CMU-ITC-83-008

Report of the ITC File System Subgroup

21 August 1983

M. Satyanarayanan

Information Technology Center
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Preface

The material presented here represents the consensus of a subgroup of the the ITC Common System Services group. This subgroup was given the charge of studying the trade-offs presented in previous debates of the CSS group, and coming up with a consistent, low-risk file system design.

The members of the subgroup were: Dave Gifford, Dave McDonald, Rick Rashid, M. Satyanarayanan, and Alfred Spector.

It is important to stress that this is a consensus design. Each member of the subgroup was at odds with individual aspects of the design, and felt that some things should be done differently. However, there is unanimous agreement that the design, as presented here is consistent, usable, and implementable with a minimum amount of risk.

The discussion of the design falls into two parts: the base design and refinements. The latter typically constitute optimizations or alternatives which are consistent with the base design, but are not essential for the functioning of the system. The decision to incorporate these (and other) refinements can be deferred until the base design has been implemented and used. The approach presented here thus provides an orderly path for enhancing performance and functionality, beginning with a simple implementation.

This document does NOT purport to be a complete design document. A number of open issues are identified at different points in the document. These and other design decisions have to be resolved before the skeleton presented here is adequate to serve as the basis for an implementation.

Finally, in an attempt to be as focussed as possible, this document ignores the issues of local disk servers, paging servers, bboards, mail, and accounting. Since they are likely to have a close relationship with the the file system, a detailed examination of these issues is in order.

1 Introduction

The ITC effort involves two relatively independent, but cooperative entities. The first of these is a collection of processing nodes connected together by a local area network. This ensemble is called VICE, and its purpose is to provide campus-wide access to shared resources. The other component is a specific type of workstation, for which the ITC will develop software. This is referred to as the VIRTUE system.

Each VICE node has a local area network to which are attached autonomous workstations. This family unit is referred to as a *Cluster*, and the trusted node is referred to as the *Cluster Server*. For security reasons, only VICE applications are run on cluster servers; no non-VICE applications are run on them. The hardware and software of the attached workstations may differ, but they can all use the services provided by VICE if they support the underlying communication and application protocols.

Within this framework, there are two file system interfaces: the VICE interface and the VIRTUE interface. The VICE interface is used by VIRTUE file systems, while the VIRTUE interface is used by application programs on workstations. Non-VIRTUE workstations may use the VICE interface as they see fit.

The design presented here makes only a small number of assumptions about the network, and about the hardware and software at workstations and cluster servers:

1. There is full logical connectivity in the network. Each workstation can access and use the services of any cluster server.

2. Each workstation possesses local secondary storage and has a file system of its own. To support workstations which do not have local secondary storage, VICE may provide a disk server. From our point of view, however, it is immaterial whether a workstation's local secondary storage is real or virtual.

3. We are only concerned with file accesses here, and explicitly ignore remote paging traffic from workstations without swap spaces of their own. If VICE provides a remote paging server for each cluster, it will be orthogonal to and independent of the file system design presented here.

4. The transport network and cluster server hardware is trustworthy and reliable. Monitoring tools are available to detect when network segments or individual cluster servers are down, and manpower is available to perform the necessary reconfiguration to maintain service to users.

2 The VICE Interface

To a first approximation, the VICE file system may be viewed as a very large timesharing file system with Unix-like concepts of files, directories, and a hierarchical naming structure.

The hierarchical file name space is partitioned into disjoint subtrees, and each such subtree is served by a single cluster server, called its *Primary Custodian*. Storage for a file, as well as the servicing of requests for it, are the responsibility of the corresponding primary custodian. Certain subtrees which contain frequently read, but rarely modified files, may have read-only replicas at other cluster servers. These read-only sites are referred to as *Secondary Custodians* for the subtree, and each of them can satisfy read requests independently. Write requests to that subtree have to be directed to the primary custodian of the subtree, which will perform the necessary coordination of secondary custodians. The unqualified term "custodian" refers to both primary and secondary custodians.

The model here is that the VICE is the master of all files stored in it, and that cached workstation copies are subordinate. However, a workstation may lock a file for reading or writing. In that case, the workstation possesses temporary mastery of that file, and need not check with VICE on individual accesses to it. Mastery always reverts to VICE in the long term; either by timeout or by explicit unlocking of the file.

2.1 VICE System Calls

In the minimal system, the VICE file system calls are:[1]

Login(*id*, *authentication information*)
One has to be logged in to a cluster server before using it.

Logout( ) Obvious.

WhereIs(*pathname*, *custodian list*, *prefix used*)
Returns the list of custodians for *pathname*, as well as the briefest prefix of *pathname* for which the *custodian list* is the same. The caller can use the *prefix used* information in future to avoid unnecessary WhereIs( ) calls.[2]

Lock(*pathname*, **READ|WRITE**)
Fails on conflict. Is automatically broken if partition occurs. Write locks should be

[1]This description uses a remote procedure call model, with call-by-value-return parameters. It is assumed that appropriate rights are possesed by the caller for each of the following operations.

[2]The order of entries in the custodian list may be significant. For example, the first entry of *custodian list* may be the primary custodian; or it may be the custodian closest to the the workstation.

directed to the primary custodian of *pathname*. Read locks may be directed to any of the custodians.

Unlock(*pathname*) Obvious. Should be directed to the custodian to which the corresponding lock request was directed.

Fetch(*pathname*, *file contents*)
Return the contents of the file at *pathname*. Action consistency guaranteed. *Pathname* must be a file. May be directed to any custodian of *pathname*.

Store(*pathname*, *file contents*)
Creates or overwrites the file at *pathname*. Action consistency guaranteed. *Pathname* must be a file. Must be directed to the primary custodian of *pathname*. The file must be write-locked by this workstation.

GetFileStat(*pathname*)
Returns meta-information about the file at *pathname*. If *pathname* is a directory, returns meta-information about all its immediate children. The available meta-information is specified in Appendix I.0.1

SetFileStat(*pathname*)
Changes meta-information about the file at *pathname*. For example, the protection on a file can be changed. *Pathname* may be a file or a directory.

Remove(*pathname*)
Deletes the file at *pathname*.

MakeDir(*pathname*)
Create the directory *pathname*.

RemoveDir(*pathname*)
Deletes the directory *pathname*.

Link(*source pathname*, *target pathname*)
Inserts a symbolic link from *source pathname* to *target pathname*.

Unlink(*pathname*) Removes the symbolic link at *pathname*.

3 VICE Implementation

In this design, the file system component of a VICE cluster server may be implemented on top of an unmodified Unix system. Each cluster server has a mechanism to map VICE pathnames for which it is a custodian to file names in the local Unix system.

The WhereIs( ) request is answered by looking up a relatively static database which maps VICE subtrees to custodians. This database is replicated at each cluster server, and updates to it occur relatively infrequently — typically once a day, rather than on a second-by-second basis. The simplest implementation would lock both the entries that are to be moved and the associated files for the duration of the transfer. Changes to the database are most likely to occur either because users are added and deleted, or because of attempts to redistribute the load on different cluster servers.

If workstations experience differential access times to different cluster servers, it is advisable to place the files most often used by a workstation on the cluster server to which accesses are the fastest. For example, the "/usr" subtree, containing all the files of a user should be on the cluster server most accessible to the workstation he most often uses. Information needed by the VICE operations staff to make and modify such placement can come from two sources: by explicit human action (for example, from a user himself when he moves, or from the campus administration), and from performance monitors embedded in VICE.

VICE does not support physical links, since they are meaningless across machines without some notion of universal low-level file identifiers. Cross-machine symbolic links pose no logical problem, but will impose a performance penalty unless workstations cache the symbolic mappings.

System files are replicated, and every cluster server is a secondary custodian. When a system file is updated, the primary custodian has to lock it at all the secondary custodians.

There are no cluster-initiated communications with workstations, with the possible exception of information messages such as "System will become unavailable in 5 minutes".

To rapidly service requests from a workstation to check whether they have the latest copy of a file, each cluster server can keep an in-core cache of recently modified filenames and timestamps.

3.1 VICE Refinements

1. A workstation may check out an entire subtree from VICE, assuming that it possesses appropriate rights. In that case the workstation possesses temporary mastery of that subtree, and need not check with VICE on individual accesses to cached files in that subtree. Mastery of the subtree always reverts to VICE in the long term; either by timeout or by explicit checkin of a subtree. Checking out a subtree has the same semantics as write-locking its root and every file in it.

   Fetches from checked out subtrees are possible, but are marked as being dubious.

   The corresponding VICE calls are:

   Checkout(*pathname*)
   Request temporary mastery of the subtree rooted at *pathname*.

Checkin(*pathname*)
Relinquish mastery of *pathname*.

2. We expect a very common sequence of actions to be to lock a file, find its date of modification, compare it with that of a cached copy, and to fetch a new copy if the cached copy is stale. A conditional fetch request can be used to combine these actions into a single request/response pair:

   CFetch(*pathname*, *boolean expression*, READ|WRITE)
   Lock file in specified mode. Evaluate *boolean expression*. If TRUE then retrieve *pathname*. *Boolean expression* may contain references to meta-information about *pathname*. *Pathname* should be a file.

3.2 Undecided VICE Issues

The following questions need to be resolved:

1. When locking, should VICE allow Multiple Readers AND one Writer, or Multiple Readers OR one Writer?

2. When multiple custodians are available for a subtree. should VICE recommend a particular one, since it has more information about traffic and loading information?

3. What special features are needed to make cross-custodian symbolic links efficient?

4. *I am sure other questions will turn up!*

4 VIRTUE Interface

The VIRTUE file system is an autonomous file system with a naming structure and functionality adequate to support single-site Unix applications. When connected to VICE, the VIRTUE file system allows transparent access to files in the VICE file system. With one exception, the VIRTUE file name space is identical to that of VICE. The exception is the subtree "/local", which contains files strictly local to the workstation. Symbolic links from the VICE name space to files in "/local" may be used to achieve workstation-sensitive file name interpretation.

4.1 VIRTUE System Calls

The following functions (basically the standard Unix calls) are available to application programs on a VIRTUE workstation:

1. **Open**(*pathname*, READ|WRITE)

2. **Close**(*Open file descriptor*)

3. **Create**(*pathname*)

4. **Read**(*Open file descriptor*, *no of bytes*)

5. **Write**(*Open file descriptor*, *data*, *no of bytes*)

6. **LSeek**(*Open file descriptor*, *offset*)
   Returns new position. Also used to find current position.

7. **GetFileStat**(*pathname*)

8. **SetFileStat**(*pathname*)

9. **Remove**(*pathname*)

10. **MakeDir**(*pathname*)

11. **RemoveDir**(*pathname*)

12. **Partition**( )
    Voluntarily partiton oneself.

13. **Unpartition**( )
    Remove voluntary partition.

14. **SetCacheMode**(UNUSABLE | READ-ONLY | READ-WRITE)
    Usually used only when partitioned. Specifies how cached file copies are to be handled.

5 VIRTUE Implementation

The VIRTUE file system is implemented on top of a standard Unix file system, and uses the latter mainly as a low-level mechanism to store and retrieve files from local secondary storage. Read, Write, and Seek requests by application programs are directed by VIRTUE to files on local secondary storage — only Open and Close requests may involve data transfer between VICE and VIRTUE. Such transfers are, however, completely transparent to the application programs.[3]

5.1 Caching Files

A file physically stored on VIRTUE workstation belongs to one of two classes: it is in the directory "/local" and hence outside the purview of VICE, or it is a cached copy of a file in VICE. The caching of files is primarily intended to minimize the volume of data communicated between VICE and VIRTUE. Ensuring the validity of cached copies is the responsibility of VIRTUE.

When servicing an open request, it is VIRTUE's responsibility to:

[3] We are likely to support remote Open of files in a later version of the system — however, that is outside the scope of this document.

1. Request VICE to lock the file in the appropriate mode.
2. Check its cache to see if it possesses a copy of the file. If so it has to check with VICE to ensure that the copy is not stale.
3. Retrieve a fresh copy if the cached copy is stale, or if no cached copy is present.

To service a Close request, VIRTUE must:

1. Write the cached copy back to VICE, if it was modified.
2. Request VICE to unlock the file, if no other process on the workstation is using it.
3. Mark the file Suspect.

At any instant of time, the cached copy of a file can be one of the following states:

Guaranteed	VICE has already locked the file on behalf of this workstation. In this state the cached copy of the file is the master copy. References to this file need not go to VICE. If modified since the fetch from VICE, the copy is marked Dirty (see below).
Suspect	We have a cached copy of a file, but are not sure that it is the latest copy. This can occur either because the workstation has not requested VICE to lock the file, or if the lock on a Guaranteed copy is lost because of partition.

An orthogonal attribute indicates whether the cached copy of a file has been modified since the most recent check for it with VICE:

Clean	No changes have been made. VIRTUE can request VICE to unlock a Guaranteed and Clean file without any data transfer.
Dirty	The copy has been modified. If the copy is Guaranteed, VIRTUE must store this copy back to VICE, before unlocking the file. If the copy is Suspect, a merge algorithm (Section 5.4) must be run to ensure that more recent updates to VICE are not lost. Typically a cached copy is Suspect and Dirty, if it was modified when VIRTUE was partitioned from the primary custodian of the file.

It is recognized that certain system files, typically the load modules of commonly used system programs, change rarely but are accessed frequently. Checking the cached copies of such files on each open is likely to impose an unnecessary performance penalty. Further, such files are usually necessary for a workstation to do useful work when partitioned. To handle this special case, VICE contains a well-known file containing a list of recent changes to these system programs, and instructions to install the latest versions in individual workstations. When a user logs in to VIRTUE (and perhaps periodically thereafter), a special application program is run to examine this file, to fetch the files corresponding to the new versions of the system programs, and to copy them into "/local".

For these files, VIRTUE bypasses the checks on opens for read to VICE and redirects the accesses to the copies in "/local."

5.2 Locating Custodians

How does VIRTUE know who the custodian for a file is? In the worst case, each file access from VICE has to be preceded by a WhereIs( ) call to a workstation's cluster server. To reduce this overhead, VIRTUE can maintain a table of VICE pathname prefixes to custodians.[4] When VIRTUE makes a WhereIs( ) call, VICE returns not only the custodian for the specified file, but also the briefest prefix of the specified pathname for which the custodian list is the same. For example, suppose the subtree "/usr/bovik" has "ClusterA" as its custodian. In response to a WhereIs( ) call for "/usr/bovik/doc/thesis/thesis.mss" VICE will return "ClusterA" and "/usr/bovik". The caller can enter the mapping "/usr/bovik" — "ClusterA" in his prefix table. Future accesses to a file in the same directory can avoid the WhereIs( ) call. When the custodian for a file changes, a file request to the old custodian will fail; at that point VIRTUE can make a WhereIs( ) call and update its table.

5.3 VIRTUE Refinements

The minimal design for VIRTUE uses the cache to minimize physical movement of data. However, each use of a cached file copy requires communication with VICE to validate the copy. The following strategies, which may be used independently of each other, can be used to reduce this interaction:

1. **Don't Care** cache state: if a cached copy is in this state, VIRTUE can blindly assume that the cached copy is valid. This is the typical state for cached copies of common system program modules. To execute such a program no references have to be made to VICE. Either the user, or an automatic aging algorithm periodically deletes such a file from the cache. The next use of the file will cause a fetch from VICE, and that will be a more recent copy.

   Part of the user profile in VIRTUE is a description of which files are to be cached in this state. For example, my login profile may contain:

   ```
   Don't Care: /bin /include /lib
   ```

2. CheckIn/CheckOut: when a user logs in to VIRTUE, his "/usr" subtree is checked out for him from VICE by the VIRTUE file system. His source and data files trickle over as cache misses occur, and are marked as Guaranteed. On logout, the "/usr" subtree is checked in to VICE. At his discretion a user may explicitly check out other subtrees.

   A user may force a CheckIn of subtrees that have been checked out to him elsewhere. This addresses the case where a user wishes to log in, while he is logged in elsewhere. In that case he assumes responsibility for changes in those subtrees which have not been reflected to VICE. To minimize this problem, a VIRTUE demon periodically stores Dirty Guaranteed cached copies into VICE.

[4] This is a completely separate table from the one which maps cached VICE files to local pathnames.

5.4 Partition

We assume the existence of a mechanism for detecting partition between a cluster server and a workstation. A simple way to implement this is to use VIRTUE-initiated handshakes with individual cluster servers. VIRTUE detects partition by timing out on a handshake, while VICE detects it by noting an abnormal delay since the last handshake.

When the custodian of a file detects the partition of a workstation possessing a lock on it, it breaks the lock. If the checkout/checkin mechanism is implemented, the custodian also performs an automatic checkin of all subtrees checked out by that workstation. Note that, in this model, it is possible for a workstation to be partitoned from some custodians, while other custodians are still available.

The default assumption made by VIRTUE is that cached copies of files whose custodians are inaccessible may neither be read nor written. A user may explicitly request VIRTUE to make such files read-only, or to make them read-write. In the latter case a merge algorithm is run by VIRTUE when the partition ends. The details of this algorithm need to be worked out. Files in "/local", of course, can always be read or written.

To run a workstation completely stand-alone, a user can request VIRTUE to deliberately partition him from VICE. The file cache is handled just as it is in the case of accidental partition. In this case, the user will indicate to VIRTUE when he wishes to terminate the partition.

5.5 Undecided VIRTUE Issues

1. How is the merge done after partition?

2. Does VIRTUE support file protection? Is this necessary if the cache is flushed after logout on public workstations?

3. Is cache storage management automatic? In that case what is the storage reclamation algorithm?

4. Does VIRTUE care to do accounting? This may be necessary if the CSS (or the Transport Group) supports remote logins between workstations, as on our SUNs.

5. What extensions do we have to provide to efficiently support diskless workstations? At the very least, VIRTUE should be able to specify file redirection to a virtual disk on the network.

6. ....*Lots of others, I am sure*........

6 Archiving

Archiving is needed for two reasons: in case of catastrophic failure at some cluster server, and in case of a user deleting data he really shouldn't have. We assume that the probability of permanent loss of data from cluster servers is sufficiently low that continuous archiving from them is unnecessary. Section 7 describes the measures that may be taken to achieve this. To protect users from their own errors, we assume that a daily incremental backup is adequate; this is the level of archival service offered at most computation centres today.

To perform this function, we postulate the existence of one or more archiving servers. These servers run on trusted machines, possess high I/O and network communication bandwidth, and support the attachment of archival media. From the point of view of the VICE file system, however, these merely appear to be workstations with read rights on all files. Each server handles the archival needs of a unique subset of primary custodians. The archival process for a custodian consists of obtaining a directory listing of all files stored on that custodian, identifying the files which have been modified since the last backup, fetching those files individually, and copying them on to archival media. We expect the archival servers to run during periods of low system load.

A significant question in this regard is the quantity of archival data generated in the system. A back-of-the envelope estimate of this can be made as follows:

> It was observed on the CMU-CSD's TOPS-10 system that, on a per-user basis, a new instance of some file was created every 75 seconds, and that the average file size was 10K bytes. Assuming an average of 2000 active users, the amount of data created or rewritten per day is roughly 20 Gbytes. This is likely to be an overestimate, since the average file size used here is based on static, not dynamic observations, and the latter is likely to be lower. A large fraction of this 20 Gbytes consists of overwritten files, and temporary files. Assuming that only about 10% of these changes get reflected in the archive, this amounts to about 2 Gbytes.

For this volume of data, videodisk technology seems to be an ideal archival media. A single videodisk can hold one day's worth of archival data, together with the indexing information needed to efficiently retrieve it. A B-tree is one possible index organization.

To retrieve an archived file, a user communicates with the archive servers, specifing a filename and a date. Since the archiving mechanism is incremental, a file will not appear on the daily archive unless it was modified at least once during that day. When trying to retrieve an accidently-deleted file, a user may be unaware of the precise date on which the file was last modified. The archive servers thus need to be able to efficiently handle the request "Get me the most recently archived copy of a file" or variants thereof. This can be done by creating a merged index of newly archived files on a weekly, monthly, and annual basis.

7 Reliability & Availability

A simple, efficient way to enhance the reliability of data stored on a cluster server is to use mirrored disks. This is a well-understood mechanism, and is widely used in spite of the extra cost involved. It is our belief that the ITC's goal of building a robust system can be be best served by adopting this technique. If mirrored disks are not used, one of the following tradeoffs will have to be accepted:

- Lower reliability
  The daily archival mechanism is then the only backup available.

- Lower performance
  Continuous archiving can be done, at the cost of much higher network bandwidth usage and longer recovery times.

To enhance availablility, we recommend the use of a buddy system: cluster servers are located in pairs, and each has adequate I/O bandwidth and processing power to serve the other cluster. Of course, some degradation in performance is to be expected when one of the buddies is down. Figure 1 shows such a cluster server pair.

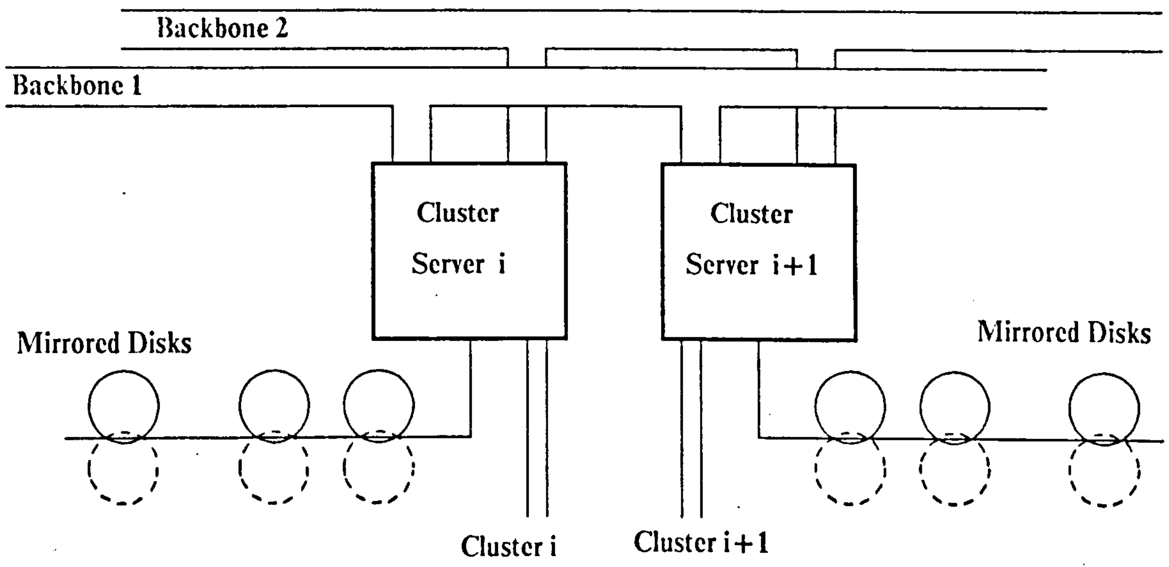

Figure 1: Normal Connections to Cluster Server Buddies

When one member of such a pair fails, a manual switching procedure connects the disks of the failed cluster server to its buddy. On recovery, the disks are manually switched back. Figure 2 illustrates the situation where one cluster server is down.

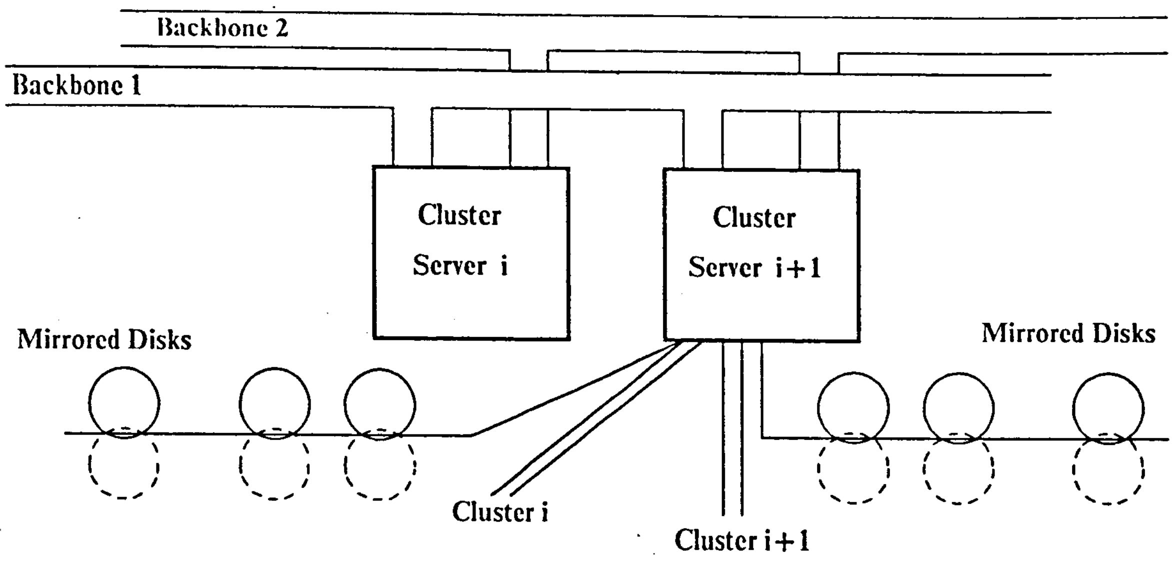

Figure 2: Connections when Cluster Server i is Down

How are requests routed when a cluster server is down? This may be done at the transport level, or in the file system. When done at the transport level, part of the reconfiguration process consists of informing a cluster server that it should masquerade as its buddy in network communications. This approach makes reconfiguration completely transparent to VIRTUE, as well as the other VICE nodes. Alternatively, the WhereIs( ) system call could represent each custodian by a pair (X, Y). Usually VIRTUE assumes that X is the custodian. When a call to X fails, Y is tried. On success, the pair is reversed, (Y,X). Future requests are now addressed to Y. When X recovers, a call to Y fails, and X is tried. This will succeed, causing the pair to be reversed again, resuming normal operation. All this is, of course, transparent to users, though visible to VIRTUE.

It may be advantageous to extend the notion of pairing cluster servers to a situation where many cluster servers are located in physical proximity to each other in some kind of "machine room." This provides a better physical operating environment for the cluster servers, and makes it easier for the VICE operations staff to perform maintenance.

I. Appendices

I.0.1 File Meta-Information

Each file stored in VICE has the following meta-information which is meaningful outside VICE:

- Filename
- Size
- Protection Information
- Timestamps of creation, access, modification
- Uninterpreted VIRTUE information.
- *I'm sure I've forgotten a few*

I.0.2 Differences between VICE and Unix File Systems

Some differences from Unix:

- There are no physical links. Only symbolic links.
- The protection mechanism may be richer, and provide a full-fledged access list mechanism.
- The notion of an i-node, and i-node operations are not available.
- The common Unix file system calls, which support byte-by-byte, or page-level access of files are absent.
- *Others*

I.0.3 Differences between VIRTUE and Unix File Systems

- Programs which depend on the fact that directories are files may not work right.
- *Others*.