Helix: The Architecture of the XMS Distributed File System

Marek Fridrich and William Older, BNR

With abstraction layering and system decomposition, all the user sees is one homogeneous system. In actuality, the architecture is supporting 15 LANs and close to 1000 workstations.

Object-oriented systems are desirable because they form a good basis for building reliable software. Helix is such a system, combining capability-based access with a user-organized directory structure to uniformly access objects distributed over a local area network. Security and system integrity are further enhanced by atomic actions that can be performed on an object or a set of objects. Abstraction layering and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices.

Helix was developed at BNR in Ottawa as a major component of the XMS system, which includes multiple, loosely coupled microcomputers, a high-speed local area network, a network operating system, and a concurrent programming language.

Before describing Helix, however, we will briefly review the key concepts of the XMS software architecture; the article by Gammage and Casey in this issue presents a more detailed discussion.

Overview of the XMS architecture

The XMS architecture supports a number of virtual address spaces called *locales*. A locale is both a recovery unit and a reconfiguration unit. Normally, a single program runs on a locale assigned to a node.

Within a program, the unit of concurrency is a *task*. A task is identified by a unique "taskid," which is generated when the task is created and invalidated when the task is destroyed. Within a locale, tasks communicate via a mechanism known as *rendezvous*. An interlocale extension of this communication mechanism is called *remote rendezvous*. The syntax and semantics of the two variants are as close as is practically possible.

The architecture encourages a *client-server* model for the structuring of applications. An application is built as a series of layers, with each layer regarded as both a server, providing services to higher layers, and a client of lower layers. To facilitate communication between clients and servers, the operating system *name server* allows a task to register its taskid together with an associated text string name of the service being provided. A client program or task can then obtain the taskid of such a public task by invoking the LOCATE ("service") primitive of the name server.

Design objectives

In simple terms, the design objective of Helix was to develop a state-of-the-art distributed file system within the framework of the XMS hardware and software architecture. The objective was broken into several requirements:

- the need to support a *diversity* of application needs (software development environment versus specialized services, big versus small, present needs versus expected future requirements);
- the need for a uniform interface to a file system that is physically distributed over a single local area network;
- the need for the file system to accommodate several different storage media, including local floppy drives, local Winchester drives, and shared file servers;

0740-7459/85/0500/0021$01.00 © 1985 IEEE

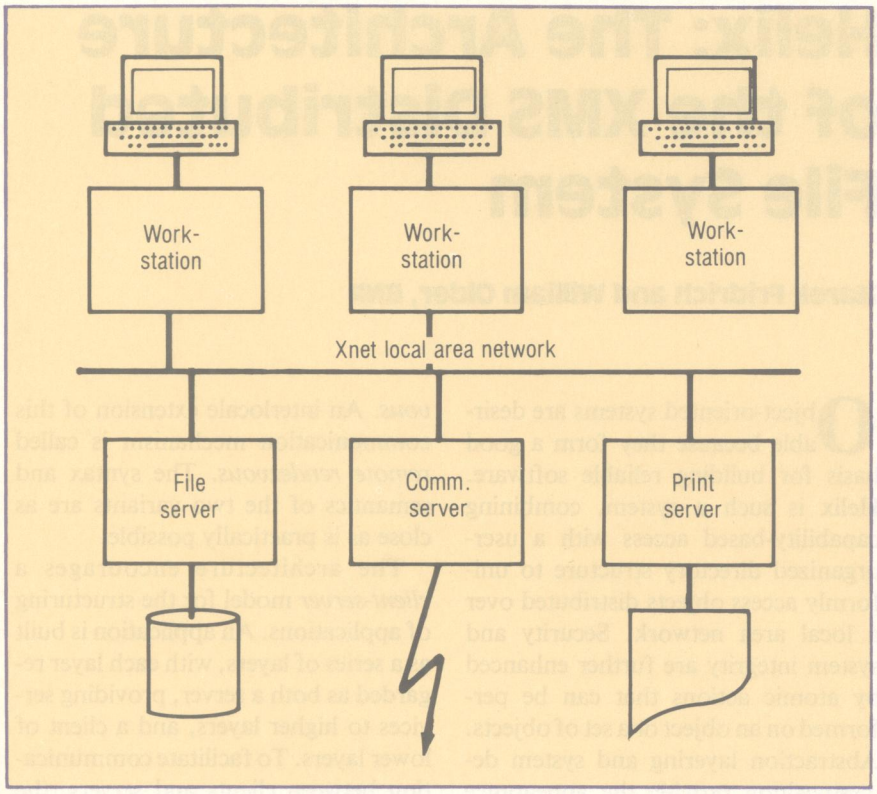

Figure 1. Typical system configuration.

- user demand for capabilities that are at least equal to, if not better than, those of existing "good" systems (AT&T Bell Laboratories' Unix, for example); and
- the need for a system with a high degree of crash resistance and media security. A direct implication is the requirement for some form of transactional support within the file system.

These requirements and pragmatic considerations, such as the need for compatibility with the UCSD file system used during early development, had a direct influence on the architecture developed.

Overview of Helix

Given the framework of the software architecture, Helix is a server providing a service to its clients. The service is concerned with the organized, efficient, and secure management of the storage and input/output system. Helix is a distributed system, responsible for managing both local devices, such as terminals and floppy or Winchester drives, and shared devices, such as file or printer servers (Figure 1).

A user may view the file system as a universe of objects, which are used to store and transfer data. Some objects, such as files, can be both read from and written to, while others, such as printers, can transfer data in only one direction. An object is identified by a system-generated capability (called an FID) or by a path name of the user's choice.

Helix objects are grouped on logical volumes. A volume is an autonomous subsystem capable of independent operation. Typically, a one-to-one correspondence exists between logical volumes and physical disks or file servers. Helix uses a policy of loose coupling among its volumes, which provides the appearance of one homogeneous system, but allows a volume to operate with no dependence on the presence (or absence) of other volumes.

Internal structuring and organization within a volume are facilitated by special objects called directories. Directories are repositories for capabilities, allowing user-generated names to be used instead. As in other systems, directories facilitate naming and efficient searching. Helix directories, however, do not have to be organized in a strictly hierarchical manner.

Once a capability is deposited into a directory, it can be referred to by the name of the access path from the root directory of the volume to the point of insertion. There is always at least one path; the first one is made at creation; additional paths may be made by subsequent linking.

File access requests are resolved in two steps. First, the user-specified path name is used to identify and retrieve a capability for the object. Second, the capability is checked for validity and compatibility with the desired access mode. If no conflicts exist, access is granted, and an *instance* of the object is created for exclusive use by the requester. A number of instances may be associated with a single object. At most, one of them has the right to update the object. The update operation (called COMMIT) is atomic, producing a new version of the object.

Helix also supports *atomic transactions* on sets of objects. These transactions are atomic in the "all-or-nothing" sense (see box at right). All Helix objects provide a standard interface; in its basic form this interface consists of two block-oriented operations called STD_READ() and STD_WRITE(). Additional operations, such as text- and record-oriented I/O, are implemented on top of the two standard operations.

Helix system architecture

A key issue in the design of any distributed system is the problem of system decomposition, including the definition of specific functional units and the placement of such units.

When designing Helix, we and our colleagues at BNR started by defining an architecture for a storage management system as a series of abstraction layers. We concentrated first on the most complex subsystem (the file server), hoping that architectures for less complex servers would fall out naturally as special cases. Our expectations were met; a pleasant surprise was

that the abstraction layering also provided an answer to the decomposition problem.

Layers of abstraction. When attempting to define an abstraction layer hierarchy, the use of the bottom-up approach is natural. At each step, the crucial decision is to first identify the next problem to be solved; the succeeding decisions, which involve alternatives and the choice of a preferred solution, tend to be easier.

The first problem is how to deal with the physical storage media, in this case, a disk drive. The *canonical disk* layer (Figure 2) abstracts the physical addressing details and makes the disk appear as an array of blocks numbered 0. . .(size-1). For performance reasons, we had to map two consecutively numbered blocks onto physically adjacent disk locations (which we will discuss in more detail later). The physical block size is a property of each individual server. Larger blocks do improve performance (file server access time) but require more buffering space on the workstation. As a compromise, we chose 1024 bytes for the filer server and 512 bytes for all local storage servers.

The next problem is the need for good performance in a shared environment. The *cache* layer (Figure 2) provides the buffering and concurrency required to achieve this goal. A Read results in a disk access only if the block specified is not present in the cache, while a Write is delayed—the block to be written is queued, with the physical disk operation taking place at a later, and hopefully, more convenient time. Since all Writes to the disk must be done before a transaction commit can be acknowledged, the cache layer supports a FLUSH() operation on a commit domain. All blocks written by a transaction can thus be physically written to the disk before the commit operation is acknowledged.

At this point, the problem of disk-space allocation cannot be postponed any longer, suggesting that the third layer should have the responsibility for storage management. If the storage-allocation map is to survive system restarts and crashes (and it should, to avoid time-consuming audits on every reboot), then the storage-allocation mechanism and the system-commitment mechanism must be tightly coupled.

Commitment and atomic actions

The concept of atomic action is extremely useful in a transactional data processing environment. To illustrate, consider the case of a program processing the transaction "transfer *X* dollars from account (file) *A* to file *B*." Should the system crash while running this program, it may leave the two accounts in an inconsistent state.

A system that supports atomic actions considerably simplifies the recovery problem. Typically, application programmers are provided with tools allowing them to specify that a group of operations be performed in one indivisible step. They are then assured that the transaction is atomic in the "all-or-nothing" sense: either all or no changes belonging to the transaction are done.

An example of an atomic action is the commit operation (COMMIT ()) on a Helix file or a set of files, which permanently applies all the changes since the last commit or open. Atomic multiserver commitment—involving objects on several volumes—is more difficult than the single-volume case, since any potential solution must overcome the problem of node crashes and unreliable communications.

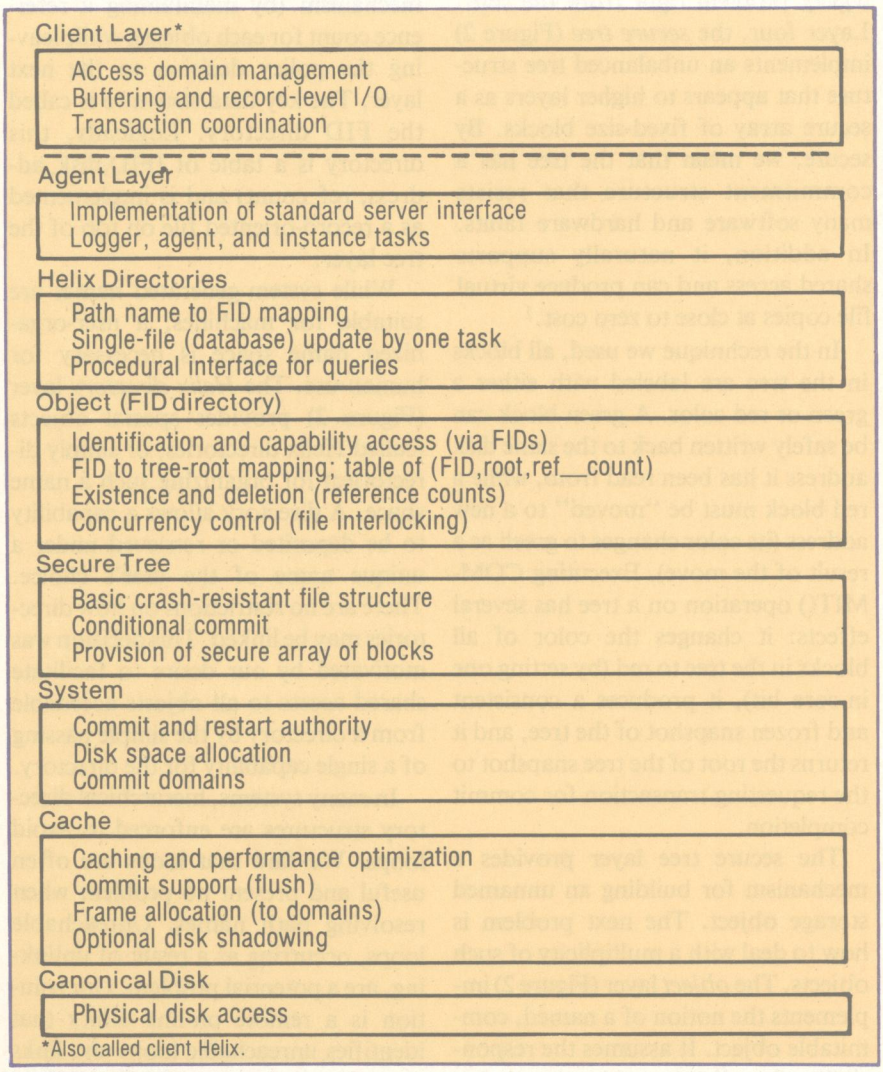

Figure 2. Abstraction layering.

The *system* layer (Figure 2) is responsible for these two basic system issues: storage allocation and final commit authority. A single "system" block holds the key information, including the disk address of the current root of the FID directory (discussed later), a reference to the bit-map of available storage, and an allocation pointer into the bit-map. The system block is time-stamped on every commit and alternately written to two known locations on the disk. The system layer is also responsible for allocating commit domains to transactions and for flushing the cache whenever a domain is committed.

The next problem was choosing a suitable file structure. In addition to meeting the standard requirements for such structures, we wanted our mechanism to solve the data and system integrity problem right from the start. Layer four, the *secure tree* (Figure 2) implements an unbalanced tree structure that appears to higher layers as a secure array of fixed-size blocks. By secure, we mean that the tree has a commitment structure that resists many software and hardware faults. In addition, it naturally supports shared access and can produce virtual file copies at close to zero cost.[1]

In the technique we used, all blocks in the tree are labeled with either a green or red color. A green block can be safely written back to the same disk address it has been read from, while a red block must be "moved" to a new address (its color changes to green as a result of the move). Executing COMMIT() operation on a tree has several effects: it changes the color of all blocks in the tree to red (by setting one in-core bit), it produces a consistent and frozen snapshot of the tree, and it returns the root of the tree snapshot to the requesting transaction for commit completion.

The secure tree layer provides a mechanism for building an unnamed storage object. The next problem is how to deal with a multiplicity of such objects. The *object* layer (Figure 2) implements the notion of a named, commitable object. It assumes the responsibility for all issues related to existence, naming, identification, commitment, and access to such objects.

We decided to identify objects by system-generated names (called FIDs). This choice has a number of advantages. First, an FID provides the dual function of access capability for an object. Second, a system-generated name is difficult to forge, and third, name conflicts are easily avoided. With respect to existence, and deletion in particular, the object layer provides a mechanism (by maintaining a reference count for each object), while leaving the policy decision to the next layer. The key data structure is called the FID directory. Logically, this directory is a table of (FID,disk_address, ref_count) and is implemented as a record-oriented file on top of the tree layer.

A pleasant surprise in the design of Helix was that abstraction layering provided an answer to the decomposition problem.

While system-generated names are suitable for machines, a user-organized name space is necessary for human use. The *Helix directory* layer (Figure 2) provides special objects (called Helix directories, or simply directories) for organizing such a name space. A directory allows a capability to be deposited or retrieved under a unique name of the user's choice. There are no restrictions on how directories may be linked. This decision was motivated by our desire to facilitate shared access to all objects accessible from a directory by the simple passing of a single capability for the directory.

In many systems, hierarchical directory structures are enforced to avoid loops. We find that loops are often useful and present no problem when resolving path names. Unreachable loops, occurring as a result of unlinking, are a potential problem. Our solution is a remote on-line utility that identifies unreachable loops and links them into directory :ADMINISTRATOR: LOSTNFOUND for manual resolution.

Each directory is logically a table of (<string_name>,FID), and is itself identified by a unique FID. To minimize complexity and maximize performance, all directories are implemented as a database within a single file object. To ensure maximum concurrency, queries such as directory listing are given a consistent read-only copy of the database, while all updates are serialized through one task.

The responsibilities of, and the rationale for, the *agent* layer (Figure 2) are closely coupled to the system decomposition problem and are discussed in the next section. Finally, the *client* layer, which is also called client Helix (Figure 2), assumes responsibility for all the remaining problems. These include buffering and record-level I/O, support for large-scale atomic actions, resource recovery, and the responsibility for obtaining and managing a client's capability list.

System decomposition. An approach based on abstraction layering has been traditionally considered a good way to design a system.[2] In a nondistributed system, few design decisions remain once the layering is developed. However, the designer of a distributed system has to decide how to map the functional units identified by the abstraction layering onto specific hardware components.

The first decision we made was not to split the functionality of any single layer among several nodes. We also decided that the system would be partitioned into two parts (Figure 3). The first part, called client Helix, is located on the client's workstation, while the second part consists of one or more Helix servers. Each server runs on the node containing the data or the physical device. Thus, our decision was reduced to deciding at which layer the split should occur.

Several choices were possible. A division at the canonical disk layer or the cache layer would produce a partitioning, in which the server would be essentially a remote disk. Without provisions for disk-space allocation, such

a server is of little use. Adding the next two layers (system and secure tree) does not help, since the resulting server is not closed in an algebraic sense. The result is little autonomy and a high communication overhead.

Decomposition is more feasible at the object layer, with the server providing capability-based access to a set of commitable objects. A system based on such decomposition is described in detail elsewhere.[1]

We decided to go one step further and make each server also responsible for managing the user-organized directories. This approach has several advantages. First, each server type can choose a locally optimal implementation of the directory structure; at the same time all servers are obliged to provide read-only access to directories in a standard canonical format.

The second advantage is that directory update is simplified, since problems such as distributed deadlocks simply do not arise. (We also avoid the local deadlock problem by treating all directory updates as transactions handled by a single task.) Finally, decomposition provides for a convenient network interface (through the name server) and minimizes communications overhead.

At this point, giving the servers more responsibilities does not make sense. For example, it is obviously desirable to do block transfers over the local network, with buffering and record level I/O being done at the workstation. In the resulting decomposition, client Helix is responsible for buffering, record-level I/O, access domain management, and some coordination functions, while servers implement directories, file interlocking, and block-level access.

At each step, the crucial decision is to identify the next problem to be solved. Decisions involving alternative solutions tend to be easier.

Having identified the decomposition, we added the agent layer to support a standard interface to all servers. The interface chosen is a tasking interface, with client Helix using remote rendezvous (see article by Gammage and Casey in this issue) when communicating with Helix servers.

Server interface and autonomy

All server types implement the standard interface consisting of three task types: logger, agent, and object instance. Upon restart or reboot, the logger task registers the server name with the name server. The logger then processes incoming LOGIN() requests by creating an agent task for each LOGIN() request that satisfies the password validity check and returns both the agent Taskid and the capability for the user directory.

The agent processes incoming requests from client Helix. A request is associated with a capability for an object (typically a directory) and a path name relative to this directory, and it specifies the operation to be performed. The server translates the path name into a capability for an object, checks the capability for validity, and performs the operation desired.

A request for object access or creation produces an object instance task. The interface supported by this task facilitates access to a variety of objects—files, directories, pipes, terminals, printers, etc. It consists of two basic operations, called STD_READ() and STD_WRITE(), supporting block transfers in multiples of the block size specific to this server.

By enforcing a standard server interface, we obtained a system with a high degree of both uniformity and autonomy. While all server types must implement the interface, they are free to choose optimal internal structuring. In most cases, the choice is made simply by eliminating or simplifying one or more of the abstraction layers; for example, since performance in a shared environment is not a concern of the private Winchester server, it does not

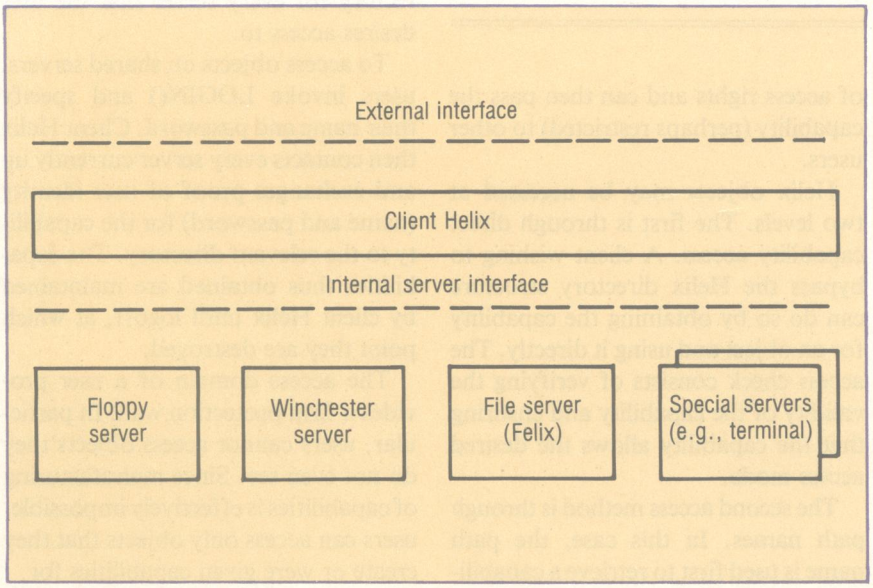

Figure 3. Helix components.

implement the cache layer. To be compatible with the UCSD file system, the floppy server is designed and structured differently. Special servers, such as a terminal or printer, are structured differently as a rule.

When initialized, each server is given a unique string name. A unique 32-bit string, based on this name (and several other factors), becomes a part of the capability for any object within that server. All other naming issues are purely local. In particular, all servers are free to implement a directory and capability mechanism of their own. Capabilities are also encrypted independently by each server (encryption is discussed in detail later).

Since Helix does not allow volumes to be cross-linked because of the need for fault tolerance, the failure of one server does not impact the others. Thus, at the server level, Helix can be characterized architecturally as a collection of autonomous volumes.

To its users, Helix appears as *one homogeneous system.* For example, users invoke only one LOGIN() command to access any and all volumes to which they have access. One interface provides access to all objects—whether local or remote. In addition, atomic actions on objects provide a high degree of transactional support.

Capability access

Our decision to use capabilities was a natural byproduct of our abstraction layering scheme. For example, because the object layer deals with a multiplicity of objects, it must be concerned with naming and may well be concerned with protection and access control. Introducing a user-controlled name space at this point did not appear to be a good idea. The FID, as a system-generated name and an access capability for objects provides a natural solution.

A good mechanism must generate capabilities that are both unique and unforgeable. Given a context, we can usually guarantee uniqueness. Making a capability unforgeable is more difficult. A pragmatic solution is to choose a name space that is sufficiently sparse to make enumerative capability generation next to impossible. (To avoid subversion, a server presented with an invalid capability must do more than just refuse access. While it may be impossible to identify the offender, reporting and logging the problem are reasonable alternatives.)

A Helix capability is a 96-bit string, with 32 bits being used to identify the volume within a local area network. Of the remaining 64 bits, four reflect the access rights, 16 identify the object within a server, 40 provide the sparseness and randomness, and four are spare. The sparseness of the name space is a guarantee that the same capability will not be generated within a certain time (our estimate is approximately 20 years). All capabilities are encrypted as they leave the server and decrypted on (re)entry.

Associated with a capability is a set of *access rights.* The full set consists of (R, W, L, U), with R/W denoting permission for read/write access to a file and L/U denoting link/unlink rights on a directory. The access rights apply recursively, allowing us to give a read-only permission to a set of objects by passing a read-only capability for the root of the directory substructure involved. The creator of an object is always given a capability with a full set of access rights and can then pass the capability (perhaps restricted) to other users.

The decision to use capabilities was a natural byproduct of the abstraction layering scheme.

Helix objects may be accessed at two levels. The first is through direct capability access. A client wishing to bypass the Helix directory structure can do so by obtaining the capability for an object and using it directly. The access check consists of verifying the validity of the capability and ensuring that the capability allows the desired access mode.

The second access method is through path names. In this case, the path name is used first to retrieve a capability. The access rights associated with the capability are then set to reflect the logical intersection of all access right sets along the path. A standard access check is then performed as just described. Capability *passing* is facilitated by the MAKE_PATH (<src>, <dest>, <rights>) primitive, and is performed in two steps. First, the capability identified by the <src> path name is retrieved, and a copy with modified access rights is made. Capabilities cannot be amplified; that is, donors cannot specify a set of access rights that they do not have. In the second step, the capability is deposited into the destination directory, subject to a check for a link permission on this directory.

A more general discussion of the advantages and disadvantages of the two basic protection mechanisms—capabilities and access lists—is given in the box on the facing page.

User access domain and security

The security of Helix is based on the notion of an *access domain* associated with a user. An access domain is the set of objects, and access rights for these objects, that can be reached from a given user directory.

A user directory is a directory with a special mark—an encrypted password of the user. In all other respects, it is just like any other directory. User directories are created by the Helix administrator, who makes one user directory for every server that the user desires access to.

To access objects on shared servers, users invoke LOGIN() and specify their name and password. Client Helix then contacts every server currently up and exchanges proof of user identity (name and password) for the capability to the relevant directory. The capabilities thus obtained are maintained by client Helix until logoff, at which point they are destroyed.

The access domain of a user provides a firm protection wall. In particular, users cannot access objects they do not even see. Since manufacturing of capabilities is effectively impossible, users can access only objects that they create or were given capabilities for.

Atomic actions

Each server supports *atomic* operations on objects. A commit of a single object is facilitated by the secure tree layer, which provides a conditional commit operation to upper layers. A full commit sequence must then include an update of the FID directory (to reflect the new tree root), followed by a commit of the FID directory at the system level. The whole operation is quite efficient, requiring only a few (typically four) disk writes.

A simple extension facilitates an atomic commit of a set of objects. The extension consists of two brackets, START_COMMIT() and END_COMMIT() (or ABORT()), provided by the object layer. A number of updates can thus be performed in one indivisible operation. Helix uses atomic set commitment internally; for example, file creation is an atomic transaction that includes a commit of both the new file and the updated Helix directory.

Helix users are given both explicit control over commitment and a mechanism for building large-scale atomic actions. The key instrument in constructing such atomic actions is the *file set*. File sets are managed through use of a simple stack-oriented mechanism. As each file is opened, it is automatically pushed onto the stack (thus being included in the currently active set). File sets are nested, with the FS_NEW() primitives marking a new set, and FS_CLOSE() closing and optionally committing all object instances enclosed.

As Gray and Cooper note, atomic multiserver commit is a difficult problem.[3,4] The approach we chose is a variant of the two-phase commit protocol, which works in the following way. First, client Helix invokes START_COMMIT() on all servers involved, followed by a commit operation on all files in the set. Communication failures or node crashes during this step are handled by timeouts performed by each server.

In step two, all servers involved are precommitted. However, no timeouts are employed during the second phase. Instead, each server involved simply waits for the commit completion command (END_COMMIT() or ABORT()), or for a manual intervention specifying the same. Deadlock problems during the commit phase are avoided by choosing an arbitrary, but fixed, global ordering in which servers are committed. Given the reliability of our local area network, this approach seems to be justified. Simplicity and reliability are its main advantages.

Access control

Any system supporting shared access must deal with access control and security. A system access table such as the one below can be used to depict the problem and its solutions. The row headings of this table represent system users; the column headings represent all relevant objects (files, directories, etc.). Entry (*i,j*) then defines whether access to object *j* is allowed by user *i*, and if so, what the permissible access modes are.

The two basic access-control approaches—capabilities and access lists— differ in how they represent and store the access matrix. Capability schemes use a row-wise representation, and the access-control information is stored with the user. Each user is provided with a capability list (directory) that identifies all objects and associated access permissions.

Access lists, on the other hand, store the control information in column-wise fashion. The information, which is kept with the object (not the user), consists of user names (not object names) and associated access rights.

Each approach has advantages and disadvantages. Access lists do not easily accommodate changes in user population; for example, the addition of a single user may require the modification of a number of access lists. With capability systems, the revocation of a capability previously granted may be difficult.

Table 1. Access mode compatibility.

New Request	Existing Instances: Nil	RC	WC	RO	WO	RX	WX
RC	Y	Y	Y	Y	Y	—	—
WC	Y	Y	Y	Y	Y	—	—
RO	Y	Y	Y	Y	—	—	—
WO	Y	Y	Y	Y	—	—	—
RX	Y	—	—	—	—	—	—
WX	Y	—	—	—	—	—	—

Concurrency control

Helix supports shared, concurrent access to objects, with the locking unit being a single file or a directory.

An access request must specify the desired mode of access. There are six access modes—(RC, WC, RO, WO, RX, WX). Each mode is a combination of an access type, read (R) or write (W), and an object lock, copy (C), original (O), or exclusive (X). Requests for access in WO or WX modes produce committable instances of the object, while the other access modes yield noncommittable instances.[1] Only one committable instance of an object may exist at any given time; at the same time, a large degree of sharing and concurrency is possible, as shown in Table 1.

Resource management

In our context, resources are all Helix objects, locks on such objects, and the auxiliary resources required, such as memory or disk storage. Re-

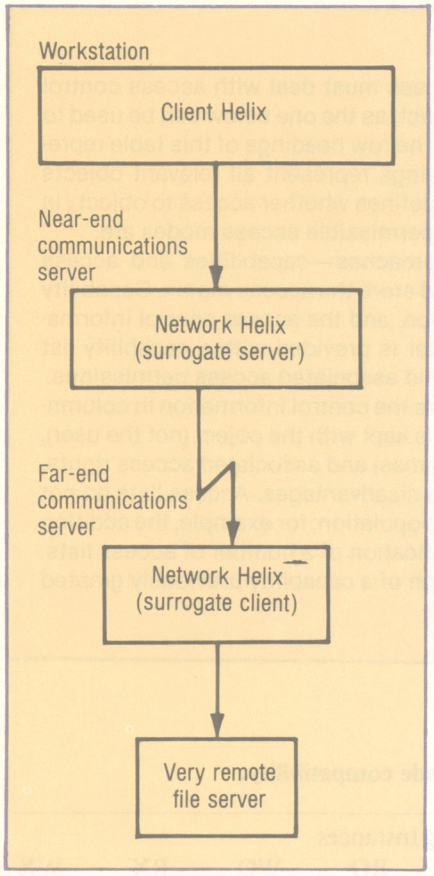

Figure 4. Very remote file access.

source management is closely coupled with the protection mechanism. For example, a client wishing to access an object must first present a capability for this object. The server involved performs a capability check, allocates the necessary resources, and creates an object instance task. This task is then responsible for the resources allocated, while its taskid is effectively a short-term capability for these resources.

The recovery problem occurs at three levels. The first problem arises when an application program fails or terminates without releasing all the resources it has acquired. File sets provide a suitable tool for an organized and efficient cleanup in this case. For example, the command interpreter marks the beginning of a new set just before invoking a program. If the program succeeds, the set can be committed; if the program fails, the set is closed/aborted, and all resources acquired by the failing program are released.

The second problem occurs when a node crashes without releasing resources on various servers. We made each resource-handling task on a server responsible for periodically checking the status of its clients—a task carried out by the agent layer. In particular, object instances and agents maintain the taskid of their respective clients and perform a periodic check.

The last problem is resource recovery within a single server, in which reclaiming disk storage is a major issue. Like many other systems, Helix uses auditing to rebuild the free storage map. An attractive feature of our commitment structure is that audits (and tape backups)can be run on line with no service interruption.

Performance

Two factors affect the performance of a distributed storage system. One is the cost of physical disk access, and the other is communication overhead.

The speed with which an object can be accessed depends on the storage allocation policy. If an object is written as a series of blocks on the same track, or several adjacent tracks, seeking can be largely eliminated. While Helix does not use such an allocation policy as a rule, it uses a more general mechanism, giving the same result as a very frequent special case.

Storage is allocated by the system layer, which allocates blocks sequentially from a bit map when requested by the secure tree layer. This allocation policy and the "write with a move" technique employed by the tree layer ensure that writes are tightly clustered. As a result, a sequentially written object is likely to be contiguous; the probability that it is contiguous is inversely proportional to the number of objects being written simultaneously. Similarly, few seeks are required when reading data (either a single object or a number of objects) that was written "at the same time," while many seeks may be required when objects of "different age" are read.

Block data transfers across the local area network are facilitated by remote rendezvous. Each STD_READ() or STD_WRITE() invokes one rendezvous; within that rendezvous, one or more 1K blocks may be transferred by a less expensive push/pull operation (similar to Thoth's "transfer"[5]).

We have measured file access time at the user level. For bulk sequential access, the typical cost is 32 ms per 1024 bytes (reading or writing). The response time for random single-block access when disk I/O and LAN transfer do not overlap is almost twice that. This figure compares favorably with performance achieved by other remote storage systems. For example, Dion quotes access times of 50 ms to read 512 bytes and 65 ms to write 512 bytes.[6]

Networking

In addition to LAN-wide access, Helix supports access to objects on file servers in remote local area networks. The implementation consists of exactly one addition—a special version of the agent layer called network Helix. This layer consists of two components; one acts as a surrogate server at the near end, and the other as a surrogate client at the remote end.

Both components, which run on the communication server, transparently extend remote rendezvous across the network, as suggested in Figure 4. At the near end, network Helix pretends to be the very remote server by implementing the standard server interface and by transparently passing all requests to the far end via virtual circuits provided by the communication server.

A very remote file server appears to its clients as a local file server, with access speed being the only difference. With 9.6K-Bps communication links, very remote access is slower by a factor of about 50.

We have described an architecture for a distributed, shared, multimedia file system with a high degree of security and crash resistance. The architecture was developed as a series of abstraction layers and with

system decomposition over a local area network.

Our initial implementation has been operational since early 1983. In late 1984, approximately 15 LANs and close to 50 file servers were supporting almost 1000 workstations.

Experience to date indicates that Helix satisfies the design objectives. In spite of the variety and the physical distribution of servers, it does indeed appear as one homogeneous system to its users. The capability-access mechanism has proved highly flexible and secure; on the other hand, we have found that with no inherent concept of file ownership and no restrictions on linking, administrative utilities and procedures are required to control the amount of disk storage used.

As a result of the commitment structure described, system integrity is so good that no "shut-down" command is provided. We routinely take down file servers by pushing "reset," or by powering down.

We have observed that atomic actions do indeed provide integrity and crash resistance. However, we feel that atomic multivolume commitment does not solve all multivolume integrity problems. A full solution must include atomic multivolume backup/restore, a concise definition of integrity regions, and the handling of boundary conditions. Moreover, multivolume commitment and volume independence are often contradictory; for example, having to restore all six file servers because of a head crash on one is not an acceptable requirement.

A single tape server with a nine-track tape drive is used to back up all file servers on one LAN. The backup is performed at logical file level at the speed of approximately 50M bytes per hour. Although, in the future, we hope to use optical disks for archiving and backup, the speed and efficiency of the tape backup process is an important improvement item now.

A pleasant surprise is the performance of remote operations. During routine operations such as editing or compiling, the subjective feeling is that local and remote operations are comparable in speed. Very remote access via network Helix, although significantly slower, has proved extremely useful and highly popular in our distributed development environment. The ease and speed with which we were able to implement networking enhancements appear to confirm the advantages of the layered architecture. □

References

1. M. Fridrich, and W. Older, "The Felix File Server," *Proc. Eighth ACM Sigops Symp. Operating Systems Principles,* Dec. 1981, pp. 37-44.
2. E. W. Dijkstra, "The Structure of 'THE'—Multiprogramming System," *Comm. ACM,* Vol 11, No. 5, May 1968, pp. 341-346.
3. J. N. Gray, "Notes on Data Base Operating Systems," *Lecture Notes in Computer Science,* Vol. 60, Springer-Verlag, New York, 1978, pp. 393-481.
4. E. Cooper, "Analysis of Distributed Commit Protocols," *Proc. ACM Sigmod Int'l Conf. Management of Data,* June 1982.
5. D. R. Cheriton et al. "Thoth, A Portable Real-Time Operating System," *Comm. ACM,* Vol. 22, No. 2, Feb. 1979, pp. 105-114.
6. J. Dion, "The Cambridge File Server," *Operating Systems Rev.,* Vol. 14, No. 4, Oct. 1980, pp. 26-35.

Further reading

R. S. Fabry, "Capability Based Addressing, " *Comm. ACM,* Vol. 17, No. 7, July 1974, pp. 403-412. [access control, particularly capability access]

F. Pollack, K. Kahn, and R. Wilkinson, "The iMAX-432 Object Filing System," *Proc. Eighth ACM Sigops Symp. Operating Systems Principles,* Dec. 1981. [object-oriented file systems]

D. P. Reed, "Implementing Atomic Actions on Decentralized Data," *Proc. Seventh ACM Sigops Symp. Operating Systems Principles,* Dec. 1979. [concepts and implementations of atomic actions]

D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Comm. ACM,* Vol. 17, No. 7, July 1974, pp. 365-375. [a good overall description of AT&T Bell Laboratories' Unix, which has become the standard by which many other systems are measured]

H. E. Sturgis, J. G. Mitchell, and J. Israel, "Issues in the Design and Use of a Distributed File System," *Operating Systems Rev.,* Vol. 14, No. 3, July 1980, pp. 55-69. [distributed file systems in a single LAN]

M. Wilkes and R. Needham, *The Cambridge CAP Computer and Its Operating System,* North Holland, New York, 1979. [access control, particularly capability access]

Marek Fridrich joined BNR in 1973 to become one of the original designers and implementers of the SL-1 PABX. In 1977, his interest in office automation led to design work in the area of text messaging and data communications. Two years later, he joined the XMS project with responsibility for the distributed file system and communications. He is now with BNR Inc. in Mountain View, California, working on networking for Northern Telecom's Meridian system.

Fridrich holds a PhD in computer science (University of Alberta, 1973), and an MSc in electrical engineering (Czech Technical University, 1965).

His address is BNR, PO Box 7277, Mountain View, CA 94039.

William J. Older has worked for BNR since 1969 in various areas of software systems. After early work on a hybrid thick-film CAD system and high-level system programming languages, he joined the SL-10 packet switching network project in 1974 as system architect and software designer/implementer.

He became interested in database problems in 1978. In 1980, he joined the XMS project to work on file servers and distributed file systems. Since 1984, he has been involved primarily in logic programming languages.

His address is BNR, PO Box 3511, Station C, Ottawa, Ontario, Canada, K1Y 4H7.