AUGMENTING AVAILABILITY IN DISTRIBUTED FILE SYSTEMS

Rafael Alonso
Daniel Barbara
Luis L. Cova

CS-TR-234-89

October 1989

Augmenting Availability in Distributed File Systems[†]

Rafael Alonso
Daniel Barbará
Luis L. Cova

Computer Science Department
Princeton University
Princeton, NJ 08544

ABSTRACT

A very important feature in distributed systems is the availability of the services offered by the system. Traditionally, distributed systems provide the services by centralizing their administration at remote servers. This leaves the door open for increased dependency on these servers and the network connecting them to the local facilities. This may be a problem in environments where availability is a concern, e.g., very large distributed systems. In this paper, we describe the design of a distributed file system called FACE, that uses the notions of **stashing** (keeping local copies of key information) and **quasi-copies** (data copies that are allowed to diverge from the primary data but in a controlled, application-dependent fashion), to augment the availability of crucial files even when the file server that contains them is no longer reachable. We also report on the first FACE prototype that has been designed and implemented via a series of enhancements to Sun's Network File System.

October 31, 1989

Augmenting Availability in Distributed File Systems[†]

Rafael Alonso
Daniel Barbará
Luis L. Cova

Computer Science Department
Princeton University
Princeton, NJ 08544

1. Introduction

Very often, distributed systems are structured in such a way that a service is transparently shared among the users. This usually is implemented by having a remote server, or group of servers, that perform some service for the user, who is unaware of their existence. Although this scheme has many advantages (e.g., centralized administration of the service), it results in total dependence on this server. Frequently, this translates to an inability to obtain the service if the communication with the remote server is not possible, as in the case of a network failure. This is true even in cases where the user has a powerful workstation on his or her desk that could be capable of performing the server's function, although perhaps with a degraded quality.

Thus, many server based designs reduce the role of a workstation to that of a terminal, used only to access the services available on the system. We believe that such a role is much too restrictive, failing to take advantage of the increasing power of these machines. Moreover, it decreases the control that a user should have over his or her resources, which could be the very reason that motivated the user to acquire the machine

---

† This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0456 and N00014-85-K-0465, by the National Science Foundation under Cooperative Agreement No. DCR-8420948, and New Jersey Governor's Commission Award No. 85-990660-6, and grants from IBM and SRI's Sarnoff Laboratory. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

in the first place.

A similar situation arises in the context of very large distributed systems (VLDS), i.e., systems that consist of a large number of processing units. We believe such systems will probably come into existence by the joining together of a number of separate smaller systems, often belonging to different organizations. 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 sub-system to join the **federation**. It does not seem likely to us that system administrators will abandon all control over their local systems to participate in a global one, regardless of how wonderful the benefits are.

To a user, autonomy means being able to continue working in spite of not being able to communicate with servers, albeit perhaps continuing in a degraded manner (e.g., *availability* of service). Seen in this light, it would seem that the notion of structuring a VLDS around the client-server model 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.

Although there is nothing intrinsically wrong with the notion of a server, we argue that a server should be the *best* place to get the service and never the *only* one. When the server can not be reached, the local machine, or nearby one, should be able to supply the service, albeit perhaps with lower quality. For instance, a computationally-bound task may be more efficiently executed in a remote server with a large computing capacity (e.g., a supercomputer). However, if that server is unavailable, the local computer is usually ready to take over and run the job, perhaps severely increasing the time required to finish it. (A degradation in the quality of service.)

In this paper we explore the issue of availability in distributed file systems. A file server may contain fast storage devices, the latest copy of all the information, archival storage, and guarantee frequent backups. But if the server is unavailable, users may be

happy having slow access to a local storage device containing hourly snapshots of at least some of their files (for instance, the most frequently used ones). The idea of saving local copies of key information for use during communication failures has been discussed in the past [Alonso88]. This technique has recently been referred to by the term **stashing** ([Birrell88],[Schroeder88]). However the actual details of maintaining a stash have not been discussed before. Most importantly, the idea of tuning the quality of the stashed data according to the needs of the application and the performance overhead that one is willing to suffer has not been studied before. We augment the usefulness of stashing by using the notion of **quasi-copies** [Alonso89], i.e., copies that are allowed to diverge from the primary data in a controlled, application dependent fashion.

In designing a distributed file system that supports stashing, the following issues must be addressed:

(1) File access: During normal operation (i.e., file server reachable), we have the option of directing the accesses to the stashed copy of the file or to always refer to the file server copy.

(2) Service quality: This involves a) selecting which files are vital for functioning when the access to the file server is cut off and b) deciding how consistent the data in these files is going to be. The more files we consider important, the more space we will have to dedicate to store them locally. The more consistent that we demand the data to be, the larger the performance cost that we will have to pay. For instance, if we demand perfect consistency, the local copies become replicated copies of the files in the remote server, and we have to preserve consistency by resorting to well known techniques (e.g., two-phase commit [Gray78])

(3) Version integration: In general, the local users can make updates to the data while the file server is unavailable. Since more than one user can be doing this, if we do not restrict updates, we will be faced with diverging copies in the moment of reconnecting the users to the file server. A mechanism is required to produce a single,

integrated copy of the file to store back in the file server.

The format of the rest of the paper is as follows. In the next section we explore in greater detail the issues presented above. In Section 3, we present the architecture of our distributed file system, called FACE. In Section 4 we discuss the first prototype of FACE and present some performance measurements. In Section 5, we present our conclusions and directions for future work.

2. Basic Concepts

In this section we present in greater detail the issues presented in the introduction. These are: file access, selecting the files that are to be stashed, selecting the consistency of the local copies, and integrating file copies.

2.1. File Accesses.

There are two ways of accessing stashed files, as Figure 2.1 shows. In Figure 2.1a, all user accesses are directed to the local disk, i.e., the stashed copy of the file. The updates are then propagated to the server, which in turn propagates them to other stashed copies. The ANDREW file system [Howard88], uses a similar configuration for caching purposes. In Figure 2.1b, accesses are always directed to the server. Updates are later propagated to the stashed copies, similarly to the primary site strategy, used in LOCUS [Popek85] Both alternatives are useful, depending on the type of files involved. The strategy of Figure 2.1a 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 2.1b is better for files that are to be shared among a community of users, and will be our choice for implementing stashing.

Finally, it is worth mentioning that although one is tempted to use the techniques of caching for purposes of stashing, these concepts should not be confused. Caching is done for performance considerations, i.e., in caching, the most frequently accessed information

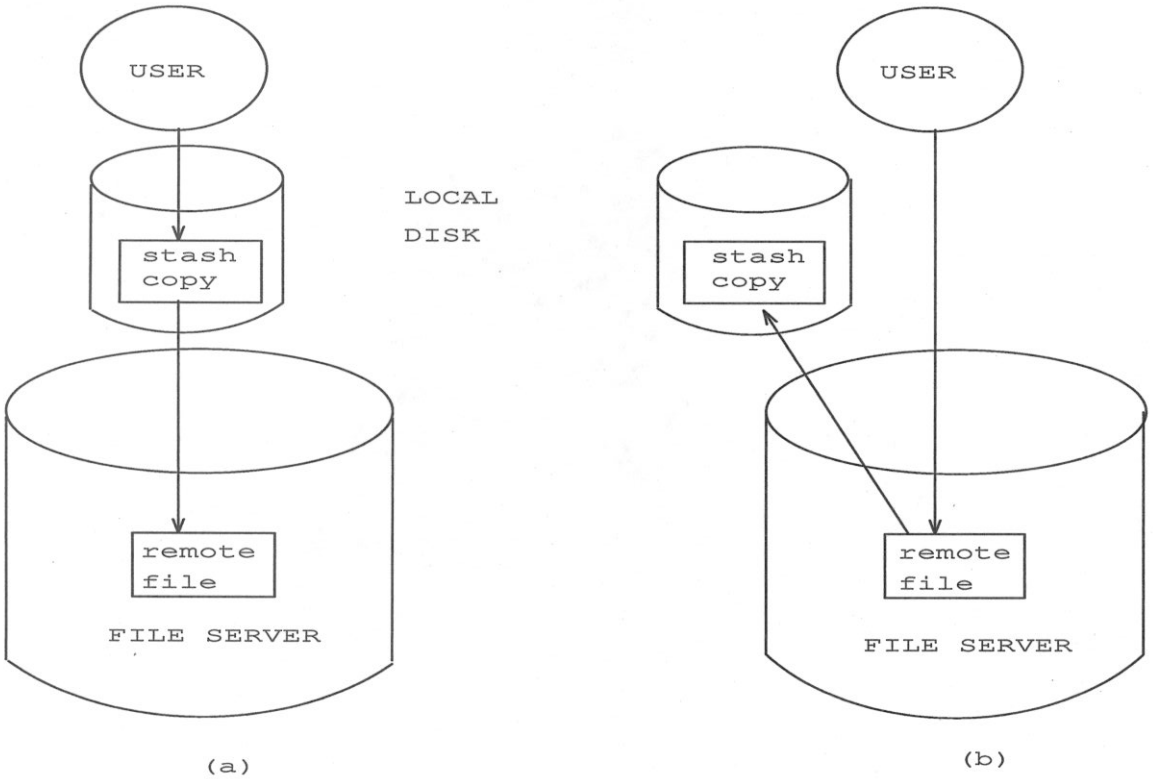

Figure 2.1: Accessing a remote file and its stashed copy

is saved. In stashing, one must save files for autonomy and fault-tolerance† reasons.

† 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.

2.2. File selection

In this section we discuss how does a user selects the files that will be stashed (or dropped from the stash). While considering this 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 names 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 [BSD86] 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. A way of implementing this facility in windowing systems is to associate a *stashing* attribute with each window. If the attribute is set a ''record_stash'' is issue on the window, thus stashing all the files touch by the commands entered in the window.

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. In the latter case, the stashing facility will be performing some of the tasks of a file caching mechanism. If there is such a caching facility already in place, then the stashing mechanism could take advantage of the cached files.

When the server is accessible, the users will be accessing the remote copy of the file. When there is a communication failure, the proper action would be to notify the users that a failure has occurred (say, by a message on their CRT screen). This will prompt most users to abort their current command and restart it immediately, using the stashed files this time. Telling the user about the failure avoids the confusion that could be caused by switching the file automatically to the stashed version. (Consider for instance a user in the middle of editing a file.)

2.3. Data Consistency

In this section we deal with the issue of the data consistency. Having files stashed can be thought as replicating them throughout the system. Obviously, this leads to the issue of mutual consistency among the copies. One principle we have to keep in mind is that the more consistent that we demand the copies to be, the higher the performance overhead that we will incur in maintaining the consistency.

While the server is reachable by all clients, the updates are all being centralized at the central copy of the file. (This is a consequence of selecting the alternative on Figure 2.1b, since in this case, all the file accesses are being done at the file server.) Even in this situation we still have the alternative of deciding what degree of consistency the stashed files are going to have. Notice that this will be the degree of consistency (or data quality) that the clients will see in the files at the moment in which the file server becomes unavailable, i.e., this will be the starting point for operation with the stashed files when the file server is no longer available. The related problem of diverging copies after the file service becomes unavailable is treated in the next subsection, when we describe the work that needs to be done after the reconnection, i.e., the reintegration of the copies. Here we only talk about consistency of the stashed files during normal operation (i.e., while the file server is reachable),

If one wants to have perfect consistency for the copies at all moments, i.e., to 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 produced at the server, all the copies should be locked and the update propagated to them. This is costly to perform. Certainly in VLDS (where the possibility of thousand of copies exist) this is an impractical solution. 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, he or she may be content with having a version that is several minutes old. Another example in which the inconsistency is tolerable can be a

system that deals with graphical information that changes dynamically. We may have several processors working in different parts of a picture, and reflecting the changes in a file kept at the file server. If we have stashed copies of the file that represent the picture in our local facility, at the moment that the file server becomes unavailable we certainly can tolerate missing some of the updates (the eye will probably not be able to notice the difference), and have a relaxed degree of consistency in exchange for less overhead in maintaining the copies. Certainly, if the application requires perfect data, we must be prepared to pay the price of using a protocol that maintains the stashed copies consistent with the server copy at all times.

To let the application decide what is the maximum inconsistency that can be tolerated, we can use quasi-copy techniques. Quasi-copies were designed originally 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 file may be client maintained for some of the copies and server maintained for others. It is worth pointing out that, since we use quasi-copies to deal with consistency, the alternative shown in Figure 2.1b is the

most natural choice because there all updates occur at the server.

There are several types of predicates that are useful in keeping quasi-copies of files. For the following definitions let x be a file at a file server and x' a stashed copy of x. Let $x(t)$ be the file contents at time t. Let $v(x(t))$ be the version number of file x at time t.

(1) *Delay Condition.* It states how much time a stashed may lag behind the file server copy. For file x, an allowable delay of α is given by the condition

$$\forall \text{times } t \geq 0 \exists k \text{ such that } 0 \leq k \leq \alpha$$
$$\text{and } x'(t) = x(t-k)$$

Since this defines a window of acceptable value, we use the notation $W(x) = \alpha$ to represent this condition.

(2) *Version Condition.* A user may want to specify a window of allowable values, not in terms of time, but of versions. For example, if a file represents a VLSI circuit, it may be useful to require a copy that is at most 2 versions old. We represent this condition as $V(x) = \beta$, where β the maximum version difference. That is , $V(x) = \beta$ is the condition

$$\forall \text{times } t \geq 0 \exists k, t_0 \text{ such that } 0 \leq k \leq \beta$$
$$\text{and } 0 \leq t_0 \leq t$$
$$\text{and } v(x(t)) = v(x(t_0)) + k$$
$$\text{and } x'(t) = x(t_0)$$

(3) *Number of changes.* Here the maximum number of updates missing in the stashed copy is bounded by the user. We represent this condition as $U(x) = \varepsilon$, where x is the file and ε is the maximum number of updates missing.

Of course, these conditions are only valid while the file server is accessible. Moreover, transmission delays should be taken into account when enforcing these conditions from the file server, i.e., in the case of sever maintained consistency. To illustrate, consider the condition $U(x) = \varepsilon$ and assume that an update is about to occur in the file server that would violate the condition, i.e., $\varepsilon + 1$ updates will be missing from some stashed copy. Hence the update to the image must be performed ''at the same time'' as the object is changed. Strictly speaking, this is not possible. Since we want to avoid using a 2-phase commit protocol (or similar strategy), we propose interpreting every condition $C(x)$ on file x at server j as

$$C(x) \quad \vee \quad W(x) = \delta \quad \vee \quad S\_FAILED(j)$$

This means that all conditions have an implicit delay window of δ, where $\delta \geq T_D$, the maximum transmission delay, and also that conditions do not have to be enforced if the server is down ($S\_FAILED(j) = TRUE$).

Of course, if the stashed copy is client maintained, none of these problems arise. The client machine will be responsible for asking for a fresh copy of the file when the delay condition is about to become false. The delay in the condition will have to take into account the maximum transmission time plus the maximum response time of the server to a service request (server's time-out). This strategy leaves the file server oblivious to the existence of the stashed copy, offloading the work to the client.

Some optimizations can be made to reduce the overhead of sending files over the network, and to improve the quality of the data stashed. The first one is to make use of cached data, if a file cache is maintained in the distributed file system for performance reasons. Very often, the cache contains the latest modifications made to the file in the local facility. We could copy the contents of the cache to the stash, whenever a local user has made a modification to the file, thereby improving the quality of the data stashed. Secondly, if the stashed files are client maintained, the client can remember the last time it accessed it. When requesting a fresh copy, this time can be included in the request. The

server will then compare this timestamp with the actual time at which the file was modified, thus determining whether sending the file is necessary or not. Finally, to avoid sending large files, one could break the file into segments and apply file comparison techniques ([Barbara89]) to find out which segments have changed since the last time the copy was send, and send only those segments.

2.4. Data Integration

Stashing is useful when the server is no longer available. At this point the client sites with stashed copies will begin to use them, possibly making updates. After the server is once again available, we need to worry about reintegration of the possible diverging copies that coexist in the system.

One trivial way of doing this would be to allow updates to occur in a single group of copies. This group can be composed by the site holding the file server and all the sites that kept connected with it, or if we truly want to operate after server failures, a group of local sites that remained connected among them. (Perhaps making sure that this group always contains the owner(s) of the file.) This strategy could be implemented using voting schemes ([Gifford79],[Barbara86]), but it would be far too restrictive for our purposes, since many local facilities could only use their stashed data for read-only purposes. This strategy corresponds to a *pessimistic* approach to recovery.

Contrary to the pessimistic strategy, we could use an *optimistic* recovery mechanism, allowing copies to diverge, and reintegrating them afterwards. The related problem in distributed database systems has received a lot of attention in the past ([Davidson85]), and some of the solutions are applicable to our problem, with the difference that instead of transactions, we deal with individual actions to the involved files. One could construct dependency graphs similar to the ones used by Davidson ([Davidson84]) and analyze them afterwards, rolling back some of the actions taken to break the conflicts. Or one could devise a patching mechanism [Garcia-Molina83] that allows the operations to be

merged into a final integrated file. But whatever solution is used, it is clear that we need to keep a log in all sites involved registering all operations that have been done to the file. In the file server, all the updates that have been made to the files and are not reflected in some stashed copy yet have to be in the log. In the clients, all operations done after losing contact with the server have to be registered.

Our intention is to implement an application dependent solution for this problem. That is, knowing the semantics of the application involved, we can analyze the logs of the different sites that made updates to the file and find out the set of operations that are conflictive. Once we do this, we will take the approach of [Garcia-Molina83] in ''patching'' the file. For this, we could a priori define a table of actions to be taken when two conflicting operations are done over the same part of the file and apply these actions until no more conflicts remain. Of course, the involved users would have to be notified of the decisions taken. Alternatively, the system could ask for human intervention, presenting the users with the conflicting actions and requesting the procedure to be follow to reintegrate. These requests can be made to the community of users that updated the file, or to a central authority. We are still doing research on this area.

3. Face Architecture

In Figure 3.1 we show a diagram of FACE's general design. There are four main components in this architecture: the distributed file system interface, the bookkeeper processes, the reintegration module, and the user-level tools (on the client side only). The distributed file system interface directs user file accesses to the appropriate underlying file system (i.e., local or remote). It also implements the system support for stashing. The client and server bookkeeper processes provide the runtime support for keeping the stashed copies of remote files within the user consistency constraints (recall the earlier discussion of client maintained and server maintained predicates). The reintegration module implements the functions presented in section 2.4, i.e., an optimistic recovery

mechanism to allow the merging of divergent stashed copies. Finally, the user-level tools provide the facilities discussed in section 2.2, which allow users to select which files to stash. In the following subsections we will discuss in more detail the distributed file system interface and the bookkeepers. We omit discussion of the integration module and user-level tools for two reasons: lack of space, and that although both subsystems have been designed, they have not yet been implemented.

3.1. Distributed File System Interface

When designing a distributed file system that supports stashing one decision that has to be made is where should the added functionality be implemented. To make this decision we have to take into account our design goals, which follow from the federate environment we are considering: scalability, local autonomy (which translates to minimal modifications to the local systems), transparency to user level processes, and portability to a variety of computer systems (heterogeneity).

Figure 3.2 shows the different layers in the file name translation process. For the first layer, *file name*, there are different naming conventions among operating systems. Therefore translation mechanisms from file names to UFID (unique file identifiers) 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. 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 *cache* also differs from implementation to implementation. In some cases (CFS [Schroeder85], ANDREW [Howard88], AMOEBA [Mullender85]) there can be a file oriented cache instead of the more traditional block cache. Forcing the caching system to handle stashing as well would not only require major modifications to the operating

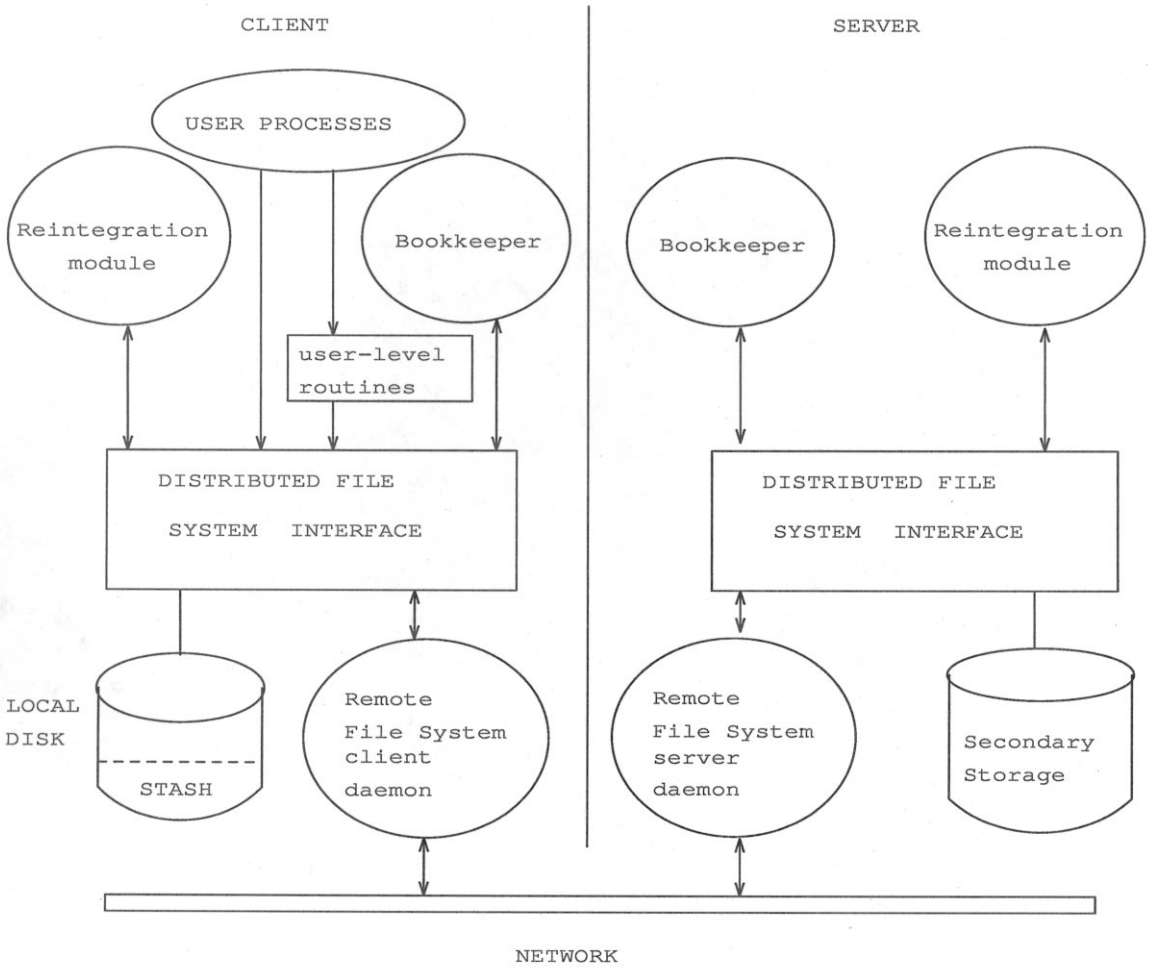

Figure 3.1: FACE general design

system, but may also result in a negative impact on the cache performance.

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

The Unique File Identifier (UFID) layer provides a standard abstraction of the file throughout the different heterogeneous file systems in the network, and is thus the right

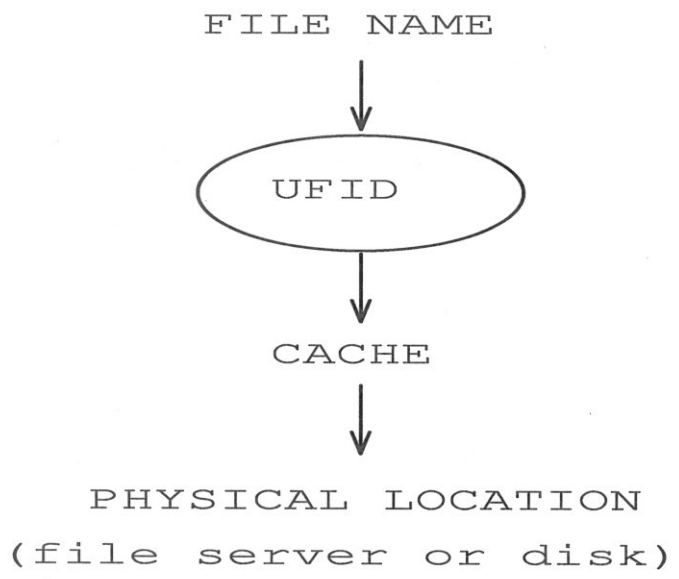

Figure 3.2: Name translation process

layer to implement stashing. In the Sun Network File System (NFS) for instance, this layer is implemented by the use of the *vnodes* numbers, an extension of the UNIX † *inodes* that corresponds of a triple: *(computer-id, file store number, inode number).*

To support stashed copies of remote files we extend the data structure that the distributed file system interface has representing remote files' UFID. Two new data fields are associated to this data structure. The first is a pointer to the UFID corresponding to the stashed copy. This UFID belongs to the local file system that the client uses as the stash space. The second field saves the quasi-copy predicate associated with the stashed file. Figure 3.3 shows the name translation process of figure 3.2 when the remote file has a

† UNIX is a trademark of A.T.&T.

stashed copy.

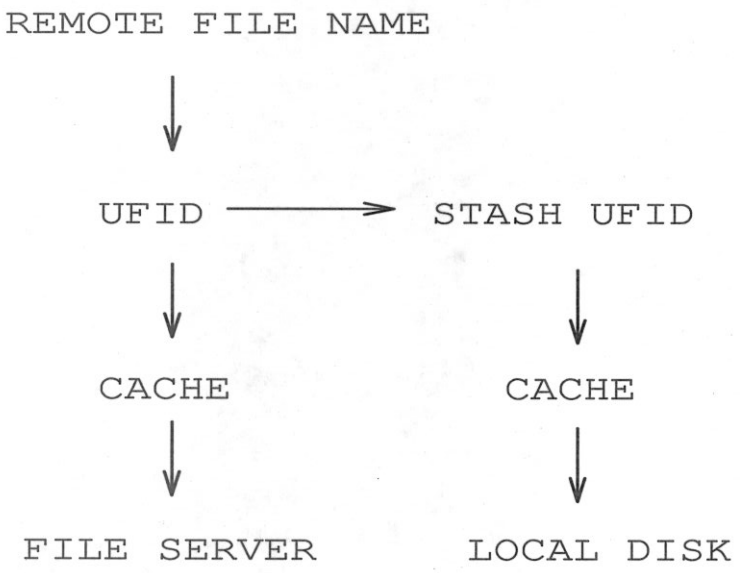

Figure 3.3: Modified name translation process to support stashing

The routines in the distributed file system interfaces have to be able to select, depending on the status of the network and the file server, which UFID to use for a given file name. Each file operation has to be implemented following the structure of figure 3.4.

There are also specific stashing routines in the distributed file system interface. The first two handle the linking and unlinking of remote files' UFID with the corresponding UFID of the stashed copies. Two others are used for allocating and deallocating the data structures for the UFID of the stashed copies in the local file system's stash space. There are also new procedures for dealing with client disengagement and re-engagement from the file server: informing the users that the file server is not available, that stashed copies

```
<file operation> :
    IF (UFID represents a remote file) AND
        (stash UFID is not NULL) AND
        (there is no connection to the file server)
    THEN
        (use the file operation corresponding to the stash UFID )
    ELSE
        (use the file operation corresponding to the remote file UFID)
```

Figure 3.4: General structure for file operations (in pseudo-code)

are in used, and of how many other users where accessing the remote files (that are stashed) at the moment of disengagement. Finally there are the entry routines to the interface (e.g., system calls) to allow users to **stash**and **unstash** files.

3.2. Bookkeeper processes

Client and server bookkeepers are processes that work with the distributed file system interface to provide the quasi-copy support for the stashing facility. The distributed file system interface, at the client or server side, informs the corresponding bookkeeper of what files the user has stashed and provides the consistency constraint associated with each file.

The management of a quasi-copy coherence will be performed in general by the server bookkeeper, although in the case of delay predicates, there is the option of unloading this task to the client bookkeeper. Regardless of which side maintains the consistency, the actions taken are the same. When the coherence condition is violated, the bookkeeper intervenes and updates the stashed copy from the remote file's content.

At the server side, if the coherence is other than a delay condition, in principle, each update to the remote file has to be intercepted to check whether the condition will be violated or not, a possibly expensive task. However this requirement can be circumvented if the server bookkeeper can be built as an internal module of the distributed file system interface and not as an external (user-level) process.

At the client side, the bookkeeper only needs to maintain expiration times for each of its remote files being stashed with client-maintained predicates. The information can be kept in a time ordered list of UFID of these remote files. 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 back them up 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. As opposed to the server bookkeeper, the client bookkeeper can be implemented as an user-level process, external to the distributed file system interface.

4. The FACE Prototype

The first FACE prototype has been implemented on the facilities of the Princeton Distributed Computing Laboratory. The system was developed on a Sun 3/50 computer running Sun's UNIX 4.2 release 3.3 (SunOS 3.3) and based on NFS [Walsh85].

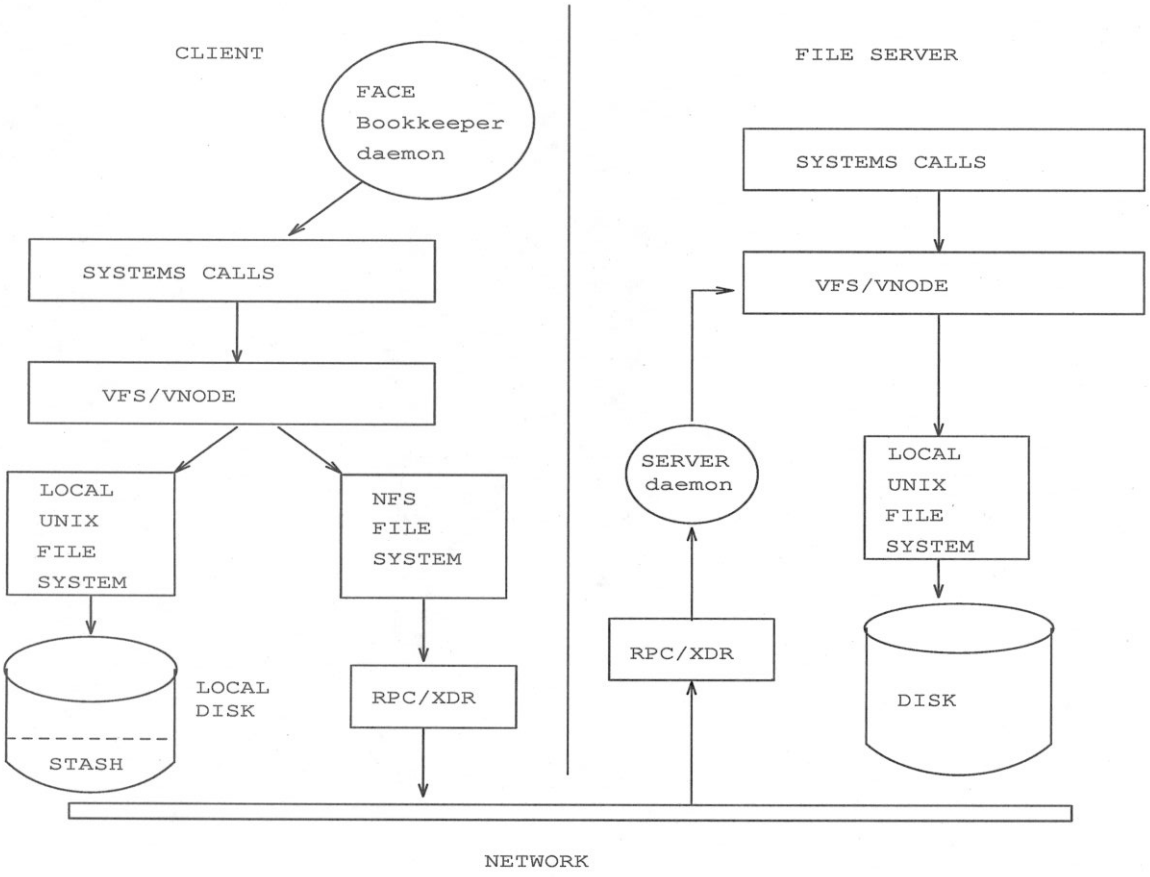

Figure 4.1: Block diagram of the FACE prototype based on NFS

The prototype is implemented to direct all file accesses to the file server during normal operation. The implementation was was made at the vnode layer [Kleiman89], being this layer the one with a widespread accepted standard. The implementation uses the client maintained strategy (see section 2.3) for keeping the consistency of the stashed files bound to the application needs. The conditions used are, of course, delay conditions. The integration module is not currently implemented.

Figure 4.1 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.). New system calls were introduced to **stash** files, and to **unstash** them. These system calls are privileged and are utilized by users through the library routines *mkstash* and *rmstash*, respectively. Also, we incorporated a bookkeeper process that maintains the consistency constraints (i.e., the quasi-copy predicates) between the remote files and the stashed copies.

file type:		remote		local		stashed copy	
kernel	operation	mean	st.dev	mean	st.dev	mean	st.dev
FACE	write	4.354	0.394	3.249	0.064	3.251	0.202
	read	1.664	0.122	1.401	0.018	1.401	0.021
	open-close	11.092	0.343	2.223	0.004	2.116	0.004
SunOS 3.3	write	4.334	0.353	3.256	0.027	not	
	read	1.698	0.058	1.406	0.018	supported	
	open-close	11.503	0.370	2.153	0.047		

Table 4.1. Comparison of system calls overhead (in milliseconds)

To evaluate the overhead caused by our code, we have measured the time to perform file system calls - open-close, read, write ([BSD86]) - in three different situations:

first with a file stored at the server (remote file), then with a file stored locally (not a stashed copy), and finally with a stashed copy of a remote file. The tests were made with the FACE kernel and the SunOS 3.3 one (without the FACE modifications). The tests consisted in 100 samples of 10,000 iterations of calling each of the system calls for a 2.5 Mbytes file. Table 4.1 shows the times (in milliseconds) that we measured. These results show that the added overhead introduce in FACE to support stashing are negligible compared to the based kernel (SunOS 3.3).

The second set of tests reported in Table 4.2 show the response time of executing a *mkstash* library routine and a *rmstash* library routine. These routines allow users to set a stashed copy of a remote file and to remove the copy, respectively. *Mkstash* invokes the **stash** system call to allocate the kernel structures corresponding to the remote file (vnode and rnode), to associate a local vnode with the remote vnode, and to allocate the local inode for the stashed copy. It then copies the remote file's content into the stashed copy†, and passes a handle - representing the stashed remote file - to the client bookkeeper. *Rmstash* invokes the **unstash** system call to remove the association between the vnode representing the remote file and the stashed copy. It them sends the bookkeeper a message indicating the the remote file is no longer stashed, and removes the stashed copy from the local disk.

The performance measurements where gathered by executing a thousand times *mkstash* followed by *rmstash*, in an otherwise empty ethernet between a Sun 3/50 acting as client and a Sun 3/180 acting as file server. There are two modes for each of the stashing routines: synchronous or asynchronous to the calling process. As table 4.2 shows, the response time of the *mkstash* routine in the asynchronous case depends on the size of the remote file. The main part of this cost is attributed to the copying of the remote file's content into the stashed copy. In the asynchronous case the *mkstash* routine allows the user

† Note that the NFS protocol is using a 2 Kbytes block size for transfers between the client and the server.

to do this first backup asynchronously from the requesting user process. The response time of the *rmstash* routine in the synchronous mode also depends on the size of the stashed copy, but in a lesser manner. Most of this cost is due to the deallocation of the data blocks pointed by the stashed copy's inode. As with the *mkstash* routine, in the asynchronous option the deallocation of the data blocks is done asynchronously from the requesting process.

file size	*mkstash*		*rmstash*	
	mean	st.dev	mean	st.dev
asynchronous	464.1	108.8	409.7	88.0
256 Bytes	456.1	78.4	489.6	120.4
512 Bytes	468.4	152.2	489.3	135.0
1 Kb	481.8	181.5	492.7	141.7
4 Kb	509.9	109.1	492.4	135.9
8 Kb	533.5	87.5	484.7	94.7
500 Kb	3,969.1	613.9	1,019.5	232.2
1 Mb	7,414.9	1,402.4	1,074.9	211.7
2 Mb	13,165.7	1,261.9	1,202.2	221.2

Table 4.2. Response time for stashing routines (in milliseconds)

5. Conclusions

In this paper, we have presented a file system design that allows the user to keep local copies of important files, decreasing the dependency over file servers. Using the notions of *stashing* and *quasi-copies*, the system allows users to tune up the quality of the service they want to receive when the file server is not reachable. We feel that one of the key points of this work is our focus on the tradeoff between availability and degradation of service. The other main contribution in this research is the design of a distributed file system which is ideally suited to VLDS's, in that it enables users to maintain local autonomy while they cooperate with others. This file system architecture has been implemented and performance figures are reported herein. These figures show that the overhead of providing the service is negligible.

The present work has been carried out as part of the **ACE** (Autonomous Cooperating Environments) project at Princeton University, and there are a number of related projects currently under way. Among them is the development of a facility for finding resources in a VLDS, which combines some concepts of name servers and multidatabases. Another involves a prototype of an information gateway that will allow a number of heterogeneous databases to share information while preserving the system administrator's flexibility to decide on software, schemas, etc. A third encompasses work on load sharing techniques that will enable users to specify how much of their resources they are willing to share with others.

While our current implementation is now complete, further work remains to be done. We described above some of our ideas for data integration (i.e., integrating two or more stashed files that have been modified during a partition). We are presently developing a tool to aid in that integration. Our current plans also call for porting our design to a number of other processors, and installing the system in some of the machines in our department.

References

Alonso88.

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

Alonso89.

Alonso, Rafael, Daniel Barbara, and Hector Garcia-Molina, ''Data Caching Issues in an Information Retrieval System,'' (To appear in) *Transactions on Database Systems*, 1989.

Barbara86.

Barbara, Daniel, and Hector Garcia-Molina, ''Mutual Exclusion in Partitioned Distributed Systems,'' *Distributed Computing*, Springer-Verlag, 1986.

Barbara89.

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.

Birrell88.

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

BSD86.

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

Davidson84.

Davidson, Susan, ''Optimism and Consistency in Partitioned Distributed Database

Systems,'' *Transactions on Database Systems*, vol. 9, no. 3, ACM, September 1984.

Davidson85.

Davidson, Susan, Hector Garcia-Molina, and D. Skeen, ''Consistency in Partitioned Networks,'' *Computing Surveys*, vol. 17, no. 3, ACM, September 1985.

Garcia-Molina83.

Garcia-Molina, Hector, Tim Allen, Barbara Blaustein, R. Chilenskas, and Daniel R. Ries, ''Data-patch: Integrating Inconsistent Copies of a Database after a Partition,'' *Proc. Third Symposium on Reliability in Distributed Software and Database Systems*, October 1983.

Gifford79.

Gifford, D.K., ''Weighted voting for replicated data,'' *Proc. Seventh Symposium on Operating Systems Principles*, December 1979.

Gray78.

Gray, J., ''Notes on Database Operating Systems,'' *Operating Systems: an Advance Course, Lectures Notes in Computer Science*, vol. 60, pp. 394-481, Springer-Verlag , R. Bayer, R. Graham, and G. Seegmuller (editors), 1978.

Howard88.

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.

Kleiman89.

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.

Mullender85.

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.

Popek85.

Popek, Gerald, and Bruce Walker (editors), *The LOCUS Distributed System Architecture,* MIT Press, 1985.

Schroeder85.

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.

Schroeder88.

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

Walsh85.

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