Palo Alto Research Center

Violet, an Experimental Decentralized System

By David K. Gifford

XEROX

Violet, an Experimental Decentralized System

by David K. Gifford*

CSL-79-12 September, 1979

© Copyright 1979 by Xerox Corporation

Over the past year we have been designing and constructing an experimental decentralized information system called Violet. The lowest levels of the Violet system make it easy to construct a distributed user application by hiding the application's decentralized environment. Violet's first application, a calendar system, provides a sophisticated user interface to a simple relational data base manager. This paper presents our experience with the design and implementation of Violet. We discuss a new algorithm for replicated data which is implemented by Violet, and discoveries we have made about desirable concurrency modes for shared files. The conclusion outlines what we consider to be desirable design features for decentralized information systems.

CR Categories: 4.3, 4.35, 4.33, 4.32, 3.70

Key words and phrases: distributed system, consistency, operating system, computer networks, system scaling, replicated data, lock compatibility, user interface, configuration independence, calendar system

* David Gifford is a graduate student at Stanford University, and this work was supported in part by the Xerox Corporation and by the Fannie and John Hertz Foundation. This paper was presented to the I.R.I.A. workshop on Integrated Office Systems on November 6, 1979 in Versailles, France.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

1. Introduction

As an investigation into the nature of decentralized information systems we decided to design and construct a system that had ambitious but attainable goals. First, we wanted to conceal the identity of physical resources so that capacity and reliability could be improved without any changes to user programs. Second, we wanted user populations to scale over several orders of magnitude in size without system redesign. Third, we wanted to maintain a high quality bit-map display for interaction, using computing power available locally to each user. Finally, we wanted each user request to result in a predictable outcome, regardless of other concurrent activities.

We have achieved these goals in a system called Violet. Violet includes a general-purpose environment that make it easy to construct decentralized user applications. The building blocks available to the architect of an application are well-defined abstractions, arranged in a hierarchy, so that simple arguments can be used to reason about the behavior of the system. The highest level of the current system is a calendar system, providing simple data base query and storage services. This system is an extremely simple data base manager, but its underlying framework is sophisticated enough to convince us that Violet's primitives are adequate to support a wide variety of applications.

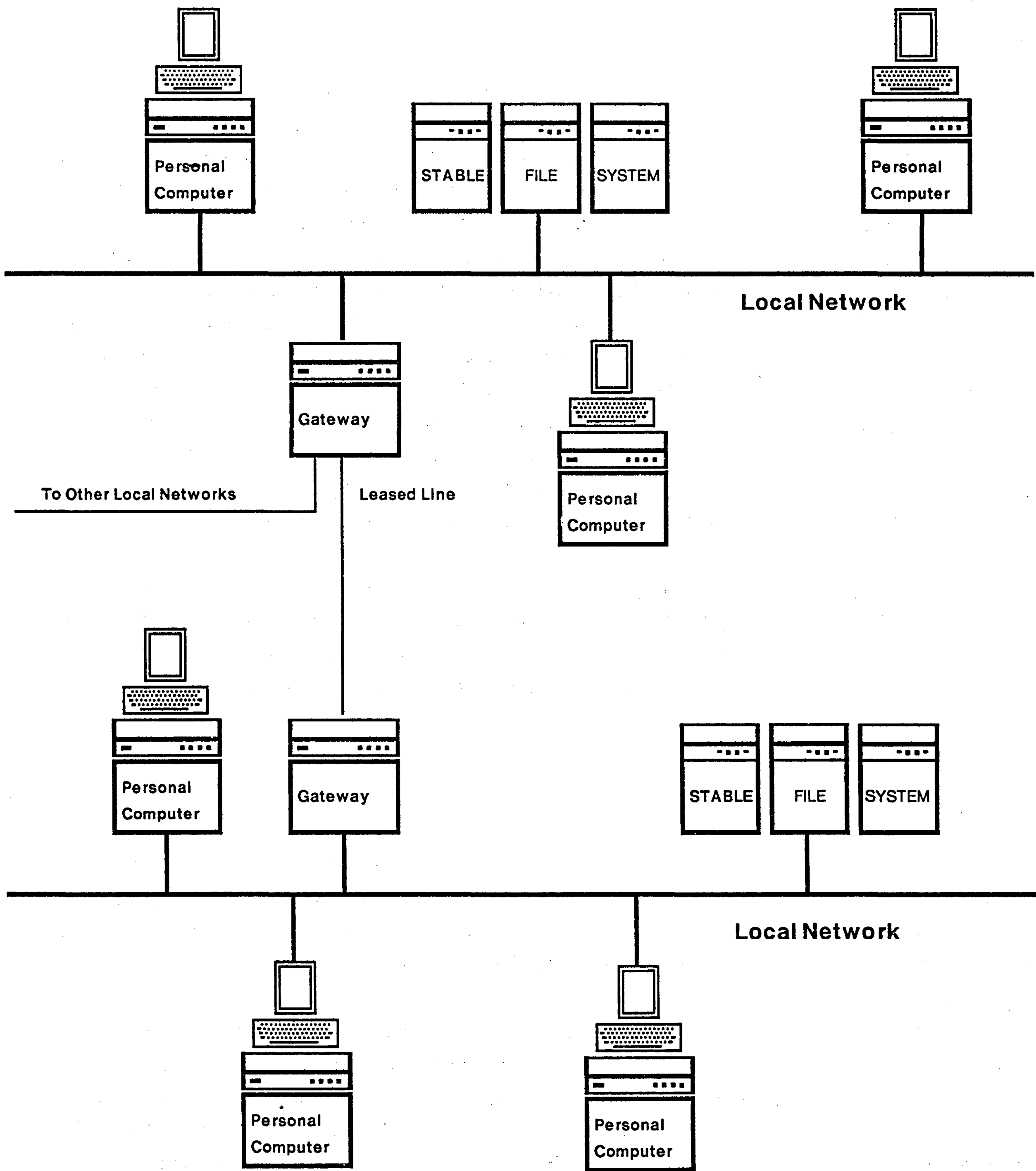
The first four sections of the paper provide a detailed view of Violet's architecture, and the last three sections discuss the architecture in practice. In order the seven sections are: a discussion of the implementation environment, an exposition of Violet's architecture, a description of our algorithm for replicated data, a section on sharing and locking, a discussion of Violet's performance, notes on the actual implementation, and a conclusion.

2. Environment

Every user of Violet has a personal computer connected to a local high-bandwidth network [7]. These local networks are joined by gateways, often with the aid of low-speed leased circuits. A collection of interconnected local networks is known as an *internetwork* [1]. Figure 1 shows a fragment of a typical internetwork. The addressing structure of an internetwork is uniform, regardless of a computer's attachment point.

In addition to user machines, there are *file servers* [4, 5] connected to an internetwork. A file server is a computer (typically with a large amount of attached disk storage) that is used solely to store files. Under user direction, Violet utilizes shared file servers to store user data. The servers provide page-level access to files: "read page" and "write page" are typical operations. The characteristics of the communications channel between client program and server determine the observed latency of the server's operations. Actual measurements reveal that the typical read latency for a 512 byte page is approximately 75 milliseconds from a file server on a directly connected network; the figure rises to 650 milliseconds for a file server connected to a remote network accessed via a 9.6 KB data circuit.

Violet uses the unique facilities of the Distributed File System [4, 5] for data storage. DFS is implemented by a loosely coupled confederacy of file servers. Operations on files are grouped into transactions, and synchronization protocols between file servers insure that either all of the actions



Typical Internetwork Environment

Figure 1

of a transaction are performed, or none are. For example, if file A is on server X, and file B is on server Y, a transaction updating A and B will either effect changes on servers X and Y or will not change data on either of them.

3. System Architecture

The system is structured in five levels. Levels correspond to components that were independently tested. Level 0 provides basic file storage facilities. Level 1 erases physical boundaries, and improves imperfect hardware to provide an idealized network virtual machine. Level 2 is a virtual memory package, and Levels 3 and 4 comprise the calendar application, with its data base manager and user interface.

To provide motivation for the abstractions, we will introduce them top down, with apologies for the occasional forward references which result. Readers may find it convenient to refer to Figure 2, a diagram of Violet's internal structure.

LEVEL 4

User Interface. Figures 3 and 4 are representative pictures of our user interface. Figure 3 shows a public "bulletin-board" calendar, Seminars.CSL. Figure 4 shows a scheduling operator applied to the union of two calendars. A pointing device is used to select menu items and calendar entries for manipulation. After a view of the calendar data base is specified, Violet paints the screen with the desired data.

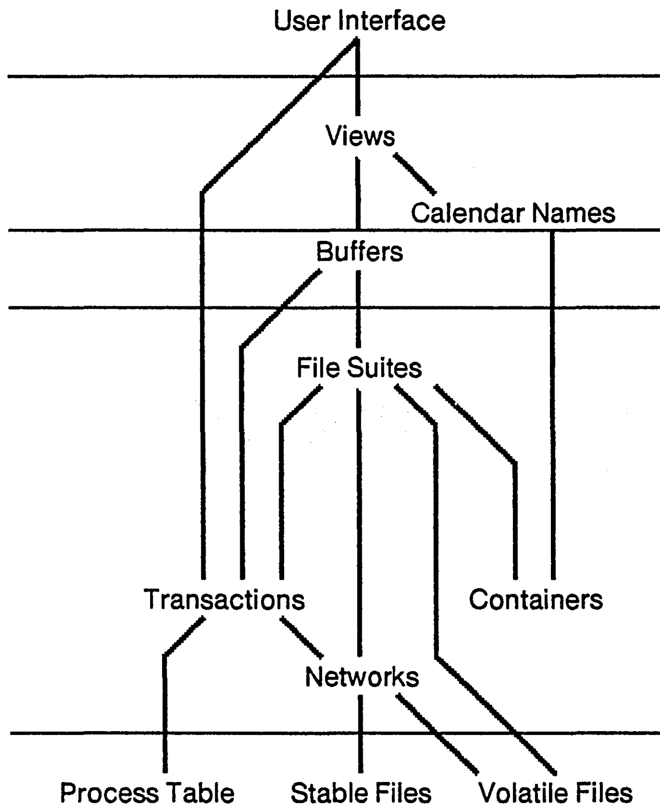
All user requests are implicitly part of the current *transaction.* When a user is satisfied with his changes he can choose to commit the transaction with the commit button, shown on the screen in Figures 3 and 4. Violet guarantees that either all of the actions of a transaction will be performed, or none of them will. If a user is unhappy with his changes, he can choose to abort the transaction and start over again.

Violet automatically keeps the display current with changes in the data base. As long as the user's transaction is in force, it appears to the user that he is the only person using the data base. However, if another user updates data that is in the current view, the current transaction is aborted, and the screen is automatically repainted to depict the change. The detailed coordination with transaction management necessary to accomplish this is described in Section 7.

LEVEL 3

Calendar Management creates and deletes calendars, and maps calendar names into file suite names (Level 1). The naming structure is fairly simple by design. Each calendar name is composed of two parts which appear as: ObjectName.GroupName. Currently we map group names into containers (Level 1), and use the object name to select a file within the container. The administration of the naming environment is decentralized; by including groups we have broken the name space into pieces that can be independently managed.

Views. Tuples are related groups of information that are stored in our data base. The only tuple type we have defined corresponds to an event, and contains such fields as start time, finish time,



The Internal Structure of Violet

Figure 2

View: Seminars.csl

Calendar

Violet Calendar System

January 1979

14 Sunday	15 Monday	16 Tuesday	17 Wednesday	18 Thursday	19 Friday	20 Saturday
		10:30 - 12:00 Prof. Steven Ward of MIT CSL Commons The MuNet: A Scalable Multiprocessor Architecture ---------- 14:30 - 15:30 Dr. Robert Bower, UCLA & TRW CSL Commons Very Large Scale Integrated Circuits: Evolutionary or Revolutionary for the 1980's ----------	13:15 - 15:00 Dave Gifford, CSL CSL Commons Dealer: The Architecture of Violet ----------	15:45 - 17:00 Don Scifres, GSL PARC Cafeteria Forum: Exploring the Light Fantastic ----------	15:00 - 16:00 Prof. Yutake Toyozawa GSL Conference Room #1077 Bistability and Anomalies in Resonant Scattering of Intense Light ----------	

?		Quit	Next Week	Previous Week	Set View	Create	Change	Delete	Commit	Abort	Copy

Figure 3

View: {Gifford.csl, Taylor.csl}

Calendar

Violet Calendar System

January 1979

14 Sunday	15 Monday	16 Tuesday	17 Wednesday	18 Thursday	19 Friday	20 Saturday
	11:00 - 12:00 Taylor.csl Unavailable ---------- 13:00 - 17:00 Taylor.csl Unavailable ----------	10:00 - 12:00 Taylor.csl Unavailable ---------- 13:00 - 15:30 Taylor.csl Unavailable ----------	11:00 - 12:00 Taylor.csl Unavailable ---------- 13:15 - 17:00 Taylor.csl, Gifford.csl Unavailable ----------	10:30 - 12:00 Taylor.csl Unavailable ---------- 13:00 - 17:00 Taylor.csl Unavailable ----------	9:00 - 17:30 Taylor.csl Unavailable ----------	

?		Quit	Next Week	Previous Week	Set View	Create	Change	Delete	Commit	Abort	Copy

Figure 4

author, event description, and so on. Sets are collections of tuples. Violet stores a set and its index in a file suite.

A view is a virtual set, synthesized from a user supplied description. The five operations on a view are create tuple, delete tuple, update tuple, get next tuple, and fetch tuple. Legal view descriptions are described by the context free grammar

view ::= *base* ∪ *view* | *base*
base ::= {*view*} | (*view*) | **Calendar**

Every view operator defines fetch and update semantics, so a user can update the synthetic set provided by the view mechanism. For example, "Gifford.CSL ∪ Taylor.CSL" describes the union of the two sets "Gifford.CSL" and "Taylor.CSL". All tuples that appear in either Gifford or Taylor appear in the union, and updates to the view are applied to both Gifford and Taylor. It would be a simple matter to define a union operator with different update semantics. The scheduling operator {*view*} synthesizes a view that groups all of the unavailable time in *view* together. Thus, "{Gifford.CSL ∪ Taylor.CSL}" could be used to schedule a meeting between Gifford and Taylor (see Figure 4).

LEVEL 2

Buffers provide a simulated virtual memory to Level 3, and in turn utilize file suites. The buffer manager guarantees that it always holds fresh data by registering itself with transaction management (Level 1).

LEVEL 1

Transactions. Transactions insure that a consistent set of updates are always applied to a file. Transactions guarantee serial consistency [2], or the illusion that there is no concurrent activity in the system.

A transaction is implicitly associated with a process—although several processes in a single processor can arrange to share a transaction. Such sharing occurs in our replicated data manager when several processes cooperate to provide a single service. A process exercises direct control over its transaction, requesting that it be aborted or committed.

Supporting this abstraction requires three distinct services. First, a process-to-transaction mapping must be maintained. Second, a centralized interface for requesting transaction state changes (commit, abort) must exist. Third, a clearinghouse for messages concerning the freshness of data must be established; it must inform concerned modules that data they previously requested has become stale. To use this mechanism, modules that save state information based on reads from the data base register themselves with transaction management, and are then notified of a transaction commit or abort. This service is used by the user interface to know when the screen is out of date and should be repainted.

Network Management performs two functions. First, it synthesizes full-duplex, perfect connections from a packet-switched internetwork composed of local networks and gateways which may lose or duplicate packets. Second, network management translates symbolic network server names into network addresses. Boggs et al [1] describe the low-level protocols and software used by Violet.

File Suites are logically arrays of bytes. The primitive operations on a file suite are create, delete, read, write, set size, and get size. File suites may be replicated for performance and reliability enhancement, as described in Section 4. Every file suite is assigned a unique name when it is created. Network management is used to communicate with the remote components of a file suite.

Containers are storage repositories for file suites, independent of specific physical storage services. A file suite's container is specified at its creation time. A file suite inherits properties from its container, including whether or not it should be replicated.

LEVEL 0

Level 0 includes the language run-time system, which supplies such necessary facilities as a segmented memory, a free-storage package, and low-level process management.

Stable Files are files that can be accessed through a transactional file system. As we mentioned earlier, stable files are implemented by DFS. Because stable files are shared, it is extremely important that concurrent file operations have well-defined properties. DFS mediates access to shared files by implicitly setting locks in response to file operations.

Volatile Files are files that can not be accessed through a transactional file system. Unfortunately, because of the complexity involved in providing transactions, many file systems do not implement them. An example is the one local to a user's personal computer. Section 4 describes how volatile files can be included in a file suite.

4. Replicated Data

It is often desirable to replicate data for additional reliability and performance. The file suite abstraction implements a read majority, write majority algorithm [3] for replicated data. A sketch of the algorithm is provided here, and the interested reader should consult [3] for further details. The algorithm assumes that there is a common transactional file system across all copies of a file suite.

A file suite is composed of a number of *representatives*, each containing a copy of the suite's data. Each representative has a number of *votes*. A *majority* is a subset whose votes sum to more than half of the total number of votes assigned. Each representative also has a *version number*. A representative is said to be *current* if it contains the most recent version of the file suite.

Every file operation directed to a replicated file suite is appropriately transformed. The central invariant of our algorithm is that any majority will always contain a current representative. Conceptually, a read checks the version numbers of a majority, and actually reads from a current representative. Writes have more stringent requirements. The first step of a write is to collect a majority of current representatives. A write then applies its update to all members of this majority. In this way it assures that some majority subset is always current.

In actuality, the performance of the replicated-data algorithm has been improved considerably from a naive implementation of the conceptual description provided above. Version numbers are read once per transaction. Background processes update obsolete representatives. An adaptive algorithm that measures response times of representatives attempts to forward a request to the fastest eligible representative.

An attractive property of our voting proposal is flexibility. By entrusting all of the votes to one representative a centralized scheme results; apportioning votes equally among representatives results in a completely decentralized scheme. The proposal also admits a variety of interesting configurations between the two extremes. For example, consider four representatives assigned the voting configuration <2, 1, 1, 1>. The first representative when paired with any other representative forms a majority subset. However, the system can tolerate the failure of the first representative. As long as a majority is available, the suite will continue to function.

Using the administrative tool of vote assignment, it is possible to blend the individual strengths of representatives to achieve desired file suite properties. Heavily weighting high-reliability representatives will tend to produce a reliable suite; heavily weighting high-performance representatives will tend to produce a high-performance suite. A thorough analysis of the vote assignment problem is beyond the scope of this paper.

A representative's version number must be accurate. If a version number became incorrect, inconsistent data could potentially corrupt the entire suite. It is straightforward to use the atomic update properties of stable files to guarantee that version numbers are maintained correctly, using the following simple algorithm. When a current representative is updated by a transaction, its version number is incremented. All current representatives have the same version number, which is interpreted as the version number of the suite. When an obsolete representative is overwritten with current data, it assumes the version number of the suite.

We have also introduced the notion of a *weak representative*, one with no votes. Such a representative can be created without administrative sanction, as it will have no material effect on the system. However, it carries a version number, as does any representative, and can be included as a member of a majority subset. Thus, when placed on a high speed device, it can serve as an effective intermediate level of store. Conceptually we like to view all data that has been temporarily promoted to a different level of store as an instance of a weak representative. Such a representative can be used for access, but changes to it will not be firm until propagated backward into a majority subset. An acceptable mode of recovery for a weak representative is invalidation, because the majority invariant always guarantees that the data can be recovered from the suite. Thus, weak representatives can be stored in volatile files.

All of the replicated data machinery we have described resides in a user's local machine. Our file servers are unaware that replication of data is being performed, and are thus unencumbered. However, it is conceivable that outside assistance could improve performance. Replication servers could assume the tasks of file management, freeing user machines from their detailed knowledge of file suite replication. One could also imagine a daemon assisting file suite management in its update tasks, potentially at times of surplus communication capacity.

In sum, we see the cardinal virtue of our replication algorithm to be its simplicity. The algorithm requires no changes to file servers, and is easy to implement.

5. Sharing and Locking

DFS mediates concurrent access to stable files by implicitly setting locks in response to file operations. These locks are held for the duration of a transaction and then released. Initially DFS employed traditional read and write locks that allowed for either one writer or n readers. The lock

compatibility matrix for this rule is:

	No Lock	Read	Write
No Lock	Yes	Yes	Yes
Read	Yes	Yes	No
Write	Yes	No	No

If a transaction attempts to set a lock that would violate the matrix it is forced to wait.

To insure that no user monopolizes a stable file, DFS will time out a transaction if other users are waiting for a file it has locked. A transaction that times out leaves stable files unchanged because its transaction is aborted. The same mechanism insures that cyclic lock dependencies (deadlocks) will be resolved by the abortion of one of the transactions.

The length of a transaction is controlled by the user, and it may consist of many interactions. As a user progresses, he acquires more locks, increasing the probability that he will conflict with another user. The net effect is to push the transaction into the lower right corner of the compatibility matrix, making the transaction less and less likely to complete.

When preparing for the first demonstration of Violet this locking strategy showed its limitations. Imagine the following scenario: there are two Violet users, each displaying the same calendar. They both have been viewing the calendar, and thus holding read locks, longer than the time-out interval in DFS. User A now updates his view, but does not commit his transaction. User A thus sets a write lock on the calendar's index, aborting User B. User B's machine, attempting to provide good service to its user, now continually asks for the index that has been denied to it. Finally, User B's machine times out User A, aborting User A's transaction. User B's machine repaints the old view of the data without User A's change. User A, having been aborted, also repaints his screen with the old information. Net progress: zero.

Two conflicting desires produce the underlying problem. First, the user interface is always trying to maintain a fresh display. Second, we want to allow a user to determine what constitutes a transaction, thus allowing him to determine how long data is unavailable. One way to solve this dilemma is to queue up all of the writes of a transaction, issuing them just before a commit. Using this strategy, User A would not acquire the write lock on the index until commit time.

We use a variant of this solution, taking advantage of our transactional file system. As writes occur during a transaction, we set intention-write locks. An intention-write lock implies that the transaction will update the datum in question at commit time. The buffering of writes till commit time is a natural by-product of our transactional file system. It still appears to the issuing transaction that a write takes place immediately. When the transaction does commit, the intention-write locks are converted into commit locks, and the writes are performed. Intention-write locks are compatible with read locks. Our new compatibility matrix is:

	No Lock	Read	I-Write	Commit
No Lock	Yes	Yes	Yes	Yes
Read	Yes	Yes	Yes	No
I-Write	Yes	Yes	No	No
Commit	Yes	No	No	No

Transactions operate in the upper left three-by-three matrix. Only during commit processing will a

transaction hold write locks.

We have chosen to make multiple intention-write locks incompatible. Eventually one of the transactions would commit, changing its intention-write lock into a commit lock. Thus, conflict is inevitable, and we chose not to postpone it.

A direct result of our new locking mechanism is the increased availability of data. It is now always possible for users to access data, except for predictably short periods during commit processing. The increased availability of data allows for more concurrent activity.

Our locking strategy is not applicable to all environments. We have provided long transactions that may be aborted, in contrast to many existing systems that provide short transactions that will always commit. Our approach is best suited for a low contention environment with users who desire to control the length of their own transactions.

6. The Performance of the Architecture

6.1 File System Properties

Above Level 1, Violet is similar to a typical time-sharing system. In fact, the interface to file suites is a copy of the CTSS [6] file system interface. Thus, the *semantics* of our interface are identical to those of a centralized file system. However, our interface's *properties* are considerably different.

We have identified three properties of our file system that serve to distinguish it from CTSS (or any other time-sharing like system). First, the servers that comprise Level 0 are under decentralized administrative control. Thus, the failure modes of Violet and a time-sharing system are considerably different. Second, the observed performance of file operations can range over an order of magnitude—approximately from 75 to 750 milliseconds. Third, the number of directly accessible files is for all practical purposes unlimited.

These properties provide for decentralized management of data-storage facilities, permitting a department (or other administrative unit) to assume full responsibility for its own storage needs. Of course, there is the potential for abuse. A local organization can be irresponsible in maintaining adequate server performance. For example, we often found ourselves calling colleagues to ascertain the state of a server when our system stopped responding.

The provisions for replicated data were designed in part to eliminate the undesirable properties of our decentralized hardware base. We do not have sufficient experience yet to judge how serious the problems are, or how much replication helps.

6.2 Level Distribution

Retaining our basic architecture, it would be possible to reconfigure the tasks for which processors are responsible. For example, the internetwork communication path could be moved from Levels 1-0 to Levels 4-3. This alternative corresponds to "sending actions" as opposed to our current approach of "sending data", and would entail operating calendar servers.

We found that the number of messages and the quantity of information that passed between Levels 1 to 0 and Levels 4 to 3 were comparable. This was a result of Violet's ability to keep pertinent calendar indexes in its virtual memory.

When assigning layers to processors the most significant effect is normally assumed to be to the cost of communication. Because our local network operates at three million bits per second, we did not find the cost of communication to be significant. The ability of the programs that compose the layers of Violet to fit in a user's local machine turned out to have the most significant effect on performance. Layers 1-4 required more main memory than many of our intended clients had available in their local machines.

7. Implementation Notes

The entire Violet system was implemented in Mesa [8], a programming language that provides integrated processes, monitors, and condition variables. The process and synchronization facilities of Mesa are used heavily by Violet. For example, the replicated data manager is a monitor that has an instantiation for every open replicated file. Each instance employs one static process, two kinds of dynamic processes, and three condition variables.

Our abstractions are implemented as Mesa modules. We took full advantage of Mesa's class structure to evolve our implementation by specifying interfaces, filling in underlying implementations as necessary. Such step-wise development afforded us the opportunity to test our assumptions about the importance of various system components by initially implementing minimal facilities, and later returning to expand them. For example, our first implementation lacked integrated transaction management, and was difficult to comprehend. The first module to use the file system created a transaction, and was responsible for passing it to other participating modules. Exceptional conditions often resulted in the invalidation of a transaction, producing chaos. Formalizing the transaction abstraction by providing a centralized interface solved these problems.

Each abstraction is responsible for behaving correctly when it is presented with concurrent requests. There are no global locks in Violet. Rather, Violet's modules use monitors to serialize access to shared data.

The user interface is implemented by two processes. One process is dedicated to maintaining the display. It collects a display-full of information using get next tuple and fetch tuple, and paints the display. It then waits on the condition variable ViewChanged. The second process watches the keyboard, and when a menu item is selected, collects characters, performs the user's request, and notifies ViewChanged. The first process then repaints the display with the updated view. Both of these processes run in the same monitor, and procedures that enter the monitor are used to synchronize user actions with display updating. Conceptually, each button on the user's display is an entry into this monitor. This mechanism insures that user requests are not processed while the display is updating.

These processes are also used to keep the display current with changes other users make in the data base. When the user's transaction is aborted, indicating there is fresh information, a new transaction is started, and ViewChanged is notified. The first process then repaints the screen with fresh information. Because the transaction abort is detected at interrupt level (a network packet arrives) it is not possible for the user interface to update the display when it first learns that it is obsolete. This consideration motivated dedicating a process to display maintenance.

Testing was the hardest part of the implementation task. An annoying part of our debugging system was that we could not discern what data structures and processes were associated with

specific parts of Violet. This information was not necessary for proper operation of the Mesa run-time support, and thus was not available. Furthermore, the lack of hardware protection in our personal computers has traditionally made certain kinds of problems difficult to locate. Thus, we tried to debug our system at design time, but we found that some implementation flaws were not uncovered until we ran the algorithms.

Interactive debugging tools were written to exercise the levels of Violet. Transaction and file management were operational after an afternoon's work with the Level 2 debugger. Our spontaneous transaction abort logic was tested by using several machines operating in this simplified environment. In addition to providing a debugging tool, the Level 3 debugger was our original teletype-style user interface.

8. Conclusion

Violet's structure reflects our understanding of the fundamental facilities that are required to support a decentralized user application. We very carefully factored necessary facilities into independent abstractions. The implementations of abstractions interact (e.g. transactions and file suites), but a client is always presented with independent interfaces for independent concepts.

Our careful factoring reduced the complexity of Violet. For example, the code that manipulates transactions is grouped into small, pure pieces, instead of being scattered throughout the system.

Our experience with Violet has resulted in a number of observations:

First, the primitives from which a decentralized system is to be constructed must be sound. A firm foundation offers conceptual power, as we demonstrated by synthesizing replicated files from a well-defined transactional file system.

Second, the notion of a transaction is fundamental to a successful concurrent system. In addition to DFS servers, we utilized non-transactional servers. Weak representatives can be stored on these servers, but for general use we found them to be largely unacceptable, because their locking protocols did not provide for sufficient concurrency. In addition, our users often restart their machines when impatient, which tended to cause irreparable damage to their calendars because of uncompleted writes.

Third, sophisticated display facilities place large performance demands on a data base system. A typical Violet screen is composed of approximately twenty tuples, and our users expect to see them appear instantaneously.

Fourth, unifying a decentralized hardware base with a file system has a great deal of conceptual simplicity. The Level 1 framework we built would allow for the construction of a wide variety of application systems.

Fifth, it is reasonable to expect that at any point some fraction of the resources of a decentralized system will be unavailable. We have introduced a new abstraction, the file suite, that masks partial system failures.

As stated at the beginning of the paper we had four goals when we built Violet: configuration independence, extensibility, a flexible user interface, and consistency. We have demonstrated a system that uses simple abstractions to achieve these goals. We are confident that more ambitious systems, if attempted with a similar architecture, would also prove to be successful.

Acknowledgments

I would like to thank my advisors, Butler Lampson and Susan Owicki, for their comments and guidance. Jay Israel, Karen Kolling, Jim Mitchell, Jim Morris, and Howard Sturgis were members of the project that implemented DFS. Scott McGregor provided implementation assistance.

References

[1] Boggs, D.R., Shoch, J.F., and Taft, E.A. Pup: An Internetwork Architecture, to appear in *I.E.E.E. Trans. on Comm. 28, 1* (January 1980).

[2] Eswaran, K.P. et al The Notions of Consistency and Predicate Locks in a Database System, *Comm. ACM 19, 11* (November 1976), pp. 624-633.

[3] Gifford, D.K. Weighted Voting for Replicated Data, to appear in *Proceedings of the Seventh Symposium on Operating System Principles, ACM Operating Systems Review.*

[4] Israel, J.E., Mitchell, J.G., and Sturgis, H.E. Separating Data From Function in a Distributed File System, Second Colloque International Sur les Systèms d'Exploitation, IRIA, Rocquencourt, France, October, 1978.

[5] Lampson, B.W., and Sturgis, H.E. Crash Recovery in a Distributed Data Storage System, *Comm. ACM,* to appear.

[6] Massachusetts Institute of Technology Information Processing Center, *CTSS Programmer's Guide,* December, 1969.

[7] Metcalfe, R.M. and Boggs, D.R. Ethernet: Packet Switching for Local Computer Networks, *Comm. ACM 19, 7* (July 1976), pp. 395-403.

[8] Mitchell, J.G. et al, *Mesa Language Manual,* CSL Report 79-3, Xerox Palo Alto Research Center, February, 1978.