution. Names and locations should be orthogonal concepts but instead of being separated cleanly they remain entangled because Unix pathnames are overloaded with locational information instead of just being purely structural or organisational.

Another problem with transparency is that a distributed Unix system constructed with the NC does not have a proper concept of a user. Users remain associated with individual systems rather than belonging to the distributed system as a whole and so, at least in this respect, the distributed system is not functionally equivalent to the systems of which it is composed. However, the NC does allow system administrators to retain control of their machines and does not require all the systems to support the same set of users. In fairness to the NC, the concept of a user is problematical for all transparently distributed Unix systems because it is so ill-defined by Unix itself and tied up with the concept of a system directory pointed at by root. Whereas Unix file names are hierarchical and extend nicely to a decentralised distributed environment, user names in Unix are taken from a flat centralised name space without structure. Unix does not provide adequate mechanisms for managing the user space of a large centralised system, let alone a decentralised distributed system. Again the problem has more to do with Unix itself than the idea of transparent distribution. An inadequate centralised mechanism cannot be extended transparently to a decentralised system. This problem is discussed in more detail in section 3.5.3.

3.4.2. NFS

A completely different approach to organising the distributed name space has been adopted by the SUN Network File System (NFS) [Sandberg86]. Disk space is limited on an individual workstation and a lot of space will be wasted in a network of workstations by duplicate copies of system files. Sharing common but perhaps infrequently accessed files (such as on-line manual pages and system source code) will clearly save space and if the access to such remote files across the network can be made comparable to the access time to a local disk, it will be possible to share the entire system file structure and even support diskless clients

with centralised file servers. SUN claim to have achieved this performance goal, and do indeed support diskless clients with NFS over Ethernet.

The NFS distributed name space is based on the concept of a remote mount. The mount system call has been extended so that it may be used to join a name space on one machine to a name space on another, across the network. However, unlike the NC approach, this is inherently asymmetrical. The system which performs the mount will extend its own naming space by gaining access to part of the naming tree of another system but that is all. Although the remote system must consent to being the target of a remote mount by publishing which parts of its naming tree it is prepared to make available and must be prepared to access and modify those parts of its naming tree on behalf of the remote client, once the mount has occurred there is no reciprocal arrangement and no single, global view of the distributed name space. Each system on the network will see a complete tree beneath its root directory `/`, but each system will see a different tree and it will not be possible to access any part of another system's file tree unless it has been published by that system and integrated into the local hierarchy with a remote mount. Even where the individual naming trees overlap, as for example with a shared subtree, at the mount point (i.e. the base of the mounted subtree) `..` will lead back to the system which is enquiring rather than the system to which the shared subtree belongs. In other words, the server which owns the subtree will interpret `..` at the mount point dynamically, with each system that shares the subtree receiving a different interpretation. However, the view of the naming tree from each system will be entirely self-consistent and this means that there will be less problem with Unix utilities such as *pwd* and *find* which depend on particular properties of the Unix naming graph (but see also section 3.5.1).

An NFS server indicates which parts of its naming tree may be remotely mounted by publishing details in a system file called `/etc/exports`. This is not a proper distributed database but merely a local facility which can be used to

control the behaviour of the NFS mount protocol daemon on each machine acting as a server. The server validates each mount request against its publication list and it is possible to restrict the access of a particular subtree to a special group of clients.

Regrettably, the client/server relationship implied by the remote mount concept is only a convention and circularities are still possible if the server machine itself performs a remote mount back to one of its clients. In this respect the NFS is no better than the NC; both systems will behave naturally if they are used as they were intended but without any enforcement of this implicit policy of no cycles, both may easily be abused. Designing a distributed algorithm that enforced a consistent naming tree and prevented the creation of circularities at mount time is probably not worth the effort. Once a distributed system has been set up its naming structure is unlikely to be changed significantly except to add new systems. A reorganisation is likely to be traumatic in any case and this provides additional inertia. A small distributed system on a LAN will have a very simple naming structure in any case so there will not be much to change. A larger system is most sensibly organised as clusters of smaller systems, each locally administered. Circularities are only likely to occur accidentally if complex inter-mounting is allowed between these domains but this can either be forbidden by design or decree. If the domains represent localities, geographically or semantically, the need for such inter-mounting will be less apparent in any case.

3.4.3. RFS

The RFS Remote File System [Rifkin86] developed by AT&T is also based on the concept of remote mount. Like NFS, there is no need for system names or directories corresponding to entire systems to appear in the file hierarchy (although there is nothing to stop a server from allowing its entire file system to be remotely mounted). However, RFS goes further than NFS in achieving

location independence by mounting resources rather than specific subtrees on specific systems. A resource is simply an extra naming level of indirection between what is being published (the RFS term is advertised) and what is being mounted. A system will advertise a portion of its file system as a particular resource and RFS will maintain a distributed database which maps resource names into physical locations. If for some reason a system becomes unavailable, there is no reason why the same resource should not be provided by another system because a potential client is not aware of the actual location of the resource. Unlike NFS, the list of advertised resources really is distributed and if the system acting as name server crashes or becomes unavailable because of a network fault, another system will be configured for this role automatically.

As an organisational aid RFS provides an additional level of structuring on resource names called a domain. A domain is simply a collection of unique resource names (i.e. a context) and it is possible to mount a resource from another domain by using a qualified name of the form `domain.resource`. However, this scheme is not recursively extensible (with domains of domains) although there is no reason in principle why it should not be. It is perhaps unfortunate that the `domain.resource` style of naming understood by the RFS database and remote mount command introduces yet another form of name to Unix but the clean separation between logical resources and physical pathnames is important. Remote mount makes it unnecessary for system names to appear in the naming tree at all.

3.4.4. LOCUS

Whereas the Newcastle Connection joins together whole Unix systems as directories in a single tree and NFS/RFS join Unix systems into a forest of overlapping trees, the LOCUS system [Walker83] adopts a third approach and is perhaps the most transparently distributed of all the systems we have considered.

However, although a LOCUS system is certainly functionally equivalent to the Unix systems of which it is composed, LOCUS is not really recursively structured as we shall see.

The LOCUS distributed naming tree is identical to an ordinary Unix tree. There is only one root directory / for the entire distributed system and therefore all the system directories and files such as `/etc/passwd` only occur once in the naming tree. This is not the performance bottleneck that it might seem because LOCUS supports replicated files and the name of a file is unrelated to the location of its nearest copy. In particular, all the important system files and even the root directory itself are likely to be replicated locally on every machine, with the LOCUS system ensuring that all updates to replicated files are propagated automatically and consistently. This facility alone is an important administrative convenience since otherwise it is all too easy for separate copies of what is supposedly the same file to get out of step.

Although LOCUS maintains the illusion of a single virtual system, it does not follow that one version of each file in the naming tree will suffice, however many times it is replicated. Files containing executable code will only execute on one type of processor, so in a heterogeneous environment it is not possible to maintain a single version of the standard Unix utilities in `/bin` (or indeed, any other program that is to be available on all machines). Instead, one version is required for each processor type but this fact must be concealed to preserve the illusion of a single system with a single `/bin` directory. This is achieved by the use of hidden directories. Each entry in `/bin` is actually a directory containing a version of the program it represents for each possible processor type. Thus, although `/bin/cp` may appear to be a simple program, it actually stands for one a series of programs with names of the form `/bin/cp@/m68`, `/bin/cp@/vax` and so on, where the `@` is an escape mechanism which allows the name to fall through into the hidden directory. When LOCUS resolves a pathname such as `/bin/cp`, it applies a

context-dependent translation to pick an appropriate file in the hidden directory. Part of the state of each process is a list of acceptable hidden directory names, effectively indicating the processor types on which that process is prepared to run.

A second LOCUS facility is made necessary by the Unix convention of storing system-related information in special files with well-known root-relative names. This equates the notion of a system rather too closely with the concept of a root subtree containing certain files. For example, `/tmp` is a directory used by convention to store temporary files and `/etc/utmp` is a file used to record who has logged on to the system and is currently using it. Although it would be possible to have a single version of both these files for the entire LOCUS system, it would not be very efficient or even appropriate to do so. There is no need for `/tmp` to be a public directory and it would be inefficient and unnecessary to go to the trouble of ensuring that every system had a globally consistent view of the contents of `/tmp`. It should be as local as possible rather than shared publicly. For different reasons, it is not desirable to have only one version of `/etc/utmp`. If the file was shared by the entire LOCUS system then it would only be possible to find out about the state of the entire distributed system rather than each local system with a private version of `/etc/utmp`. Consequently, even at the risk of violating transparency, it is desirable to make pathnames such as `/tmp` or `/etc/utmp` special references to a unique version of the file on the local system. LOCUS achieves this by maintaining a special sub-tree of site-specific files for each system in the global naming tree and trapping the conventional Unix names for these objects with a special form of symbolic link. Thus, `/etc/utmp` in the shared root directory is actually a symbolic link to `<LOCAL>/utmp` where the special name `<LOCAL>` is automatically translated into a site-specific pathname in a context-dependent way. (Again, this translation is part of the state of each process, and may be manipulated by some new system calls.) See also section 3.5.4 for an alternative approach.

The hidden directory mechanism and <LOCAL> facility are controlled breaches of transparency, making the individual systems which makes up the distributed LOCUS system temporarily visible for reasons of expediency, efficiency or necessity. However, the illusion of a single system is otherwise remarkably complete. In particular, the sharing of a common /etc/passwd file means that there need be no concept of a user belonging to a particular system because the individual systems have effectively ceased to exist. They have all coalesced to form a single virtual Unix system distributed across the network and a LOCUS user belongs to this system. User ids and group ids remain globally unique under LOCUS. Similarly, process ids are also allocated in such a way that they too remain unique across the distributed system. This makes it possible to send signals to remote processes without ambiguity, preserving the illusion of a single centralised system.

But the almost total transparency of LOCUS has been achieved at a price. Although it is recursively structured, the structure is effectively flat. Because a LOCUS system does not have any sub-systems but is merely transparently equivalent to a single system, the only way in which two LOCUS systems may be joined is by merging them into one system, resolving all the conflicts that will occur in the globally unique id spaces they both assume. It is simply not feasible to carry on joining LOCUS systems together in this way, resolving more and more conflicts and growing system files like /etc/passwd indefinitely. Without introducing sub-structure the sheer size of the resulting system will make it unmanageable but the very transparency of LOCUS prevents such sub-structure from being added for then the LOCUS distributed system will not be identical to its component systems.

Herein lies a paradox. Transparency may be a good thing in a small system, but does it scale for a really large system? To be recursively extensible a system must either have no structure (i.e. be based on flat naming spaces) or else an

infinitely extensible structure (i.e. be constructed from relative rather than absolute pathnames). No compromise is possible because any finite limit on structure will set a barrier to recursive extension.

3.5. Some Impediments to Transparency

Although the extensible hierarchical naming provided by the Unix file system makes it easy to devise a transparent scheme for naming remote files, there are many other subtle details which must be attended to before a genuinely transparent distributed Unix system can be constructed [Marshall86]. Not every aspect of Unix is recursive and indeed some features of the system call semantics depend on flat naming spaces and are therefore extremely awkward to extend to a distributed system without altering the interface and violating transparency [Stroud86]. This difficulty only arises because an existing system, warts and all, is being used as the basis for the transparent distributed system. Such an evolutionary approach protects investment in software and expertise at the cost of requiring backwards compatibility with unfortunate features which were never designed with distribution in mind. A revolutionary design for a distributed system, built from scratch without the need to be compatible with any previous system, should not suffer from this kind of problem in theory. However, there is still a need for the designers of revolutionary systems to be aware of these issues, since otherwise they might fall into the same trap by accident. If a distributed system is not properly recursive then it will be difficult to merge two such systems for precisely the same reason that it is difficult to combine two centralised systems into a distributed system transparently.

With this is mind, and in no particular order, we will consider some of the finer points to be observed in constructing a transparent distributed Unix system and extract some general principles about the construction of a distributed name space in the process.

3.5.1. Naming Graph Semantics

Any alterations to the structure of the Unix naming graph may have subtle and unexpected consequences for programs that assume a certain property. For example, the *pwd* program used to print the root-relative pathname of the current directory assumes that the current directory is always below root, and the *find* program used to exhaustively search a portion of the directory hierarchy assumes that the graph is tree-structured apart from multiple links to the same leaf nodes. Both of these properties can be violated by the mechanism used to construct an otherwise transparent distributed Unix system. However, because these are special cases, it could be argued that such utilities should be altered non-transparently in order to preserve full transparency for the rest of the system, rather than abandoning the concept of transparency altogether because it cannot be made to work all the time.

In general, it is impossible to tell which aspects of the Unix naming graph semantics have been taken for granted in the design of a particular algorithm. Consequently, unless every aspect of these semantics is preserved (i.e. the transparency is complete), something may break. In practice this is not such a problem because Unix provides an environment of cooperating software tools, rather than an uncoordinated bunch of competing utilities. Consequently, deep knowledge about the file system semantics is only concentrated in a very few system utilities (such as *pwd* and *find*), and these can be dealt with on an individual basis.

3.5.1.1. pwd

Any distributed Unix system (such as the Newcastle Connection) which preserves the tree-structure of individual Unix name spaces by making them sub-directories of a directory above their root directory will have to cope with Unix utilities that assume the current directory is below the root directory. In

particular, any utility which attempts to deduce the full pathname of the current context using the *pwd* algorithm must be prepared to cope with the possibility that the current directory is positioned above root or in a subtree which is parallel to the root context in some sort of cousin rather than son relationship.

The correct algorithm for discovering the full pathname to the current context in an arbitrary tree where the root context is not necessarily positioned at the base of the tree is as follows:

(a) Work up from `.` using the normal *pwd* algorithm until you either find `/` or else encounter a directory which is its own parent (i.e. the base of the naming tree).

(b) Assuming that `/` is not encountered en-route, start again from `/` and work out how many `..` steps are required to reach the base of the tree.

(c) Prefix the appropriate number of `/..` stages to the pathname deduced in (a). This is the full pathname to an arbitrary point in an extended naming tree.

This algorithm assumes that the tree has only one directory which is its own parent, i.e. the base of the tree is unique. Consequently, the points discovered at stages (a) and (b) will coincide. However, if this was not the case the tree would not be a tree and so the concept of a unique full pathname would be meaningless anyway.

Notice that this algorithm results in the most general pathname for the current context via the base of the tree rather than the most direct or canonical pathname. In general, although any two points in a tree will always have the base of the tree as a common ancestor, there may be a less remote point in the tree at

which their paths back to the base coincide. Discovering this most recent common ancestor amounts to calculating the canonical pathname between the two points.

Given a mechanism for comparing two pathnames for identity it is possible to obtain the canonical pathname from the absolute pathname by progressively simplifying the more complex name until no further reductions are possible without invalidating or changing the meaning of the name. If the system relies on globally unique identifiers to distinguish objects, and if it is possible to derive the identifier for an object from a pathname which denotes it, then it is possible to streamline the simplification algorithm further. During stage (a) the unique identifier for each ancestor encountered on the route back to the base of the tree is stored so that as each potential common ancestor is visited at stage (b) its unique identifier may be checked against the known ancestors from stage (a).

But even this algorithm may visit more nodes than is strictly necessary because, although ideally there is no need to search up the tree beyond the common ancestor, there is no way of recognising this point in advance. Since the cost of visiting a very remote node such as the base of the tree may be very high, especially if the naming tree covers a large distributed system dispersed across a wide area network, it is better to avoid ever having to calculate the full pathname of the current context from first principles by keeping track of it at all times. If the starting location is known (and ultimately this will be supplied as part of the bootstrapping process) it will be possible to apply an incremental algorithm as the context moves relative to this point and this will be much more practical. The pathname can then be made available via a system call. Modern versions of the Unix shell support *pwd* directly as a built-in command and do indeed keep track of the pathname to the current context at all times. This is possible because the *cd* command to change directories must also be built into the shell (since running *cd* in a child process would have no effect on the parent shell).

Regardless of the algorithm used to calculate the full pathname to the current context, hopefully every program that requires this information will either invoke *pwd* directly or else use a library function (such as *getwd* which is defined in the SVID). Consequently, the introduction of `/..` and other directories above the root should only affect a couple of utilities at most.

3.5.1.2. find

Some Unix systems impose a closed root and equate `/..` with `/` automatically. This makes it impossible to create directories above root. Consequently, remote systems may only be positioned below root in the naming tree. If the distributed name space is symmetrical and allows system `A` to access system `B` and vice-versa then it will be possible to construct a circular pathname from `A` through `B` to `A` again. For example, if by convention all remote systems were found in a `/net` directory then from system `A` the pathname `/net/B/net/A` would denote A's root directory and would be the beginning of an infinite loop. If the naming graph is not tree-structured then any program which attempts to visit all the nodes in a sub-graph systematically, such as *find* or an archiving program like *tar*, will not work correctly because it will be based on a recursive algorithm for traversing a tree rather than an arbitrary graph.

Programs such as *find* need to be able to detect the second time an inode on a given device is visited in order to handle links correctly. This could be achieved by using a bitmap for each device. However, it would be wasteful of memory to keep a bitmap for every device visited and in any case device numbers are only unique within a single system. Without an explicit system identification embedded in the device number there would be a danger of confusion when traversing a name space that spanned several systems because the same device number could occur several times for different devices. It would also be reasonable because of the way mount works for the *find* algorithm to assume that having exhaustively visited

all the files on one device there would be no further references to that device from elsewhere in the naming tree. A given device may only be mounted in one place at a time on a single system but although this is also true for a network of systems it is no longer apparent that this is so because device numbers are no longer unique.

This problem is much harder to solve than the *pwd* problem because the programs involved are fairly complex and there are more of them to deal with. Until recently, Unix provided no primitive function or software tool (apart from *find* which is rather cumbersome to use in practice) for recursively enumerating all the nodes in a naming sub-tree systematically. Any program which needed this functionality was written on an ad-hoc basis. Although the SVID now provides such a function (*ftw*) it is unlikely that old programs will be converted to use it. Even if they were, it would be difficult to modify *ftw* so that it worked correctly in a distributed environment because of the inadequate Unix facilities for identifying files uniquely, a direct consequence of making the flat name space of device and inode numbers visible.

The best solution is probably not to tackle this problem of identification at all but simply to prevent pathnames from passing through more than one remote system. This will prevent loops in the naming graph caused by the connections between systems but will also mean that name resolution is no longer transparent for complex pathnames. We will explore the implications of this further in chapter 5. However, in the meantime, a lingering difficulty is how to deal with pathnames that cannot be resolved such as `/net/B/net/A`. One approach would be to make the point at which the naming graph loops behave like an unreadable directory. (In our example, that would mean that `/net/B/net` was unreadable and consequently `/net/B/net/A` could not be resolved.) This is a transparent solution but it is not a truthful solution and could have paradoxical consequences. It might be less confusing to violate transparency by introducing a new type of file system object to denote a remote system. Of

course, this would require modifications to all the programs which know about the various types of object in the file system (and in particular the *ls* program which gives detailed information about the contents of directories). Unix is not easily extensible in this way. The file system is not a general purpose repository for arbitrary types of object. Instead, it supports a very limited number of primitive objects and knowledge about the semantics of these objects is scattered throughout the system in various utilities rather than concentrated in one place.

The best approach would probably be to adopt the solution used to prevent looping symbolic links and introduce a new error code. This error code would be returned whenever an attempt was made to use a pathname which passed through more than one remote system. Unix has a uniform convention for reporting the failure of a system call and since most programs are not interested in the detailed cause of a problem they would be unaffected. Only the list of error messages printed by the *perror* subroutine would need to be altered (although this would require relinking every program which used *perror* unless dynamic linking was supported).

This is a general solution to the problem. A more specific solution might be appropriate for particular utilities. For example, SUN have added a flag to *find* which restricts its search to local file systems. However, there is no compulsion to use such a flag, and consequently loops must still be dealt with when they occur. Furthermore, this is not a transparent solution since requiring the use of a new flag would break old commands which invoked *find* indirectly. Restricting *find* to local file systems by default is backwards compatible but rather defeats the purpose of a transparent remote file system!

3.5.2. Low-Level Identifiers

We have already alluded to some of the difficulties caused by the fact that Unix makes low-level identifiers visible to programmers. It is possible to map

pathnames into the device and inode number of the object they denote. Programs as familiar as the Unix copy command *cp* use this facility to check whether the source and destination of the copy operation are identical in order to prevent accidentally destroying the contents of a file. Because device numbers are only unique within a single system, it would be possible in a distributed system for a local file and a remote file to share the same identity, even though they were quite distinct. Although in theory it might seem unlikely that two particular files on different machines could share the same inode number given that inodes are effectively allocated independently and at random from a relatively large address space, that in itself would be no reason for not addressing the problem. However, in practice assumptions of independent random allocation are not always valid because it is possible to create disk backups by taking a physical copy of the image on the disk rather than a logical copy of its contents. Files with the same name are guaranteed to have the same inode number on a backup disk created in this way and hence clashes will be inevitable if individual files are copied between a disk and its backup.

It would be easy to solve this problem by adding a system number to the device and inode number already provided as identification but this would not be transparent. Such a system identifier would have to be globally unique in any case to allow further systems to be added to the network without the danger of a clash of identity. A better approach would be to provide an extensible pathname rather than a fixed hierarchy of values as a low-level identifier. However, this would not be transparent either. Without modifying the existing interface, the only viable solution is to encode the identifiers of remote objects so that they can be distinguished from those of local objects. This is only possible if the address space for low-level identifiers is sparsely populated and very few programs (and preferably none) are interested in the exact value of the low-level identifiers they manipulate. Fortunately, Unix appears to have these characteristics. In

particular, most Unix systems only support a handful of devices so very few of the values possible in the device number field of an identifier are used in practice.

Although it would be possible to distinguish remote identifiers from local identifiers by setting an otherwise unused bit in the device number, it would still be possible to confuse two remote identifiers from different systems. The encoding scheme used to distinguish remote identifiers from local identifers must be sufficiently ingenious to allow the remote system to be identified precisely. This will ensure that identifiers are not ambiguous but in fact there is a more subtle reason for this requirement. Some versions of Unix include the *ustat* system call which uses a low-level identifier to obtain statistics about a device. In order to handle remote devices correctly the device number must include a system identifier.

There is simply not enough space in a fixed size identifier to encode the necessary information but it is possible to store an index into an auxiliary table instead. However, this poses various problems in itself which are not actually specific to Unix but must be solved by any non-trivial naming system. An identifier can only be interpreted correctly while the corresponding table entry exists. Without an explicit mechanism for destroying identifiers their lifetime is theoretically infinite. Furthermore, since identifiers are supposed to be absolute (i.e. have the same meaning everywhere), the table must be known throughout the system (or indeed the distributed system) so that identifiers can be interpreted correctly from any context. Finally, it must be impossible to forge identifiers and hence compromise the integrity of the system.

Although these problems might be soluble if the low-level identifiers provided by Unix were large enough to include time-stamps or could be encrypted (and whether this was so would depend on the size of the system in any case so that such an approach would not scale well), in practice, limitations on the size of

identifier available make it impossible to provide full transparency. Nor is it really necessary to do so. Although in theory identifiers may be manipulated by programs in arbitrary ways, in practice they will only be used in one or two standard ways and providing these work as expected the system will be transparent to all intents and purposes. For example, it would be easy to implement a scheme which gave identifiers transient non-unique values which were only valid within the context of the process which generated them and this would work perfectly well in practice even though it was not truly transparent. Given knowledge of the mechanism used to implement this pseudo-transparency it would obviously be possible to write a pathological program that violated the transparency but this would not be a reasonable thing to do nor could such a program be created by accident. Some variant of this approach to solving the low-level identifier problem has been adopted by all the distributed Unix systems which have tackled this issue [Marshall86, Rifkin86].

3.5.3. Ownership and Authorisation

The Unix permission system is based on the use of numeric values called user ids. (There is also a system of group ids but that does not concern us here.) Both files and processes are owned by a particular user id. However, for human convenience and in the interests of usability Unix also provides a mapping between user names and user ids which is understood by all the appropriate utilities. Consequently, it is always possible to work in terms of user names (such as `robert`) rather than user ids (such as `42`) except when using system calls directly from within programs because the Unix kernel itself does not understand user names. This was perhaps an unfortunate design choice but it was made long ago and no alternative approach has been proposed. The Unix facilities are not really adequate for a large centralised system and because they do not scale well within a single system they do not work well between systems either.

The correspondence between user ids and user names is recorded in a file called `/etc/passwd`. This file is not shared between systems so a particular mapping between user name and user id is only valid on a particular system (or more precisely, is only valid whilst the root directory is positioned at a particular point in the naming tree). The user name `robert` may denote a completely different person on another system or may correspond to a different user id. If distribution is occurring at the operating system level rather than the application level, it will be numeric user ids rather than textual user names that get passed between machines. However, a user id on one machine may denote a completely different person on another and it is not reasonable to require a common `/etc/passwd` file or a unique identifier for every user of the distributed system (although NFS and LOCUS impose just this requirement). Instead, all user id values must be intercepted as they are passed between systems and translated accordingly. Ideally, this mapping should occur in both directions to ensure both that local users have appropriate permissions on remote systems and also that ownership of remote objects is reported in terms of local users. However, the mapping will not necessarily be one-to-one; indeed, if a whole class of users are only allowed guest status on a remote machine it will be many-to-one. Furthermore, if a remote user is not allowed to use the local machine there will be no suitable inverse mapping at all so a special user id must be provided to denote remote objects which do not belong to anyone on the local system.

If Unix used pathnames instead of numeric values to represent user ids it would be easier to represent such remote values providing system names were visible in the naming tree explicitly. For example, `/../A/robert` could denote user `robert` on system `A` and similarly `/../B/robert` could denote `robert` on system `B`. However, if systems `A` and `B` were under the same management, so that user `robert` on each referred to the same individual, it would be more appropriate to use an unqualified `robert` within this context. From outside the

domain of A and B a fully qualified pathname would still be required. With such a scheme, the natural hierarchy for user names might not follow the system hierarchy exactly although it would probably coincide with some grouping of systems at a higher level. It would be necessary to recognise when names passed out of their defining domain and qualify them accordingly or alternatively to use absolute pathnames for users which were valid from everywhere in the system. Again, this problem is not unique to Unix and is usually solved by using fully qualified absolute names which are guaranteed to be unique and have the same meaning everywhere. However, the whole concept of an absolute name is alien to the idea of recursively joining systems together because it imposes universal constraints on the choice of names rather than purely local constraints.

3.5.4. Remote Execution

Providing transparent access to remote files is one thing but being able to run programs transparently on any processor is quite another. At the file system level of an operating system there is no concept of executing a program. The file system is simply responsible for reading the contents of files into memory and does not need to know whether such a request comes from a program which wants to read some data or the operating system which wants to execute a program. If distribution occurs at this level then there can be no concept of remote execution. An operating system built on top of a transparently remote file system will execute all programs locally, paging them across the network with the assistance of the remote file system as necessary. For this reason, NFS and RFS, which are both transparent remote filing systems, provide remote paging rather than remote execution (although SUN have recently added a non-transparent remote execution facility to NFS called REX).

On the other hand, if distribution occurs at the operating system level then the concept of executing a file must be distributed in the same way that the concept of

reading a file is distributed. However, because an operating system not designed with distribution in mind has no concept of other systems, transparency dictates that the choice of execution site must be made automatically. Apart from the local system, in the absence of an automatic load balancing facility, the only sensible choice is the system where the program resides to avoid incurring the expense of copying its object code across the network. The NC has implemented true remote execution in this way. However, LOCUS has gone furthest of all towards being a transparent distributed system in the strict sense discussed in section 1.1 by providing a "change working processor" command to control where programs are executed. (Of course, such a facility is non-transparent being an addition to the functionality provided by the original system. Automatic load-balancing would provide full transparency.)

Remote execution poses some interesting problems for the name resolution algorithms. As discussed in section 2.3.2, Unix has confused the concept of root as a naming context with the idea of a system and many programs use root-relative names to access system information such as the mapping between user ids and user names in `/etc/passwd`. Executing a program does not change the meaning of root and yet certain system programs which are supposed to report information about the system they run on will only work correctly if root is moved to that system. This would break the interpretation of other root-relative pathnames. Unix should provide a special naming context for local files which works regardless, irrespective of where programs are executed from (in effect, this would be a closure), or else move the information which is presently stored in system files below the kernel boundary so that root becomes a pure naming context as it should be. In the absence of such a facility, the NC provides a special form of the *exec* system call called *excr* which moves root to the remote execution site and this covers most of these special cases. Because the interpretation of low-level identifiers such as user ids and process ids is also tied to the location of root,

moving root in this way also is a useful way of reaching remote objects which cannot be named with pathnames.

3.5.5. Summary

To summarise, building a transparently distributed operating system involves solving various subtle problems mainly concerned with notions of identity. If a naming mechanism does not scale within a system it will be difficult to extend it between systems. Consequently, a pragmatic rather than a fanatical attitude to transparency is required in order to build realistic distributed systems out of existing systems. Furthermore, many of these problems remain non-trivial, even when designing a distributed system from scratch.

Although some of the difficulties discussed in this section are caused by weaknesses in the Unix system call interface, where a lack of recursive generality makes transparency difficult to achieve in practice, many of the issues raised would apply to any distributed naming system. If names are not globally unique but rather are relative to some context and if it is possible to pass names around between contexts then they must be transformed en-route so that they still denote the same object. Similarly, if names have a transient significance or are only valid within an implicit context because they rely on hidden state information then they must not be used outside their defining context or after they have expired.

3.6. Conclusions

In this chapter, we have discussed the problems of joining name spaces together, both within a single system and between systems to construct a transparent distributed system. Because joining name spaces together is an inherently recursive process it is not surprising to find that the same problems must be tackled irrespective of the level at which systems are joined. It is

therefore natural to base the design of a transparent distributed Unix system on an extension of the local mechanism for joining name spaces together, namely the idea of mount. However, there are other ways of combining the name spaces of individual Unix systems to form a transparent distributed system. The main problem is whether to preserve the notion of system. The Newcastle Connection maps systems onto directories and provides a single tree structure for the entire distributed system. NFS and RFS use the concept of a remote mount to share portions of the naming tree between systems but make no attempt to present a consistent global picture. LOCUS hides the distribution and the individual systems entirely by maintaining the illusion of a single Unix system. However, this avoids the problem because there is no way of joining together two LOCUS systems short of merging them entirely. Making the notion of a system explicit in the naming hierarchy is a violation of transparency because Unix has no support for such a concept but without such a notion it is impossible to build distributed systems recursively. The real problem is that the Unix concept of a system is not recursive so that there is no mechanism for introducing sub-structure into a large centralised system which can be generalised to a distributed system.

Chapter 4
Distributed Systems and Global Identifiers

In chapter 3 we considered evolutionary approaches to building distributed systems by joining existing systems together. This involved exploring ways in which the naming mechanisms of centralised systems, designed without distribution in mind, could be extended to cope with remote objects. Structuring mechanisms which simplify the administration of a large centralised system may be generalised to a distributed system quite easily but in general a naming system designed for a centralised environment is not adequate for a distributed environment.

In this chapter we will consider some distributed naming systems which have been designed from scratch without needing to be compatible with the naming mechanisms of a centralised system. In particular, we will explore mechanisms for joining name spaces together and resolving naming conflicts. Most of these mechanisms rely on the concept of a universally unique identifier and we will examine whether such universal uniqueness can be achieved in practice.

4.1. Global Naming and Name Resolution

Given the name of some remote resource in a distributed system, there are two stages involved in making use of that resource. Firstly, the name must be translated into the location of the resource and secondly, messages must be sent to this location in order to perform operations on the object. The second of these stages is well understood. Typically, locations are identified by unique network addresses and routing algorithms make it possible to send a message across the network to reach any particular location [Shoch78]. However, the first problem, naming and locating objects, involves the design of a distributed naming service

and some sort of naming scheme, and this is what we will concentrate on in this chapter.

It is worth observing that the action of resolving a name can be captured within this model as an operation performed on a name server resource. This poses the bootstrapping problem of locating the name server itself. As discussed in section 2.1.2, bootstrapping must be solved outside the naming system. For example, the name server may reside at a fixed well-known address or may simply respond to a broadcast request issued as part of the initialisation sequence whenever a new system is installed as part of the distributed system.

Of course, it would be possible to design a system that simply referred to objects directly by their location (or address) instead of by a more abstract name. However, such a system would be unfriendly to use and awkward to reconfigure because it would be impossible to reassign objects to new locations without changing their name. Names provide a useful level of indirection which distances an application from the objects it manipulates.

In real life, human beings have various ways of resolving ambiguous names. Although some attempts have been made to model the human naming process in computer systems [Sollins85], it is simpler to assume that all objects can be named unambiguously. An easy way to ensure this is to give every object a unique name, thereby guaranteeing that there can be no ambiguity since no two objects can have the same name. This is known as **absolute naming**. Alternatively, with **relative naming** the name of an object depends on the system which accesses it so that systems are in effect naming contexts. Within the distributed system as a whole, objects can have more than one name and two distinct objects can share the same name. To resolve this ambiguity, relative names must be qualified with the name of the system from which they are valid if they are to be used from outside that system.

Relative naming encourages a decentralised approach to storing name bindings. Only a limited set of names are valid from a particular system and consequently there is no need to store the entire name database at a centralised location. However, unlike absolute names which have the same meaning everywhere (i.e. denote the same object), it is not possible to pass a relative name from one system to another because it may no longer be valid or may even refer to another object. A compromise which imposes more structure on an absolute name space is the use of hierarchical names.

For example, the Xerox Grapevine mail system [Birrell82] recognises names of the form `name.registry` whereas the more recent Xerox Clearinghouse name server [Oppen81] (developed as a result of experience with Grapevine [Schroeder84]) recognises longer names of the form `name@domain@organisation`. In both cases, the hierarchy is of fixed depth and not extensible like Unix pathnames.

The Xerox name servers are intended to store high level names for objects such as people, services and machines. It is therefore reasonable to assume that new names will be created relatively infrequently so that uniqueness within a certain level of the hierarchy can be ensured by coordinating all name allocation through a central administrator. Consequently, this sort of name service is not suitable for implementing a file system where names are chosen privately by individuals, and the use of directories supports an arbitrary number of levels in the naming hierarchy. For this reason, although the Xerox distributed systems are constructed around a centralised name service, paradoxically they do not support a uniform naming convention for every object they contain. Instead, they use a uniform scheme for naming and locating services but thereafter each service is responsible for managing its own name space.

Although some attempts have been made to design a Universal Directory Service [Lantz85] that provides a uniform naming mechanism for all the objects in a distributed system, it is perhaps more realistic to recognise that different naming domains have different dynamic characteristics (choice of names, frequency of update, etc.). Since these characteristics will lead to different design choices and trade-offs in an optimal implementation, it is sometimes better to implement specialised name services for each application if only to simplify administration and improve efficiency.

4.2. Allocating Unique Identifiers

A name server maps possibly ambiguous names into unique identifiers used by the distributed system to identify and locate objects. Consequently, an application such as a distributed file system requires a mechanism for allocating unique identifiers to file objects. Ideally, the allocation algorithm will be decentralised to make the most of the distributed environment. Individual systems should be able to allocate globally unique identifiers independently without fear of conflict.

One approach is to use structured identifiers containing the address of the server where the object resides and an identifier for the object which is only valid at that server but is otherwise guaranteed to be unique within this limited scope [Watson81]. This is effectively a relative address. It is very easy to locate objects from their identifiers but then the identifier is not location independent. If an object moves to a different server then its identity will change. This dependence on the physical address of the server can be alleviated somewhat if the network supports the use of logical addresses, in effect names at a different level of the system. Alternatively, a multicast address could be used to identify a group of servers, one of which would respond to requests for a particular object. This approach to naming has been taken by the Stanford V kernel [Cheriton84b].

Even if objects are allowed to move around between systems, unique identifiers can still be based on server addresses without necessarily tying the object to a particular location. For example, in a homogeneous network based on Ethernet [DEC81] each machine is assumed to have a unique 48-bit address [Dalal81]. Machines on an Ethernet can independently allocate identifiers which are guaranteed to be globally unique by making this address part of the identifier for all the objects which they create. Two identifiers generated by the same machine are guaranteed to be unique by basing the rest of the identifier on a strictly increasing logical clock. This need not be synchronised with the clocks on other machines but must never supply the same value twice, even in the face of machine crashes.

By making the machine which creates an object responsible for giving it an identifier, this scheme ensures that identifiers are allocated uniquely. However, if objects may subsequently migrate to other machines, their identifier will only reflect their creation site and not their present location. Consequently, there will still be the problem of locating the object and, because it is possible for a distributed system by its very nature to be in an inconsistent state, it is always possible that the result of resolving a name will be incorrect. Applications must be designed accordingly and, in the interests of efficiency, distributed systems often use hints and other caching techniques to speed up the resolution process at the risk of occasionally getting the wrong result [Terry85].

Many of the issues involved in choosing the exact form of a unique identifier (or UID) and using it to locate objects in a distributed system are discussed in a paper about the design of this aspect of the Apollo Aegis distributed system [Leach82]. The Apollo hardware provides a 20-bit unique node identifier which is concatenated with a 36-bit clock value to form the basis of a 64-bit UID used to identify all the objects in the system. Rather than providing a centralised name server to map UIDs into locations, objects are located by a series of heuristics

augmented by the use of a hint manager. In this respect, the Apollo system is not as sophisticated as the Xerox designs (which are always able to locate an object given its name) but on the other hand there is no natural partitioning of the name space to simplify the design of the database. Although UIDs must ultimately be translated into location-specific structured names, the designers of Aegis felt that this binding should be delayed as long as possible so that the unbound UIDs could be used uniformly throughout the system (except at the lowest levels). Absolute location-independent identifiers have the advantage that they can be passed freely from process to process across machine boundaries so that when an object migrates, there is no need to locate and update all the references to it from elsewhere in the system. This simplifies the problem of unmounting portions of the object space stored on physical volumes and moving them between machines.

4.3. Combining Name Spaces

Given a mechanism for naming and locating objects unambiguously, it is possible to construct a self-contained distributed name space. However, what happens when it is necessary to combine two such name spaces into one, so that the objects in each of the constituent distributed systems are equally accessible in the composite system? Any assumptions about uniqueness used to justify the construction of names or identifiers may no longer be valid in the composite system, and the resulting ambiguities must be resolved. We will now explore some of the problems which arise and the solutions which have been proposed.

4.3.1. Adding an Extra Level of Hierarchy

With a hierarchical naming scheme, an obvious approach to combining two systems is to add an extra level to the hierarchy containing two domains, one for each system. Since names need only be unique within a domain, any clashes between the two names spaces will not cause problems providing names are not used outside the domain for which they are defined. This approach was taken by

the telephone system when first area codes and then country codes were introduced. Extra prefixes were added to the standard form of a telephone number to give every telephone in the world a unique name under an absolute naming scheme.

For example, the absolute telephone number of the Computing Laboratory at Newcastle University is +44 91 232 9233 where 44 is the country code for the UK and 91 is the area code for Tyneside. This number may be used from anywhere in the world that supports international dialling with the understanding that the + prefix on the 44 be replaced by whatever the local convention is for reaching the base of the telephone naming tree, that is, the context for resolving internationally agreed country codes. This prefix (equivalent to `/..` in Unix naming terms) varies between countries. In France it is 19 and in the UK it is 010. In this sense, international telephone numbers are hierarchical names defined absolutely except at the outermost level which has a relative name (although the + is arguably an access code for the international network which is not part of the name).

Of course, within the UK there is no need to use the absolute name for a UK telephone number (the Computing Laboratory becomes 0 91 232 9233 where 0 is another nationally defined prefix for getting to the national level of the naming tree, this time equivalent to `/` in Unix naming terms) and within the area to which the telephone number refers there is no need to even use the area code (from within Tyneside, the Computing Laboratory may be dialled as simply 232 9233 without any prefix). This scheme is possible because at each level of the hierarchy names (or in this case numbers) are centrally controlled and guaranteed to be unique. There are not two countries with the same code 44 nor two areas within the UK with the same area code 91. (There may be several

countries which each recognise an area code of 91 but this is not ambiguous because an extra level of hierarchy has been added to resolve the conflict.)

4.3.2. Heterogeneity

The standardisation of international dialling prefixes for telephone numbers has only affected naming at the outermost level. Each national telephone agency is still responsible for defining the form of its own national telephone numbers. These may include a regional code as well as an area code or may be completely flat. The important point is that there need only be agreement on the form of names at the level in the system at which the name spaces are joined.

Unfortunately, it is not so easy to join heterogeneous name spaces together in general. Telephone numbers are taken from the limited alphabet of ten numeric symbols. Consequently, it is possible to use alien telephone numbers from within another telephone number space. They will be transmitted across the local number space untouched and only interpreted in the alien number space to which they refer. However, arbitrary semantic checks (such as the length of the number dialled) or the presence of extra symbols (such as # and *) will cause problems and this sort of difficulty is much more likely to arise when the names come from a richer name space.

For example, heterogeneous file systems may not support the same alphabet for generating filenames, may use a different character for separating the components of a name or may simply impose different restrictions on the length and form of a name or one of its components. VMS file names are much more restrictive than Unix pathnames and this makes it difficult to combine a Unix system with a VMS system transparently. From Unix, VMS names must appear to be Unix names and vice-versa. This requires some form of mapping to be defined at the boundary between the systems. (It is sometimes even necessary to map file names in a distributed Unix system because of differences in directory

representation and hence the maximum length of a filename [Fraser-Campbell86, Weinberger86].)

A similar problem arises when trying to combine two different network architectures (such as SNA and OSI) at a gateway [Williamson87]. There must either be a way of representing OSI names as SNA names or else the gateway must maintain a mapping between the two name spaces, and intercept all attempts to pass a name from one domain to the other.

In general, full transparency is probably unattainable between truly heterogeneous systems. Joining systems together involves seeking a common abstraction which can be transparently extended by enlarging the name space. If the systems are really different this may be impossible. Even if it is possible to achieve some measure of transparency it may only work in one direction if one system offers a superset of the other's functionality because it will be impossible to emulate the more general system on the more limited system.

4.3.3. Dealing with Old Names

Even ignoring the particular problems caused by heterogeneity, it is not always possible to join two homogeneous systems together by adding an extra level of hierarchy. Quite apart from the need to alter all the software to handle names with extra structure (especially if the system only recognised a fixed depth hierarchy originally), there is also the problem of old names embedded in arbitrary programs and files throughout the system. It is usually quite impossible to locate all these names and resolve potential ambiguities by translating them into the new form.

One way of dealing with old names is to treat them like unqualified telephone numbers found jotted down on bits of paper and assume that the names refer to the domain in which they are found. This is only workable if both the original

system and the merged system use a fixed depth name hierarchy and require all names to be absolute and fully qualified (i.e. names cannot be abbreviated by assuming a default context). If a tree of nested systems is built in accordance with these requirements, it will always be possible to identify which level in the tree a given absolute name belongs to by simply counting up the levels from the leaves of the branch in which the absolute name is found. Even so, there is always a danger that because all names are absolute, applications software will not unreasonably assume they have the same meaning everywhere and will quite inadvertently pass an old absolute name from one branch of the tree to another by ad-hoc means, without converting it to its correct, fully qualified absolute form.

Transparency hides the boundaries between name spaces but there must be a mechanism to intercept and convert unqualified names at the boundary before they can escape into a context in which they are ambiguous. Unless each system only manipulates names through an abstract interface which can easily be intercepted at system boundaries, this problem is very difficult to solve. The Flex system described in section 2.2.2 uses capabilities rather than character strings for names and the underlying capability machine on which it is implemented provides just such an abstract interface. Although Flex is difficult to implement efficiently on conventional hardware, this capability machine solves many naming problems. For example, Flex supports remote capabilities and this makes it possible to pass names between systems without ambiguity because they can be automatically transformed en-route to point back to the original system.

Without capabilities or a similar mechanism, old names are difficult to deal with because combining name spaces makes absolute names into relative names. Lampson has proposed a mechanism which prevents this by ensuring that absolute names from old name spaces remain absolute in the new name space [Lampson86].

Lampson's solution relies on an external mechanism for identifying distributed name spaces uniquely. Absolute names are qualified with the unique identifier for the base of the naming tree to which they belong. When naming trees are combined, the unique identifier for the base of each old tree is recorded together with its absolute name in the new tree. In this way, each naming tree contains a historical record of every naming tree that ever had a separate existence but now forms part of this tree. Although old absolute names might still exist for such trees, it is possible to intercept all such old names as they are used and translate them dynamically into new names.

This is best illustrated by an example from Lampson's paper. Suppose both DEC and IBM have adopted the Lampson naming scheme and DEC names begin with `#333/DEC/` whereas IBM names begin `#666/IBM/`. If the DEC and IBM name spaces are combined into an ANSI name space which begins `#999/ANSI/` then the new tree must record the fact that `#333` is now known as `#999/ANSI/DEC` and `#666` is now `#999/ANSI/IBM`. When an attempt is made to resolve an old name such as `#333/DEC/SRC/Lampson`, the mismatch between the old UID `#333` for the base of tree and the new UID `#999` will be detected and the name will be translated into `#999/ANSI/DEC/SRC/Lampson` before it is resolved.

Recording historical information about name spaces is only feasible if name spaces are combined infrequently so that the amount of historical information that must be stored about old names remains manageable. This may be a reasonable assumption but Lampson's scheme also requires universally unique identifiers, unique not just within a single system but between every distributed system that might ever be constructed. This is much less realistic as we shall see in section 4.6.

4.3.4. Merging Name Spaces

Without the possibility of extending the hierarchy by an extra level, name spaces must be merged at the outermost level. Hopefully, this level will be sparsely populated with names so that clashes can be avoided. This is particularly likely if the names are taken from a rich alphabet.

For example, the Xerox name services support proper names rather than numbers and with forethought and a centralised agency to control the names of organisations for Clearinghouse and registries for Grapevine there should be few problems. IBM are unlikely to choose an organisation name of Xerox (assuming they were to adopt Clearinghouse as their naming standard) although a city name such as Newcastle is certainly ambiguous without further qualification (there are at least two places called Newcastle in the UK alone).

On the other hand, with a less rich alphabet (such as the numerical area codes for telephone numbers) clashes are more likely. It may be possible to add a few new area codes to the American phone system as America acquires new states, but if it were to acquire a whole continent with its own telephone system complete with a large set of area codes, clashes would be inevitable and a new level of hierarchy would have to be introduced.

A flat space of identifiers, perhaps based on unique network addresses, will only extend if the uniqueness criterion continues to hold. This might be reasonable when combining identifier spaces based on 48-bit Ethernet addresses, but nobody would expect 8-bit Cambridge Ring addresses to remain unique. On the other hand, it is not feasible to construct an internet from more than 256 Cambridge Ring stations unless something is done to extend the address space, perhaps by adding a network number to the station number. Any network which supports absolute addresses must identify each system uniquely since otherwise it will be possible to deliver messages for the same address to more than one

location. Consequently, such an internet address may be used as the basis of a unique identifier, with the advantage that it is more abstract and hardware independent than say a 48-bit Ethernet address, making it possible to combine heterogeneous network hardware into a single logical network and build a consistent distributed name space based on unique identifiers.

For example, the Arpanet supports 32-bit internet addresses and these are used as the underlying unique identifier in a hierarchical naming system based on domains [Mockapetris83]. Unfortunately, a scheme based on logical identifiers is not infallible. There is nothing to stop an individual site from setting up a network unilaterally which conforms to the ARPA model but uses its own block of addresses which may well be in use elsewhere on the ARPANET.

This is precisely what has happened at Newcastle where the supposedly unique identifier 42, already offically allocated to another site, has been unofficially adopted for use internally. So long as the Newcastle network remains distinct from the official ARPANET this will cause no problems because addresses generated at Newcastle which are based on the identifier 42 will be used entirely locally within Newcastle and not propagated to the outside world. However, if the Newcastle network were to be joined to a larger internet then these private addresses would become public and their value would matter. Newcastle would have to choose (or rather be given) a portion of the address space that was unique in this wider context and the logical internet address of all the machines at Newcastle would have to be changed. Fortunately, this would not be too traumatic because no distributed systems have been constructed at Newcastle using unique identifiers based on these logical addresses. However, this would not be true in general. It should be possible to combine systems built out of unique identifiers without having to change the addresses on which the unique identifiers are based and this can only be achieved by allocating blocks of

addresses that are guaranteed not to clash in advance, through a centralised agency.

4.3.5. Random Identifiers

There is one interesting technique for generating unique identifiers which neatly sidesteps the problems of joining name spaces together. If it is possible to generate identifiers randomly so that the probability of two systems independently generating the same identifier is negligible then systems may be joined together without worrying about name clashes.

For example, the Amoeba system [Mullender85] uses this technique to generate interprocess communication port identifiers and protects resources by relying on the fact that it is impossible to guess a valid port number in a reasonable time. Mullender claims that for a large network with 2000 processes, each with an average of 5 ports, a random 48-bit port identifier could be broken by brute force in 2.8×10^{10} tries on average. At a rate of 50 tries per second, it would take almost 18 years of continual trying to find just one port. The size of the random identifier could be adjusted to suit the number of objects in the system and their expected lifetime, giving a probabilistic guarantee that identifiers would not clash.

Using this approach, several Amoeba systems have successfully been joined together over X.25 [Renesse86] without needing to worry about whether identifiers clash. The only problem has been locating an object given its random identifier. Because it is not feasible to use a broadcast algorithm over a WAN the Amoeba solution is non-transparent and involves explicitly publishing identifiers in remote domains.

Random identifiers are the most pragmatic solution to the unique identifier problem. They do not require a centralised agency to allocate numbers nor do they

need agreement over the particular random number generation technique used (since one random number should be very much like another). However, the random numbers really must be random or else the assumption which guarantees the correctness of the distributed system will break down. The guarantee is only probabilistic in any case but the probability that it is violated can be made arbitrarily small providing an upper bound can be put on the number of objects that will ever be in the system and their lifetimes. For a system which grew indefinitely with objects persisting for ever this approach might not be feasible but the magnitude of such a system would be well outside the capacity of current (or even forseeable) technology.

4.4. Reorganising Name Spaces

As well as being combined to form larger name spaces, hierarchical name spaces may also be reorganised by detaching particular subtrees and moving them to another portion of the tree. This poses the same problem of dealing with old names which refer to a part of the tree which has moved. The standard solution is to use indirection in the form of an alias or symbolic link. A pointer to the new location of the subtree is left behind at its old location (similar to the pre-recorded forwarding messages provided automatically by telephone companies when telephone numbers are changed - "the number you have reached no longer exists - please redial as ...") and the name resolution process automatically and invisibly indirects to the new location. To take another example from Lampson's paper [Lampson86], if DEC were to buy IBM and move the entire IBM name space so that it became a subtree of the DEC name space, then a symbolic link pointing to `#999/ANSI/DEC/IBM` would be left behind at `#999/ANSI/IBM`, so that pathnames of the form `#999/ANSI/IBM/...` were automatically translated to `#999/ANSI/DEC/IBM/...` (with old names of the form `#666/IBM/...` going through two stages of rearrangement but still working).

This approach works very well but means that a given object may now have two or more names, only one of which is a true direct absolute name, since the others will pass through symbolic links. This is a considerable weakening of the tree structure of a hierarchical name space and means that although objects still have a canonical pathname (their unique direct absolute name), it is no longer possible to simplify arbitrary pathnames into canonical form without knowing about all the symbolic links in the system. This knowledge is distributed at the nodes where the symbolic links reside and these must be visited as part of the name resolution process to redirect the name back to its true path. Without caching this can be expensive; the advantage of the canonical name algorithm was that it relied on a global property of the naming graph and could therefore be applied statically without arbitrary knowledge about remote names. However, caching the name and value of symbolic links at the base of the tree should at least make it possible to resolve such pathnames directly without being led down false branches in the naming tree only to be redirected elsewhere.

4.5. The Power of Indirection

The use of indirect naming objects which are intercepted and interpreted by the naming system automatically is a useful way of extending a name space. Effectively, this is how distributed Unix systems such as the Newcastle Connection and NFS are constructed and how the Unix mount mechanism works. In a distributed system, this scheme can be used to leave forwarding addresses for objects which have migrated elsewhere (as in the Emerald system [Black86]) and may also be used to extend a centralised system to a distributed environment. For example, some work has been done using this technique to extend the Smalltalk object manager to handle remote objects [Decouchant86]. Similarly, the Flex architecture has been extended to include remote capabilities which are accessed indirectly via local procedure objects that masquerade as a local copy of the remote object [Foster86]. Message passing kernels such as Accent [Rashid81] or

Chorus [Guillemont82] send messages between machines using an indirect network transfer agent and the ANSA project is also exploring the use of indirection for this and several other aspects of communication [Herbert87]. There are even parallels with virtual memory systems. The LOOM virtual memory system for Smalltalk [Kaehler86] uses stub objects which are automatically paged in from the disk transparently by the object manager as required. In effect, these are indirect objects that cause a page fault rather than a network transfer. Indirection is a very powerful mechanism for extending the semantics of a system without altering its functionality. Whoever said "any problem in computer science can be solved by adding enough levels of indirection" was probably correct!

4.6. Are Globally Unique Identifiers Realistic?

From the work discussed in this chapter it would appear that if it were possible to construct name spaces using globally unique identifiers that really were globally unique across all time and space then such name spaces could be joined together freely without their internal identifiers clashing. This of course assumes a certain degree of homogeneity, namely that all interested parties would agree to a common naming scheme and allow a centralised authority to control at least the top level of their name space. Even in the atmosphere of good will fostered by international standardisation efforts such as OSI, this degree of cooperation would be unprecedented. Indeed, the concept of a unique identifier or well-known address is alien to the OSI model which prefers the use of locally defined service access points, as witnessed by problems with the Ethernet standard and the demise of a type field controlled by Xerox in favour of a length field and a non-unique link service access point value.

In fact, it is likely that there will always be a need for gateways between heterogeneous name spaces which will have to map from one form of name to

another and intercept directory service requests although it will be difficult to prevent names from being passed from one naming space to another by other means. Consequently, there will always be the problem of pathnames that pass through several naming spaces or domains recursively, although since the number of rival naming schemes will hopefully be relatively small, it will be possible to use structured names of the form `name.domain` internally.

Ignoring the political problems of securing international agreement, is it otherwise possible to allocate globally unique identifiers to objects and hence construct vast distributed systems that can span the world with a single naming graph? Perhaps in theory, but in practice, people make mistakes, Murphy's Law will intervene and something will go wrong.

In theory, there is no problem with allocating globally unique identifiers. Nobody has yet managed to create a computer system containing an infinite number of objects and any finite collection of objects can be mapped one-to-one onto a subset of the integers. Each object may be uniquely named by its image under this mapping. Estimates vary, but if for the sake of argument the observable Universe contains 10^{72} (or 2^{120}) particles, a 120 bit unique identifier should be more than enough for most computer systems (although it might be somewhat unwieldy and space inefficient to use such an identifier exclusively).

The problem of course is counting (or rather naming) each object. The fact that such a mapping exists in theory does not mean that it is known in practice. There are many such mappings and the problem is agreeing on a particular one, so that two sets of objects may be merged without any name clashes. Knowing that an object has a unique name does not help to discover what that name is.

In practice, unique identifiers are usually generated from a machine address (to guarantee uniqueness between machines) and a timestamp (to guarantee uniqueness within a particular machine). Timestamps must be strictly increasing

and this may require special hardware and software to prevent human error and avoid the problems caused by machine crashes. The granularity of the time stamp must tread the delicate tightrope between being too short (so that all possible timestamp values are used up too soon), and too long (so that UIDs cannot be generated quickly enough).

Unique machine addresses are usually based on network hardware. For example, Ethernet addresses are supposed to be unique 48-bit quantities and if it were possible to join together all the machines in the world onto a single Ethernet, no two machines would have the same address. Or at least that is the theory. In practice, uniqueness is ensured by allocating 24-bit blocks of the Ethernet address space to individual computer manufacturers on application to a centralised authority. The manufacturers are then responsible for ensuring that they do not use the same address twice. However, it is possible for something to go wrong in the manufacturing process and at least one manufacturer (who shall remain nameless) is known to have allocated the same address twice by accident. Quite apart from this, most Ethernet hardware allows the network address to be altered by software which makes it impossible to guarantee uniqueness and allows a malicious node to impersonate another.

Relying on an Ethernet address (or any other kind of hardware address) to ensure uniqueness is only possible in a homogeneous network. In practice, this is not realistic, except perhaps for a proprietary system based on proprietary hardware. When an internet is constructed from a mixture of different networks, each with their own addressing convention, a logical internet address must be used to distinguish systems and this can be used as the basis of a unique identifier. However, moving away from physical hardware addresses increases the scope for human intervention and hence error in the allocation of values. For example, the 32-bit internet addresses used by the ARPANET are associated with network hosts simply by an entry in an editable file under Berkeley Unix. It is

much easier to change a logical address than a physical address because of the extra levels of indirection between the abstract network protocol and the physical communications medium. Although this makes it much easier to reconfigure and merge networks, it causes several problems for distributed systems built over such networks if they use such logical addresses as the basis of their unique identifiers. All the identifiers in the system must be tracked down and modified whenever changes are made to the logical addresses of hosts on the network. This is simply not practical for a large distributed system but unfortunately a unique logical address may be the only thing that distinguishes network hosts in a large network.

Another approach to ensuring uniqueness is to use a random number as part of the identifier (as discussed in section 4.3.5). If such values are genuinely random then the probability of two systems inadvertently picking the same identifier can be made arbitrarily small by making the random component large enough. However, this is aesthetically unpleasing because it makes what should be a deterministic problem into a nondeterministic problem and introduces the possibility of errors resulting from undetected name clashes. It may be a pragmatic solution to the difficulty but it is disappointing to find no deterministic solution. Perhaps there is an analogy with Shannon's Statistical Theory of Communication here: it is possible to transmit a message down a noisy channel with an arbitrarily small probability of error by use of a suitable encoding scheme but the probability can never be reduced to zero.

4.7. Conclusions

We have discussed various techniques for generating unique identifiers and explored the ways in which systems based on such identifiers may be combined. Unless identifiers are universally unique, it is difficult to combine naming spaces transparently but we have argued that universal uniqueness is difficult to

achieve in practice. The problems of joining name spaces together which we discussed in chapter 3 for evolutionary distributed systems must still be solved in the design of revolutionary systems. There are no easy answers. This should come as no surprise; after all, a truly transparent distributed system should be indistinguishable from a centralised system. Both will define a self-contained name space so that in both cases joining two such systems will involve merging name spaces or at least providing mechanisms for crossing name space boundaries transparently. Combining two centralised systems to form a distributed system should be exactly analogous to combining two distributed systems to form a larger distributed system.

Although it might be argued that combining individual systems was a much more common event than combining whole distributed systems, it is still just as important for the composite system to be transparently indistinguishable from the systems of which it is composed. Any extension mechanism should be recursively applicable at more than one level of system abstraction. Consequently, in the next two chapters we will explore whether it is feasible to build distributed systems recursively.

Chapter 5

Recursive Transparency and the Newcastle Connection

In this chapter we will explore the idea of building transparent distributed systems recursively. If a transparent distributed system is really functionally equivalent to the systems of which it is composed then it should be possible to use it recursively as a component of a larger distributed system. To make this idea more concrete, we will explore its implications for the Newcastle Connection by studying how closely a recursive implementation of a distributed Unix system built with the NC conforms to the Unix semantics and whether such a distributed Unix system is indeed functionally equivalent to the systems of which it is composed. We will also briefly consider how other distributed Unix systems have tackled these problems before outlining a solution which will be examined in more detail in the next chapter.

5.1. Recursive Transparency

An operating system such as Unix manages the resources of a machine and makes them available to application programs as a series of abstractions invoked through a well-defined system call interface. In effect, the operating system is an interpreter for the objects and operations defined by a virtual machine. The idea of transparent distribution is to extend this system call interface without altering its functionality so as to allow an application running on one machine to access objects on another machine. Ideally, all the individual systems should appear as one system with a single interface.

Transparent distribution can be achieved by inserting a layer of software between applications and the operating system which is transparent in the sense that it looks exactly like the operating system to an application (and exactly like

an application to the operating system). Such a layer must intercept every system call and decide whether it refers to an object on a local system or a remote system. Local operations will be passed on to the underlying operating system on the local machine whilst remote operations will be sent across the network as Remote Procedure Calls (or RPCs) to a server on the remote machine which performs the operation and returns the result.

The server is really part of the transparent distribution layer on the remote machine but will appear to be an application to the remote operating system. It would therefore be possible to insert another transparent distribution layer between the server and the operating system. This would give the server access to remote resources and allow it to create servers for itself. However, just like an application on the local machine, the server on the remote machine should be unable to tell whether it is accessing remote resources if the transparent distribution layer is really transparent. This is what we mean by the term **recursive transparency**. A server which runs on top of a transparent distributed layer is said to be **connected**. Conversely, a server which runs on top of the operating system directly is said to be **unconnected.**

Of course, a server is no ordinary application because it is really part of the transparent layer on the remote machine. It may seem strange to make it a client of another transparent distributed layer and especially to do so when its code already involves many of the details of the transparent distribution layer (such as the format of RPC messages). However, the client part of a transparent distributed layer is only responsible for intercepting system calls. It is not concerned with the nature of the application whose system calls it is intercepting and is quite different from the server part of the transparent distribution layer. Consequently, it should be possible to maintain a strict separation between the client and server part of the distribution layer and make no attempt to merge them in a single server program. A connected server built in this way will not be

able to tell which of its resources are remote and which are local. This can cause problems as we shall see when we have investigated recursive transparency in the context of a real system. In particular, we will consider whether connected servers work with the NC, a transparent distributed layer for Unix.

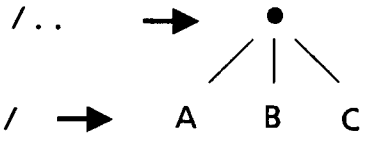
5.2. Connected Servers and the Newcastle Connection

The NC joins together a collection of individual Unix systems into a single distributed system by extending the name space on each machine. Entries in the local naming tree may refer to remote systems and a pathname can start on one system and cross the network to another. It is possible to access remote files and to run programs on remote machines as if they were local, in other words transparently.

The NC is implemented in the way described in section 5.1 as a layer of software on the local machine which intercepts every system call and determines whether it refers to a local object or a remote object. For the purposes of this analysis, system calls fall into three categories: those which take pathname arguments, those which take arguments such as file descriptors which have been derived indirectly from a pathname by a previous system call and those which take arguments such as user ids or process ids which implicitly refer to "this system". Pathnames may be examined to see whether they contain a reference to a remote system or start from a remote context. File descriptors will have been created by a previous system call also intercepted by the NC and may therefore be looked up in a table maintained by the NC. However, the third category is problematical because there is no obvious recursive structure in a system identifier which can be extended to a distributed system. The best solution is to assume that such identifiers refer to the system where the root directory / is located. This is in keeping with the way in which Unix utilities expect to find system information in files with root-relative pathnames such as `/etc/passwd`.

Once a system call has been analysed in this way it can be redirected with an RPC to a server on a remote machine if necessary. If the server is connected it will subject the system call to further analysis and possibly redirect it to a second remote system. In particular, this makes it possible for a pathname to span several remote systems without the local system which first analyses the pathname needing to know anything about the naming structure of the distributed system as a whole beyond its immediate neighbours. However, the standard implementation of the NC has unconnected servers which run directly on top of a Unix kernel without the insertion of a transparent NC layer (although experimental versions with connected servers do exist). Consequently, a pathname which passes through two remote systems will not be analysed by the unconnected server on the first of these remote systems. Similarly, it will not be possible to reach a second remote system with a pathname that starts from a remote context. Such violations of transparency caused by the use of doubly remote pathnames break the illusion of a single distributed Unix system provided by the NC.

For example, suppose that three Unix systems A, B and C are arranged at the same level in a distributed Unix naming tree constructed with the NC. From



system A, C can be named directly as /../C and indirectly via B as /../B/../C. The indirect pathname will fail because it involves accessing the remote system /../C from an unconnected server on B. In this case, it is rather perverse to use the indirect pathname when it is possible to name C directly although such redundant pathnames are sometimes generated by programs (or even people) accidentally. Indeed, a fully transparent distributed system which allowed

arbitrary links between directory entries on different machines would make it only too easy to use such pathnames by accident because the system boundaries would be invisible.

In section 3.5.1.2 we argued that loops in the distributed naming graph could be prevented by not interpreting doubly remote pathnames. However, sometimes the use of such indirect pathnames is unavoidable. In the rest of this section we will discuss three such occasions.

5.2.1. Remote Execution

As discussed in section 3.5.4, the NC implements true remote execution and always runs a program on the machine where it resides. This effectively adds an extra level of indirection during name resolution, making it impossible for a remotely executing program to name some objects without using pathnames that pass through more than one remote system. To understand why this is so, we must consider the effect that remote execution has on naming.

The current directory and root directory of a Unix process are not altered when it executes a new program. Consequently, when a process moves to a remote machine by executing a remote program, if its root and current directories were originally on the local machine, they will now be remote as far as the remote machine executing the client program is concerned and must therefore be accessed via servers. Unix only provides the root and current directory as starting contexts for naming files, so if these contexts are already remote and servers are not connected, it will not be possible to name remote files on any other systems, including files on the machine where the program is now executing. All pathname calls will be sent to the server where their starting context is located but that server will be unable to handle further remote names because it is unconnected.

For example, with the system configuration above, suppose a process on A executes a program on B. Because the root and current directory of the process remain on A, all pathname operations will be passed from B back to a server on A, including pathnames beginning /../B. If the server on A is unconnected it will be unable to handle such names and so there will be no way for the process now running on B to name objects on B (unless the current directory is moved to B before performing the remote execution).

Even with a connected server, there is no satisfactory way of naming local objects because Unix does not implement closures or provide a naming context for the local system. This makes it impossible to write a portable program that will always create files on the machine where it runs. Root-relative pathnames are not good enough because they must include system names. For example, a name beginning /../B would only name local objects if the program which used it was running on B. This is not location transparent and a program which used such names could not be moved to another machine without alteration. (As discussed in section 3.4.4, LOCUS has extended Unix naming to include a special <LOCAL> facility to solve this problem.)

In an attempt to get round these naming difficulties (and also to tackle some of the other problems of remote execution discussed in section 3.5.4) the NC provides a special version of the Unix *exec* system call named *excr* which stands for "execute with changed root". If a process executes a program with *excr* rather than *exec*, its root directory will be moved to the machine where the program resides (i.e. the machine where execution will take place under the NC interpretation of the Unix *exec* semantics). This allows both local and remote files to be named with root-relative pathnames but unfortunately, because of weaknesses in the Unix concept of a system, moving the root can have other strange side-effects on things like the meaning of process ids and user ids. Nevertheless, it could be argued that making *excr* the default remote execution

semantics fixes more problems than it creates. This would tend to suggest that in not moving the root directory during a remote *exec* the NC is interpreting the Unix *exec* semantics too literally. In particular, the NC is ignoring the fact that Unix often uses root to mean two things, an absolute location and the idea of "this system". For a single system the two are equivalent but for a distributed Unix system they may be different.

5.2.2. Network Heterogeneity

Another occasion when doubly remote names might be required would be if the network was heterogeneous and not all the systems used the same network protocol. For example, with the configuration above, suppose that `A` and `C` use different protocols and cannot communicate with each other directly but `B` understands both protocols. Then it would be reasonable to expect a pathname of the form `/../B/../C` to allow interworking between `A` and `C` using `B` as an explicit gateway. However, without connected servers this would not work, even though it is possible to access both `A` and `C` from `B`. One solution would be to invoke all commands involving `A` and `C` from `B` but this might not always be convenient if `B` was inaccessible. Fortunately, remote execution provides a way round this difficulty although *excr* must be used because of the naming difficulties described above. Since using *excr* will change the meaning of `/`, all root-relative pathname arguments must be re-written accordingly.

For example, here are three possible commands for copying a file from `A` to `C`:

```
(1) cp /foo /../C/bar
(2) cp /foo /../B/../C/bar
(3) excr /../B cp /../A/foo /../C/bar
```

The first is the most natural but fails because `A` cannot communicate with `C` directly and `B` cannot be used as a transparent gateway. The second will also fail

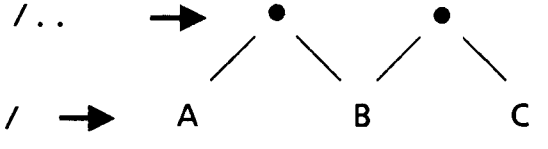
because it requires a connected server at B. Only the third possibility which uses *excr* to invoke the copy program *cp* from B will work but it is significantly more complex for the user than the other two commands.

This situation might seem rather bizarre and contrived but it has actually arisen in practice at Newcastle on a single Ethernet. The same problem would occur on a much larger scale if two large networks using incompatible protocols were joined together to form a distributed system. It is usual to ignore this problem when designing a distributed system and assume that all communication occurs over a fully connected internet. For a really large distributed system this may not be a realistic assumption. Making network boundaries visible at the application level in this way is not very attractive but at least it makes it possible to construct some kind of distributed system under these circumstances.

5.2.3. Name Space Management

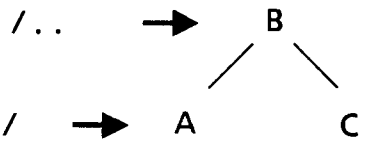
A similar situation might arise if the replicated parts of the distributed naming tree were inconsistent. Such an inconsistency could easily arise by accident rather than design, especially since the naming tree must be maintained by the collective action of all the system administrators.

For example, suppose that system C could only be named from system B and not from system A. Then the only way in which a process on A could access files on C would be via B with a pathname of the form /../B/../C. Although it could be



argued that limiting knowledge about system names and addresses to centrally administered machines such as B would be desirable (or at least convenient),

deliberate inconsistencies should be frowned upon because they violate the recursive model of distribution on which the NC is based. The proper way of dealing with this situation would be to return to a tree structure by making B the parent of A and C, but this would require a connected server at B. This is



unfortunate because for a very large distributed system it would make sense to divide up the name space into smaller domains which could be independently managed.

5.2.4. Summary of Connected Servers

Connected servers (or the equivalent functionality) are necessary before a transparent distributed system can be said to be completely transparent. In particular, connected servers are needed to make remote pathnames work properly during remote execution and to resolve redundant pathnames correctly. They are also required if the system naming tree is structured so that not every system can name every other system directly. In these circumstances, the only way of reaching an object in one domain from another would be via a connected server on a machine that knew about both domains.

In the rest of this chapter we will examine the concept of a connected server in more detail to see whether recursive transparency works in practice and whether it is the best way of achieving the full level of transparency we require. In particular, we will examine various aspects of the Unix semantics to see whether it is possible to implement a transparent distributed Unix system recursively.

5.3. Connected Servers and Unix Pathnames

Unix pathnames have no direction. A single pathname can move up or down the naming tree and in the presence of symbolic links may even jump from one part of the tree to another quite unexpectedly. As a result it can be difficult to analyse a complex pathname spanning several systems in order to determine which system it ultimately refers to.

When the NC intercepts a system call with a remote pathname argument, the pathname is only analysed to identify the first reference to a remote system it contains. The remainder of the pathname is passed to a server on that remote system in an RPC. If the server is unconnected, the pathname will not be analysed further but will simply be treated as if it were local. This leads to the breaches of transparency with doubly remote pathnames discussed in section 5.2. In contrast, a connected server will analyse the pathname further and will be able to access further remote systems via its own servers. Consequently, arbitrarily complex pathnames can be resolved (eventually) via a chain of connected servers. However, it is not always appropriate to follow such a chain of servers if the pathname loops back on itself with . . or encounters a symbolic link.

For example, suppose that a process has moved its current directory to a remote system. All pathnames which are not root-relative will be passed to the server on the machine where the current directory now resides. If the process now moves its current directory back to the local system (or to some other remote system) using a pathname which is not root-relative (such as . .), the "change directory" RPC will fail if the server is not connected. However, if the server has been connected it will create itself a server on the new system to hold the directory context, even if this is the local system. Consequently, all pathnames relative to the current directory will now pass through at least two servers. Even the simple sequence

```
cd /../remote
cd ../local
```

will lead to a process accessing all files named from the current directory via two servers, one of which will be on the local machine.

Operations which affect the naming context are special and should be treated with caution. However, any operation on a pathname involving . . or a root-relative symbolic link can cause similar problems. If a pathname passes from one remote system to another (or simply loops back to the local system) then a connected server will create a server for itself on the second system, even if there is already a server on that system created more directly by the client. An object can always be named in two ways, from root or from the current directory, but if one of these directories is remote then the two pathnames will not necessarily lead to the same server.

These examples demonstrate that the use of connected servers can lead to objects being accessed indirectly via more than one server when a more direct route is possible. Although this is inefficient, this is not in itself sufficient reason to abandon the concepts of recursive transparency and connected servers. However, as we shall see in the next section, the presence of more than one server on the same machine can cause semantic difficulties and violate transparency and so an alternative approach is needed to achieve the equivalent functionality of a connected server.

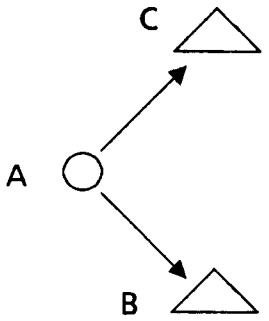
5.4. Multiple Servers

An NC server is an extension of the state of a local process on a remote machine. It acts as an agent on behalf of its client and has no independent existence of its own. Every process using the NC will have a private server, not shared with any other process, on each of the remote systems it accesses. This

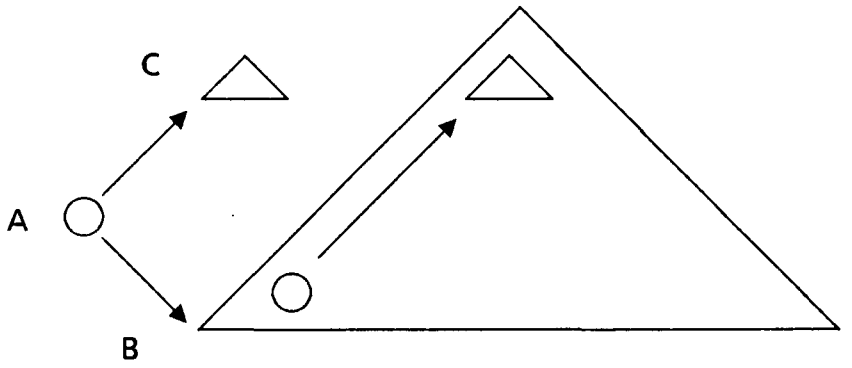
combination of a local process and its remote servers is called a Distributed Sequential Process (or DSP) because, although it is a distributed collection of processes, only one process is active at a time with the flow of control moving between process and server just as it moves between procedures in an ordinary program.

As we have just seen, if it is possible to name a system in two different ways and a DSP built with connected servers accesses the system in both ways, the DSP will end up having two servers on that machine. This might appear to be contrary to the definition of a DSP (one server per process per machine) but in fact there will be two DSPs. One of these will be recursively nested in the other and will appear as a simple server at the higher level of abstraction.

Returning to our original example of three systems A, B and C arranged symmetrically in a tree, if A accesses B and C directly as /../B and /../C respectively, the result will be a DSP with client at A and servers at B and C.



If A then accesses C indirectly as /../B/../C, a second server will be created on C but this will actually belong to the DSP consisting of client at B and server at C. This entire DSP will be indistinguishable from the server at B in the higher level DSP with client at A. Since a system can always be named in two ways (relative to root or relative to the current directory) this situation can easily arise in practice. But what harm can come of having two servers on the same machine? Will the recursively structured DSP continue to be indistinguishable from a single Unix



process, or will it breach the normal Unix semantics in some way? In order to answer these questions, we must consider how a DSP is implemented.

A DSP consists of a collection of processes spread across several Unix systems. However, in the interests of transparency, this collection must masquerade as a single virtual process running on a single virtual Unix system. On each individual Unix system the identity of this virtual process is synonymous with the identity of the real process that runs as its representative on that system. Clearly if a single DSP has two component processes representing it on a given system, there will be a conflict of identity and so it is in this area of the Unix semantics that we must look for difficulties.

5.4.1. Access Rights and Ownership

Access rights are closely associated with identity. Unix permissions are based on the concept of users and groups and every process has a user id and group id. A server forming part of a DSP effectively has a user id belonging to another system, specifically the user id of its client. As discussed in section 3.5.3, such a remote user could be represented quite naturally with a pathname of the form `/../remote/user` but unfortunately Unix uses small integers taken from a flat name space to represent user ids and a name space without any structure cannot be recursively extended to include the notion of a system. Instead, the NC derives the identity of each server from the identity of its client by mapping what is

effectively a pathname made up of the address of the client system and the user id of the client process on that system into a specific local user id on the server system. This allows each system administrator to control which remote systems and individuals are allowed to access his system over the network. There is no need for each system to support the same set of users as there is for some distributed systems.

This scheme works well with unconnected servers where there is only one level of mapping but is complicated by the presence of connected servers. If a connected server becomes the client of another server in a recursively structured DSP, the identity of the second server will be derived from the local identity of the first server rather than the identity of the original client to whom the entire DSP conceptually belongs. Otherwise the intermediate server would not be connected transparently. In other words, it would be aware that it was a server and therefore special. Because there is a natural tendency to give less permission to remote users of a system, it follows that if there are two servers for the same client of a recursive DSP on a given system, it is quite likely that they will have different permissions associated with them, especially if one has been created indirectly by another server rather than directly by the original client.

For example, returning once more to the three systems A, B and C arranged symmetrically as subdirectories of /.., suppose that there is a user id robert on A and C but not on B. If the two roberts are the same person, then it makes sense for C to map the conceptual user path /A/robert into the local user robert. However, knowing nothing about robert, B will map /A/robert into a.n.other, a default guest user with minimal access rights. Naturally C will map /B/a.n.other into the local version of a.n.other. There is no conceivable reason for C to map /B/a.n.other onto robert and indeed the real Robert would not be pleased if his files on C were compromised in this way! However, as a consequence of this, if user robert on A creates a server on C by

using a pathname beginning `/../C`, the resulting server on `C` will have the permissions of `robert` on `C` but if `robert` on `A` then uses a pathname beginning `/../B/../C` to refer to the same object, a second server will be created on `C` with only the permissions of `a.n.other`. In other words, the access that a user on `A` has to objects on `C` depends on whether the pathname used to access them passes through `B`.

Although there is clearly a difficulty here, it is not immediately obvious that this apparent paradox is a violation of Unix semantics and a clinching argument for dispensing with connected servers. Even on a single Unix system, a user must have search permission on all the directories mentioned by a pathname in order to use that pathname and therefore some pathnames to an object will work and others will not. This is not quite the same as both pathnames working but granting different access rights but there is some similarity. Indeed, one might argue that it is quite reasonable to associate different permissions with different pathnames and that this is a good way of providing security on a "need to know" basis.

A more convincing paradox can arise with the *chown* system call which changes the ownership of a file. If *chown* is applied to a remote file, the new owner of the file must belong to the remote system rather than the local system. Consequently, the user id supplied as parameter to the *chown* call on the local system must be mapped to an appropriate user id on the remote system. As before, the mapping will depend on the pathname used to name the file and so the owner of the file after the *chown* call will depend on this pathname too.

With our example above

```
chown robert /../C/file
```

will result in the file being owned by `robert` but

```
chown robert /../B/../C/file
```

will result in it being owned by a.n.other. This is certainly not the Unix semantics.

Connected servers can interact with guest users such as a.n.other in another interesting way. As explained in section 3.5.3, just as there is a need to map user ids when creating servers on remote machines, so there is a need to perform the inverse mapping when reporting the ownership of remote files. However, user id mapping is an expensive operation and as a compromise the NC simply reports whether or not a remote file is owned by the local process. When the ownership of a remote file is examined, the result is either the user id of the local process or else a special default value to indicate that the remote file is not owned by the local process. Since there is already a user id which is used to indicate a user for whom there is no mapping (namely the default guest user a.n.other), it is convenient to overload this value and use it for this purpose too. So long as no process using the NC runs as a.n.other this overloading will not cause any problems. Although servers may run as a.n.other they are usually unconnected and do not use the NC. However, a connected server running as a.n.other can cause the ownership of remote files to be reported incorrectly.

For example, with the user id mappings used in the previous example, suppose that /file on C is owned by lindsay. Examined from A as /../C/file by robert it will correctly appear to be owned by a.n.other. If a connected server running as a.n.other on B examines the same file, the NC will also report that the file is owned by a.n.other (i.e. a user other than the equivalent of a.n.other on C). However, because the server on B runs as a.n.other, it will mistakenly think that it owns the file and this will cause the NC to report that the server's client on A owns the file too. Since a connected server on B created by robert on A will run as a.n.other (because robert is not mapped by B), it

follows that if `robert` on `A` examines `/../B/../C/file` it will appear to be owned by `robert` even though it is actually owned by `lindsay`!

This problem is really caused by the fact that the user id `a.n.other` is being used to represent two things: the default access rights for an unmapped user id and the owner of a file on a remote system. It would be possible to resolve this ambiguity by using different user ids for these two purposes but it is only necessary to do so because connected servers break the assumption that made it safe to overload one value in the first place.

All of these problems could be avoided by defining a consistent user id mapping within each naming domain. If the mapping relation was transitive then all the servers on a particular machine for a given DSP would be created with the same user id, irrespective of the route by which they were created. However, achieving a consistent mapping might involve an unusual degree of cooperation between the individual system administrators. In effect, each system would have to support the union rather than the intersection of all the other systems' users. In the limit this would mean that the systems would share the same list of users. Nevertheless, this cooperation would only need to extend to users within a given naming domain because those systems which belong to two naming domains form bottlenecks through which all names must eventually pass if they are to reach another naming domain. (It must be assumed that there are no circularities in the tree of naming domains since otherwise the problem will rapidly become intractable.)

5.4.2. Resource Allocation and Locking

Another problem area with multiple servers also concerns the question of identity. If a process acquires some resource from the kernel, the kernel will keep a record of the allocation by recording that the resource is now owned by a particular process id, a value taken from another flat numeric naming space. As

with user ids, it is not possible to identify a remote process with a pathname and consequently, when a resource is acquired remotely by a server on behalf of a client, the remote process can only be identified with the server's process id. However, this will do just as well providing that all manipulations of the resource are made through the same server. In general, this is no problem even in the presence of multiple servers on the same machine because Unix resources are strictly local to a process and cannot be shared with other processes. They are usually acquired as the result of a pathname operation and consequently the NC knows exactly which resources belong to which servers and there is no possibility of confusion. So long as the same pathname is always used to access it, all operations involving a given resource will automatically be passed on to the correct server, if necessary via a chain of servers. However, difficulties will arise if the effect of acquiring the resource is visible outside a single Unix process and therefore operations such as locking which affect the global state of a Unix system will cause problems.

If a Unix process acquires an exclusive lock on a file using some pathname then even if the file can be named in other ways, other Unix processes will not be able to lock the same file. However, it is reasonable to expect that the process owning the lock should be able to relock the file (possibly using a different pathname) without ill effect. Within a single system, a Unix kernel is able to tell whether two pathnames are equivalent and whether two processes are the same because it works in terms of low-level identifiers which are unique within a single system. Unfortunately, there is no way that the NC can persuade a Unix kernel that two distinct Unix processes are actually part of the same process on some virtual Unix system and so it follows that an attempt to lock the same file twice using two different pathnames which pass through different systems to two distinct servers on the same machine will fail, in deadlock if the process tries to wait until the lock is released. However, unlike the Unix kernel, there is no way

that a transparently connected DSP can recognise that two pathnames actually refer to the same system and prevent this from happening because it uses localised purely relative naming. There is no mechanism for taking a global view of the system, especially dynamically on each pathname access.

This time there really is a problem with emulating the semantics of Unix in the recursive system although it is not clear how likely this problem is to occur in practice. A sequence of locking calls involving the same object named in different ways will succeed if the object is local but will fail if the pathnames pass through remote systems by different routes. It is also very difficult to get round this problem because there is no possibility of mapping the process identity so as to fool the underlying kernel. Obviously it is possible to conceive of forms of locking or resource allocation which use an explicit unique id to represent the owner of the resource and do not suffer from this problem but this would not be Unix. In any case, such operations would not be based on pathnames and would therefore be difficult to distribute transparently. The problem here is that the form of identity used to record ownership should include a system identifier. However, without unique ids, it is not possible to tell whether two pathnames are equivalent and hence identify systems unambiguously, especially in the presence of connected servers. Even if the kernel did understand pathname identifiers, it would be unreasonable to expect it to be able to recognise equivalent but not identical pathnames if a DSP cannot do so. We will return to the problem of telling whether two pathnames are equivalent in section 6.4.

5.4.3. Flattening the Recursion

Multiple servers on the same machine only arise because connected servers impose a strictly recursive interpretation upon the DSP model. It is perhaps worth considering what might happen if the recursion was flattened by preventing more than one server per DSP from being created on each machine.

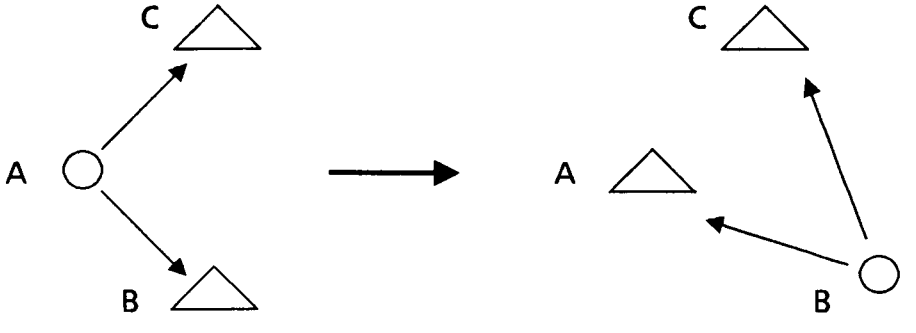
This would involve distributing the list of servers making up the DSP between each server so that every server knew about every other. Every time a new server was created its details would need to be propagated to the other servers. This could be done using a broadcast or multicast protocol. Alternatively, the knowledge could be propagated on a "need to know" or "lazy" basis. The current list of servers (or the most recent changes to the list) could be incorporated into the RPC protocol so that each server received the information the next time it was invoked to perform some remote operation. One difficulty with this approach is that it would require unique names in order to identify all the systems correctly. There would also be reliability implications because the distributed list of servers would not always be in a consistent state. However, the main problem concerns the authorisation difficulties mentioned earlier. If the first server on a given system is created indirectly by another server, it will not necessarily be given the permissions it would have received had it been created by a more direct route. In section 5.7 we will discuss a mechanism called DIY which tackles this problem by ensuring that all server creations are initiated by the most direct route from a central location.

Of course, it would be possible to go even further and abandon the DSP model altogether. Multiple servers are only a problem because servers are private with one server per system per process representing the state of the DSP everywhere. If the concept of multiple private servers is abandoned in favour of a single public server on each system then the problem does not arise. However, this public server must be able to identify its client and without unique identifiers this involves solving the same problem of determining whether two pathnames from an arbitrary naming graph denote the same object. Unique identifiers are not acceptable for this purpose because they are not recursive and therefore do not scale or allow the name space to be divided up into individually managed domains.

5.5. Remote Execution

Remote execution is another area where connected servers fail to work correctly and the idea of recursive transparency breaks down. However, in this case the reason is that the NC server must assume it is unconnected in order to implement remote execution correctly. This is because remote execution involves a sort of bootstrapping process in which a new NC layer is inserted between the server program and the Unix kernel as the server becomes a client.

The NC implements true remote execution rather than remote paging and this involves rearranging the DSP. The client and the server on the system where the remote execution takes place must change places because the controlling point of the DSP from which all system calls are generated will move to the remote machine. The server process will *exec* the new client program whilst the original client process becomes a server acting as an agent for the new client. When the



original client has more than one server, those servers not involved in the remote *exec* must re-establish communications with the new client as directly as possible. With a single naming domain and network address space this is relatively straightforward because direct communication is always possible. However, it is during this rearrangement operation that the NC code assumes that its servers are not connected.

The NC layer of software inserted between programs and the Unix kernel to provide transparent distribution contains various data structures which are used

to control name interpretation and indicate which of the resources owned by the process are local and which are remote. Clearly, it is important that this state information be preserved as one program executes another. When a remote execution takes place, these data structures must be rearranged as the client part of the DSP moves from one system to another to reflect the fact that resources which were local are now remote and resources on the system where the *exec* is taking place are now local.

The server on the system on which the remote *exec* takes place will become the new client and must therefore construct an appropriate data structure for the new NC layer in the client which will reflect the location of all resources relative to the new client. The code in the server which constructs this data structure assumes that all the resources which belong to the server itself are local. This is certainly correct if the server is unconnected. However, if the server is connected then some of its resources may actually be remote although there will be no way of telling which resources are local and which are remote because the NC layer attached to the server is transparent. Consequently, the NC data structure which a connected server constructs for the program it is executing will not be correct.

More seriously, there will actually be two conflicting data structures because the NC layer attached to the connected server will construct its own view of the server's resources as part of the NC algorithm for what is an ordinary local *exec* from this viewpoint. Just as the connected server is unaware of its own remote resources, so its NC layer is unaware of the fact that the server is part of a DSP and will inherit other servers as it becomes a client after the *exec*. Transparency works in both directions and the correct overall picture can only be obtained by merging both views. However, both views are correct in their own right because they belong to separate DSPs which are part of different virtual Unix system abstractions. But unless these two distinct layers of abstraction are preserved in the new client, one or other of the views they represent will be compromised and

the whole algorithm will break down. In effect, servers in one DSP or the other will be forgotten about and certain resources held by those servers which should be remote will be treated incorrectly as if they were local. The most likely effect is that the program being remotely executed simply does not work properly.

As an example, consider the three systems A, B and C again, arranged symmetrically in a tree. Suppose that from A we wish to execute the *cat* program remotely on B and use it to list its standard input. Because standard input will be opened on A before *cat* is executed on B, its file descriptor (which may refer to a local or remote file) will be one of the resources that will be inherited across the *exec* boundary. The NC will have a record of whether the file descriptor is local or remote and this must be adjusted to reflect the rearrangement of the client and server. All of the following examples will work because none of them requires connected servers and consequently only one level of DSP and virtual Unix system is involved:

```
/../B/bin/cat < file
/../B/bin/cat < /../B/file
/../B/bin/cat < /../C/file
```

On the other hand, if there is a connected server on B and an indirect pathname is used to name a file on C then there will be two levels of DSP. The server on B will be the client of another server on C which will hold the true file descriptor for the standard input of the *cat* command:

```
/../B/bin/cat < /../B/../C/file
```

In the topmost DSP with A as client in which the remote *exec* takes place the file descriptor will be remote on A and local on B. Consequently, after the remote execution has taken place B will have forgotten about its server on C and will treat the file descriptor as if it were local. This is incorrect and without a valid file

descriptor for its standard input the *cat* program will terminate without printing anything.

At the risk of causing further confusion, here is a command that will work with a connected server at B:

```
/../C/bin/cat < /../B/../C/file
```

Because the connected server is not involved in the remote execution, both A beforehand and C afterwards will regard the file descriptor as being remote on B and even though it is actually handled by a server on C (which will be ridiculously inefficient because *cat* is already running locally on C) this will not cause any problems. In other words, the algorithm only breaks down when the *exec* in one DSP takes place at a server which is part of another DSP and itself owns remote resources on other servers. Furthermore, the key point is that a remote resource owned by a connected server cannot be passed across an *exec* boundary. If it is acquired after the *exec* there will be no problem. Consequently the following examples will work with connected servers:

```
/../B/bin/cat /../C/file
/../B/bin/cat /../B/file
/../B/bin/cat /../B/../C/file
```

In each case, the file to be listed is an argument to the *cat* program and is therefore opened after the *exec*. None of the examples would work without connected servers but each will involve at least two servers (the third will involve three) where at most one server or even purely local access would be possible. Connected servers do not give the most efficient solution by any means. The DIY mechanism proposed in section 5.7 would be required to sort out the optimal route from the client to the server and prevent the absurdity of creating a server next to the client on B in two of the above examples.

One further possibility, namely a connected server executing a program which it thinks is local but is actually remote, will also fail to work correctly. What should happen is that as the server tries to turn itself into a client it discovers at the last minute in its NC layer that it should remain a server since the new client is actually being created on another system. The DIY mechanism described in section 5.7 avoids the problem of nested remote *exec*s by tracing all pathnames to the server on the system where the program resides.

What can be done about this problem of connected servers and remote execution? Clearly it is not possible or practical to support two (or more!) NC layers simultaneously. However, it should be possible to reimplement the file server "exec with NC data" operation so that it merged the two levels of DSP rather than replacing the state information of one with the other. This would mean that the NC had to be able to tolerate the existence of two servers on the same machine because it would not be possible to merge two servers from different levels of DSP. However, the NC itself would never create such multiple servers directly. They would only arise as a result of merging two recursively structured DSPs and duplicate servers from the nested DSP would only exist as long as their resources existed; they would never be used to acquire new resources. (Presumably it would be sensible to promote a server from the inferior DSP if no duplicate existed at a higher level. However, as explained earlier, such a server would not necessarily have the same access rights as a server created by a more direct route.) Such a merge algorithm would be able to support connected servers without significantly compromising transparency. Only the new "exec with NC data" operation would not be transparent because by definition it is not part of the Unix system call interface. However, some implementation of this operation is required even when an unconnected server is implemented directly on top of a Unix kernel so this is not really a problem.

5.6. Other Distributed Unix Systems

So far we have only considered the implications that recursive transparency has for the NC. Before looking at possible solutions to the problems it poses, we will consider the other distributed Unix systems discussed in section 3.5 to see how they have tackled these issues.

Both NFS and RFS are based on the idea of a remote mount. This means that they can exploit the existing kernel mechanism for crossing mount points and deal with . . correctly. RFS is closest to the NC in that its RPC protocol works in terms of pathnames. In other words, if a remote mount point is encountered while a pathname is being resolved, the entire system call is continued on the appropriate remote system. However, this mechanism is not recursive because the RFS server on the remote machine does not cross further remote mount points in the same way. NFS looks up pathnames in their entirety before generating an RPC for the required operation. Furthermore, it looks up remote names one segment at a time, effectively reading remote directories and resolving names locally, rather than passing the pathname across to the remote system for resolution. One consequence of this is that the local system must be aware of all the mount points on the remote system, even those for local file systems. This can be very expensive and an administrative nightmare, even for a moderately sized distributed system, and proposals have been made to hide at least internal file system boundaries on remote systems so that only the root of each remote system need be mounted in the local file system. Using one RPC to resolve each remote segment of a pathname and then a further RPC to perform a remote operation is also very expensive and NFS is only able to function efficiently because caching is used to avoid the need for name lookup as much as possible. LOCUS also resolves pathnames locally by reading remote directories and therefore has the same problems.

Both LOCUS and NFS require a single user id space across all systems and hence overcome some of the identification problems arising out of mapping user ids between systems. However, this requirement causes problems when two independently managed systems are combined and, although conflicts can be resolved by re-allocating user ids as systems are merged, this approach simply does not scale since every system must know about every other. RFS does at least support user id mapping like the NC.

One important difference between the NC and the other implementations of distributed Unix is that only the NC uses private servers, one per process per machine. The other distributed Unix systems effectively have a single public server on each machine which provides the remote file system abstraction for every client. The implications of this were discussed briefly at the end of section 5.4.3. The question of inadvertently creating multiple servers on the same machine for a single client does not arise. However, because a public server manages remote resources on behalf of many clients, each request for service must be accompanied by the identity of the client making it. Consequently, the same semantic conflicts can occur if a client can reach a server by more than one route but may only be identifed using the route by which it reached the server. In these circumstances, identity would be represented by a pathname and in order to discover whether two clients were identical it would be necessary to test whether their identifying pathnames led back to the same system. For an arbitrary naming graph this problem is not soluble without introducing unique identifiers (which are contrary to our recursive philosophy) as we shall see in section 6.4.

In conclusion, the problems of recursive transparency are not unique to the NC but other distributed Unix systems do not have a solution to offer either. Instead, they avoid the issue by making simplifying assumptions or imposing unreasonable restrictions on the construction of the distributed system.

5.7. Towards a Solution - DIY

We have seen in this chapter that although the functionality of a connected server is needed in order to make a distributed system fully transparent and to allow a name space to be partitioned into domains, the idea of recursive transparency and connected servers simply does not work in practice. Connected servers can lead to more than one server being created on the same machine and this can cause semantic conflicts, especially over the notion of identity. Remote execution cannot be implemented transparently with a connected server either, and pathnames involving . . which pass in and out of systems cannot be resolved satisfactorily.

The solution to these problems is to relax the strict view of transparency that leads to the notion of recursive transparency and attempt to flatten the recursion. To avoid creating extra servers unnecessarily, a connected server must be aware of the fact that it is part of a DSP. Because other servers belonging to that DSP might exist on other systems, the server must be careful about creating new servers. Indeed, it would be better if all servers were created directly by the local process, assuming there is only one naming domain and all systems are equally accessible. The distribution layer cannot be added transparently to the server because it is transparency that causes the extra servers to be created. Instead, the distribution layer must somehow be merged with the server code.

The problem is that name resolution and performing remote operations have been combined into a single RPC. Following a pathname through a chain of connected servers will reach the correct system in the end but not by the most direct route. Separating out the process of name resolution into a name lookup RPC to servers that know whether pathnames are local or remote would be simpler but less efficient because every remote operation would require an extra RPC to look up the name first. Since most remote objects can be accessed directly

with only one server, this approach will effectively double the number of RPC calls.

If name resolution and performing remote operations are to be combined into a single RPC in the interests of efficiency, a routing layer based on pathnames must be added to the RPC protocol. If a pathname operation is directed to the wrong server (i.e. the pathname is not local to that system), the routing layer will generate an exception and indicate a better pathname for the client to try again with. Such an exception (called "Do It Yourself" or simply "DIY") will be sufficient for a single naming domain because the client will always be able to interpret the name itself. A DIY mechanism handles the current directory problem nicely and also copes with redundant pathnames which go in and out of systems because it ensures that servers are always created and accessed by the most direct route through the naming graph. For multiple naming domains, the routing layer must be able to determine whether the improved pathname is indeed accessible to the client (lies within the same naming domain) or whether a new server must be created. So long as there is only one chain of servers leading into each naming domain from the client there is no danger of redundant servers being created. However, this may be hard to guarantee when the overall naming tree contains loops and it is possible to reach a naming domain via two different routes which are sufficiently indirect for the algorithm to break down.

5.8. Conclusion

In the next chapter we will explore the idea of combining name resolution and performing remote operations into a single RPC in more abstract terms. Although we have used Unix and the NC to introduce the topic of recursive transparency, we believe that the problem is more general than this and not simply an artifact of strange characteristics of Unix. Identity and name equivalence are fundamental issues in the design of any naming system but we have shown that it

can be difficult for a recursively structured distributed system to achieve transparency in these areas if it is implemented recursively using connected servers. We must therefore find a way of achieving the same degree of transparency without using a recursive implementation. The DIY mechanism introduced in section 5.7 is the approach we shall take.

Chapter 6

An Abstract Approach to Recursive Transparency

Chapter 5 introduced the concept of recursive transparency using Unix and the NC as an example of a transparently distributed system. In this chapter we will reconsider the topic of recursive transparency in the more general context of an abstract model of recursive distributed systems. Our model is object-oriented and its basic computational step is to perform an operation on an object. Objects are identified by name, and names must be resolved in order to locate objects. We will explore the implications of recursion for this model and consider mechanisms for combining name resolution with the RPC used to perform remote operations. This requires adding a pathname-based routing layer to the RPC protocol. We will also examine algorithms for simplifying pathnames statically and determining whether two pathnames denote the same object in a distributed system constructed without the aid of globally unique identifiers.

6.1. Introduction

In section 5.1 we argued that an operating system can be thought of as an interpreter for the objects and operations defined by a virtual machine. Programs which use the services of an operating system by issuing a system call are in effect performing operations on objects. We may therefore describe the system call interface more abstractly in terms of a *perform* operator. The expression *perform(OP, NAME, ARGS)* indicates that operation *OP* is to be performed on object *NAME* with arguments *ARGS*.

A distributed system constructed from many such systems introduces the concept of a location and the idea of local and remote objects. Although the component systems of a transparent distributed system only allow operations to

be performed directly on local objects, the goal of transparency is to allow remote objects to be accessed and manipulated from any system. Consequently, constructing a transparent distributed system involves generalising the naming scheme of individual systems to include remote objects and extending the *perform* operation accordingly. This requires a *resolve* operator which maps the name of a remote object into the address of a system and the name of the object on that system. Given such a *resolve* operator and the ability to send messages between systems, it is possible to construct a transparent distributed system by designing an RPC protocol and using a client/server model to perform operations on remote objects.

The essence of this construction technique is the way in which the local *perform* operation is extended to handle remote objects. One possible implementation of remote *perform* is given by the following algorithm:

```
perform(OP, NAME, ARGS)
{
    [address, name] ← resolve(NAME)
    if (address = my-address)
        → OP(name, ARGS)
    else
        → rpc(address, OP, name, ARGS)
    fi
}
```

Notice that the *resolve* operation which converts the *NAME* argument denoting a remote object into an *[address, name]* pair is quite distinct from the RPC that actually leads to the desired operation being performed on a remote system. Also, because *resolve* returns a name which is guaranteed to be local on the remote system, there is no need for the server on that machine to resolve the name further. (In other words, the server need not be connected.) Instead, it need only evaluate *OP(name, ARGS)* locally and return the result. This may seem unnecessarily restrictive and lacking in generality but it is a natural consequence of the strict separation of the *perform* and *resolve* operators. In the rest of this

chapter, we will explore ways in which this separation may be relaxed and develop a more recursive way of constructing distributed systems.

6.2. Name Resolution, Recursion and Transparency

If a distributed system is to be functionally equivalent to the systems of which it is composed then it must use the same form of naming. Furthermore, if distribution is to be transparent, the names of remote objects must be indistinguishable from the names of local objects. When the independently managed name spaces of individual systems are combined to form a distributed name space some name clashes may occur. Consequently, the *resolve* operator must be able to map between names in the distributed system and names local to a particular system. This allows names in the distributed system to coincide with names on local systems provided that all names are interpreted by *resolve* so that it is not possible to mix names belonging to individual systems with names from the distributed system as a whole.

Such an overlap between the name spaces is desirable because if objects local to a given system have the same name in the distributed system, programs which are tied to that system by the use of local names will continue to work in the distributed system because the same names will denote the same objects. If local system names are relative to a system naming context then it will be easy for names to retain their meaning in the distributed system providing each system retains its own system naming context. Unfortunately, this is arguably a breach in functional transparency because the component systems in the distributed system will still be visible since each will have its own distinct view of the naming space. The distributed system as a whole will not present itself as a single system naming context.

For a flat name space, names are either local or remote and the *resolve* operator simply looks them up in a table. With a hierarchical name space a more

structured approach is possible. In chapter 2 we described how pathnames were a natural naming mechanism for a hierarchical system and in chapter 3 we showed how pathnames could readily be extended in a recursive and transparent manner to a distributed system by introducing the concept of a remote context. Implementing *resolve* for pathnames with remote contexts involves following the pathname from its starting context until a remote context is encountered. This is mapped into the address of a system and the name of a context on that system at which the process of resolution can continue with the remainder of the pathname until finally the object denoted by the pathname has been identified. The problem with this approach is that the organisation of the name space reflects the location of objects in the system. Any pathname which passes through a particular remote context will denote objects on that system (or a more remote system reached from it). Names are not entirely location transparent. The solution to this problem is to allow remote leaf nodes as well as remote contexts. If a leaf node can refer to an arbitrary remote node, then the structure of the naming graph can be made independent of the location of objects. Names should not be confused with locations but this is what pathnames have a tendency to do.

Name resolution will inevitably involve consulting some table or directory of names and addresses at some stage, irrespective of the form of name used by the distributed system. This table will either be stored locally on each system or else stored at some centralised point accessible via a name server. A name server is a single point of failure but storing the table locally involves replicating it across all systems and maintaining consistency between the various copies (assuming that all systems share the same view of the name space). It is also possible to use a mixture of these two approaches, perhaps employing a recursive hierarchy of name servers. Maintaining consistency across such a naming scheme can be difficult, especially if the naming structure is dynamic and changes frequently,

but this problem is somewhat orthogonal to the topic of this thesis. Here we are only concerned with the recursive aspects of this approach.

In the absence of a centralised name-server (or series of name-servers) with absolute (or collective) knowledge of the location of all objects in the distributed system and the ability to map a remote name into an address and local name, it is possible to implement *resolve* in a decentralised way by letting every system provide a limited name server capability. A *partial-resolve* is performed locally which maps a given name into another name and a location at which this second name may be resolved further. This forms the basis of a recursive implementation of the full *resolve* which uses the RPC mechanism to pass a name through a series of systems so that (hopefully) it gets progressively simpler (i.e. "more local") at each stage until finally the local system for the name is reached and the object the name denotes is located.

It is obviously desirable that this process of resolution gets closer at every stage so that it converges rather than diverges but in the presence of aliases names may sometimes become temporarily more complicated. Aliases are problematical because although they should ideally have a static meaning which is independent of the client that interprets them, in practice some naming systems allow aliases to depend on a dynamic context belonging to the client who expands the definition. (Root-relative symbolic links in Unix are an example of such a feature.) If the server responsible for resolving the alias does not know the location of this context, it will be unable to give a location at which the resolution can be continued.

Quite apart from this complication, in the absence of centralised control over naming, it may be difficult to define a measure of how close a name is to the object it denotes and hence guarantee that the resolution process will eventually terminate. However, assuming that the naming space has been set up

consistently so that this problem does not arise, the *resolve* algorithm may be described as follows:

```
resolve(NAME)
{
    [address, name, resolved?] ← partial-resolve(NAME)
    if (resolved?)
        → [address, name]
    else
        → rpc(address, RESOLVE, name)
    fi
}
```

The *partial-resolve* function returns a flag which indicates whether it succeeded in resolving the entire name or whether it has only come up with a "closer" location from which the resolution may be continued. In a truly decentralised system, the *resolved?* flag would simply depend on whether *NAME* proved to be entirely local or not. The *partial-resolve* function would have no knowledge about other systems. However, this approach would be less efficient than an implementation that included non-local knowledge because the only way now of guaranteeing that an object was located at a particular site would be to issue a *RESOLVE* RPC to that site. Consequently, every remote operation would now involve at least two RPCs, namely one to resolve the name of the object and one to perform the operation at the remote site. Even worse, both of these RPCs would be directed to the same site. We will return to this point in the next section. However, it is worth noting a special case. If there is only one level of distribution so that remote names cannot span more than one system, the *partial-resolve* function will always be able to locate an object directly and no name resolution RPCs will be necessary.

There is a problem with this version of the *resolve* algorithm. It generates a recursive chain of RPC calls which might inadvertently lead back to a system which had already been visited if the pathname contained a loop. An alternative

approach which would have only one outstanding RPC at a time would use a loop as follows:

```
resolve(NAME)
{
    [address, name] ← [my-address, NAME]
    repeat
        if (address = my-address)
            [address, name, resolved?] ← partial-resolve(name)
        else
            [address, name, resolved?] ← rpc(address, PARTIAL-RESOLVE, name)
        fi
    until resolved?
    → [address, name]
}
```

This version of the algorithm uses a *PARTIAL-RESOLVE* RPC rather than relying on a recursive algorithm to completely resolve the name. Each RPC merely checks whether a name is local and if not returns an *[address, name]* pair which is "closer" in the sense discussed above. The algorithm is controlled from a single centralised point (the local system that initiated the operation in the first place) rather than distributed throughout the named systems recursively. In fact, this new algorithm will involve no more RPCs than the original algorithm and may well involve less if a name loops back to the same system twice.

For example, if A and B are systems which have been mapped into remote contexts of the same name, then an example of a pathname which looped back on itself would be /B/A/x. This could be resolved from A using one RPC to B with the new iterative algorithm but would require two RPCs (from A to B and then from B back to A) with the original recursive algorithm. Eliminating the recursion in this way is desirable because it reduces the number of RPCs and prevents unnecessary callbacks.

So far, little has been said about the address part of the result of these operations. However, there is an implicit assumption that all the systems can communicate equally with each other and that the addresses have a unique global meaning. This makes it possible to pass these values around freely as the

results of RPCs. If the address space is partitioned into domains or the values are not globally meaningful, more care is needed. The RPC mechanism must be aware of when a value is passed from one domain to another and must either massage the value accordingly or substitute a surrogate that can be used correctly but transparently. Such surrogates will take the form of `value@system` and may be recursively nested. The RPC layer is responsible for forwarding messages addressed to such surrogate addresses and mapping surrogate values as they are passed between systems. In fact, the *PARTIAL-RESOLVE* algorithm should only be used within a single domain with the fully recursive *RESOLVE* algorithm being used between domains.

6.3. Combining Perform and Resolve

In discussing recursive implementations of *resolve* we have so far deliberately kept name resolution distinct from performing operations on remote objects. However, as we remarked earlier, a completely recursive implementation of *resolve* is inefficient because every name would have to be checked on the system it purported to belong to before its location could be guaranteed. The final *RESOLVE* RPC which verified the location would immediately be followed by a *PERFORM* RPC to the same system to perform the required remote operation. If the two operations were combined, an RPC could be saved.

The algorithm for a combined *perform* and *resolve* uses a purely local version of *resolve* which either detects a local name or supplies an address where a better name may be tried. It is in fact equivalent to the fully decentralised version of the *partial-resolve* function with no knowledge of remote systems but has been renamed *local-resolve* to avoid confusion.

```
perform(OP, NAME, ARGS)
{
    [address, name] ← local-resolve(NAME)
    if (address = my-address)
        → OP(name, ARGS)
    else
        → rpc(address, PERFORM, OP, name, ARGS)
    fi
}
```

This is clearly very similar to the algorithms that have gone before. In fact, apart from the use of *local-resolve*, the only real difference is that an explicit *PERFORM* RPC is used to invoke this same code at the remote site recursively.

Of course, just as before, loops in the naming structure will cause a recursive algorithm to loop back to a system which has already been visited. However, as with *PARTIAL-RESOLVE*, the algorithm can be restructured so that all the RPCs are coordinated from one place, namely the local site at which the operation was initiated. The modified algorithm then becomes:

```
perform(OP, NAME, ARGS)
{
    name ← NAME
    repeat
        [address, name] ← local-resolve(name)
        if (address = my-address)
            → OP(name, ARGS)
        else
            repeat
                [result, address, name] ← rpc(address, PERFORM, OP, name, ARGS)
            while (result = DIY and address != my-address)
        fi
    while (result = DIY)
    → result
}
```

When a server receives a *PERFORM* RPC it must resolve the name argument locally to determine whether the object on which the operation is to be performed is indeed local (to the server). If the object is local then the operation will be performed and the result returned but if it is remote then the address of a new remote system and a name on that system will be returned with an indication that the client should try again at that location. This assumes that all systems are equally accessible so that the client is able to communicate directly with any

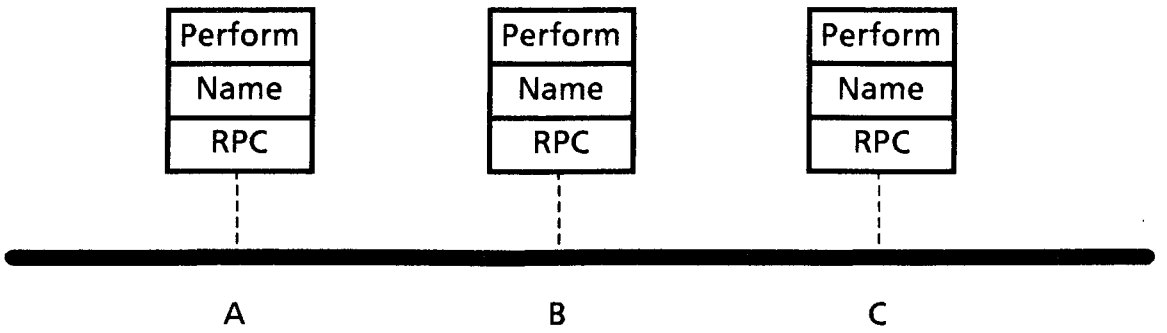
system that a server knows about. (If the systems are not equally accessible and the server is aware that the remote system is inaccessible to its client, the algorithm must be started again with the server becoming a client in a nested distributed system. As before, the iterative algorithm can only be used within a single domain and the recursive algorithm must be used between domains.)

Assuming that all systems are equally accessible (i.e. assuming a single domain) the server algorithm may be described as follows:

```
server-perform(OP, NAME, ARGS)
{
    [address, name] ← local-resolve(name)
    if (address = my-address)
        → [OP(name, ARGS), nil, nil]
    else
        → [DIY, address, name]
    fi
}
```

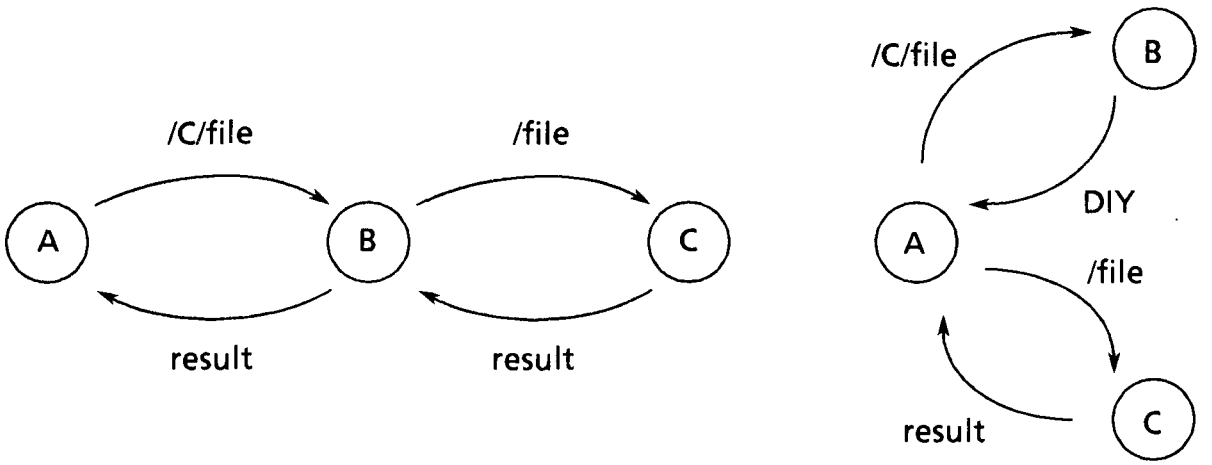
Notice that if *local-resolve* discovers that *NAME* is local to the server, the operation is performed locally and the last two components of the result structure are irrelevant. The result of a *PERFORM* RPC is really a union of two possible types: a success value indicating the result of the operation or a DIY exceptional value indicating a "better" (presumably "closer") place to try the operation.

It is perhaps easier to visualise this combined implementation of *perform* and *resolve* as three distinct layers. The top level actually performs the operation (locally or remotely), the middle level resolves pathnames and the bottom level relays RPCs between addresses on the network.



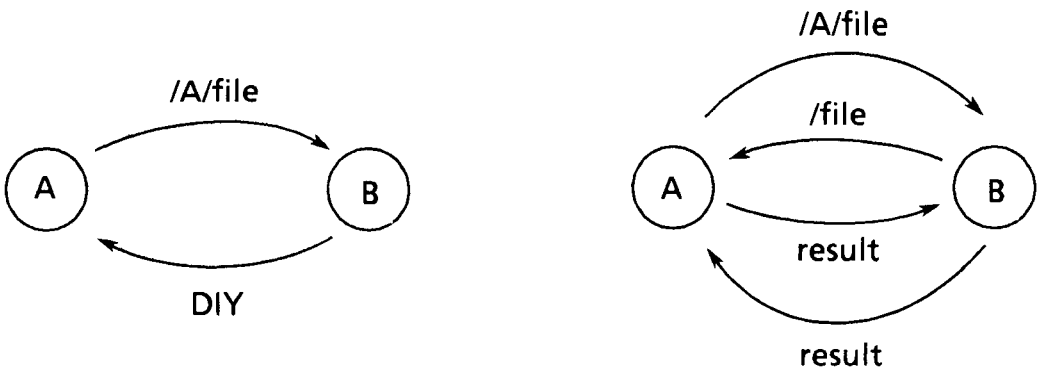
Each *perform* operation enters the name layer on the local system and eventually emerges from this layer on the system on which the object referred to is local. In the meantime, the call will have been moved between systems by the RPC layer according to the results of the *local-resolve* operation performed in the name layer of each system. The recursive and iterative implementations only differ in the routing algorithm used between systems: the iterative algorithm always routes RPCs via the client system using a DIY mechanism, but the recursive algorithm simply allows them to pass freely and directly between systems at each stage of the resolution.

For example, consider performing an operation on an object named `/B/C/file` from system A. The name of the object will be passed to the name layer on A and resolved into the name `/C/file` on system B. The RPC layer will pass this name to B and the name layer at B will further resolve it as the name `/file` at the address C. At this point the difference between the two possible implementations will manifest itself. A recursive implementation of the RPC layer will pass the message on directly to C; an iterative implementation will return a DIY message to the effect "try again with `/file` at address C" to the RPC layer on A.



(a) recursive (b) iterative

Since every RPC must return eventually, no extra inter-system messages or RPCs are caused by the iterative algorithm and for a name that loops back to the local system such as `/B/A/file` fewer RPCs are required. However, instead of



(a) iterative (b) recursive

spreading the sending and receiving of messages evenly between the systems referenced by a given pathname, the local system will take most of the load, generating one RPC for each system in the pathname. Arguably, this is reasonable because it avoids penalising remote systems unnecessarily for the effect of remote operations performed by the local system.

Another advantage of the iterative algorithm is that in the absence of a mechanism for resolving identical names statically it avoids creating more than

one server at each remote system for a given client on the local system, simply because all RPCs and hence server creations are initiated by the local client system. Although it would be possible to avoid this with the recursive algorithm, it would require making every server for a particular client (or at least the RPC layer on each system) aware of every other server for that client so that new servers were only created when absolutely necessary. In fact, because of possible permission problems caused by creating servers indirectly (via other servers), a hybrid algorithm would probably be required, with all servers being created non-recursively from the local client system. The iterative algorithm is a much better solution.

6.4. Other Pathname Algorithms

We have discussed various ways of resolving pathnames and developed an algorithm which combines name resolution and performing remote operations into a single RPC protocol implemented in three layers. We have in fact developed a dynamic mechanism for simplifying pathnames based on the idea of bouncing a name resolution RPC between systems until the most direct path to an object is found. If it were possible to simplify a pathname statically before passing it to the *resolve* algorithm, there would be no need to access systems mentioned in any redundant part of the pathname and less RPCs would be generated. Instead of using a dynamic sequence of RPCs to simplify the name, it could be analysed statically and transformed into a canonical form. It would then be possible to access the remote system denoted by the pathname by the most direct route, without any unnecessary RPCs and without passing through any unnecessary servers.

In practice the overhead of resolving pathnames is reduced by two factors. Firstly, people tend to use pathnames in their simplest form (although computers are not so considerate, so machine-generated pathnames may still be a problem).

If no simplification is possible then nothing will be gained by static analysis. Secondly, because of the overhead associated with pathname resolution, even on a single system not every operation requires a pathname. Instead, it is possible to translate the pathname into a lower level name (effectively a capability) which is simpler to resolve but has purely local and transient significance.

Such a capability is used for the duration of an extended sequence of operations on an object more permanently referred to by a pathname and captures the dynamic state of the computation. Its creation and subsequent destruction mark the beginning and end of this extended sequence. A typical example of such a facility is the concept of opening a file to get a file descriptor which is then used in place of the filename in a series of read or write operations before the file is finally closed and the descriptor is destroyed. Although the widespread use of such descriptors means that in practice pathname resolution is less frequent an operation than might be expected, it is still important to establish the most direct route to the server which holds the descriptor for the remote object and it is therefore worth considering an alternative approach to this simplification.

In section 2.3.4 we discussed a canonical form for Unix pathnames and in section 2.3.5 we showed how . . could be eliminated to give a simple scheme for reducing pathnames to their canonical form. For a suitable naming graph these transformations would provide a useful simplification algorithm which could be applied statically rather than dynamically without accessing remote systems. However, the canonical pathname is only unique if the naming graph is tree-structured and consequently canonicalisation does not guarantee the most direct path for an arbitrary graph, nor does it guarantee a unique path. In order to solve the identity problem we must find a way of discovering whether two pathnames denote the same object.

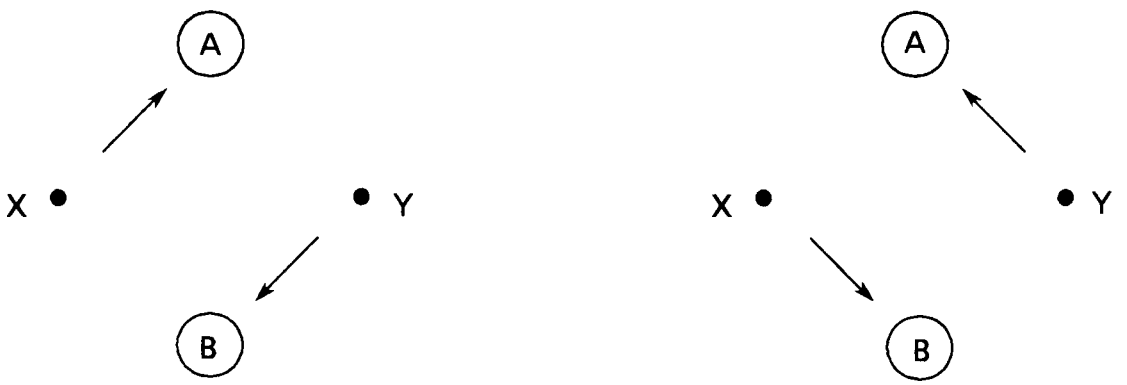
Given unique identifiers the problem is trivial. Each object will have an identity which is guaranteed unique amongst all possible systems. One way of achieving this would be to give identities a hierarchical structure which included a unique system identifier. Mapping a pathname into the identity of the object it denotes might require an RPC but, since each object has only one identity and no two objects share the same identity, it is possible to compare identities directly to determine whether pathnames are equivalent.

Even without a unique system identifier it is still possible to compare pathnames from within a single process (or some other form of localised context) providing there is only one naming path to each remote system. Every time a new remote system is accessed the transparent distribution layer attached to the process can allocate an arbitrary but unique (at least within this context) identifier for that system. This can be used to qualify any internally unique identifiers which are issued by the remote system so that they may be distinguished from identifiers issued by other remote systems. However, such qualified identifiers are only unique relative to a process rather than absolutely unique and are therefore only valid during the lifetime of the process which created them. They may not be published or used by other processes.

However, if the naming structure of the distributed system is a general graph rather than a tree (so that a given system can have more than one name), it will be impossible to tell whether two pathnames denote the same object without introducing unique identifiers for all the systems. Although it is reasonable to assume that individual systems are able to generate their own private sequence of unique identifiers internally (e.g. timestamps), there is nothing to stop two systems from independently generating the same "unique" identifier. Identifiers which are only unique within a localised context cannot be used unambiguously outside that context. Consequently, in order to prove that two pathnames are equivalent, it is not sufficient to derive a system-specific unique identifier from

each of them and compare these for equality, although this test is certainly a necessary condition for equivalence and could therefore be used to prove that two pathnames were not equivalent.

For example, consider an algorithm which relies on the idea of marking the object at the end of one pathname and then checking to see whether the object at the end of the other pathname had been marked in the same way. The mark is an internally unique token generated by the system checking the pathnames for equivalence. However, because such tokens are not globally unique and are used outside the context in which they are unambiguous, the algorithm is vulnerable to an anti-symmetry argument. If system X generates some token T which is used to mark object A before visiting object B, there is nothing to stop system Y from independently generating the same token T simultaneously and using it to mark object B before visiting object A. Both systems would find the second object



(a) mark with T (b) check mark

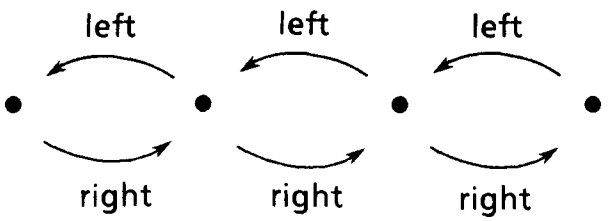
marked with the correct token and would incorrectly deduce that the two objects were equivalent when they were not. Even if the objects added their own qualifying mark to the token T they could still both generate the same qualifier independently.

Admittedly this kind of failure is pathological in the extreme. By choosing the identifying token randomly from a sufficiently large population the probability of

the algorithm failing by accident could be made arbitrarily small. However, the probability of a clash can never be made zero and so what was originally a deterministic problem has now only a probabilistic solution.

It could be argued that without globally unique identifiers the problem we are trying to solve is ill-defined and therefore cannot have a solution. Being able to identify and distinguish individual systems amounts to defining a mapping from each system onto a unique identifier. Without unique identifiers it is not possible to define what is meant by two systems being the same and so it is impossible to construct an algorithm which can do so. However, it is possible to come very close to constructing an algorithm that works by approximating the idea of a unique identifier with a random identifier as we have just seen. Furthermore, for certain name spaces it is possible to determine whether two pathnames are the same from knowledge of the naming graph structure.

For example, consider a graph where every node has two neighbours, `left` and `right`, which are inverses of each other. Although there are no absolute



names, all pathnames may be reduced to a canonical form very easily (so many steps to the left or so many steps to the right) and so two pathnames relative to the same system can be compared for equivalence by reducing them to their canonical form.

It is always possible to introduce unique identifiers by providing a centralised server which resides at a well-known address and supplies guaranteed unique tokens to order. However, this incurs the additional overhead of allocating

identifiers dynamically as needed rather than statically in advance. The identifiers would have to be random and transient in significance because if they had any meaning and were associated with an absolute naming scheme, clients would have to identify themselves to the server in order to get the correct identifier. Furthermore, this is a centralised solution to a distributed problem. Such a server would be critical to the correct functioning of a distributed system which depended on it and there could only be one such server. This would cause problems when two distributed systems, each with their own server, were joined together.

6.5. Summary and Conclusions

In this chapter we have developed an abstract model of a transparently distributed system in terms of two operators, *perform* and *resolve*. This has enabled us to ignore the semantic details of any particular system and instead to concentrate on the mechanisms by which a layer of software providing transparent distribution can intercept operations, decide whether they refer to local or remote objects and redirect them to the appropriate system accordingly. The concept of name resolution, captured by the *resolve* operator, is central to this process.

Although our model is general enough to include an implementation based on a centralised name server, in keeping with our recursive philosophy we have concentrated on a more decentralised approach in which each system has limited knowledge of its immediate neighbours in the naming graph. Name resolution may then proceed recursively by following a name from system to system or iteratively by returning an indication to the calling system that a name is not local together with an indication of a "better" system to try. As demonstrated in the previous chapter, a recursive implementation can lead to semantic difficulties because it is liable to create multiple servers on the same machine and otherwise

confuse the notion of identity. Consequently, an iterative approach is preferable. Iteration will also involve less inter-system messages if a name loops back on itself. However, an iterative algorithm is unable to cope with a name space structured into sub-spaces or domains because it assumes that the naming graph is fully connected and every system knows about every other. Consequently, a hybrid approach is required with iteration used within domains and recursion between domains. However, this requires the domains to be organised in a tree structure in order to guarantee a unique naming path between any two systems.

Once an object has been located by resolving its name, an RPC must be directed to the system where it is to be found in order to perform the required operation on the object. With a decentralised approach to name resolution, it makes sense to combine the *perform* and *resolve* operators into a single RPC protocol so as to avoid sending messages to the same system twice. The same arguments about recursion and iteration apply and the result is an RPC design which uses pathnames as addresses and includes a routing layer based on the idea of DIY, a special exception which indicates that a remote operation should be retried on another system.

The combined *perform* and *resolve* algorithm is effectively a dynamic mechanism for simplifying pathnames by flattening the recursive structure of the name space. The most direct route which an RPC can take through the naming layer to a particular system corresponds to the canonical pathname for that system. In chapter 2 we showed that it was possible to simplify pathnames to their canonical form statically and hence avoid making any RPCs but this is only appropriate for a tree-structured graph. The more general problem of determining whether two pathnames from an arbitrary naming graph are equivalent is basically insoluble without introducing globally unique identifiers.

Although we have been taking for granted the basic hypothesis that recursive structuring is a good thing, this viewpoint has created many of the problems we have had to solve. The fact that these difficulties do exist is an argument against recursive structuring in favour of globally unique identifiers.

Chapter 7
Conclusions

7.1. Summary of Thesis

This thesis has analysed many of the surprisingly subtle naming issues that arise in the construction of transparently distributed systems. In Chapter 1 we argued that distributed systems should be constructed transparently. A transparently distributed system built of existing systems should be functionally equivalent to the systems of which it is composed. Naming is of fundamental importance in achieving this.

In Chapter 2 we showed that the purpose of a naming system was to map high level user names into internal system identifiers. A hierarchical naming structure based on the use of contexts makes it possible to localise portions of the name space and control which names are visible at any one time. It is important to be able to navigate in such a hierarchy and name one context from another. Most systems provide a generic name for the parent context such as . . but we argued that it was better to name the context explicitly. This makes it possible to simplify redundant pathnames automatically without requiring global knowledge of the entire naming graph. If the graph is tree structured then pathnames may be reduced to a unique canonical form which is the most direct route through the tree. Such a canonical pathname may be used to identify an object unambiguously. However, the presence of naming aliases in the form of multiple paths to the same node in a more general graph makes it impossible to derive a unique canonical form. In a general graph some simplification of redundant pathnames is still possible but there is no longer any guarantee that a pathname can be reduced to its simplest form.

Chapter 3 considered the problems of joining hierarchical name spaces together. A naming system maps external names to internal identifiers and name spaces may be joined at either level. This involves resolving naming conflicts. Internal identifers may need to be qualifed with the identity of the name space to which they belong or else be replaced with identifiers which are unique in the wider scope of the combined name spaces. Naming conflicts are not such a problem externally because external names tend to be contextual. However, a mechanism must be found for allowing external names to cross name space boundaries. Ideally this should hide the boundary between name spaces so that names remain location transparent. No restrictions should be imposed on the grouping of names into a given context, irrespective of the location of the objects they denote. Such granularity is difficult to achieve in practice since arbitrary references between name spaces make garbage collection and other forms of integrity checking difficult. Instead, a compromise which exploits locality of reference may be adopted, reducing the granularity of inter-name space references by making whole sub-spaces rather than just individual objects visible through a mount mechanism.

The techniques used to join together name spaces within a single system may also be used to join together whole systems to construct a transparent distributed system. One of the difficulties in achieving full transparency is the presence of an explicit naming context for system objects in the naming graph. It is difficult to preserve individual system contexts in the naming graph of the distributed system as a whole without violating transparency because a single system does not need an explicit mechanism for identifying and distinguishing other systems since there are none. This is more a problem of scale than of distribution because the idea of a unique system context is not recursive or extensible. It would be difficult to administer a large centralised system without a mechanism for dividing it up into subsystems. If it existed, such a mechanism could easily be

extended and used to structure the design of a transparently distributed system but without it some compromise is necessary.

Unix uses the root directory / as both a system naming context and a globally agreed starting point for absolute pathnames. However, the implicit assumption that these two meanings are the same is only valid for a single system and is liable to break down for a transparently distributed system. The various distributed Unix systems described in chapter 3 approached this problem in different ways.

The Newcastle Connection represents systems as remote contexts and groups them together into a new context which has no absolute name in its own right but may only be named relative to an existing system. This approach preserves the identity of individual systems in a shared global naming hierarchy.

An alternative approach adopted by NFS and RFS uses the concept of remote mount to allow individual systems to share parts of their name space with other systems while still retaining a single system view of the world. With this second approach, there is no common view of the naming tree and each system may have a private name space which is not visible to any other system.

Neither of these approaches is completely transparent because in the first case individual systems are still visible and in the second case there is no single view of the distributed name space. However, a third approach adopted by LOCUS subsumes the individual systems entirely into a single virtual system. A distributed system built with LOCUS has only one system naming tree and it is shared by all the components of the distributed system. This is the only approach which is completely transparent but it is essentially flat. It is not possible to join two such virtual systems together in a hierarchy. Instead, they must be flattened into a single system by resolving naming conflicts and renaming objects if

necessary. It would be difficult to administer a large LOCUS system because it would be monolithic with no sub-structure.

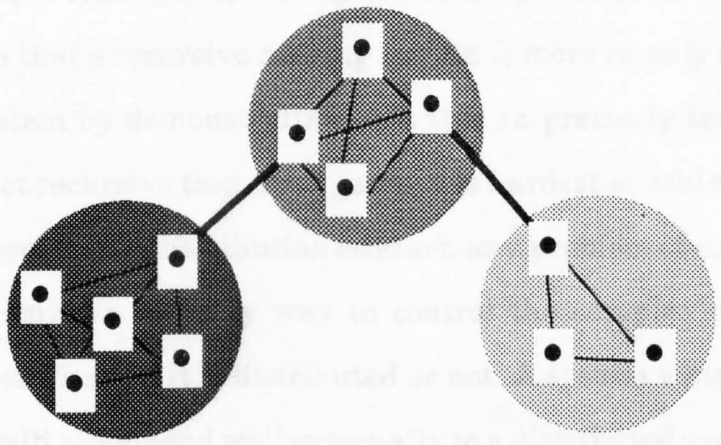
Chapter 3 also contained a detailed discussion of various aspects of the Unix semantics which do not extend easily to a transparently distributed system. Most of these problems arise in areas where Unix is not recursively structured and therefore less suitable as a component of a distributed system. Although the details were specific to Unix, these issues must be tackled in the design of any distributed system regardless of whether it is composed of existing systems or built from scratch.

Chapter 4 examined distributed systems constructed from scratch rather than built out of existing centralised systems. Such systems are usually built assuming the existence of globally unique identifiers, and various techniques for allocating such identifiers and guaranteeing their uniqueness were discussed. In particular, the problem of combining such systems transparently without identifier clashes was examined. The most pragmatic solution appeared to be generating unique identifiers at random, giving only a probabilistic guarantee that no clashes would occur. We argued that although globally unique identifiers offer a theoretical solution, they cannot be relied on in practice, and concluded that the problem of combining whole distributed systems was really no different from that of combining individual systems if distribution is transparent.

Chapter 5 explored the idea of constructing transparent distributed systems recursively by considering the implications this might have for the Newcastle Connection, an implementation of a transparent distributed Unix system. Semantic difficulties arise from the fact that in a recursively structured system based on the notion of localised pathnames the concept of identity becomes confused when it is possible to name objects in more than one way. For the Newcastle Connection, identity is tied up in the concept of a Distributed

Sequential Process (DSP) consisting of a client process on the local machine and a server process on each of the remote systems which the DSP has accessed. We showed how a recursive implementation allows two servers representing the same DSP to be created simultaneously on the same machine and how this crisis of identity is responsible for various semantic problems (related to the notion of identity) where the transparency breaks down and the distributed system is no longer functionally equivalent to the systems of which it is composed. In fact, one aspect of the Newcastle Connection, remote execution, cannot be implemented transparently (and hence recursively) in any case because the server programs need to be aware of any servers they may have acquired for themselves, contrary to the notion of transparency. Other distributed Unix systems have either not tackled this problem or else have relied on globally unique system identifiers to avoid the difficulty.

Chapter 6 left Unix behind and examined the idea of recursive transparency in more abstract terms. A distributed system must *resolve* names to locate objects and then *perform* operations on those objects. The implications of combining *resolve* and *perform* into a single operation, in effect an RPC based on pathname routing, were explored, first for a flat system structure and then for a recursively structured distributed system. It was shown that it is only possible to guarantee that a given system can be reached by one naming path (hence identifying the client system uniquely) if the overall naming graph remains tree structured. This restriction can be lifted slightly by grouping systems into naming domains and allowing each system to have limited knowledge about the global graph structure in the form of information about its enclosing naming domain, provided that an overall tree structure is retained between naming domains. Without a way of giving systems a global identification, guaranteed unique amongst all possible systems, it is not in general possible to tell whether two pathnames in an arbitrary naming graph denote the same object or not and hence the crisis of



identity remains, although once again a pragmatic solution based on random identifiers is possible. This is an argument against recursive structuring in favour of globally unique identifiers.

Bringing together some of the ideas discussed in this thesis we see that by using globally unique identifiers it is certainly possible to recognise pathnames which denote the same object because there is a proper notion of identity. However, except in those special circumstances where uniqueness can truly be guaranteed across all possible systems, it is not possible to join two distributed systems constructed with unique identifiers together to form a single system and still guarantee that all identifiers are unique. Consequently, the problems of identifying individual name spaces and mapping from one space of unique identifiers to another must still be tackled. But putting a limit on uniqueness amounts to introducing contextual names. This in effect opens up Pandora's box and introduces the whole range of problems discussed in chapters 5 and 6.

7.2. Contributions of Thesis

Naming is of fundamental importance in the construction of a transparently distributed system. If a distributed system is built out of existing systems then the naming characteristics of the component systems will determine to what extent the composite system is functionally equivalent to the systems of which it

is composed, in other words the degree of transparency which can be achieved. We have shown that a recursive naming system is more readily extensible than a flat naming system by demonstrating that it is in precisely those areas in which a system is not recursive that transparency is hardest to achieve. In fact, this is not so much a problem of distribution so much as a problem of scale. The introduction of sub-structure is the only way to control the complexity of a large system regardless of whether it is distributed or not. A system which does not scale well internally will not extend well externally to a distributed system.

Naming is inextricably linked with the notion of identity. In any system it is vital to be able to identify objects uniquely and unambiguously. It is usually possible to translate a name into a unique identifer. However, such a facility is not recursive because unstructured identifiers are only unique within their defining context. The obvious solution to this problem is to identify objects with extensible sequences of unique identifiers with one component for each level of the hierarchy. This is in effect a pathname. However, such an identifier can only be guaranteed unique if the naming graph of the overall system is tree-structured. Although this would be a natural consequence of a distributed system being genuinely constructed recursively out of existing systems, in practice this is usually not realistic. When systems are joined together their naming graphs are connected in several places and the overall structure is not a tree. Naming facilities such as aliases or links (multiple names for the same object) could not be provided in a pure tree-structure.

We have shown that within a tree-structured graph it is possible to use the concept of a canonical path to determine whether two pathnames are equivalent statically. However, for a more general naming graph and a distributed system with a high level of parallelism it is impossible to tell whether two pathnames are equivalent without introducing globally unique identifiers or some other form of synchronisation. For a very large scale distributed system, global uniqueness

would be difficult if not impossible to achieve in practice and consequently it is not reasonable to expect a deterministic solution to this problem for such a system. Probabilistic algorithms which use random numbers to approximate globally unique identifiers are an attractive alternative because their behaviour is predictable and the probability of an error can be made arbitrarily small.

The problem of name resolution in a distributed system amounts to simplifying an arbitrary pathname dynamically using limited contextual information. The most natural implementation of such an algorithm is recursive but this will not work correctly if the pathname is redundant, compromising the notion of identity and leading to various semantic difficulties which violate the transparency of the distributed system. Although it is possible to flatten the recursion somewhat by partitioning the name space into sub-domains, a recursive algorithm will still be required between domains and this will lead to the same difficulties if the partitioning is not tree-structured.

Although structured identifiers offer a solution to the problem of resolving name clashes when name spaces are combined, they only work correctly under the unrealistic and restrictive assumption that the overall name space is tree-structured. Universally unique identifiers are also unrealistic because they require centralised coordination and are prone to human error unless guaranteed unique by hardware. The only pragmatic approach is to rely on random identifiers which offer no more than a probabilistic guarantee that naming and identity will not be compromised.

The table on the next page illustrates the various techniques for identifying objects discussed by this thesis and summarises their relative advantages and disadvantages.

In conclusion, the main contribution of this thesis has been to examine the difficult problem of joining name spaces together without accepting the easy

solution of assuming unique identifiers (which on closer inspection poses severe management problems). Very little work appears to have been done in this area.

Technique	Advantages	Disadvantages
canonical pathnames	allow static name resolution	requires rigid tree structure and cannot cope with aliases or ..
unique identifiers	work well for a single naming domain	flat structure clashes when combining name spaces
globally unique identifiers	universal panacea	unrealistic for large systems requiring centralised coordination and rigidly enforced uniqueness
recursive name lookup	natural solution for recursive name system	cannot handle redundant names without confusing notion of identity
iterative name lookup (DIY)	solves identity problem for single naming domain	cannot cope with multiple domains
combined recursive and interative approach	best deterministic solution possible without requiring global uniqueness	domains must be joined in a pure tree structure
random identifiers	good approximation to global uniqueness without management problems	no longer deterministic but probablility of error can be made arbitrarily small provided an upper bound can be placed on the size of the system

7.3. Future Work

Although this thesis does not include detailed descriptions of any implementation work, the insights it contains are based on extensive practical work with the Newcastle Connection. Specifically the author was responsible for the first implementation of the NC inside the Unix kernel and indeed the first port of the NC code to another machine. He has also participated in much of the development work, especially in the area of interworking with heterogeneous implementations of Unix. Only the detailed knowledge thus gained about the practical problems of implementing transparently distributed systems has made it possible to write this thesis.

Given this experience with the Newcastle Connection, an obvious direction for future work would be to incorporate the DIY mechanisms discussed in chapter 6 into the RPC protocol used by the NC. This would make it possible to provide the functionality of connected servers required for full transparency without compromising Unix semantics by confusing the notion of identity. It would also address the problem of constructing a large distributed Unix system out of many independent naming domains. However, it is not clear that the effort involved in implementing the extra functionality, particularly in the area of remote execution, would be justified by the use made of such a system.

Another possibility would be to monitor a real Unix system and determine how the various Unix naming facilities are used in practice. For example, by examining typical directory structures and the use of context-dependent pathnames it would be possible to predict whether it would be useful to introduce an alternative naming mechanism such as a closure or eliminate . . and use canonical pathnames instead. Similarly, statistics collected about pathname resolution could be used to determine the impact that introducing DIY mechanisms would have on the performance of a distributed Unix system. Some

work already done in this area [Floyd87] indicates that pathnames tend to be quite dense and name resolution is an expensive operation, making some sort of caching essential for performance reasons.

The impact of caching on the design of a transparently distributed system is also worth exploring. One of the difficulties with adding distribution to a system transparently is picking an appropriate level at which to intercept operations on objects. Since the layer of software inserted at this point must effectively simulate the name resolution alogorithms of the underlying system, it is important to minimise the extent to which existing mechanisms are duplicated. However, despite "end-to-end" arguments [Saltzer84], optimisations such as caching tend to be applied at the lowest level of the system but distribution is added at a higher level to maximise the functionality which is captured by the transparent layer. This trade-off deserves more investigation.

One way of exploring these issues would be to develop from scratch a more recursively extensible system based on the knowledge gained in this thesis of the limitations of Unix. This would address such areas as user and process identification whilst avoiding the problems caused by making low-level identifiers visible. Such a system might use random numbers to solve some of the identity problems caused by the use of local identifiers in a large distributed system. However, one practical problem in evaluating such a system would be that its particular advantages would only become apparent if it was deployed on a grand scale on top of many computers in an environment containing many sets of users and many system administrators. This would not come about unless the system was clearly better than Unix and offered more than just recursive extensibility. Even then it is not clear that a system as pervasive as Unix (which was once described as the "Fortran of Operating Systems") could ever be replaced on such a scale. Certainly, the implementation effort and political intrigue required would be very considerable. Unfortunately, research into the scaling

properties and management difficulties of very large distributed systems requires the resources of a multinational organisation and a leisurely timescale.

A more realistic line of research would be to develop a formal model of naming which treated names as first-class objects and captured the notion of passing names between contexts. Treating names as typed objects makes it possible to hide their internal representation and prevent them from being forged or passed around between contexts by ad-hoc means. The Flex system contains several interesting ideas such as remote capabilities and closures which offer promising directions for future work. Basing such a model on Flex or a similar programming environment would demonstrate that the ideas in this thesis are not just applicable to the design of distributed operating systems but are also useful in the design of other types of system too. It might be more realistic to construct a large-scale experiment with an IPSE than with an operating system.

7.4. Concluding Remarks

Given the opportunity, the domain of a naming space based on globally unique identifiers may be enlarged by adding an extra level of hierarchy. This solution has been adopted by the telephone network on several occasions but it requires the ability to recognise old names and prevent them from being used out of context. This could involve a major effort in redesigning the system and it is perhaps more realistic to assume that this problem will arise from the start, not just once but possibly indefinitely, and therefore to base the system design on localised contextual names which will scale more easily. However, as we have seen, these introduce their own problems, and in particular the problem of identity. We are left with the unhappy conclusion that

> "Global identifiers apparently work but don't scale,
> Local identifiers scale, but apparently don't work."

References

[AT&T85]

AT&T, *The System V Interface Definition*, AT&T, Indianapolis, USA, Spring 1985.

[Barak86]

A. Barak, D. Malki, and R. Wheeler, "AFS, BFS, CFS... or Distributed File Systems for Unix", *Proceedings EUUG Autumn Conference 1986*, pp. 461-472, Manchester, UK, September 1986.

[Birrell82]

A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder, "Grapevine: An Exercise in Distributed Computing", *Communications of the ACM*, vol. 25, no. 4, pp. 260-274, April 1982. Also Xerox PARC CSL-82-4

[Black86]

A. Black, N. Hutchinson, E. Jul, and H. Levy, "Object Structure in the Emerald System", *Proceedings OOPSLA'86 in ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 78-86, Portland, USA, November 1986.

[Brownbridge82]

D. R. Brownbridge, L. F. Marshall, and B. Randell, "The Newcastle Connection - or UNIXes of the World Unite", *Software Practice and Experience*, vol. 12, no. 12, pp. 1147-1162, December 1982.

[Brownbridge84]

D. R. Brownbridge, "Recursive Structures in Computer Systems", PhD Thesis, Computing Laboratory, University of Newcastle upon Tyne, September 1984.

[Cerf83]

V. G. Cerf and E. Cain, "The DoD Internet Architecture Model", *Computer Networks*, vol. 7, pp. 307-318, North Holland, 1983.

[Cheriton84a]

D. R. Cheriton, "The V Kernel: A Software Base for Distributed Systems", *IEEE Software*, pp. 19-42, April 1984.

[Cheriton84b]

D. R. Cheriton and T. Mann, "Uniform Access to Distributed Name Interpretation", *Proceedings 4th International Conference on Distributed Computing Systems*, May 1984.

[Codd79]

E. F. Codd, "Extending the Database Model to Capture More Meaning", *ACM Transactions on Database Systems*, vol. 4, no. 4, December 1979.

[Copeland86]

G. P. Copeland and S. N. Khoshafian, "Object Identity", *Proceedings OOPSLA '86 in ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 406-416, Portland, USA, November 1986.

[DEC81]

DEC, Intel, and Xerox, "The Ethernet, a Local Area Network: Data Link Layer and Physical Layer Specifications - Version 1.0", *ACM Computer Communication Review*, vol. 11, no. 3, pp. 20-66, July 1981.

[Dalal81]

Y. K. Dalal and R. S. Printis, "48-bit Absolute Internet and Ethernet Host Numbers", *Proceedings 7th Data Communications Conference*, October 1981.

[Decouchant86]

D. Decouchant, "Design of a Distributed Object Manager for the Smalltalk-80 System", *Proceedings OOPSLA'86 in ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 444-452, Portland, USA, November 1986.

[Floyd87]

R. Floyd and C. S. Ellis, "The High Cost of Opens in the UNIX Environment", CS-1987-5, Department of Computer Science, Duke University, Durham, North Carolina, USA, February 1987.

[Foster82]

J. M. Foster, I. F. Currie, and P. W. Edwards, "Flex: A Working Computer with an Architecture based on Procedure Values", *Proceedings International Workshop on High-Level Architectures*, pp. 181-185, Fort Lauderdale, Florida, December 1982.

[Foster86]

J. M. Foster and I. F. Currie, "Remote Capabilities in Computer Networks", RSRE Memorandum No. 3947, RSRE Malvern, UK, March 1986.

[Fraser-Campbell86]

W. Fraser-Campbell and M. B. Rosen, "An Implementation of NFS under System V.2", *Proceedings EUUG Conference Spring 1986*, Florence, Italy, April 1986.

[Guillemont82]

M. Guillemont, "The CHORUS Distributed Operating System: Design and Implementation", *ACM International Symposium on Local Computer Networks*, pp. 207-223, Florence, Italy, April 1982.

[Hall85]

J. A. Hall, P. Hitchcock, and R. Took, "An Overview of the Aspect Architecture", in *Integrated Project Support Environments*, ed. J. McDermid, pp. 86-99, IEE, 1985.

[Herbert87]

A. Herbert, and J. Monk, *ANSA Reference Manual*, ANSA, Cambridge, UK, June 1987.

[Kaehler86]

E. Kaehler, "Virtual Memory on a Narrow Machine for an Object-Oriented Language", *Proceedings OOPSLA'86 in ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 87-106, Portland, USA, November 1986.

[Lampson86]

B. W. Lampson, "Designing a Global Name Service", *Proceedings 5th ACM Symposium on Principles of Distributed Computing*, pp.1-10, Calgary, Canada, August 1986.

[Landin64]

P. J. Landin, "The Mechanical Evaluation of Expressions", *Computer Journal*, vol. 6, no. 4, pp. 308-320, January 1964.

[Lantz85]

K. A. Lantz, J. L. Edighoffer, and B. L. Hitson, "Towards a Universal Directory Service", STAN-CS-85-1086, Department of Computer Science, Stanford University, Stanford, California, August 1985.

[Leach82]

P. J. Leach, B. L. Stumpf, J. A. Hamilton, and P. H. Levine, "UIDS as Internal Names in a Distributed File System", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 34-41, Ottawa, Canada, August 1982.

[Marshall86]

L. F. Marshall and R. J. Stroud, "Remote File Systems Are Not Enough!", *Proceedings EUUG Autumn Conference 1986*, pp. 93-99, Manchester, UK, September 1986.

[Mockapetris83]

P. Mockapetris, "Domain Names - Concepts and Facilities", RFC 882, November 1983.

[Mullender85]

S. J. Mullender, "Principles of Distributed Operating System Design", SMC, Amsterdam, October 1985. PhD Thesis

[Nowitz78]

D. A. Nowitz and M. E. Lesk, "A Dial-Up network of UNIX Systems", in *UNIX Programmers Manual V7*, August 1978.

[Oppen81]

D. C. Oppen and Y. K. Dalal, "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment", Xerox Office Products Division, Palo Alto, 1981.

[Randell83]

B. Randell, "Recursively Structured Distributed Computer Systems", *Proceedings 3rd Symposium on Reliability on Distributed Software and Database Systems*, pp. 3-11, IEEE, October 1983.

[Rashid81]

R. F. Rashid and G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel", *Proceedings 8th ACM Symposium on Operating Systems Principles*, pp. 64-75, Asylomar, 1981. (ACM Operating Systems Review, vol. 15, no. 5)

[Renesse86]

R. van Renesse and J. M. van Staveren, "Wide-Area Communication under Amoeba", Internal Report IR 117, Department of Mathematics and Computer Science, Vrije University, Amsterdam, December 1986.

[Rifkin86]

A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah, and K. Yueh, "RFS Architectural Overview", *Proceedings Usenix Conference Summer 1986*, pp. 248-259, Atlanta, USA, 1986.

[Ritchie74]

D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", *Communications of the ACM*, vol. 17, no. 7, pp. 365-375, July 1974.

[Saltzer78]

J. H. Saltzer, "Naming and Binding of Objects", in *Operating Systems, An Advanced Course*, ed. R. Bayer, R. M. Graham, G. Seegmuller, Springer, 1978. (Lecture Notes in Computer Science, vol. 60)

[Saltzer84]

J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-To-End Arguments in System Design", *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277-288, November 1984.

[Sandberg86]

R. Sandberg, "The Sun Network Filesystem: Design, Implementation and Experience", *Proceedings EUUG Conference Spring 1986*, Florence, Italy, April 1986.

[Schroeder84]

M. D. Schroeder, A. D. Birrell, and R. M. Needham, "Experience With Grapevine: The Growth of a Distributed System", *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 3-23, February 1984.

[Shoch78]

J. F. Shoch, "Internetwork Naming, Addressing and Routing", *Proceedings 17th IEEE Computer Society International Conference (COMPCON)*, pp. 72-79, September 1978.

[Sollins85]

K. R. Sollins, "Distributed Name Management", PhD Thesis MIT/LCS/TR-331, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 1985.

[Stroud86]

R. J. Stroud, "Beyond Unix", Internal Report SRM/409, Computing Laboratory, University of Newcastle upon Tyne, May 1986.

[Tanenbaum85]

A. S. Tanenbaum and R. van Renesse, "Distributed Operating Systems", *ACM Computing Surveys*, vol. 17, no. 4, December 1985.

[Tanenbaum86]

A. S. Tanenbaum and S. J. Mullender, "The Design of a Capability-Based Distributed Operating System", *Computer Journal*, vol. 29, no. 4, pp. 289-300, 1986.

[Terry85]

D. B.. Terry, "Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments", PhD Thesis, Xerox Palo Alto Research Centre, CSL-85-1, February 1985.

[Vandome86]

G. Vandome, "Comparative Study of some UNIX Distributed File Systems", *Proceedings EUUG Autumn Conference 1986*, pp. 73-82, Manchester, UK, September 1986.

[Walker83]

B. Walker, G. J. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System", *Proceedings 9th ACM Symposium on Operating System Principles*, pp. 49-70, Bretton Woods, New Hampshire, October 1983.

[Watson81]

R. W. Watson, "Identifiers (Naming) in Distributed Systems", in *Distributed Systems - Architecture and Implementation*, pp. 191-210, Springer-Verlag, New York, 1981. (Lecture Notes in Computer Science, vol. 105, Chapter 9.)

[Weinberger86]

P. J. Weinberger, "The 8th Edition Network File System", *Proceedings EUUG Conference Spring 1986*, Florence, Italy, April 1986.

[Williamson87]

R. Williamson, "The SNA/OSI Gateway", Systems Research Group Seminar, Computing Laboratory, University of Newcastle upon Tyne, February 1987.

[Wupit83]

A. Wupit, "Comparison of UNIX network systems", *Proceedings 6th SIGSMALL, ACM*, San Diego, USA, December 1983.

[Xerox81]

Xerox, "Internet Transport Standard", XSIS 028112, Stamford, Connecticut, USA, December 1981.

[Xerox85]

Xerox, "Viewpoint Series Reference Library", 600P88821, Rank Xerox, UK, 1985.

[Zimmerman80]

H. Zimmerman, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection", *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425-432, April 1980.