FACE: ENHANCING DISTRIBUTED FILE SYSTEMS FOR AUTONOMOUS COMPUTING ENVIRONMENTS

Rafael Alonso
Daniel Barbará
Luis L. Cova

CS-TR-214-89

March 1989

FACE:
Enhancing Distributed File Systems for Autonomous Computing Environments[†]

Rafael Alonso
Daniel Barbará
Luis L. Cova

Computer Science Department
Princeton University
Princeton, NJ 08544

ABSTRACT

As distributed systems become larger, the appropriate system architectures are those that provide a great deal of support for local node autonomy. Since in such autonomous computing environments we expect that servers will have the freedom to deny service to any user, we believe that the proper view of a server is not that it is the only place in which to obtain a service or resource, but rather a server is the best place to do so. In the case of CPU servers it is clear how the above may be accomplished: a job that is denied service at a server will simply be run locally, although perhaps in a degraded fashion. It is less clear what to do in the case of a file server. A possible approach consists of storing at the server the latest copy of all user files, while keeping local copies of older versions of the most crucial information. Thus, even after a failure users may be able to proceed with their work, although in a degraded manner.

The idea of keeping local copies of key information has been called **stashing** in the literature. We augment the usefulness of stashing by combining it with the idea of **quasi-copies** (remote data copies that are allowed to diverge from the primary data but in a controlled, application-dependent fashion), which eases the cost of maintaining replicated data. In this paper we present the design of FACE, a distributed file system which allows users the option of using stashed files that are kept sufficiently consistent to fulfill user needs. The first FACE prototype has been designed and is currently being built via a series of enhancements to Sun's NFS.

March 11, 1989

FACE:
Enhancing Distributed File Systems for Autonomous Computing Environments†

Rafael Alonso
Daniel Barbará
Luis L. Cova

Computer Science Department
Princeton University
Princeton, NJ 08544

1. Introduction

Structuring a distributed system is a demanding task, even if the size of the system is quite limited. But the work becomes much more difficult when the scale of the system is very large. In discussing the architecture of such very large distributed systems (VLDS), one should consider for a moment how those systems may arise in practice. We claim that, although many large-scale systems will be formed by the growth of previously small distributed systems, most will be created by the joining together of a number of separate smaller systems.

In a system that has grown from within, designers may implement an architecture of their choice. However, in an environment composed of a multitude of cooperating sub-systems the autonomy of each entity must be respected in order to convince the owners of each of the separate sub-systems to join the distributed **federation**. It does not seem likely to us that system administrators will abandon all control over their local systems to participate in even the most wonderful global system. Thus, we believe that any viable distributed system architecture must support the notion of autonomy if it is to scale at all in the real world ([Alonso1988a]).

For the purposes of this discussion, we will (informally) define autonomy as the degree of freedom that a system has in denying remote requests on local resources. (This is neither a complete nor a precise definition of this concept but should suffice for our purposes.) Thus, a site has greater or lesser autonomy according to the degree in which it can restrict external access to its resources. Seen in this light, it would seem that the notion of structuring a VLDS around a set of **servers** may not be a wise decision. If a server is now allowed to refuse service to its clients, an unsuspecting user may find him or herself unable to complete his or her work because an unknown machine on a remote site does not wish to cooperate‡.

A little thought will reveal that what we are really arguing against is not the notion of a server *per se*, but rather against the idea that a server is the **only** place providing a resource. We would rather consider a server as the **best** place to obtain a resource or service, not the only place. For example, if a machine needs to execute a computationally-bound task it may request

---

† This research is supported by New Jersey Governor's Commission Award No. 85-990660-6, and grants from IBM and SRI's Sarnoff Laboratory.

‡ This situation has been described by Leslie Lamport as "*one where my program can crash because of some computer that I've never heard of is down.*"[Birrel1988]

remote execution in a computation server. However, if that server is unavailable, the home machine is usually ready to complete the task locally, albeit perhaps with degraded performance.

A less obvious example is that of a file server. Such a server may contain fast storage devices, the latest copy of all the information, archival storage, and guarantee frequent backups. But, if the file server cannot be used, users may be happy if they have slow access to a local storage device containing hourly snapshots of at least some of their files (for example, the most frequently accessed ones).

The idea of saving local copies of key information for use during communication failures has been discussed in the past. This technique has recently been baptized by other workers as **stashing** ([Birrel1988],[Schroeder1988]). However, the actual details of maintaining a stash have not been fully discussed in the literature. One of the issues that have not been addressed is that of how to maintain consistency between the stashed information and the primary copy.

One technique for reducing the overhead of maintaining replicated copies of information is to use old versions of the data. Using information that is stale but still useful is not novel. For example, using old data is already common practice in databases (i.e., snapshots [Zipf1949]), name servers [Birrell1982], routing [Tanenbaum1988], and caches in distributed systems [Terry1987] to name a few applications. However, in these applications the degree of inconsistency between the local version and the primary copy is left unspecified. Clearly, there is a wide spectrum of possibilities between fully consistent data and data that is simply known to be out of date.

To resolve this issue of consistency, we have developed in previous work the notion of **quasi-copies** [Alonso1988b]. Quasi-copies are replicated ''copies'' that may be somewhat out of date but are guaranteed to meet a certain consistency predicate. (We will give a more detailed description of quasi-copies in a later section.) Although the idea of quasi-copies was developed in the context of distributed databases, we feel that the notion of **controlled inconsistency** can be quite useful for distributed applications. In particular, we feel that this idea is quite appropriate for a distributed file system.

Currently, we are developing a distributed file system which uses quasi-copy techniques to support stashing. This work is being carried out as part of the **ACE** (**Autonomous Computing Environments**) project at Princeton University. In this project we have been working on a number of issues related to VLDS, in particular those that stress the role that autonomy should play in such systems. Our work to date in this area has resulted in new ideas for data sharing [Alonso1988b], resource sharing [Alonso1988c], negotiation [Alonso1989a], etc.

The distributed file system we are developing is called **FACE** (a **F**ilesystem for **ACE**). The first prototype of FACE consists of a series of enhancements to Sun's Network File System (NFS) [Walsh1985] in order to add a stashing capability to it. The current version of FACE has been completely designed, and its implementation is well under way. In this paper we present some of the issues that arose in the context of our design and current implementation, as well as give the details of our architecture.

The format of the rest of this paper is as follows. In the next section we describe the use of file stashing from a user's perspective. In Section 3 we discuss a number of issues that arise in an implementation. This is followed by a discussion of the FACE file system architecture, and of the modifications being made to NFS. Finally, we present our conclusions.

2. Using a Stashing File System

In this section we describe the use of a stashing file system from the user's perspective. (A detailed description of the system's view of the process appears in a latter section.) There are a number of points that we need to cover here. First, how does a user select the files that will be stashed (or dropped from the stash). Second, we need to specify what the user will experience when the communicating links to the primary file server are severed and the stashed information begins to be used. Third, since we only have a limited amount of memory available for the stashed data, at some point the stashed information will be insufficient; we need to address the issue of what alternatives will be made available to the user in this situation. And finally, we have to describe what happens after communications are restored.

While considering the points just described, we must keep in mind that the user community of most systems will be quite heterogeneous, and various users may differ in their level of computer sophistication. The view of the stashing process that a sophisticated user may be expected to have is quite different from that of an inexperienced one.

There are a number of ways in which a more experienced user will be able to select the files that he wants stashed. There are both static and dynamic types of choices. The static approach is more suitable for files which the user almost always needs stashed (for example, the operating system files, a compiler, a favorite editor). The method for specifying such ''permanently stashed'' files is via a list of the pathnames for those files that will be kept in a well-known file in the user's login directory (e.g., a ''.stashrc'' file). While the knowledgeable user will add and delete file names from the .stashrc file, less experienced users can simply use a default .stashrc provided by the system administrator.

Of course, we require a more dynamic way of specifying files to be stashed. In general, users will be involved in completing a certain task (or a collection of tasks) that is important enough that they do not want its completion jeopardized by a possible communication failure. Users who understand their application very well can simply use the stash command (i.e., ''stash filename'') to select the files to be stashed. Those involved in more complex projects will probably already have a mechanism equivalent to the UNIX make(1) facility [BSD1986] in order to keep track of the files in their application. Stashing can also be done by extending the stash command so that it can understand the format of the ''make'' file and then proceed to stash every file that is mentioned in the make file (either data files or commands).

Somewhat less sophisticated users may not be as inclined to think beforehand about their applications and the files that they will require. For them, another approach may be offered. Just before the user is about to start on his usual activity cycle (for example, edit, compile, run, etc.) the user will invoke the ''record_stash'' command. From this point on until a ''end_record_stash'' command is given, every file (data or otherwise) that is used will be added to the collection of files to be stashed. Thus, once the user has gone through one application cycle he can be fairly certain that he will be able to continue his work (at least for the near future).

But even this approach may not be appropriate for the least sophisticated users. For them, there are two choices. The first is that the application itself takes care of the stashing. For example, a spreadsheet program can, on its user's behalf, stash a copy of itself as well as one of the files being processed. This approach may result in much wasted space, but it certainly requires very little input from the user. Alternatively, the system can save a ''working set'' of files, i.e., stash the files that are accessed in the last δ commands, where δ is a predefined, tunable

parameter.

Next, we address the issue of what happens when there is a communication failure and the stashing mechanism takes over. Consider a user in the middle of editing a file. If, once communications fail, his file is replaced by the stashed version, the user will quickly notice (or worse, notice sometime later or not at all) that the file he was editing has somehow changed. This would be disconcerting to any but a sophisticated user. Thus, in most cases the proper action would be to notify all users that a failure has occurred (say, by a message on their CRT screen), which will prompt most users to abort their current command and restart it immediately (albeit with stashed data). The most sophisticated users will have the option of trying to continue with the new file without stopping their application. This may be appropriate for certain applications although clearly not for all.

If the communication failure lasts long enough, the user will eventually try to access a file that is not in the stash. At this point the user will be notified that the data is inaccessible, and most users (naive or otherwise) will simply have to stop (hopefully much later than if they had not had access to a stash), or turn to other computational activities.

Finally, after communications are restored users are left in the following situation. In general, there will be two copies of any files that they have modified during the communication failure. For the most naive users, the proper approach is to simply pick the stashed copy to replace the original one. For more knowledgeable users, we propose presenting them with the two files and asking them to pick which version will supplant the other. And for the most sophisticated users, tools could be made available to patch together both versions of the data. This last approach would be preferred in general. However it is difficult to implement, specially in cases where there are multiple users updating stashed data during the failure. This last topic is the focus of continuing research.

3. Implementation Issues.

At first glance, one would be tempted to use the techniques of caching for purposes of stashing. However, stashing and caching are two different concepts and should not be confused. Caching is done for performance considerations, i.e., in caching, the most frequently accessed information is saved [Howard1988]. In stashing, one must save files for autonomy and fault-tolerance† reasons. That is, we would like to save the essential files to keep working after the node is no longer able to communicate with the server. In stashing, locality of reference and "hot spots" do not play an important role. Stashing should be viewed as a "pre-fetching" technique used to save enough information so work can proceed in the event of disengagement from the file server.

In this section we present the issues involved in implementing stashing on a distributed file system. We will be dealing with four topics: where to direct file accesses, the consistency of the copies, the management of stashing after communication failures, and saving space in the storage reserved for stashing.

There are two ways of accessing stashed files, as Figure 3.1 shows. In Figure 3.1a, all user accesses are directed to the local disk, i.e., the stashed copy of the file. The updates are then

† In general, many autonomy issues are similar to issues of fault-tolerance. For example, a lack of response from a site could be handled in a similar fashion regardless of whether it was caused by a network failure or by denial of access.

propagated to the server, which in turn propagates them to other stashed copies. The Andrew file system [Howard1988], uses a similar configuration for caching purposes. In Figure 3.1b, accesses are always directed to the server. Updates are later propagated to the stashed copies. Both alternatives are useful, depending on the type of files involved. The strategy of Figure 3.1a is especially good for handling private files belonging to a single user, since it is very unlikely that more than one stashed copy of these files exist in the network. The strategy on Figure 3.1b is better for files that are to be shared among a community of users.

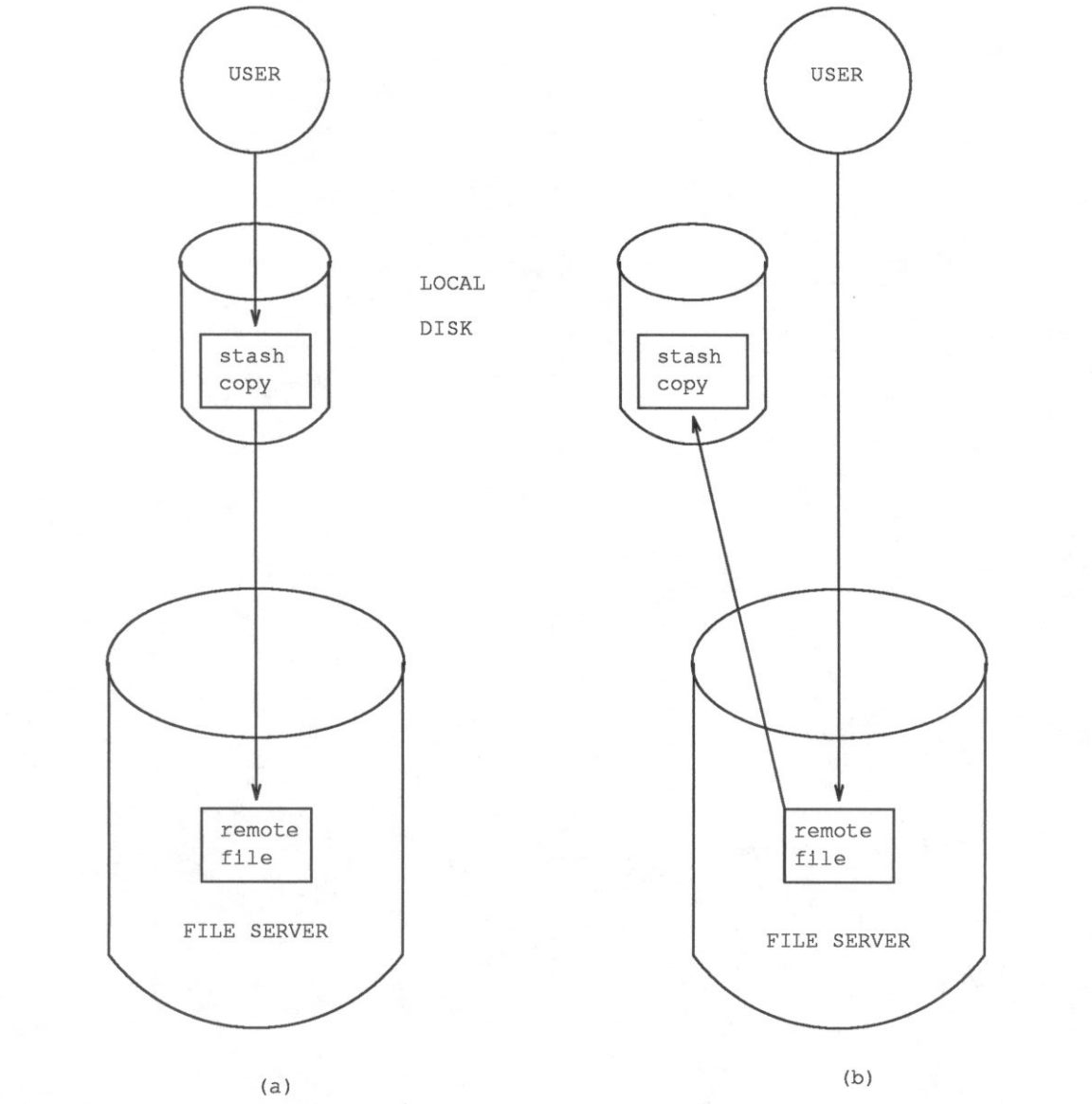

Figure 3.1: Accessing a remote file and its stashed copy

For either of the two alternatives, if one wants to have perfect consistency for the copies, i.e., keep them identical to the file in the server, there is no alternative but to use a two-phase commit protocol (or an equivalent mechanism). That is, every time an update is to be produced, all the copies should be locked and then the update propagated to all of them. Obviously, this would be costly to implement. Moreover, for many applications, we would be paying a high price for an unnecessary service. For instance, if a user is working to meet a paper deadline and the network fails, it is preferable that some version of the paper is still available, even if it is one hour old, than not having any information accessible. The worst case scenario is that all modifications made in the last hour are not reflected in the stash, and have to be repeated. But the inconsistency is certainly tolerable.

To let the application decide what is the maximum inconsistency that can be tolerated, we use quasi-copy techniques [Alonso1988b]. Quasi-copies were originally designed to deal with consistency issues in databases, but the concept is also applicable to a file system. With quasi-copies, it is assumed that a central location (server) exists, where all the updates are processed, and several copies are spread throughout the network. A predicate is associated with each copy, establishing the degree of inconsistency which can be tolerated. For instance, the predicate can state that the copy must not be more than ten minutes old. The user is free to choose from a spectrum of consistency specifications. These may range from demanding a perfect, consistent copy, to settling for a stale snapshot. The system guarantees that this predicate is not violated when updates occur. This can be done in one of two ways. In the first one, the server constantly watches for updates and becomes responsible for propagating them when the predicate is about to be violated. Alternatively, the clients could be responsible for the consistency of their copies, requesting fresh ones periodically. (Notice that this is only possible for age-dependent predicates.) We call the two ways of maintaining consistency *server maintained* and *client maintained* respectively. The same object may be client maintained for some of the copies and server maintained for others. Finally, it is worth pointing out that, since we use quasi-copies to deal with consistency, the alternative shown in Figure 3.1b is the most natural choice because there all updates occur at the server.

To handle quasi-copies in our file system, we introduce a software component called the *bookkeeper*. This module is in charge of keeping track of predicates associated with the files being stashed in the system. The bookkeeper in a client will keep track of predicates of stashed files whose consistency is client maintained. It will request that the server's bookkeeper send a fresh copy when the predicate is triggered. The bookkeeper in the server keeps track of the stashed copies whose consistency is server maintained. When an update is made to any of these files, the bookkeeper determines whether the predicate is still valid or not. In the latter case, a new copy is sent to the client. We have opted for implementing only the client maintained alternative in the initial prototype.

Some optimizations to reduce the overhead of sending files over the network are possible. For instance, in client maintained consistency, the client bookkeeper can remember the last time at which the server's copy was accessed. When requesting a fresh copy, the client bookkeeper may include this time in its request. The server bookkeeper will then compare this timestamp with the actual time at which the file was last modified, thus determining whether sending the file is necessary or not. To avoid sending large files, one could break the file into segments and apply file comparison techniques ([Barbara1989]) to find out which segments have changed since the last time the copy was sent, and only those segments need be transmitted.

The next issue is what to do when the client node is disengaged from the network. When this happens, accesses to the file must be redirected to the stashed copy. Moreover, as explained in the previous section, the user should be alerted that this is happening. Any updates that occur from there on are reflected in the stashed copy and not in the central copy. Later on, when the communication is reestablished, these updates can be reflected in the central version. However, if more than one stashed copy was updated, some mechanism is required to integrate all the versions. It could be as simple as making one of the versions persist or as complex as using some semantic analysis to merge them. In our current implementation, the last version written supersedes all others.

Finally, as we mentioned in Section 2, the storage space reserved for stashing is limited, so we need to maximize its usefulness. Some of the longest stashed files are immutable, or change very slowly with time (operating system files, editors, etc.). A helpful technique in saving space is to copy these files to the stash in compressed format.

4. Architecture

In Figure 4.1 we show a block diagram of FACE's general design. User processes interact with the system call interface to perform file and stashing operations. These operations may be invoked directly by user processes or by the FACE user-level routines. (These routines consist of a simplified implementation of the facilities explained in Section 2.) The bookkeeper is a daemon process that provides the runtime support for keeping the stashed copies of remote files within the user consistency requirements. The distributed file system interface directs user file accesses to the proper file system (UNIX, MSDOS, etc.). This layer also manages the necessary data structures for the stashed copies and the local disk partition where they reside (the stash partition). Our stashing facility is client maintained, i.e., it is the client machine who is responsible for setting up the data structures and disk space for the stashed copies. It is also its job to maintain the user consistency constraints.

The first FACE prototype is being implemented by modifying Sun's Network File System (NFS) [Walsh1985]. For completeness we provide a brief description of NFS in the next section. This will be followed by a complete description of the prototype, and we conclude with a detailed example.

4.1. NFS Overview

NFS is a system by which different computers share a file space. Its semantics are similar to those of the UNIX file system [Leffler1989]. Currently, several operating systems support the NFS protocols. Among them we find SunOS, Mt. Xinu's BSD UNIX implementation, NeXT's Mach, and MSDOS [PC-NFS1989].

In NFS all file activities are centered around the **vnode**, a data structure whose role is equivalent to the inode in traditional UNIX file systems (for an extended description of vnodes see [Kleiman1989]). A key idea in NFS is that of splitting the kernel functionality into two well defined parts. One consists of file system dependent information (e.g., inodes in the case of UNIX file systems), and another comprising implementation independent data structures (vnodes).

In Figure 4.2 we present the architecture of NFS. User processes perform file operations by using system calls. These system calls operate on the Virtual File System (VFS). VFSs consist of data structures and operations for each file system being incorporated. This is similar to the

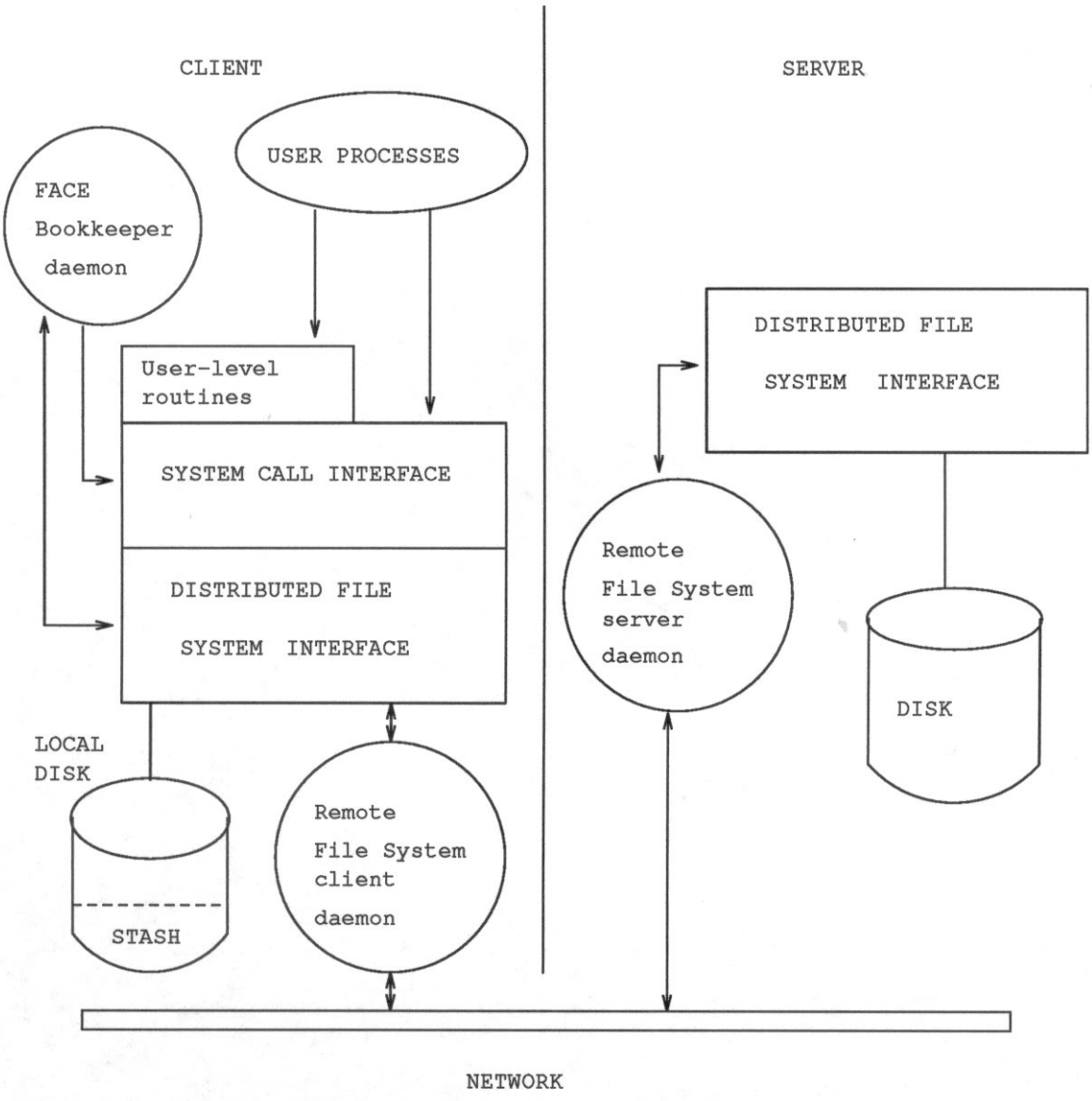

Figure 4.1: The architecture of FACE

mount table information in standard UNIX. Each file in a VFS is represented by a vnode, and any file operations on it are translated to the specific routines of the particular VFS to which the file belongs. For example, if the vnode represents a local UNIX file, then a read system call is translated to the appropriate routines to handle inodes (e.g., iget, bread,etc.) [Leffler1989]. If instead the vnode represents a remote file the appropriate Sun's RPC and XDR routines are invoked [ONC/NFS1988].

Figure 4.3 shows the contents of the VFS data structure and the vnode data structure. Each structure contains a pointer to an array of function entry points for the specific routines of each

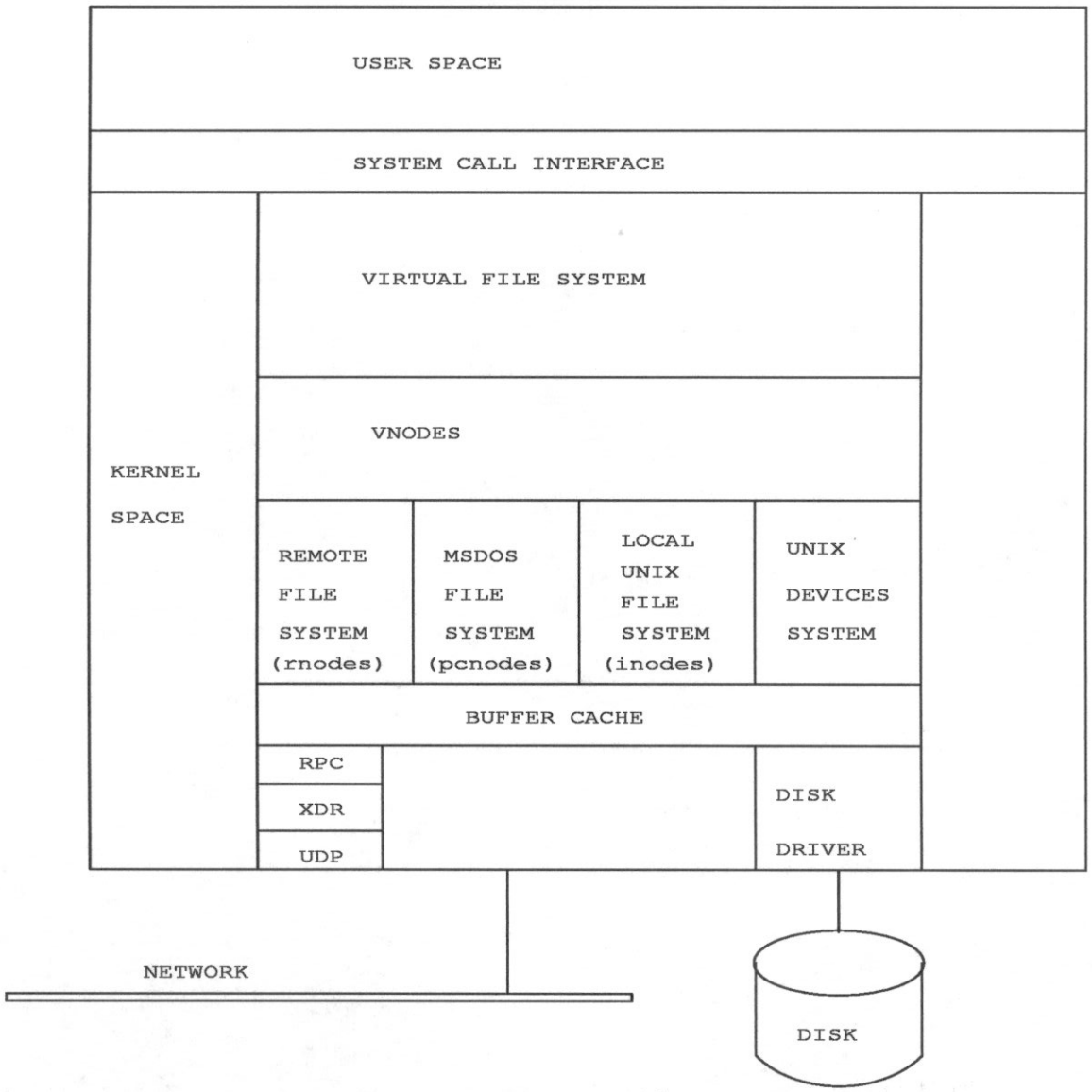

Figure 4.2: NFS architecture

particular VFS or vnode. The private data field also points to file system-specific information.

4.2. FACE Prototype

The first FACE prototype is being implemented on the facilities of the Princeton Distributed Computing Laboratory. The system is being developed on a Sun 3/50 computer running Sun's UNIX 4.2 release 3.3 and based on NFS.

The first decision we had to make was where to modify NFS to satisfy our design goals: fast prototyping, transparency at the user level, portability to a variety of computer systems, and

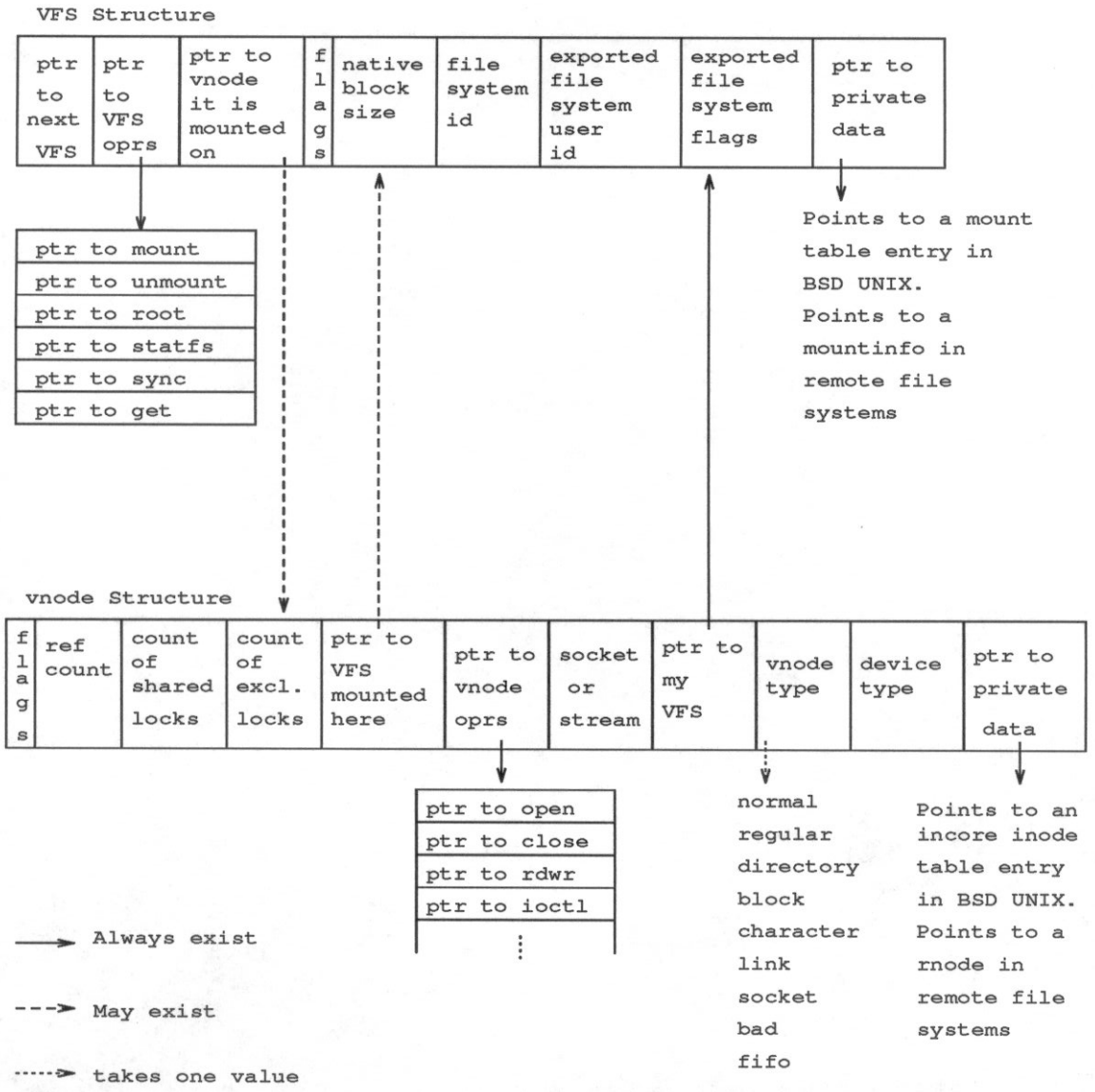

Figure 4.3: VFS and vnode structures

minimal modifications to the underlying implementation.

Figure 4.4 shows the different layers that exist in NFS as viewed from the file name translation process. From all these layers, the only one that has a widespread accepted standard is the vnode layer. For the first layer, *file name* , there are different naming conventions among operating systems (UNIX, MSDOS, MVS, etc.). Therefore translation mechanisms from file names to vnodes are different for each type of system. It is hard to implement stashing at this level, since it would consist in keeping two disjoint file spaces together by modifying the user-level applications. Clearly, user processes would have to be able to use the remote file name or the name of

the stashed copy. Although this could be provided by "enhanced" library functions, it would have to be done on a per language basis. Clearly, our goals of portability and transparency at the user level would not be met.

The *buffer cache* also differs from implementation to implementation. In some cases (Cedar [Schroeder1985], Andrew [Howard1988], Amoeba [Mullender1985], and iPcress [Staelin1989]) there can be a file oriented cache instead of a block cache. Forcing the caching system to handle stashing as well would not only require major modifications to the operating system, but may also result in a negative impact on the cache performance.

Finally the *physical location* layer is even more implementation dependent than the previously two discussed layers, for obvious reasons (heterogeneous I/O devices).

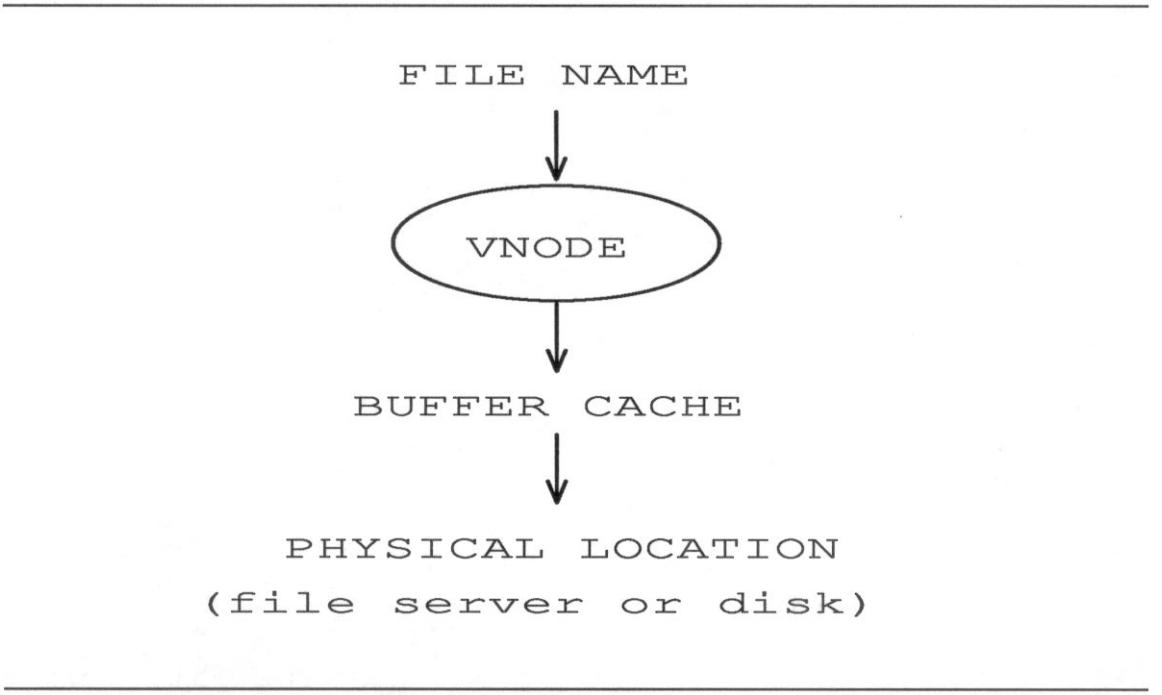

Figure 4.4: NFS name translation

Figure 4.5 shows a more detailed block diagram of the FACE prototype. The server side does not change at all from the NFS implementation since, as mentioned before, the stashing facility is client maintained. In the client side, the VFS/vnode component is modified. Some extra data structures are added to support the stashing facility. Several routines are modified and new ones are added to perform stashing operations (e.g., initiating and maintaining the necessary data structures, redirecting users file accesses to the stashed copies when the client machine disengages from the file server, etc.). Also, new system calls are introduced as well as a daemon process in charge of all the bookkeeping operations (the bookkeeper). This process maintains the consistency constraints (i.e., the quasi-copy predicates) between the remote files and the stashed copies. We now explain in more detail each of these changes.

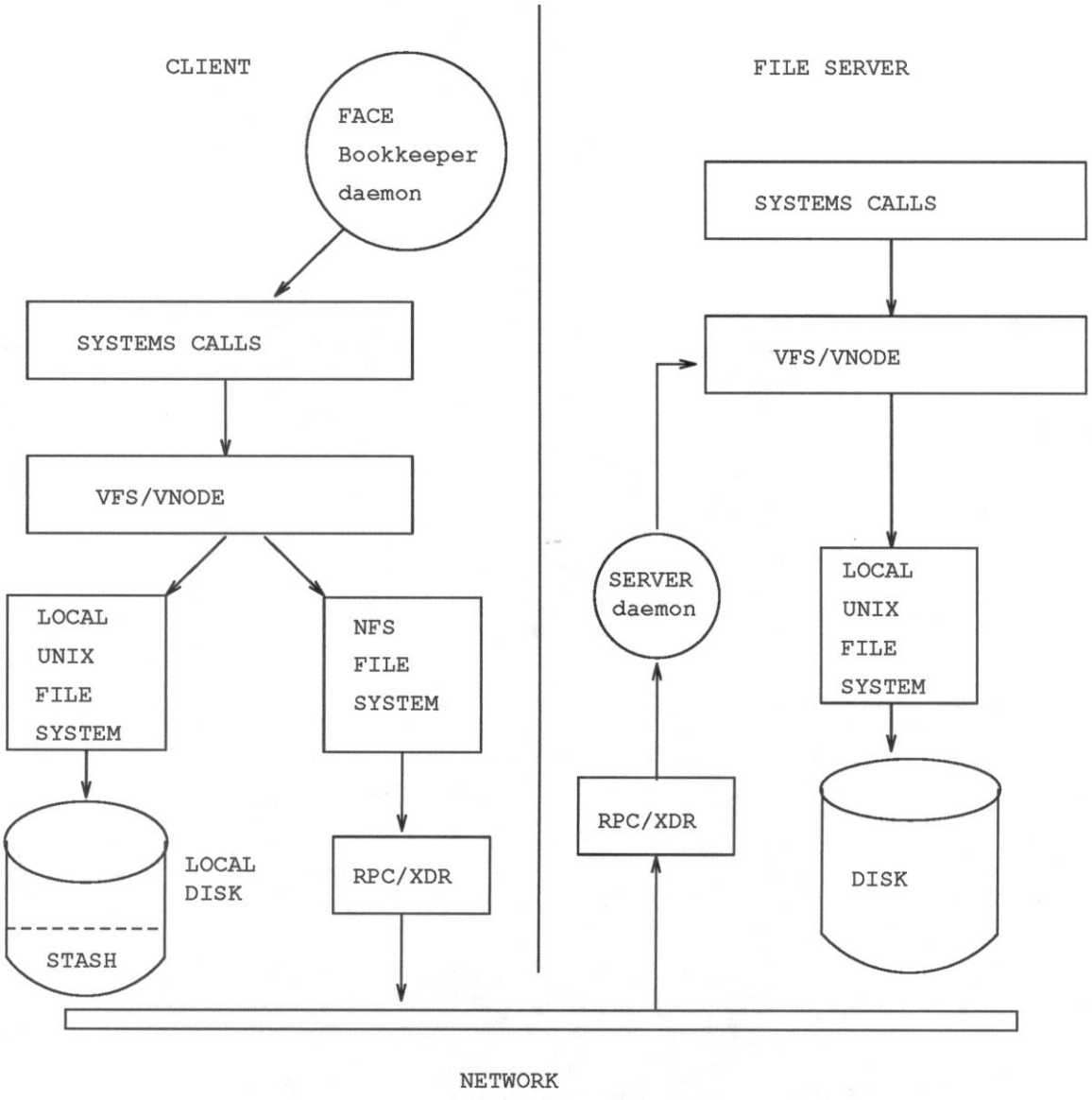

Figure 4.5: Block diagram of the FACE prototype based on NFS

4.2.1. New Data Structures

Figure 4.6 shows the data structures added to the NFS' data structures to support stashing. Four new data fields have been added to the rnode data structure that represent remote files at the client side. The first is a pointer to the vnode corresponding to the stashed copy (*stash vp*). The vnode pointed by *stash vp* belongs to the VFS of the client local UNIX file system. The second field, *predicate*, contains the user consistency constraint (the quasi-copy predicate). For this prototype, only time related predicates are allowed, i.e, the length of time a stashed copy is valid since it was last copied from the file server. The third field, *strategy*, indicates the user's

preference toward directing all of her/his file accesses to the stashed space during normal activity, i.e., using the stash copy as either a backup facility or a file cache. For the latter option, the stashed copy will not be automatically kept consistent with the remote file when the user make changes to it. This option is better used for remote files that change very slowly over time and that are only used for reading by the client. Finally, a *valid* field is added to indicate if the stashed copy conforms or not with the user predicate. This field is only added for performance reasons.

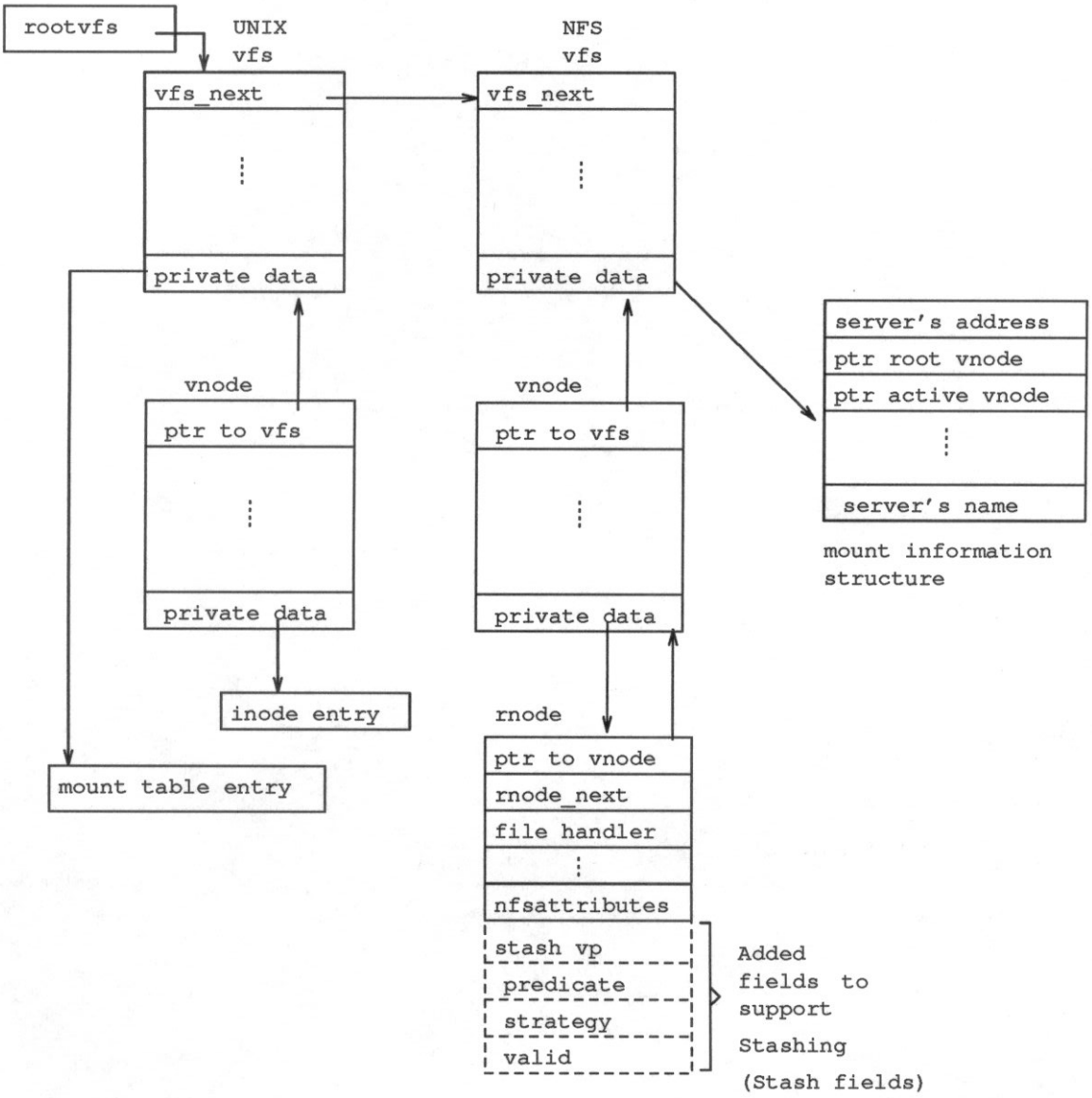

Figure 4.6: New NFS data structures to allow stashing

4.2.2. New and Modified Kernel Routines

In NFS, kernel code for all file related operations consists of macro definitions. These macros translate vnode operations to actions on the underlying file system (see Figure 4.3). In our prototype we redefined these macros to allow the use of either the remote files or the stashed copies. In pseudo-code, these macro definitions are presented in Figure 4.7.

```
DEFINE <file operation>
    IF (vnode represents a remote file) AND
        (strategy is local or there is no connection to the file server) AND
        (stash vp is not NULL) AND
        (valid is true)
    THEN
        (use <file operation> pointed by the rnode's stash vp pointer)
    ELSE
        (use <file operation> pointed by the current vnode)
```

Figure 4.7: New NFS macro definitions (in pseudo-code)

There are a number of new routines in the kernel. The first two handle the linking and unlinking of rnodes of remote files with the corresponding vnodes of stashed copies. Two others are used for allocating and de-allocating the new vnodes of the stashed copies in the local file system and stash partition. There are also routines for saving and deleting values from the rnode stash fields, setting up all the ancestors in the stash partition, etc. There are also many new procedures for dealing with client disengagement and re-engagement from the file server (informing the users that the file server is not available and that stashed copies are in used, informing the users of how many other users where accessing the remote files that are stashed at the moment of disengagement, allowing data patching between modified stash copies and the corresponding remote files, etc.).

4.2.3. New System Calls

The new system calls added to the NFS system calls interface in order to support stashing are the following:

status = **mkstash**(*fd,predicate,strategy,user_credentials*);

the **mkstash** call assigns the values provided by the user (*predicate* and *strategy*) to the corresponding rnode fields of the remote file to be stashed. The remote file is identified by the file descriptor (*fd*) returned by the open(1) [BSD1986] system call. It also creates a new vnode in the stash partition, with its corresponding inode entry. If the the data structures for the stashed copy are successfully allocated, a message is sent to the bookkeeper telling it that the file represented by *fd* has being stashed. *Status* has the return value of the system call that informs the calling process of the success of the call. The *user_credentials* are used to check access

permission to the remote file.

status = **rmstash**(*fd,user_credentials*);

the **rmstash** call removes the corresponding vnode of the stashed copy and its inode entry off the remote file represented by the file descriptor *fd*. It also clears the stash fields in *fd*'s rnode. It sends a message to the bookkeeper indicating that the file represented by *fd* is no longer stashed. *Status* and *user_credentials* have the same role as in the previous call.

status = **strategy**(*fd,strategy,user_credentials*);

all subsequent file accesses to the remote file represented by *fd*, during normal activity, are directed to the stashed copy if *strategy* is local. Otherwise, the accesses are directed to the file server. *Status* and *user_credentials* have the same role as in the **mkstash** call.

4.2.4. The Bookkeeper Daemon Process

The bookkeeper daemon process is a user level program that provides quasi-copy support for the stashing facility. The bookkeeper keeps information about all the files that have been stashed, i.e., file descriptors to the remote files that have a stashed copy. It also keeps the expiration times for each stashed copy. Thus, the bookkeeper can keep track of predicates, determining the validity of the contents of the stashed copies, and obtaining fresh information from the file server. The information is kept in a ordered linked list of file descriptors of the remote files being stashed (Figure 4.8).

The bookkeeper sends itself an alarm signal with the value of the earliest expiration time. When the signal arrives, the bookkeeper checks all those remote files which do not conform with the user consistency constraint, marks them invalid, and tries to backup them from the file server. For each successful backup the bookkeeper rearranges the ordered linked list and marks the stashed copy as valid. When the bookkeeper finishes with this procedure it sends a new alarm signal for the next expiration time.

4.2.5. An Example

To clarify the design presented in this section we conclude with an example of how the data structures are used to support stashing. Figure 4.9 represents the interconnection between the data structures of a remote file and its stashed copy. Figure 4.9c shows the commands being issued by an user of our FACE prototype. The user changes directory to "/user", edits a remote file ("a") and then requests a stash copy of it. Figure 4.9a shows the vnode and VFS data structures for the remote file. There are three file systems mounted: root ("/") from which the machine boots, a remote file system ("/user") and the stash partition ("/stash"). "/" and "/user" are UNIX 4.3 BSD file systems and reside in the client local disk. "/user" is a NFS file system representing a UNIX 4.2 file server. The vnode labeled "/root" is the vnode for the root directory of the user mounted file system. The vnode labeled "/user" is the vnode corresponding to the mount point for the remote file system "/user." The "/user/a" vnode represents the remote file that has been edited by the user. Once the user issues the command - *stash a 1hour remote* - the added data structures in the rnode of "/user/a" (*stash vp*, *predicate* and *strategy*) are filled by the system call **mkstash**. This system call allocates a vnode in the "/stash" file system for the stash file "/stash/user/a" (Figure 4.9b). It also tells the bookkeeper process that a new stash copy has been created. The bookkeeper stores the needed information to backup the content of the stashed copy with the content of the remote file. It uses the value in the predicate field of the rnode of "/user/a" to send itself an alarm signal for when the stash copy "/stash/user/a" not longer

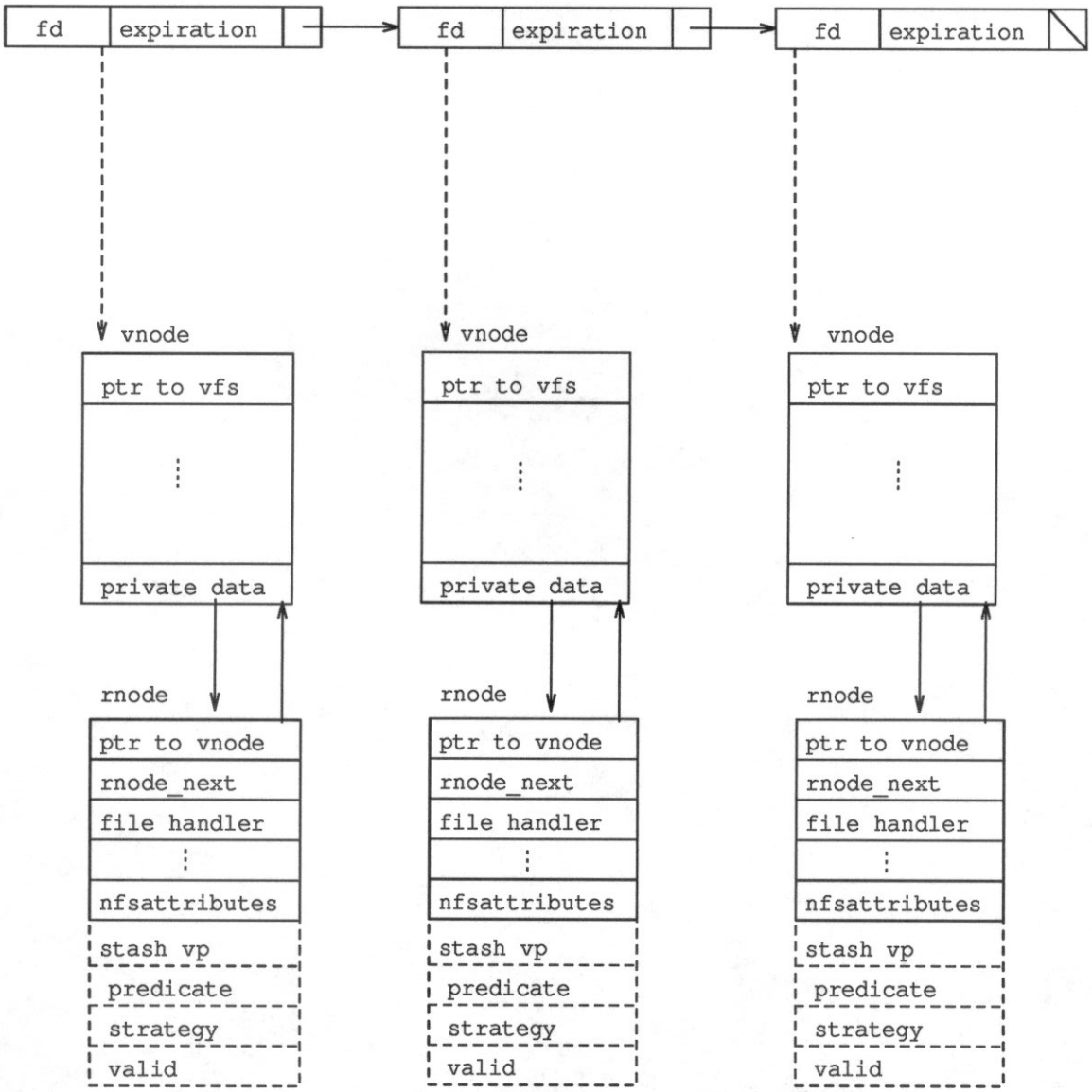

Figure 4.8: Bookkeeper internal information

conforms with the user consistency constraint.

If the strategy field is set to remote then all file accesses to "/user/a" are done exactly as in NFS, i.e., to the file server. At the moment that the file server is not longer reachable, the user is informed of such an event and subsequent accesses are done to the stash copy "/stash/user/a" through the *stash vp* field. The stash copy is used until the bookkeeper marks it as invalid (unless the user overrides this action) or the server is again reachable. Therefore, the facility is transparent to user applications.

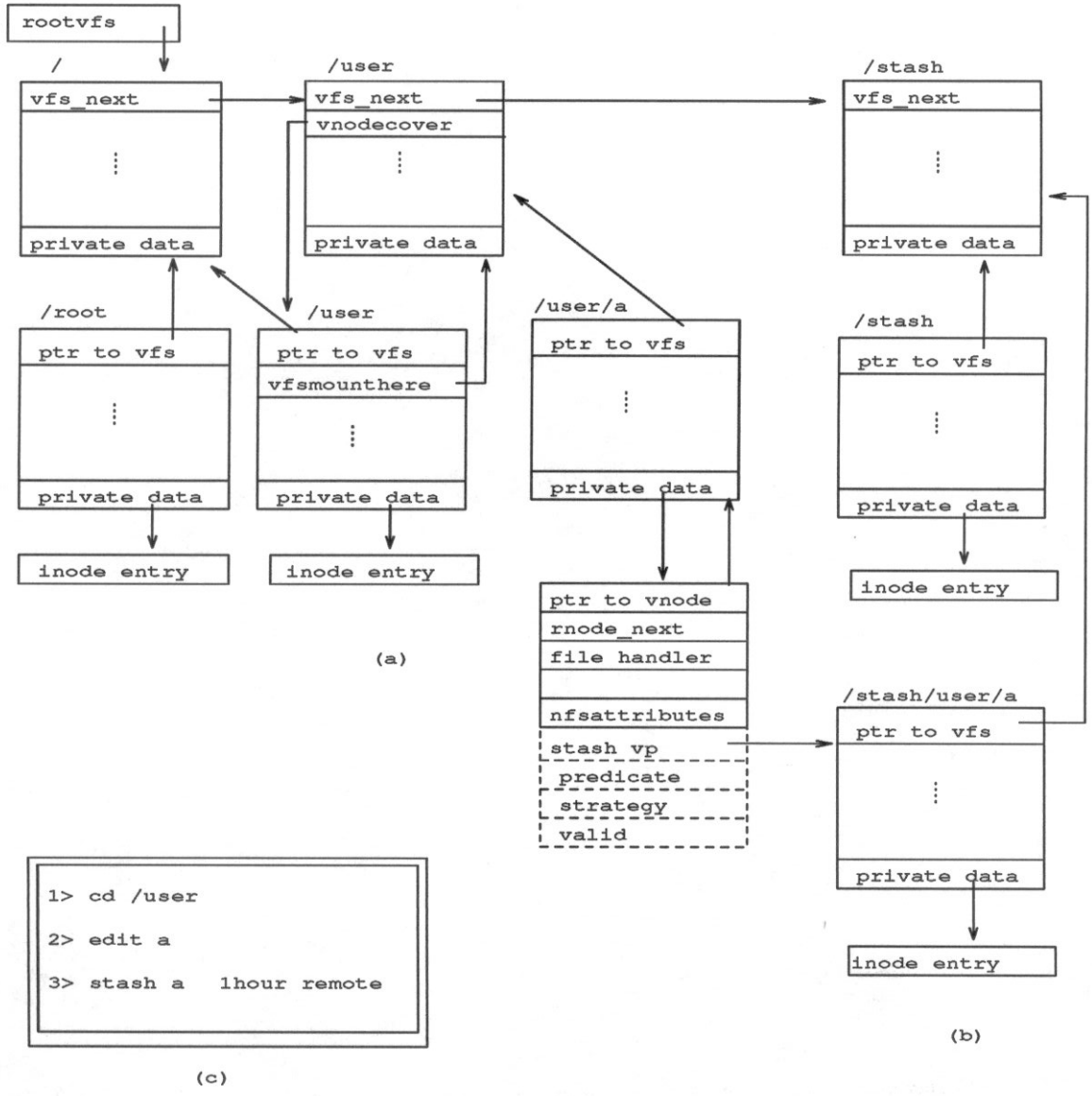

Figure 4.9: An example

5. Conclusions

We have presented in this paper the design of the first prototype of FACE, a distributed file system for autonomous computing environments. This system provides file stashing to increase the availability of important information when file servers are not reachable. This feature enhances both the autonomy of the local nodes as well as the degree of fault tolerance of the overall system.

We deal with the issue of stashed copy consistency, by incorporating into our design quasi-copy techniques. Thus, users may decide the level of consistency that stashed copies must

have in order to meet the demands of their particular applications. We feel that this is an important feature with broad usefulness. It also liberates the system from the burden of keeping perfectly consistent copies in cases where perfect consistency is not required.

The goals that governed our design were fast prototyping, transparency at the user level, portability to a variety of computer systems and minimal modifications to the underlying implementation. These considerations led us to implement the initial version of FACE by modifying NFS, partly to simplify our task, and partly because the latter has become a *de facto* standard for sharing files in distributed environments.

After the first prototype has been fully tested, we plan to upgrade FACE by substituting the underlying UNIX file system by a file system currently being developed at Princeton (iPcress [Staelin1989]) that provides logging of users' accesses. This feature will help us design an automatic selection mechanism for choosing which files should be stashed.

References

Alonso1988a.

Alonso, Rafael and Luis L. Cova, ''Resource Sharing in a Distributed Environment,'' *Proceedings for the 1988 ACM SIGOPS European workshop*, Cambridge, England, September, 1988.

Alonso1988b.

Alonso, Rafael, Daniel Barbara, Hector Garcia-Molina, and Soraya Abad-Mota, ''Quasi-Copies: Efficient Data Sharing for Information Retrieval Systems,'' *Proceedings of the International Conference on Extending Database Technology*, Venice, Italy, 1988.

Alonso1988c.

Alonso, Rafael and Luis L. Cova, ''Sharing Jobs Among Independently Owned Processors,'' *Proceeding of the 8th. International Conference on Distributed Computing Systems*, pp. 282-288, Computer Society Press, San Jose, California, June 1988.

Alonso1989.

Alonso, Rafael and Daniel Barbara, ''Negotiating Data Access In Federated Database Systems,'' *Proceedings of the fifth Conference on Data Engineering*, Los Angeles, California, February 1989.

Barbara1989.

Barbara, Daniel and Richard J. Lipton, ''Randomized Technique for Remote File Comparison,'' *Proceedings of the 9th. International Conference on Distributed Computing Systems*, Computer Society Press, Newport Beach, California, June 1989.

Birrel1988.

Birrel, Andrew, ''Position Paper for the ACM SIGOPS Workshop 1988,'' *Proceedings for the 1988 ACM SIGOPS European workshop*, Cambridge, England, September 1988.

Birrell1982.

Birrell, A., 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.

BSD1986.

4.3 Berkeley Software Distribution, *UNIX User's Reference Manual,* USENIX Association, Berkeley, California, April 1986.

Howard1988.

Howard, John H., Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayan, Robert N. Sidebottom, and M. West, ''Scale and Performance in a Distributed File System,'' *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51-81, February 1988.

Kleiman1989.

Kleiman, S.R., ''Vnodes: An Architecture for Multiple File System Types in Sun UNIX,'' *Tutorial T6: Open Network Computing and NFS*, USENIX, San Diego, California, February 1989.

Leffler1989.

Leffler, Samuel J., Marshall Kirk McKusick, Michael J. Karels, and John S. Quaterman, in *4.3 BSD UNIX Operating System*, Addison-Wesley, 1st. edition, 1989.

Mullender1985.

Mullender, Sape J. and Andrew S. Tanenbaum, ''A Distributed File Service Based on Optimistic Concurrency Control,'' *Operating Systems Review*, vol. 19, no. 5, ACM, December 1985.

ONC/NFS1988.

SUN Microsystems, Inc., ''ONC/NFS Protocol Specifications and Service Manual,'' Part No. 800-3084-10, Revision A, of 26 August 1988.

PC-NFS1989.

Sun Microsystems, Inc., *PC-NFS User's Manual*, 1989.

Schroeder1985.

Schroeder, Michael D., David K. Gifford, and Roger M. Needham, ''A Caching File System for a Programmer's Workstation,'' *Operating Systems Review*, vol. 19, no. 5, December 1985.

Schroeder1988.

Schroeder, M.D., ''Autonomy or Interdependence in Distributed Systems?,'' *Proceedings for the 1988 ACM SIGOPS European workshop*, Cambridge, England, September 1988.

Staelin1989.

Staelin, Carl and Hector Garcia-Molina, ''The iPcress File System,'' Technical report, Princeton University Computer Science Department, Princeton, New Jersey, March 1989.

Tanenbaum1988.

Tanenbaum, Andrew S., *Computer Networks,* Prentice-Hall, 2nd. edition, 1988.

Terry1987.

Terry, Douglas B., ''Caching Hints in Distributed Systems,'' *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 48-54, January 1987.

Walsh1985.

Walsh, Dan, Bob Lyon, and Gary Sager, ''Overview of the Sun Network File System,'' *Proceedings USENIX Winter Conference 1985*, January 1985.

Zipf1949.

Zipf, George Kingsley, B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms, ''A Snapshot Differential Refresh Algorithm,'' *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 53-60, Addison-Wesley Press, Washington, D.C. , May 1986.