An Experimental Implementation of the Tilde Naming System

Douglas Comer Purdue University
Ralph E. Droms Bucknell University
Thomas P. Murtagh Williams College

ABSTRACT: The Tilde naming system identifies files in a distributed computing system in a novel way, providing a consistent mechanism for referencing both local and remote files that is independent of the details of the underlying computing environment. The Tilde naming system achieves this independence by providing each user with control of a private name evaluation environment. In contrast, several contemporary distributed systems try to disassociate file names from the structure of the computing environment by hiding complex, heterogeneous computing systems beneath a uniform, global name evaluation environment. The private naming environments supported by Tilde naming incorporate local and remote names into a single naming mechanism while improving software portability and retaining the familiar advantages of hierarchical file naming systems.

We have constructed an experimental implementation of the Tilde naming system, and created a computing environment sufficiently rich to support significant software development. With this prototype, we have been able to study the effects of the

Tilde naming scheme on the system's user interface, software project development, information sharing and other issues. This paper summarizes Tilde naming and discusses insights into naming mechanisms gained through the use of the experimental system.

---

1. Introduction

One important element of the UNIX computing environment is its hierarchical file naming mechanism [Quarterman et al. 1986; Thompson 1978], which gives the user the ability to group related files under a single directory, and allows the identification of files relative to specific locations within the naming hierarchy. There are, however, problems with UNIX file naming — primarily with scaling and in distributed systems. The file naming mechanism in UNIX interprets names through a single, global naming hierarchy shared by all users. As UNIX file systems are extended to distributed computing environments, it becomes increasingly difficult to address the issues of scale raised in both the implementation and administration of such a naming environment. We refer to such a globally shared naming environment as an *absolute* naming mechanism, because each file name has an interpretation that is independent of the machine, user or process that uses it.

The alternative to absolute file names is *relative* naming. Many contemporary operating systems, including UNIX, already incorporate forms of relative naming. The UNIX current working directory, the UNIX C-shell command interpreter's "*~userid*" mechanism [4.2 Manual 1983], VAX/VMS device aliases [DEC 1982] and the VM/SP CMS mini-disk search order [IBM 1986] are all examples of relative file naming mechanisms. However, in most systems these and other relative naming mechanisms are included as conveniences for the user, while an underlying absolute file naming system is considered the primary name evaluation mechanism.

It is possible to build a file naming system in which *relative* names provide the primary naming mechanism. The Tilde naming system [Comer and Droms 1985; Comer et al. 1989; Comer and Murtagh 1986; Droms 1986] developed as part of the TILDE project [Comer 1984] is just such a relative naming system. The Tilde naming system is based on a collection of small, disjoint, hierarchical namespaces, which replace the single, global namespace of UNIX and UNIX-like systems.

There are two primary motivations for this fundamental difference between Tilde naming and more familiar naming mechanisms. First, the local naming environment and name evaluation mechanism remove the constraint that a file's name be unique across the entire distributed system. Allowing relative names eliminates many of the obstacles faced in scaling the system to a large distributed environment without introducing location dependencies into file names. Second, the organization of files into disjoint namespaces that can be identified by local names provides a valuable style of abstraction for the management of large software subsystems. Thus, our new naming mechanism provides identification of local and remote files independent of the structure of the distributed computing environment, while enhancing software system modularity and portability.

The primary penalty associated with Tilde naming is the added burden to the user of managing a local naming environment. To evaluate methods for minimizing the impact of name management, and to explore the effect of Tilde naming on file naming and software portability, we have constructed an experimental implementation of the Tilde naming system. This paper concentrates on our experiences with the experimental system.

We have organized the remainder of this paper into three parts. The next two sections reviews the characteristics of the naming mechanisms in several contemporary distributed file systems and summarizes the Tilde naming system. Section 4.1 describes the Tilde naming prototype system. The third part of the paper, sections 5 and 6, illustrates the use of Tilde naming mechanisms, discusses insights into relative naming gained through the use of the experimental system, and presents future directions for this research.

2. *Background*

Many distributed computing systems rely on shared access to files as the primary form of persistent shared data. The majority of these systems, including the Newcastle Connection [Brombridge et al. 1982], IBIS [Tichy and Ruan 1984], Locus [Halker et al. 1993], Athena [Saltzer 1987] Vice/Virtue (Andrew) [Satyanarayanan et al. 1985], and Sprite [Welch and Ousterhout 1986] use an *absolute* file naming mechanism, in which all file are organized into a single, globally shared namespace.

Ideally, the naming service in a distributed computing system should provide location independent naming and scale to large systems. Location independence is important to the users of a computing system, in that increased location independence implies increased flexibility and decreases dependency on the specific architecture of the underlying system. To the extent that a system does not exhibit location independence, users must be conscious of changes in the computing environment, and software systems must be altered in response to changes in the computing system itself. Location independence involves two components: storage site independence — avoiding file names that depend, explicitly or implicitly, upon the physical location at which a file is stored — and evaluation site independence — ensuring that the interpretation of a file name is independent of the processor on which it is issued. The rapid growth of large, heterogeneous networks that can span administrative and geographic boundaries motivates the need for distributed file systems that can grow to include thousands of computers in distinct administrative domains.

Systems such as Locus, Athena, Andrew and Sprite all attempt to extend the UNIX model of a single, global, absolute file naming environment to a distributed computing system. These naming systems hide the distributed nature of the underlying computing system by disassociating a file's name from its location. The price for this abstract view of the computing environment is that globally shared naming contexts do not scale well. The implementation of a globally shared naming environment can become unwieldy when extended to large networks. In addition, the more

difficult administrative problem of sharing a single naming hierarchy among computers managed by separate administrative structures becomes unworkable in systems that cross administrative boundaries.

IBIS and the Newcastle Connection explicitly include information location in a file name, by including in the file's name the name of the site at which the file is stored. Such a scheme scales easily to a large system. When such techniques are employed, however, the user must know explicitly where a file is located before the file can be identified. If the file is moved to a new storage site, all references to that file's name must be modified to reflect its new location.

The alternative to absolute file naming is relative naming, in which the interpretation of file names is controlled by the user. As an example, the Network File System (NFS) uses a relative naming scheme, where file names are evaluated in an environment established at each processor. This naming context is tied to the processor, however, so that processes executed on remote hosts use a (possibly) different naming environment, which is likely to yield unexpected results.

The other major proposal for a relative file naming system is called QuickSilver. Both naming systems support forms of relative naming in which names are interpreted relative to an environment associated with the user rather than the processor. In their work, the designers of the QuickSilver file system have emphasized issues related to the efficient implementation of relative naming. We have devoted less effort to implementation issues and more to aspects of the naming system that determine the impact on users of the replacement of absolute naming by relative naming. Section 3. includes a brief discussion of the major differences between Tilde and QuickSilver. Cabrera and Wyllie give a complete description of the QuickSilver file naming system [1987] The Tilde naming scheme is discussed in detail in the remainder of this paper.

3. The Tilde Naming System

In the Tilde naming system each process establishes and manages a relative naming environment called a *Tilde forest*. Based on the way in which the UNIX command processor runs programs for the user (as we will explain in more detail in section 4.1) the Tilde forest can be thought of as a per-user naming environment. The Tilde forest is composed of disjoint naming hierarchies known as *Tilde trees*. In a hierarchical file system, users often structure their directories in such a way that all the files associated with a particular activity or project can be found in one subtree of the system's directory hierarchy, which we refer to as a *project subtree*. Our intent in partitioning the naming hierarchy is that each Tilde tree will correspond to one project subtree. Nothing in the design of the Tilde naming system enforces the organization of Tilde trees as project subtrees. Rather, the mechanism is designed to encourage and support the collection of the components of a software subsystem into a single Tilde tree, just as directories in a hierarchical file system encourage the organization of related files into directories.

A Tilde forest represents the entire file naming environment for a process. That is, a process can only access files in the Tilde trees within its Tilde forest. File names are of the form *~tree/path* where *~tree* selects a Tilde tree, and *path* gives a complete path from the root of the Tilde tree to the file itself. The Tilde naming system also supports the resolution of file names relative to a current working directory, which are resolved in the same way as UNIX relative file names. The first component of a file name, the *Tilde name*, is interpreted relative to the set of trees in the Tilde forest. The process chooses its own Tilde names as identifiers for the roots of the Tilde trees in its Tilde forest.

Each Tilde tree also has an unique, absolute name by which it can be identified independent of any user's Tilde forest. The name resolution mechanism identifies a Tilde tree by this unique name, which we call a *Medusa name*.[1] A Tilde forest is represented by a list of bindings from Tilde Names to Medusa

1. So named because Medusa names are so ugly that the user should not see them.

names, so that a file's Tilde Name is first resolved to a Medusa name, which the resolution mechanism then uses to locate the tree itself.

In our model of a distributed system, each user establishes computing sessions on processors belonging to a local network of processors. This local network is in turn connected to a more loosely integrated internet of processors. The network connections between processors within a local network will generally be more reliable and will provide higher transmission rates than the connections available to the other machines on the network. More importantly, the machines within a local network will generally be operated by a single organization or by closely cooperating organizations. Our goal is to provide completely location transparent access to files within the local network and to provide access to remote files using mechanisms identical to those used within the local network. To accomplish this, each Tilde tree's Medusa name must be independent of the tree's location within the local network but may depend on the local network within which the tree is stored.

Figure 1 shows an example of a process and its Tilde forest. In the figure, the Tilde forest extends across the local network to Tilde trees residing on several remote nodes. The naming environment provided by the Tilde forest is, therefore, not limited to a single node, but can span the entire network.

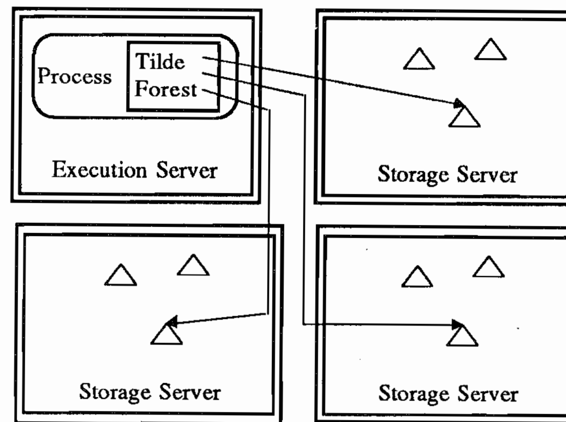

Figure 1: A process and its Tilde Forest

Figure 2 shows the internal structure of a identified tree by the Tilde name ˜system, and illustrates the resolution of the file name ˜system/cmd/ls. The process shown in the figure has three trees in its Tilde forest. The resolution begins by parsing the file name into two components, ˜system and /cmd/ls. The Tilde tree named by the component ˜system is identified by the entry in the Tilde forest that maps ˜system to the root of the tree. The remaining component of the file name, /cmd/ls, then identifies the desired file within the Tilde tree through the file name resolution mechanism.

There are two key differences between Tilde naming as described above and QuickSilver. First, the user controlled naming environment in our system, the Tilde forest, is a run-time structure associated with each process while the closest corresponding entity in QuickSilver, the user index, is a persistent entity stored in the file system. Our choice of this more dynamic form for the user's file naming environment was motivated by the desire to allow the user to take full advantage of relative naming by creating transient naming environments where appropriate. Examples of situations where this flexibility have proved useful are included in Section 5.

The other major difference between the two systems centers around our use of the notion of a project subtree as supported by the Tilde tree. The Tilde tree serves several purposes in the design of Tilde naming. First, it represents a recognition of the fact that it is sometimes inappropriate to associate file names with

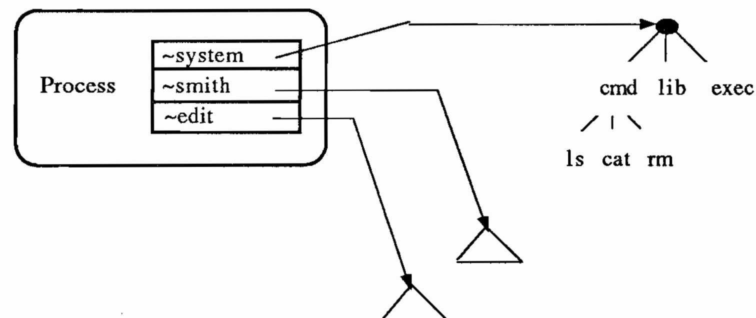

Figure 2: Tilde Name Resolution Mechanism

users. If a group of files together represent some independent software system or other project, it is appropriate to recognize that collection as an independent object in the system. The Tilde tree makes this possible, at least from the point of view of naming. The absolute name system underlying QuickSilver uses pairs composed of a user name and a path relative to that user's naming environment as file names. Every file's name is associated with some user. As a result, while QuickSilver is designed to gracefully handle situations in which a user moves from one domain in the system to another, it runs into problems if a user simply leaves. Subtrees of a user's namespace that were of interest to other users become orphans when their owner departs. If such a tree is to remain available, it must be migrated to some other user's private tree. All users of the migrated subtree must then update their private namespaces to reflect the new location of the subtree.

More importantly, the Tilde tree provides a means to control naming conflicts between software systems by localizing embedded, self-referential file names in a way that makes no assumptions about the organization of a user's naming environment. Consider what may occur if users are allowed as they are in QuickSilver to make arbitrary binding between file names within their environment and files. Suppose that one user constructs some useful program in the directory `/utilities/src/useful` and its subdirectories. Further, assume that like many complex programs this program makes reference to data files stored in `/utilities/src/useful`. Now, suppose that some other user wishes to include the program in his namespace. It is not sufficient to simply make a link to the program itself. Instead a link must be made to the directory `/utilities/src/useful` using a path identical to that chosen by the creator. Even if the second user likes to keep source materials in a directory separate from `/utilities` or prefers some name other than utilities (such as `/bin`) for his directory of executables, he must include the directories `/utilities` and `/utilities/src` in his namespace. It is not hard to imagine that users will quickly find their namespaces crowded with extra directories included simply to ensure that their namespaces mimic those of other users with whom they wish to share files. By organizing software into Tilde

trees, all the components of a software system are incorporated into the namespace in the Tilde forest, independent of the rest of the user's namespace.

4. A Prototype Implementation of the Tilde Naming System

We have constructed a prototype implementation of the Tilde naming system, based on the Berkeley Software Distribution (BSD) of the UNIX operating system and NFS. The prototype runs on a network of VAX computers and Sun workstations. The goal of the prototype was to gain insight and experience with distributed naming within a computing environment capable of supporting the typical edit-compile-test program development cycle. In addition to implementing support of Tilde naming in the modified UNIX kernel, we converted a subset of UNIX commands to work under the experimental Tilde environment.

The prototype includes several main components:

Modified UNIX kernel:
: The Tilde system functions, including simulation of Tilde trees, interpretation of Tilde file names and management of Tilde forests are implemented through changes to the UNIX kernel.

Modified command interpreter:
: The user interface to the Tilde naming system is incorporated into the UNIX command interpreter, *csh* [4.2 Manual 1983] along with other modifications to convert *csh* to use the Tilde file naming scheme.

Modified UNIX utilities:
: The directories containing the executables of a subset of the UNIX utilities together with associated source code, documentation and other ancillary files were reorganized into project subtrees stored as Tilde trees. Thus, users of our prototype operate within an environment that includes UNIX compilers, editors, debuggers, utilities and other software subsystems to support significant software development. This process required no modifications to these

utilities except for the substitution of Tilde system names for embedded UNIX file system names.

The key problems in the development of the Tilde prototype included the implementation of Tilde trees and the Tilde forest, the interpretation of Tilde file names, the use of NFS as a remote file access mechanism for Tilde tree access and the implementation of the user interface to Tilde naming. We will concentrate on interpretation of file names and the management of the Tilde naming environment in this paper; other details of the prototype implementation are discussed in "Naming of Files in Distributed Systems" [Droms 1986].

4.1. File Name Resolution

The resolution of a file name in the Tilde system is performed in three steps. First, the Tilde tree component of the name involved must be resolved into the Medusa name for the tree. Next, the system must locate the tree identified by the Medusa name within the system. Finally, the system interprets the remainder of the path relative to the tree's root as in a conventional hierarchical file system.

Our prototype implementation of Tilde naming is layered on top of a standard UNIX file system including support for Sun's NFS protocol [SUN 1989] The Tilde name evaluation mechanism is independent of the data transport mechanism, so that an implementation of Tilde naming can be built on top of any distributed file access mechanism. The current implementation is based on the mounting of remote directories and could therefore use any of the systems such as AFS [Morris et al. 1986] RFS [Rifkin et al. 1986] or NFS that provide this mechanism. We chose NFS for the prototype because of the authors' familiarity with NFS and the availability of source code for the hardware and software used in the development of the prototype.

Our implementation uses the facilities of the UNIX file system and NFS to accomplish the last two steps in the resolution of a Tilde name. To illustrate this process, consider the resolution of the file name `~system/cmd/ls` in the context of the process whose Tilde naming environment is shown in figure 3.

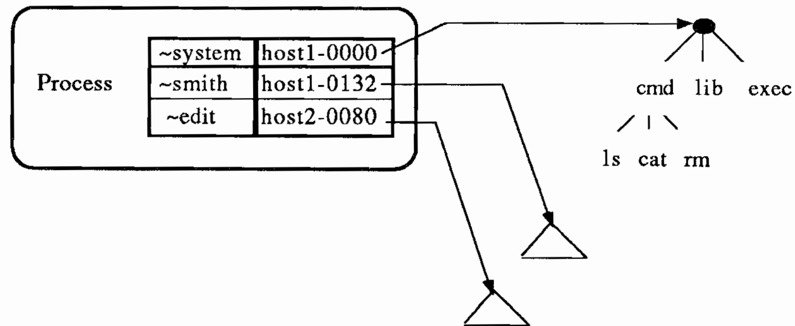

Figure 3: Tilde Names and Medusa Names in the Tilde Forest

In the figure, the process is shown to have a Tilde forest with three bindings. These bindings are stored in a data structure added to the per-process data maintained by the UNIX kernel. When the file name `~system/cmd/ls` is processed by the kernel primitive to open a file, the kernel searches this table of bindings and determines that the Medusa name for the tree name `~system` within the user's Forest is `host1-0000`.

In our prototype, each Medusa name has two components: the name of the processor within the local network on which the tree was created and a tree identification number guaranteed to be unique among the trees created on that processor. Such names appear to violate the goals of our system in two ways. First, including a host name as a component of a tree's name certainly seems to make the name storage site dependent. In our implementation, however, the host name component (`host1` in the example) represents the host on which the tree was *created*, not the site at which the tree is currently stored. The host name is only included to provide a simple, distributed Medusa name generation mechanism. Once the Tilde tree has been created, it retains its original Medusa name even if it is moved to another storage site within the local network. Thus, even if a Tilde tree is moved while it is a part of any Tilde forests, the bindings in the various Tilde forests that refer to the Tilde tree need not be modified to reflect that new location.

Second, while we have achieved location independence, it is apparently at the cost of reintroducing a globally unique, flat,

absolute namespace represented by Medusa names. In the Tilde naming system, however, Medusa names are hidden from the user, who identifies files using Tilde names, not Medusa names. We view Medusa names as a part of the underlying implementation — similar to "i-node numbers" or "disk block numbers".

Given a tree's Medusa name, the system still needs a way to locate and access the Tilde tree in the system. If the Medusa name does not contain any information about the associates Tilde tree's current location, how can the tree be located in the local network? As indicated above, our implementation takes advantage of NFS and the underlying UNIX file system to accomplish this.

The hosts participating in the Tilde naming system all use NFS as a transport mechanism, and all organize their local file naming hierarchies as shown in figure 4. Note that the figure denotes the underlying UNIX file naming environment from our prototype — not the Tilde naming system that the user sees. Each host using the Tilde naming system collects its Tilde trees under a single directory dedicated to that host. In the figure, three hosts, `host1`, `host2` and `host3` are represented by host directories. The individual host directories are all collected into a single directory, called `/.tilde`, on each host. Thus, on `host1` the subdirectories `/.tilde` and `/.tilde/host1` are part of the local storage attached to `host1`, while `/.tilde/host2` and `/.tilde/host3` represent remote links to directories residing on `host2` and `host3`. Figure 4 also shows that Tilde tree `host2-0079`, created on `host2`, has migrated to `host1` while retaining its original Medusa name.

Within each host subdirectory, Tilde trees are represented by a subtree whose root is given that tree's Medusa name. With this common structure in place on all participating hosts, any host can search the set of Tilde trees currently managed by all hosts by scanning the subdirectories under the `/.tilde` directory. We can now fill in the missing step in the Tilde naming evaluation mechanism — once the evaluation mechanism has located the Medusa name of the desired Tilde tree, the UNIX subtree that represents that Tilde tree can be located by scanning for a UNIX directory with that Medusa name in the set of directories managed by the participating Tilde system hosts.

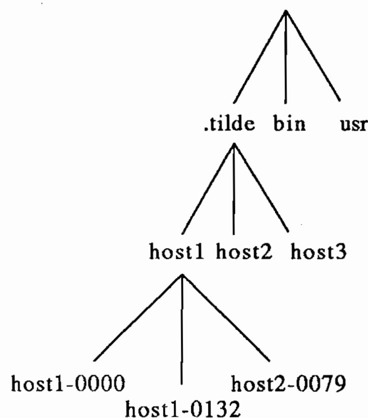

Figure 4: The Organization of Host Directories

To create a new Tilde tree, a host simply creates a new subdirectory representing the Tilde tree in its local host directory. The host constructs a unique Medusa name for that Tilde tree and identifies the subdirectory by the Medusa name of the Tilde tree it represents. Because the collection of subdirectories in the .tilde directories represents a global database of Tilde tree and Medusa names, creation of a new Tilde tree subdirectory in one of the host directories implicitly adds the new Tilde tree to the global set of Tilde trees.

Migrating a Tilde tree requires simply moving the subdirectory representing the Tilde tree to a new host directory. As figure 4 shows, the Tilde tree retains its original Medusa name, regardless of the host on which it resides. As long as the Tilde tree resides in some host directory, it is a part of the set of Tilde tress and can be located by the Tilde name evaluation mechanism.

In essence, our prototype uses a linear search mechanism to locate Tilde trees. The search mechanism does include some optimizations. One optimization is the use of a cache in the file evaluation mechanism that retains the last known location of Tilde trees. The name evaluation mechanism consults the cache and first probes for a Tilde tree at its last known location. If that probe fails, or the Tilde tree is not represented in the cache, the name evaluation mechanism next looks for the Tilde tree on the host at which the tree was created (as recorded in the Tilde tree's

Medusa name) by simply probing for the Tilde tree in the directory in which the appropriate host's Tilde trees are stored. Our use of caching is predicated on the assumption that Tilde trees will tend to remain on the same machine on which they were created.

4.2. Tilde Forest Management

The details of the Tilde forest management mechanisms are the result of a conscious effort to build an experimental system suitable for extension. As a result, the kernel primitives are minimal and the *csh* interface is extensible. the Tilde forest management system consists of four main components:

1. Kernel Primitives — New kernel primitives are provided to enable a process to manage its tilde forest
2. Shell built-in functions — *csh* commands give the user access to the kernel primitives for management of the *csh* process' Tilde forest.
3. Command scripts — The primitive *csh* interface is extended through the use of the *csh* command language to provide more powerful Tilde forest management tools
4. Tilde Tree Registry — A simple data base is provided through which elements of the collection of trees managed by the local network can be identified.

The first two components were constructed as part of the initial prototype implementation, and are discussed here, while the remaining components were developed through the use of the prototype and are covered in section 5.

The local network employs the UNIX model of process creation, in which a new child process is instantiated as an exact copy of the requesting parent process. The child process inherits the execution environment of the parent process, including open files and current working directory. Under Tilde naming, this inheritance model is extended to include the Tilde forest, so that a new process inherits its initial Tilde forest from its parent. A child process, therefore, initially resolves file names in the same environment as its parent process, just as the child process shares other parts of its execution environment with its parent process.

Once a process begins independent execution, it can dynamically alter its local Tilde forest without affecting the naming environment of any other processes.

A process' requirements for modification of its Tilde forest are simple. As mentioned earlier, the Tilde forest consists of a list of bindings from Tilde names to Medusa names. A process can modify its Tilde forest by adding a new binding to the forest or by deleting an existing binding from the forest. A process can also list the bindings in its Tilde forest to obtain information about its current naming environment. Adding a new binding between a Tilde name and a Medusa name effectively adds a new Tilde tree to the process' naming environment, making files in that new Tilde tree accessible to the process. Similarly, deleting a binding from the forest effectively removes the tree from the process' naming environment. These simple management mechanisms can be combined into more complex operations, as we will see in a later section.

Tilde hosts use a process-based command interpreter mechanism, an extension of the UNIX *csh*, which executes commands on behalf of the user as separate, child processes. Command processes, instantiated by the command processor, inherit the command processor's Tilde forest and, therefore, execute in the same naming environment as the command processor itself. The command processor's Tilde forest can be thought of as the user's Tilde forest. Modifications to the command processor's forest alter the naming environment perceived by the user.

The basic *csh* interface consists of a small set of builtin commands providing functions similar to the Tilde forest management kernel primitives available to a process. That is, by issuing a simple command a user can add a new tree to his forest, delete a tree from his forest, or list the current contents of his forest. Thus, the Tilde version of *csh* includes additional command interpretation to recognize user requests for forest modification and to execute the appropriate kernel commands for modifying *csh* process' Tilde forest as requested.

A user's initial Tilde forest, established by the local network's session initiation mechanism, consists of a small set of Tilde trees, including a tree in which the user can store session customization information. This mechanism is analogous to initial profile

mechanisms such as the UNIX `.login` or the VM/CMS `PROFILE EXEC` files. Once the initial forest is established, the user can establish a Tilde forest composed of bindings to a working collection of Tilde trees either interactively or through command scripts included as part of the session customization information.

The inheritance of a process' Tilde forest extends to processes created on remote hosts as well as processes created locally. This ensures that name interpretation is evaluation site independent. Regardless of which processor within the system a process is spawned on (or migrates to), the interpretation of names is always determined by the Tilde forest established by the user.

5. *Experience with Tilde Naming*

We now present an informal summary of our experiences with our experimental Tilde naming implementation. First, we describe extensions made to the *csh* interface that support two different classes of users: *novice users*, who work in a static environment of globally shared Tilde trees, and *expert users*, who modify their Tilde forests dynamically to share Tilde trees with other groups of users. Section 5.3 explains the Tilde Tree Registry mechanism for identification of Tilde trees. The next subsections illustrates the use of Tilde naming to solve naming conflicts between software subsystems. This section then concludes with a description of a uniform organization of the components of a Tilde tree that we developed to enhance the project subtree abstraction.

5.1. *Novice Users*

The goal of minimizing the impact of Tilde naming on novice users led us to design static, unobtrusive mechanisms that can be used and managed with little interaction on the part of the user. Ideally, the user interface should consist of a single command, perhaps under the control of a system administrator, that establishes a standard, consistent Tilde forest at the initiation of a computing session. The novice user or the system administrator can add that single command to the user's session initiation file, so

that the user need never issue any explicit Tilde forest management commands. As a result, the novice user can function in the Tilde naming environment without being aware of the change from an absolute naming system.

We take advantage of the ability of *csh* to read sequences of commands from *command scripts* to extend the basic Tilde naming interface mechanisms to include simple primitives for management of a standard Tilde forest. For example, the local network administrator can maintain command scripts that define lists of trees, such as standard system trees or users' home trees, so that the user can, with a single *csh* command, incorporate standard sets of trees into his forest when initiating a computing session. These lists of Tilde trees also allow the system administrator to install new system trees by simply changing the lists of standard trees.

A user who wishes to establish a personalized forest, different from the standard system forests, at the beginning of each computing session, defines his own list of trees in a local file. Reading the file during session initiation then automatically sets up the desired local naming environment. The mechanism for listing the user's Tilde forest is integrated with the other forest management primitives, so that a personalized forest can be created by interactively constructing the desired Tilde forest and saving a listing of the forest in a local file.

5.2. *Expert Users*

More experienced users use the ability to manage their private Tilde forests to better control their naming environments. The ability to quickly switch between multiple naming environments is especially useful to software developers. For example, when developing a software subsystem for distribution to other local networks, the user will want to test the components of the system in a minimal Tilde naming environment composed of only commonly available Tilde trees, while developing the code in a more extensive environment. This management function is accomplished by listing both Tilde forests in files, and creating a command script that deletes the existing forest and reads in a new forest from a specified file. Users collaborating on a project can

introduce a new version of a project by adding the project's Tilde tree to their Tilde forest, and then return cleanly to their usual environment by removing the Tilde tree.

We found several styles of command scripts to be useful in enhancing a user's interface to the Tilde naming system. A user who makes modifications to his Tilde forest may want to maintain a consistent environment across computing sessions, retaining changes to the environment rather than establishing a standard, default environment. If the user saves a copy of his forest when he terminates a session, and reestablishes that forest when he initiates the next session, the user's naming environment can be retained across sessions. Switching between two Tilde forests can be easily accomplished with two scripts, generated by saving the current Tilde forest, each containing commands to establish a specific Tilde forest.

5.3 The Tilde Tree Registry

The environment management primitives discussed in the previous sections assume a mechanism for the identification of Tilde trees that are not in the Tilde forest. While the kernel and *csh* primitives for forest management use Medusa Names for this purpose, Medusa names are not an appropriate mechanism for users to identify Tilde trees. Medusa names are not mnemonic and are not chosen by the user; they exist only for the convenience of the underlying name resolution mechanism. Therefore, in the Tilde naming system prototype, we constructed another mechanism for the identification of Tilde trees.

The *Tilde Tree Registry* is a collection of information describing each of the trees managed by the local network. Each Tilde tree has an entry in the Registry that lists the tree's distinguishing characteristics. The Registry accepts database-like queries and returns information about identified trees. The Registry mechanism is coordinated with the user interface to the Tilde forest, so that the results of Registry requests can be composed with Tilde forest management commands into more powerful mechanisms. Figure 5 illustrates the information retained in the Tilde tree Registry database. The information stored in the Registry includes the Tilde tree's owner and creation date, a short

comment describing the contents of the Tilde tree, and the tree's Tilde and Medusa names.

5.4 Establishing New Project Subtrees

Frequently, new subtrees must be added to hierarchical file systems to install new software subsystems, add a new user to a computing environment or develop an experimental version of an existing software subsystem. In a globally shared hierarchical naming system, each new subtree must be given a system-wide unique name. In the Tilde system, this need for the selection of common names is eliminated in many cases. Furthermore, in cases where it remains necessary to select a common name the number of users who must agree on a common name is reduced.

First, in cases where the files in a tree contain no file names that reference files within the tree itself, each user is completely free to assign any Tilde name to the tree when adding it to the user's Tilde forest. In the more difficult cases where the tree does contain self-references its users will all be forced to use a common Tilde name for the tree. Because naming is user-relative, however, the name selected need not be a system-wide unique name. It needs only be unique within the Tilde forests of the users who wish to share the tree. Thus, the resolution of each name conflict is localized to a group of users who are directly interested in the files being named. This is critical to the administrative scaling of the system since it implies that the bulk of name assignment problems can be resolved locally.

One interesting case we have observed in which the use of relative naming makes it possible to avoid what would become a name conflict in an absolute naming system is the development of

Medusa Name	Tilde Name	Owner Date	Creation	Comment
host2-0000	˜system	sysadmin	11/02/85	host2's system Tilde tree
host1-0000	˜system	sysadmin	01/04/86	host1's system Tilde tree
host2-0080	˜edit	jones	01/10/86	experimental Edit
host2-0079	˜edit	sysadmin	01/05/86	production Edit
host1-0132	˜smith	smith	12/10/86	user smith's home Tilde tree

Figure 5: Tilde Tree Registry Database

a new version of an installed software subsystem. Suppose that there is a large user community that depends on the availability and stability of the original, or production, version of some subsystem which is also being revised as a development project. One solution is to create a copy of the subsystem's project subtree, which can then be modified without affecting the users of the original version. Figure 6 gives an example of a global naming hierarchy in which a new, experimental version of the `edit` project subtree has been installed under the directory `edit-exp`.

In Figure 6, references to the components of the experimental version of the editor subsystem must all be changed from `/cmds/edit` to `/cmds/edit-exp` to reflect the new name of the root of the project subtree. In typical software subsystems, these references are scattered throughout the subsystem source code, and generated in obscure and arcane ways, so that it is inconvenient to locate and alter all these internal references.

In the Tilde naming environment, the experimental version of the editor can be created as a new Tilde tree containing copies of all the components of the production Tilde tree. The experimental Tilde tree is entered into the Tilde tree Registry with the same

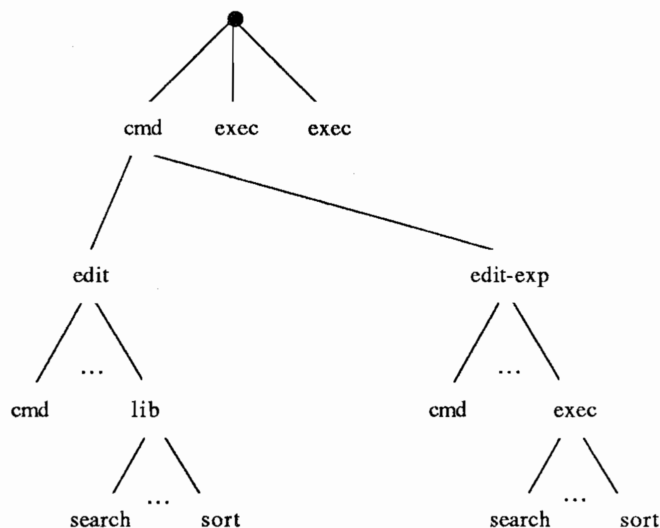

Figure 6: Creation of an Experimental Version of a Software Subsystem

Tilde name as the production tree, but with a unique Medusa name differentiating the experimental and production trees. In this instance, then, a user developing the new version of the editor can bind the development tree into his Tilde forest under the Tilde name `~edit`, while the rest of the user community will continue to bind the production tree to the name `~edit`. Figure 7 gives an example of two processes, both of which use `~edit` to reference the production and experimental versions of the editor subsystem. In the figure, the two Tilde trees are identified by their respective Medusa names.

The two processes are able to use the two versions of the editor without making changes to the software subsystems; the bindings in the Tilde forests cause internal references between editor system components to resolve to files in the appropriate Tilde tree.

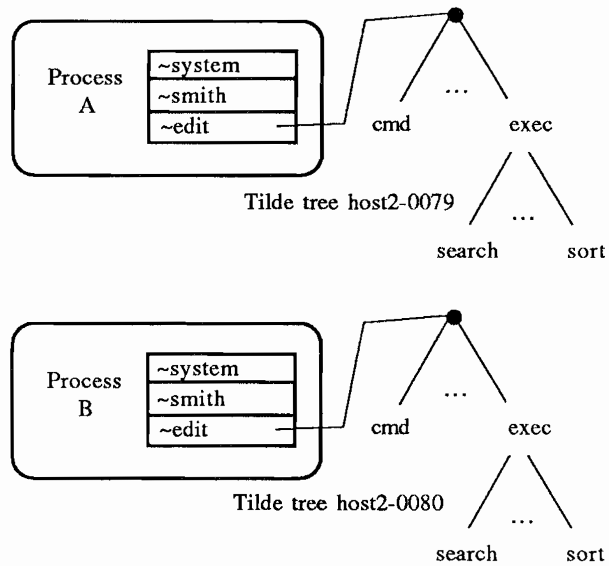

Figure 7: Creation of an Experimental Tilde Tree

5.5 *Organization of Tilde Trees as Project Subtrees*

The set of Tilde trees in our prototype system represents a reorganization of the files supplied with the UNIX operating system. The trees are organized according to the project subtree paradigm, so that each Tilde tree represents a separate software subsystem. Components of a software subsystem are collected into a single Tilde tree, rather than stored in directories shared by many software subsystems.

As work on the experimental system progressed, we found that many software subsystems share a common internal organization. Reorganizing the files in these subsystems according to a standard structure creates a uniform external interface for the software packages. Components of software subsystems fall into seven categories:

- Commands — commands executed by the user; the system's external interface.
- Executables — executable modules not directly invoked by the users.
- Databases — text-oriented data files shared by the system components.
- Libraries — collections of non-textual information (e.g., separately compiled modules)
- Sources — used to construct the system components.
- Included files — text modules shared among system source files.
- Documentation — all documentation such as user manual entries and descriptive papers.

These categories are defined based on our experience with UNIX-based software subsystems. We do not claim that this organization would be useful in other environments.

The Tilde tree structure shown in figure b illustrates the organization of the components of the C compiler within the naming structure suggested by the categories listed above. In the figure, application programs like the compiler and the assembler are stored as `~cc/cmd/cc` and `~cc/cmd/as`. Executable

components not usually invoked directly by the user are stored as ~cc/exec/ccom and ~cc/exec/c2. Source code files are stored under ~cc/src in subdirectories that reflect the subdirectories at the root of the Tilde tree, so the source for cc is stored in ~cc/src/cmd-exec/Cc. Header files are stored in ~cc/include and documentation files like cc.1 and ls.1 are stored in ~cc/doc.

6. Conclusions

The design of flexible, effective and convenient naming systems is a difficult problem, especially in distributed computing environments. Through our experience with distributed computing environments, we have established several important features desirable in naming mechanisms. The Tilde naming system incorporates these features.

The Tilde naming system increases the independence of file naming from the structure of the distributed computing environment and enhances software portability by isolating the

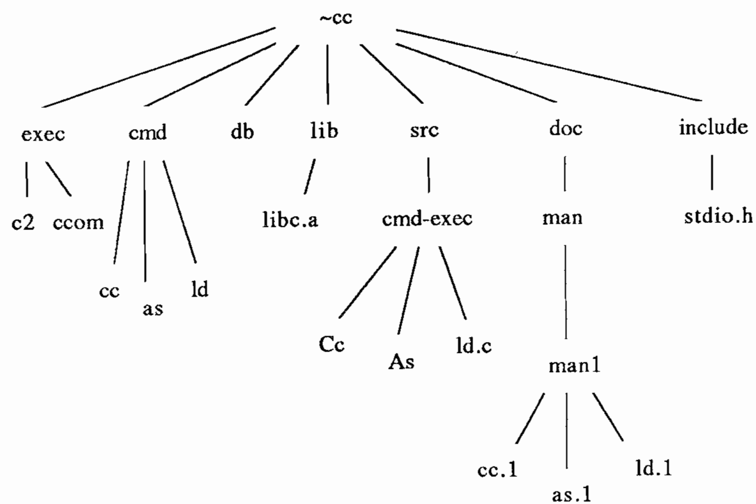

Figure 8:
Structure of the C Compiler Tilde Tree

identification of a file from the *access* to that file. The local, per-process Tilde naming environment provides storage and resolution site independence. These advantages incur an additional overhead of name environment management; this overhead is minimized partly through careful design of the process and user interfaces to the naming mechanism and partly through the recognition of the importance of a mechanism for grouping related files — the Tilde tree.

Experience with the prototype implementation described in this paper has reinforced our conviction that the basic design of the Tilde naming system is sound, and has further directed our research into distributed naming systems. We have experimented with extensions to the user interface of the Tilde naming environment, and have constructed new software subsystems that use the enhanced portability provided by Tilde naming. As a result of our effort to convert existing UNIX software to the Tilde naming system, we have identified a useful strategy for the organization of the files within a Tilde tree.

It might appear that the introduction of Tilde naming increase rather than decreases the complexity of name management. UNIX already includes a rich set of mechanisms with which users can construct name evaluation environments. How does Tilde naming make the task of managing a user's naming environment easier?

Many of the name mappings provided by Tilde naming can, in fact, be implemented using existing mechanisms such as environment variables, symbolic links and remotely mounted directories. Tilde naming unifies the management of a user's naming environment under a single, explicit mechanism, eliminating the need for other, ad hoc mechanisms. And, because Tilde naming is a per-user, rather than a per-processor, naming mechanism, a user's naming environment can be replicated on remote hosts, where the per-processor naming environment constructed of local symbolic links and a local mount structure may be different than the naming environment on the local host.

There are several directions for future work in this area:

- The current experimental system uses a broadcast mechanism to locate a Tilde tree in the local network. As we discussed in section 4.1, this mechanism will not scale well to

larger local networks, and should be replaced by a dynamic location mechanism. The use of "hints" as described in the design of QuickSilver could provide the basis for such an efficient location mechanism.

- Trusted software can be spoofed by the current Tilde forest inheritance mechanism. To ensure security, a trusted process must be able to verify its file naming environment by examining and specifying its Tilde forest.
- There is no inherent limitation that precludes the extension of the Tilde naming environment across administrative boundaries between local networks. However, the details of Tilde tree identification and location in an inter-network environment will require changes in our underlying transport and identification mechanisms. For example, as we suggested in section 3, a tree's Medusa name may be dependent on the local network at which the tree is stored (but not on the tree's specific location within the local network).
- We have explained how the Tilde naming mechanism provides a system-independent naming environment through inheritance of the Tilde forest by child processes. There is another mechanism for remote execution, called the Client-Server model, which uses inter-process communication between clients and servers for the execution of services, as opposed to the command interpreter's use of child processes. Inheritance of the Tilde forest does not apply, since the two communicating processes do not share a common ancestor in the process tree.

Members of the DASH project [Korb 1984] experimented with the sharing of Tilde forests between Client and Server processes [Wills 1986]. The DASH system uses a Client-Server mechanism for service execution, in which Client processes on workstations request services from Server processes on hosts. To preserve its naming environment, a Client must transmit its execution environment to the Server, so that the service can be performed in the expected environment. For example, a Client can share its Tilde forest with a Server by first listing the forest, sending a representation of the forest bindings to the Server, and requesting that the Server establish those bindings in its Tilde forest. This

mechanism for passing a Tilde forest between processes is equivalent to the inheritance of an initial Tilde forest by a child process.

References

D. R. Brownbridge, L. F. Marshall, and B Randell. The Newcastle Connection or UNIXes of the World Unite! *Software-Practice and Experience*, 12:1147-1162, 1982.

L. F. Cabrera and J. Wyllie. QuickSilver Distributed File Services: An Architecture for Horizontal Growth. newblock Almaden Research Center Research Report RJ 5578, IBM, April 1987.

D. Comer. Transparent Integrated Local and Distributed Environments (TILDE) Project Overview. Technical Report CSD-TR-466, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, 1984.

D. Comer and R. Droms. Tilde Trees in the UNIX Environment. *Proc. of the Winter 1985 Usenix Conf.*, 1985.

D. Comer, R. Droms, and T. Murtagh. Process-Relative File Naming. *ACM Transactions on Computing Systems*, (submitted August, 1989).

D. Comer and T. Murtagh. The Tilde File Naming Scheme. *Proc. of the IEEE Sixth Int. Conf. on Distributed Computing Systems*, pages 509-514, Cambridge, MA, May 1986.

Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA. *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version*, 1983.

Digital Equipment Corporation, Maynard, MA. *VAX/VMS Command Language User's Guide, Volume 2a, Command Language and System Messages*, 1982.

R. Droms. *Naming of Files in Distributed Systems*. PhD thesis, Purdue University, West Lafayette, Indiana, 1986.

IBM Corporation, Information Development, Dept. G60, Endicott, NY. *VM/SP CMS Command Reference*, 1986. SC19-6209-4.

J. T. Korb. An Overview of the DASH Intelligent Terminal. Technical Report CSD-TR-492, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, September 1984.

J. H. Morris et al. Andrew: a distributed personal computing environment. *Comm. of the ACM*, 29(3):184-281, March 1986.

J. S. Quarterman, A. Silberschatz, and J. L. Peterson. 4.2BSD and 4.3BSD as Examples of the UNIX System. *ACM Computing Surveys*, 17(4):379-418, December 1985.

A. P. Rifkin et al. RFS architectural overview. *Proc. of the Summer 1986 Usenix Conf.*, pages 248-259, 1986.

J. Saltzer, November 1987. Private communication.

R. Sandberg. The Sun Network File System: Design, Implementation and Experience. FE146-0/20K, Sun Microsystems, Inc., Mountain View, California, November 1987.

R. Sandberg et al. Design and implementation of the Sun Network File System. *Proc. of the Summer 1985 Usenix Conf.*, 1985.

M. Satyanarayanan et al. The ITC Distributed File System: Principles and Design. *Proc. of the Tenth ACM Symposium on Operating Systems Design*, December 1985.

Sun Microsystems, Inc. NFS: Network File System protocol specification. RFC 1094, NIC, March 1989.

K. Thompson. UNIX Implementation. *The Bell System Technical Journal*, 57(6):1931-1946, July-August 1978.

W. Tichy and Z. Ruan. Towards a Distributed File System. *Proc. of the Summer 1984 Usenix Conf.*, pages 87-97, 1984.

B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. *Proc. of the Ninth ACM Symposium on Operating System Principles*, October 1983.

B. Welch and J. Ousterhout. Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System. *Proc. of the IEEE Sixth Int. Conf. on Distributed Computing Systems*, pages 184-189, May 1986.

C. Wills. The Use of Services in the TILDE Environment. Technical Report CSD-TR-656, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, December 1986.

[*submitted Nov. 8, 1989; revised May 30, 1990; accepted July 28, 1990*]