M.I.T. LABORATORY FOR COMPUTER SCIENCE
Computer Systems Research Division

June 25, 1980
Request for Comments No. 192

# *SWALLOW:* A DISTRIBUTED DATA STORAGE SYSTEM FOR A LOCAL NETWORK

David Reed
Liba Svobodova

This paper has been submitted to the International Workshop on Local Networks sponsored by the IBM Zurich Research Laboratory in August 1980.

# *SWALLOW*: A DISTRIBUTED DATA STORAGE SYSTEM FOR A LOCAL NETWORK

David Reed
Liba Svobodova

M.I.T. Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

## ABSTRACT

SWALLOW is an experimental project that will test feasibility of several advanced ideas on design of object-oriented distributed systems. Its purpose is to provide a reliable, secure and efficient storage in a distributed environment consisting of many personal machines and one or more shared repositories. SWALLOW implements a uniform interface to all objects accessible from a personal computer: these objects can be stored either on the local storage device or in one of the repositories. The repositories provide stable, reliable, long-term storage. The access control to objects in the repositories is based on encrypting the data; encryption is used to prevent both unauthorized release of information and unauthorized modification. SWALLOW can handle efficiently both very small and very large objects and it provides mechanisms for updating of a group of objects at one or more physical nodes in a single atomic action.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

## 1. Introduction

SWALLOW is an experimental distributed data storage system being developed at the Computer Systems Research Group of the Laboratory for Computer Science at M.I.T. The purpose of this project is to test several advanced ideas on design of reliable secure object-oriented distributed systems. The project has emerged from a broader research on computer networks [CLAR 79] and distributed processing [SVOB 79] conducted by this research group over the past several years.

SWALLOW is intended for a distributed environment that consists of a network of cooperating but highly autonomous computers. The term *autonomy* characterizes the ability of the individual network nodes to operate independently when disconnected from the network, but also the ability to administer their own resources: the allocation of the local hardware resources, and most importantly, the processing and release of the local information.

SWALLOW can be viewed as a set of standard protocols that cooperating computers may use to manage their data. It is not intended to be used directly by human users: it defines a functional interface for other programs. If SWALLOW is to be successful in the stated environment, this interface must provide a natural support for a broad class of application programs. In particular, SWALLOW provides:

*Uniform interface* - the read and write operations defined for the objects managed by SWALLOW make the location of data stored in the system transparent. However, the owners of data are allowed to control the location of data either directly, by specifying where the data are to reside, or indirectly, by specifying the desired properties of the stored data, in particular, stability and security.

*Reliability* - SWALLOW provides storage for data objects that is extremely stable. In addition, only those nodes that hold data needed by a computation need be available to run that computation, so availability is enhanced.

*Atomic actions* - SWALLOW provides synchronization and recovery mechanisms so that any arbitrary set of operations on an arbitrary set of objects may be combined into an atomic action. Network failures and node crashes do not compromise proper synchronization and recovery of these atomic actions.

*Protection* - SWALLOW provides encryption-based protection mechanisms. These mechanisms are decentralized, so that there is no critical central authority that can compromise the security of every user of SWALLOW.

*Support for small objects* - SWALLOW can support a very large number of objects whose average size is relatively small. These objects are not only stored and retrieved as separate entities, but different access restrictions can be specified and enforced for each individual object.

SWALLOW supports only untyped objects. An object is represented as a history of the states assumed by the object since its creation, following the model developed by Reed [REED 78, REED 79]. This model provides the basis for synchronization and recovery in the implementation of atomic actions. The very high reliability of SWALLOW is accomplished through a combination of an extensive use of stable storage and careful internode protocols.

## 1.1 Related work

The projects most closely related to SWALLOW are the WFS [SWIN 79, PAXT 79] and Juniper [LAMP 79, ISRA 78]. Both of these projects implement a file server for a similar environment as postulated in our project. The WFS provides to its clients a page-level access to files stored on a remote server. Only operations on single pages are guaranteed to be atomic. However, a transaction system that runs in a client computer was developed as an extention of WFS [PAXT 79]. This system supports atomic actions that involve not just multiple pages but multiple files. The Juniper falls in between: it is a distributed server that implements multi-file atomic actions without participation of the clients.

SWALLOW is closer to Juniper, but it goes substantially beyond the facilities of any of these servers in the following:

-it provides a uniform interface to remote data (maintained by shared servers) *and* local data (maintained by the client computers)

-it supports objects of arbitrary sizes, with provisions for large numbers of small objects

-it associates protection mechanisms with individual objects *and* with atomic actions on collections of objects

-it is integrated with other kinds of servers, namely, the name lookup and authentication servers.

Our project draws also on other previous and current work in the areas of management of distributed databases, file systems, crash recovery, protection, and management of small objects. However, while most work that is concerned with reliable update of distributed databases concentrates on perfecting the *protocols* for concurrency control and commitment synchronization, our intent is to design and implement the actual *mechanisms* that are necessary to make any of these protocols feasible. The main consideration is the robustness of these mechanisms; we use techniques similar to those employed in Juniper [LAMS 79A] and the transaction manager for the System R [GRAY 79]. Our protection scheme assumes an existence of an authentication and key distribution server; protocols for secure key distribution were developed by Needham and Schroeder [NEED 78].

## 2. Object model

SWALLOW allows the construction of atomic actions. Each atomic action is a (possibly distributed) computation that can read or write an arbitrary set of objects in the system. By definition, an atomic action, though composed of individual steps that read and modify objects, cannot be decomposed from the point of view of computations outside the atomic action. This view implies two properties:

> *Atomicity in the face of failure*—an atomic action either executes in its entirety without failure or else it has no effect. No updates are exposed for reading until the atomic action is guaranteed to complete.
>
> *Atomicity in the face of concurrency*—the accesses to objects that are part of the same atomic action either all precede or all follow any other access not part of the atomic action. This implies that the effect of applying a set of atomic actions to the state of the system is equivalent to some *serial schedule* of those actions.

SWALLOW provides for atomic actions by using the object model developed by Reed [REED 78, REED 79]. Here we recapitulate the important features of that model, though our treatment is necessarily sketchy. In Reed's model, each object is represented as a sequence of versions, called a *version history*. Each write to an object produces a distinct version; the versions themselves are immutable once created. Reading an object is done by selecting the appropriate version, according to the mechanism discussed below.

All accesses to objects in the system are totally ordered by assigning each access a number called its *pseudo-time*. Each atomic action is assigned at its start a range of pseudo-times that does not overlap with the range assigned to any other atomic action. Successive accesses in an atomic action are assigned pseudo-times from that range in increasing order.

The versions of each object are ordered by the pseudo-times of the write requests that created them.

Reading an object at pseudo-time $p$ selects the version of maximal creation pseudo-time $q$, such that $q < p$. Write and read requests belonging to concurrent atomic actions are executed in the (real time) order of arrival at the object. Write requests that arrive at an object out of pseudo-time order may be refused. This happens if a write request with pseudo-time $p_2$ arrives at an object that has a version created at pseudo-time $p_1 < p_2$, which version has been returned as the result of a read at pseudo-time $p_3 > p_2$. If a write request is refused, the atomic action issuing the write request is aborted, unless the write is not required for correct completion of the atomic action. Consequently, the effect of a set of atomic actions that correctly complete is equivalent to a serial schedule of the atomic actions according to their pseudo-times.

The ability to abort a partially completed atomic action that encounters a failure (a synchroniztion failure like the refused write, a network or node failure detetected by a timeout, or a protection violation) is provided by grouping the versions created by an atomic action into a set called a *possibility*. A possibility is a group of versions that are not yet available to atomic actions outside the one that created them. When an atomic action completes successfully, it *commits* its possibility, making all of the versions it created available to all other atomic actions. On unsuccessful (non-) completion, the versions in the possibility are aborted. If an atomic action attempts to read a version belonging to an uncommitted possibility, it must wait until the state of that possibility is known. Thus, none of the updates made by an atomic action is exposed until the atomic action succeeds.

Possibilities are implemented by requiring that each version refer to a *commit record* that represents the state of all of the versions of the possibility. Initially the commit record is in the UNKNOWN state. The possibility is committed by changing the commit record state from UNKNOWN to COMMITTED. It is aborted by changing the commit record state from UNKNOWN to ABORTED. No other state changes are allowed.

## 3. Organization of SWALLOW

Figure 1 illustrates the overall structure of the SWALLOW system. Each client computer that uses SWALLOW accesses stored objects via a module called the *broker*, which is implemented on each client. The data owned by that computer is stored either on local secondary storage or on a shared server called the *repository*. SWALLOW can have several repositories; individual client computers may have access to different subsets of them.

The broker controls the location of, and mediates all accesses to, the data owned by its client. These functions are achieved through interactions with the repositories and other servers: the name lookup server and the authentication server.

The repository provides large quantities of stable storage, but it supports only the minimum necessary set of functions. In particular, a repository is not responsible for protecting the data stored in it from unauthorized release, and, although it implements some mechanisms needed for construction of atomic actions, the responsibility for managing atomic actions rests with the brokers.

## 3.1 Organization of the repositories

We are assuming that the information stored in the individual client computers is not *directly* accessible from other nodes, that is, it is not possible to change or even examine the raw data (of course, an application program can make local data available to programs running at different nodes). Also, most of the clients will be "personal computers" that may not have sufficient facilities to store their own data reliably. Thus the SWALLOW repository provides two main functions:

1. reliable and secure long-term storage
2. direct data sharing

The repository implements the object model described in the preceding section; this provides the basic mechanism for access synchronization to shared data. Protection of shared data is achieved through encryption, as will be discussed later.

For reliable long-term storage of objects, the physical storage in the repository must be stable, that is, the information stored in it must not decay over time. In addition, it is necessary to ensure that information written to it is either written completely and correctly or not at all, that is, that the operations on stable storage are *atomic*. Since no physical device provides storage with these properties, the atomic stable storage must be implemented as an abstraction, using hardware components with less desirable properties [LAMS 79A]. In particular, atomic stable storage must be designed to tolerate processor crashes during write operations and decays of the storage media. This is accomplished by writing the data twice, into decay-independent sets.

An operation that is most difficult to perform atomically is an in-place update of stored information. An atomic update means that either the content of the updated entity is changed into the new value or, if the operation fails, the value of this entity is left unchanged. That is, atomicity guarantees that a stored entity is never left in an inconsistent state where the old value has been lost and the new value is incorrect. To perform an atomic update, the two copies of stored information in the two decay-independent sets must be changed strictly sequentially, i.e. the first write must complete successfully (correct data written to correct address) before the second write is initiated. If the storage model does not have to support an update operation, the problem of atomicity is simplified. It is still necessary to have two copies for stability, and the ability to detect and correct bad writes, but the two writes into the two decay-independent sets can be done concurrently. This simplification is a strong motivation not only for the adaptation of Reed's object model, but for overall organization of the repository around a stable *append-only* storage called the *version storage* that contains not just the data objects but also all the information needed for crash recovery.

The given object model will require a large amount of storage. Thus, it is important to utilize storage devices that are: 1) inexpensive, 2) are easy to store offline. To provide fast access to an arbitrary version of an arbitrary object, a random access device is needed. A promising option is to use optical disks. Since optical disks provide *write-once* storage, the append-only model of version storage is crucial to making such an implementation feasible.

## 4. Protocols

The protocols of SWALLOW are designed to ensure that atomic actions involving multiple sites work correctly in the face of arbitrary failures of networks and nodes. We require that only those brokers and repositories involved in atomic action need be operational and accessible for that action to succeed. The set of brokers, repositories, and network communications facilities used need not be known in advance.

The protocols used by SWALLOW are based on those suggested by Reed [REED 78, REED 79]. We are primarily concerned with performance and reliability issues in the design of the protocols. The basic protocol on which all higher-level protocols are built is the SWALLOW Message Protocol (SMP). The purpose of the SWALLOW Message Protocol is to provide a means for sending single arbitrary-sized messages from one computation to some "logical port" on another machine. It is a datagram-style protocol, with some differences we will discuss. Another basic protocol, the pseudo-time clock protocol (PCP), synchronizes the logical clocks at each broker that are used to assign pseudo-times to atomic actions. The high-level protocols of SWALLOW fall into two categories—the protocols for storing and retrieving objects from remote repositories, and the protocols that implement atomic actions.

### 4.1 SWALLOW Message Protocol

The SWALLOW Message Protocol is used for all communications between components of the SWALLOW system. The protocol is specialized to the needs of SWALLOW, and built on top of the Internet Datagram Protocol defined by the DARPA Internetwork Working Group [POST 80]. The protocol defines a set of *ports* at each computer node in the system, and allows any computation at a node to send a message to a port at any node.

Messages are not limited to a single packet. However, if a single packet will hold a message, then only that packet will be transmitted. No connection setup is required. If a message requires multiple packets, then the SMP flow control mechanism controls the rate at which the second and successive packets are delivered, to avoid overflowing the receiver's buffer capacity. The flow control is a windowing scheme, where the window is controlled by the receiver.

There is no attempt to mask all failures at the message protocol level. A variety of surprising effects can occur, such as duplicate delivery of messages, lost messages, and reordering of messages within the network. These problems are naturally handled by the higher level protocols. It is not necessary to detect duplicate messages since requests pertaining to data accessing and management of atomic actions are processed in such a way that they are idempotent—doing the same operation over again has no effect. Lost messages are handled correctly by the higher level protocols because each request has an associated response that serves as a positive acknowledgement of *both* the delivery of the message and the completion of the action requested.

### 4.2 Synchronization of Pseudo-time clocks

Generation of mutualy exclusive pseudo-time environments for atomic actions is the responsibility of the brokers. Each broker generates pseudo-times for its atomic actions by using a *pseudo-time clock*—a cell that maintains a monotonically increasing integer value. There are two requirements on the values obtained by reading the pseudo-time clocks. Each value obtained must be unique over the set of all values obtained from all pseudo-time clocks. This can be satisfied by assigning a unique identifier to each pseudo-time clock, which identifier is then returned as the low-order bits of any value read from the pseudo-time clock. The second requirement is that two readings that occur at distinguishable real times (that is, where it is apparent from outside of the system that one value was read before the other) should reflect the order of the readings. Thus the pseudo-time clocks must be approximately synchronized. The protocol used to achieve this approximate synchronization is based on the technique suggested by Lamport [LAMP 78], though SWALLOW has somewhat less rigorous requirements for synchronization.

### 4.3 Protocol for accessing data on repositories

The brokers access data stored on a repository by means of read and write requests. The version to be read or created is specified in the request by an object unique identifier and a pseudo-time. In addition, each request includes a commit record unique identifier. The pseudo-time and commit record unique identifier depend on the atomic action responsible for the access.

A read-request contains simply a specification of the version desired. If the repository can provide the proper version immediately, it will respond with a message that consists of the value of the version. If the version is very old and has migrated to offline storage, or if the version desired has been created by an atomic action that has not yet committed, the read request must be delayed and the repository will respond with a status response. A status response is also returned when the version does not exist: either the specified object has never been created, has never been initialized, or has been deleted by a request with an earlier pseudo-time. However, the repository may not respond at all, if it or the network encounters a failure. The broker is free to retransmit a read-request at any time, since repeating a read on an object has no side-effects. The response to a read-request includes the object unique identifier and pseudo-time, which are sufficient to allow the broker to ignore extra reponses that may be generated by retransmission or duplication of the message in the network.

A write-request specifies which version is to be created, and contains the value of that version. If the repository can create that version, it responds with a positive acknowledgment when the version has been created, added to the possibility denoted by the commit record identifier, and is in stable storage. If the version has already been created (as in the case when a write-request is duplicated or retransmitted), that fact is detected because the object unique identifier, pseudo-time, and commit record unique identifier match an existing version. Such a request is also positively acknowledged, because the retransmission may have been due to a lost acknowledgment. If the version cannot be created because it conflicts with some other version or because the object cannot be accessed, a negative acknowledgment is returned.

The responses to both read and write requests serve two purposes. They transmit the result of the action requested at the repository, and they implicitly provide confirmation that the repository received the requests. No low-level acknowledgment is used to provide this latter confirmation. The number of network packets transmitted is thus reduced significantly.

Since the values of objects may be of an arbitrary size, messages that transmit such values (e.g., write-requests and responses to read-requests) may be arbitrarily long. In the common case where objects are small, the responses will fit in one packet, so the interaction associated with a such an access will require only two packets—one packet in and one packet out. No lower-level acknowledgment or connection setup will be required, so both the delay and network overhead will be minimized. For large versions, the flow control mechanisms of the message protocol allow the data to flow at a maximum possible rate between the repository's stable storage device and the broker's stable storage device.

The repository is designed in such a way that it can process and transmit large versions piecemeal.

As individual fragments are delivered via SMP packets, they are stored on the repository's stable storage. The SMP flow control mechanism adapts the rate of transmission to the characteristics of the repository stable storage. The fragments can be of different sizes. Further, since objects are never updated in place, different versions of the same object can be fragmented differently; this allows for maximum flexibility in flow control.

## 4.4 Protocol for managing atomic actions

As we noted earlier, a major function of SWALLOW is to coordinate atomic actions. The tools for doing this coordination, pseudo-times and commit records, are managed by the SWALLOW protocols.

*Commit records*

Commit records are managed by repositories. Each commit record's state is maintained at a single repository, and is manipulated by remotely originated requests. When an atomic action is begun, the initiating broker requests creation of a commit record at some chosen repository. Thereafter, each version created by that atomic action is tagged with the name of the associated commit record. When the atomic action completes successfully, a COMMIT request is sent to the commit record's site. If the COMMIT request is honored, then the versions created by the atomic action become accessible to other atomic actions. The atomic action will be aborted if either an explicit ABORT request is received or a timeout elapses at the commit record.

As in Reed's model, an atomic action passes its *commit point* when the commit record is updated to the *committed* state. Versions become committed later, when they are informed of the final state of the commit record; the state is transmitted from commit record to version by two mechanisms.

- If a read is attempted on an uncommitted version, a request is made to the commit record's site to get the commit record's state; if *committed*, the version then is committed.

- When a commit record is committed, it broadcasts this fact to all versions tagged by the commit record; this broadcast may not get to all such versions, so this mechanism is viewed only as an optimization.

In SWALLOW, we have modified the protocol originally suggested by Reed for managing commit records, in order to enhance performance. In Reed's model, each version created in an atomic action is added to a list maintained with the commit record. This list serves two purposes—it is used to allow deletion of the commit record when all references to it are deleted and it is used to broadcast the final state of the commit record to its versions as soon as it is known. The modification in SWALLOW distributes this list. Each site that contains versions tagged with a particular commit record maintains a *commit record representative*, as shown in Figure 2. The commit record representative contains a list of the tagged versions at its site. The primary commit record contains only a list of *local* tagged versions and a list of sites that contain commit record representatives. The broadcast of a commit record's final state thus will usually require many fewer messages.

When a new version is created at a site, if there is no commit record representative at that site, one is created. Creating the representative requires an interaction with the primary commit record site. Once a commit record representative exists, new versions can be created without any further interactions with the primary site—each version is simply added to the represntative's list. This also substantially reduces the message traffic.

The cost of creating $N$ new versions in an atomic action at $S$ sites can be broken down as follows (this is the case where no failures occur).

| Messages | Purpose |
|---|---|
| 2 | create commit record |
| $2N$ | create versions. |
| $2S$-2 | create commit record representatives |
| 2 | commit commit record |
| $2S$-2 | commit versions, delete commit record. |

Thus the total number of messages is $2N+4S$.

*Pseudo-times*

Pseudo-times associated with the accesses of an atomic action are chosen by the brokers that perform the accesses. If multiple brokers are involved in an atomic action, then they must coordinate their use of pseudo-times. This coordination is done by a broker-broker protocol that hands off the state of an atomic action from one broker to another. In the initial version of SWALLOW, an atomic action is active at only one broker at a time. This restriction guarantees that successive accesses in an atomic action are made in successive pseudo-times.

When an atomic action is begun, the broker initiating the action receives two independent capabilities. These are a) the right to read and create versions of objects as part of the atomic action and b) the right to commit the atomic action. The first capability is usually passed when an atomic action becomes active at a new broker; the second is usually retained at the initiating broker because that broker usually knows best when the atomic action is correctly completed. Both may be passed, however. The right to abort an atomic action is part of capability a) above, since each broker that performs accesses on behalf of an atomic action has the option of failing to perform its part and thus aborting the action implicitly.

When a client of SWALLOW makes a request to another site to perform a part of an atomic action, it includes in its request the capabilities needed to perform that part. These capabilities are obtained in a "sealed package" from the client's broker, and passed in the request message. The broker at the site that is the target of the request can then unseal the package and perform accesses using the enclosed capabilities. Thus, the interbroker communication is piggybacked on the interclient communication, rather than requiring extra messages.

## 5. Protection

Protection of information in the SWALLOW system is based on the systematic use of encryption. We have chosen an encryption-based protection mechanism because it maximizes the opportunity for node autonomy in implementing protection policies. Our goal is to have each broker responsible for protecting the information it manages, with minimal responsibility placed on shared nodes such as repositories. The primary threats to information owned by the clients of a broker occur when that information is shipped on the network and when that information is stored on a shared repository. Another important threat to the information in the system is unauthorized tampering with the atomic action mechanism that results in inconsistency. We consider two generic classes of threats—unauthorized disclosure and unauthorized modification of the stored information. A third class of threats—unauthorized denial of service—can be detected, but not corrected within the system. An intruder can easily intercept, change, delete and introduce packets into the network. SWALLOW does not allow such interference to result in unauthorized disclosure of information, and prevents unauthorized modification of information.

We assume that clients communicate using encrypted messages, and that some sort of authentication server allows the clients to authenticate such communications. Our concern here is primarily with the broker-repository communications. This communication can be broken down into two kinds, as before—those directly concerned with accessing versions of objects and those associated with commit records.

The broker-repository communication requires a special kind of authentication. The broker needs to detect the case where an intruder tries to impersonate a repository. A repository that does not remember the data stored there can have an effect equivalent to unauthorized modification or unauthorized denial of service. Assuming the broker trusts the repository, the broker needs to determine whether a response to its request is (a) from the correct repository, and (b) a response to that request (not a duplicated response to another request). This authentication of responses can be done by generating a digital signature at the repository that signs the request-response pair. Needham and Schroeder discuss a variety of such techniques.[NEED 78].

There are two kinds of accesses to objects, reading and writing. Unauthorized disclosure of objects on the repository is prevented by encrypting each version of an object under an object key (OBK) known only to authorized brokers. Versions transmitted from broker to repository are encrypted, so intercepted network messages do not expose the contents of versions. The repository does no authentication of requests to read objects. This simplifies the repository, at the expense of relegating more work to the broker.

In principle, encrypting objects in a key known only to the broker also protects against unauthorized modification, since a version created with the wrong key would be detected when an authorized user attempted to read it and failed. Since all versions of an object are retained (at least on backup tape), we could allow any broker to create a new version of any object on the repository, leaving it to the broker reading an unauthorized version to detect and ignore that version. In practice, we may not want repositories filled up with unauthorized versions that are not detected until the next read operation—this is a rather severe performance penalty.

If a conventional encryption scheme is used, such as the Federal Data Encryption Standard [NBS 77], the above protocols have the property that anyone who knows the key for an object can both read and write the object. Use of a public-key system [DIFF 76] would allow distinguishing the capability to produce a new version from the capability to read an old version, since different keys are used to encrypt and to decrypt in a public-key system.

To prevent the unauthorized creation of garbage on the repository, the SWALLOW repository authenticates requests to create new versions. The repository maintains a key with each object (called the write authentication key, WAK). The WAK is used to encrypt write requests, so that the repository can authenticate them. A broker that writes an object must know both its OBK and its WAK. This use of a second encryption can be used with a conventional encryption scheme to distiguish the right to read from the right to create new versions. In a public-key system, this second encryption serves only to detect unauthorized writes early—the release of the WAK for an object only makes the repository perform more poorly in space and time.

The broker is free to maintain the keys associated with objects in any way it chooses. In the initial version of the broker, the keys associated with objects will be maintained in a special object called the key file. The key file is simply a mapping from object id to the keys associated with it. The key file will be stored under a single master encryption key as an object on some repository.

Encryption is also used to authenticate requests associated with commit records. The primary purpose of this authentication is to ensure that an atomic action is committed only if all parts of it complete correctly. Our current scheme for this involves keeping a key (the per-atomic-action key or PAK) with each atomic action. The PAK is kept secret by the brokers involved in the atomic action until the atomic action has created all of its versions. Each version is encrypted using the OBK and the PAK in combination (double encryption will always work; but XOR'ing the two keys works for a DES-based system). No computation outside the atomic action knows the PAK, so the versions created by the atomic action are not visible until the PAK is made public. As part of the commit request sent to the commit record, the PAK is supplied. When the commit record informs a version of its final state, the PAK is transmitted and stored with each version. The information returned when a version is read includes the PAK, so the broker, knowing the OBK already, can access the committed version.

## 6. Conclusion

We are implementing SWALLOW to demonstrate that the concepts involved, that is, uniform interface, atomic actions, stability and protection in an environment that consists of a large number of small objects residing at different physical nodes, can be implemented in a practical system. Since the organization of SWALLOW is radically different from traditional storage systems, the only way to understand how well it will perform in practice is to build it, and then use it in constructing some applications. In the first stage, we will implement a prototype system with most of the features of SWALLOW on a set of Altos [LAMS 79B], with at least one repository node, and several brokers/client nodes. Altos were chosen because of the existence of both solid hardware and well-developed support software. However, our laboratory is developing a more powerful personal computer, the Nu machine [WARD 80], and an abstraction-oriented language for distributed processing in a network of autonomous computers is under development [LISK 79]. Thus the full SWALLOW system will be constructed on the Nu machines, using a language explicitely designed for that kind of programming.

## References

CLAR 78 Clark, D.D., Pogran, K.T., Reed, D.P., "An Introduction to Local Area Networks", *Proc. of the IEEE*, Vol 66, No. 11, November 1978, pp. 1497-1517.

DIFF 76 Diffie, W. and Hellman, M., "New directions in cryptography," *IEEE Trans. Inf. Theory IT-22*, 11 (November 1976), pp. 644-654.

| | |
|---|---|
| GRAY 79 | Gray, J. et. al., "The Recovery Manager of a Data Management System," IBM Research Laboratory Technical Report RJ2623, August 1979. |
| ISRA 78 | Israel, J.E., Mitchell, J.G., Sturgis, H.E., "Separating Data from Function in a Distributed File System," *Proc. of Second International Symposium on Operating Systems*, IRIA, October 1978. |
| LAMP 78 | Lamport, L. "Time, clocks, and the ordering of events in a distributed system." *Comm. of the ACM 21*, 7 (July 1979), pp. 558-565. |
| LAMS 79A | Lampson, B.W., Sturgis, H.E., "Crash Recovery in a Distributed Data Storage System", XEROX Palo Alto Research Center, April 1979. To be published in *Comm. of the ACM.* |
| LAMS 79B | Lampson, B.W., Sproull, R.F., "An Open Operating System for a Single-User Machine," *Proc. of Seventh ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, December 1979, pp.98-105. |
| LISK 79 | Liskov, B. "Primitives for Distributed Computing," *Proc. of Seventh ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, December 1979, pp.33-42. |
| NBS 77 | National Bureau of Standards, *Data Encryption Standard*, Federal Information Processing Standards Publication 46, 1977. |
| NEED 78 | Needham, R., Schroeder, M., "Using Encryption for Authentication in Large Networks of Computers," *Comm. of the ACM*, Vol. 21, No. 12, December 1978, pp.993-999. |
| PAXT 79 | Paxton, W.H., "A Client-Based Transaction System to Maintain Data Integrity," *Proc. of Seventh ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, December 1979, pp.18-23. |
| REED 78 | Reed, D.P. "Naming and Synchronization in a Decentralized Computer System." Ph.D. thesis, M.I.T. Department of Electrical Engineering and Computer Science, September 1978. Also available as M.I.T. Laboratory for Computer Science Technical Report TR-205. |
| REED 79 | Reed, D.P. "Implementing Atomic Actions on Decentralized Data." paper presented |

at Seventh ACM Symposium on Operating Systems Principles, Pacifc Grove, CA, December 1979. Submitted to *Comm. of the ACM.*

| | |
|---|---|
| SVOB 79 | Svobodova, L., Liskov, B., Clark, D., "Distributed Computer Systems: Structure and Semantics," M.I.T. Laboratory for Computer Science Technical Report TR-215, March 1979. |
| SWIN 79 | Swinehart, D., McDaniel, G., Boggs, D., "WFS: A Simple Shared File System for a Distributed Environment," *Proc. of Seventh ACM Symposium on Operating Systems Principles,* Pacific Grove, CA, December 1979, pp.9-17. |
| WARD 80 | Ward, S.A., Terman, C.J., "An Approach to Personal Computing," Digest of Papers, COMPCON Spring '80, February 1980, pp.460-465. |

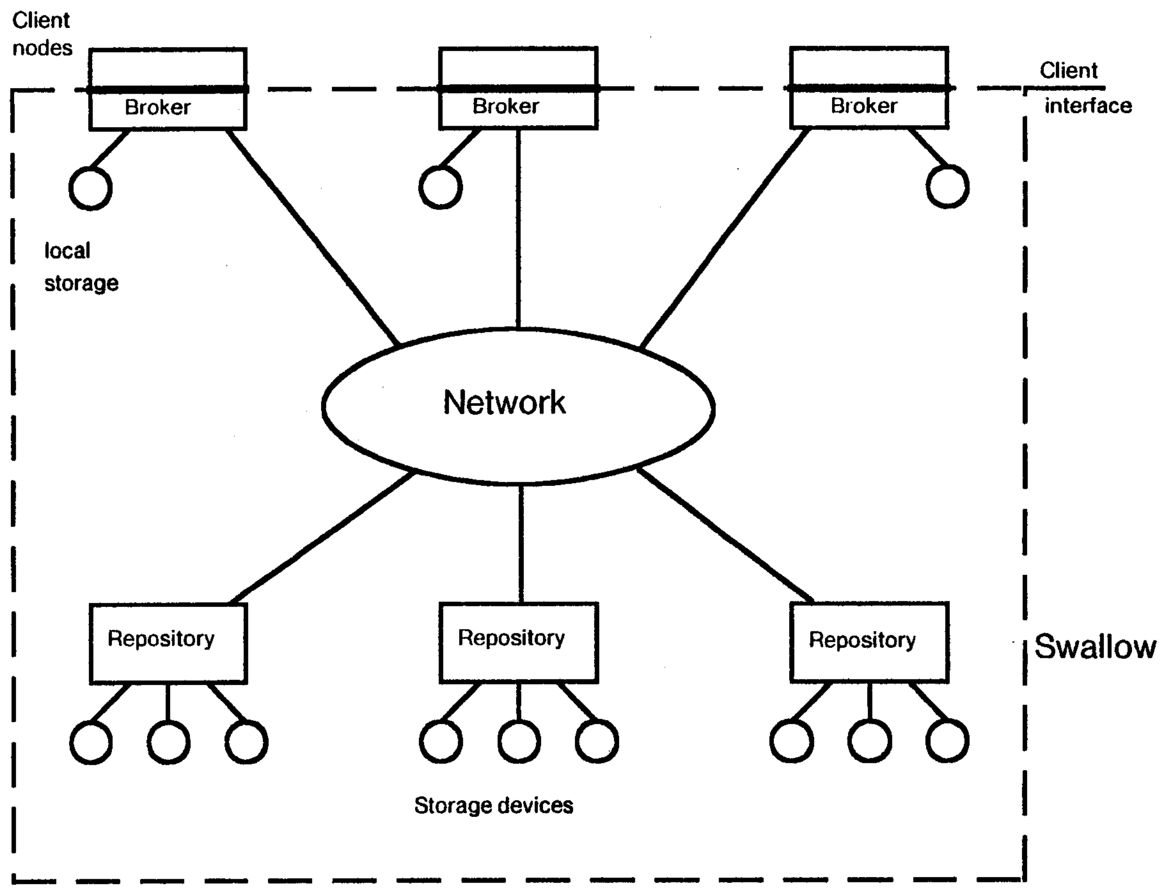

**Figure 1: Structure of Swallow System**

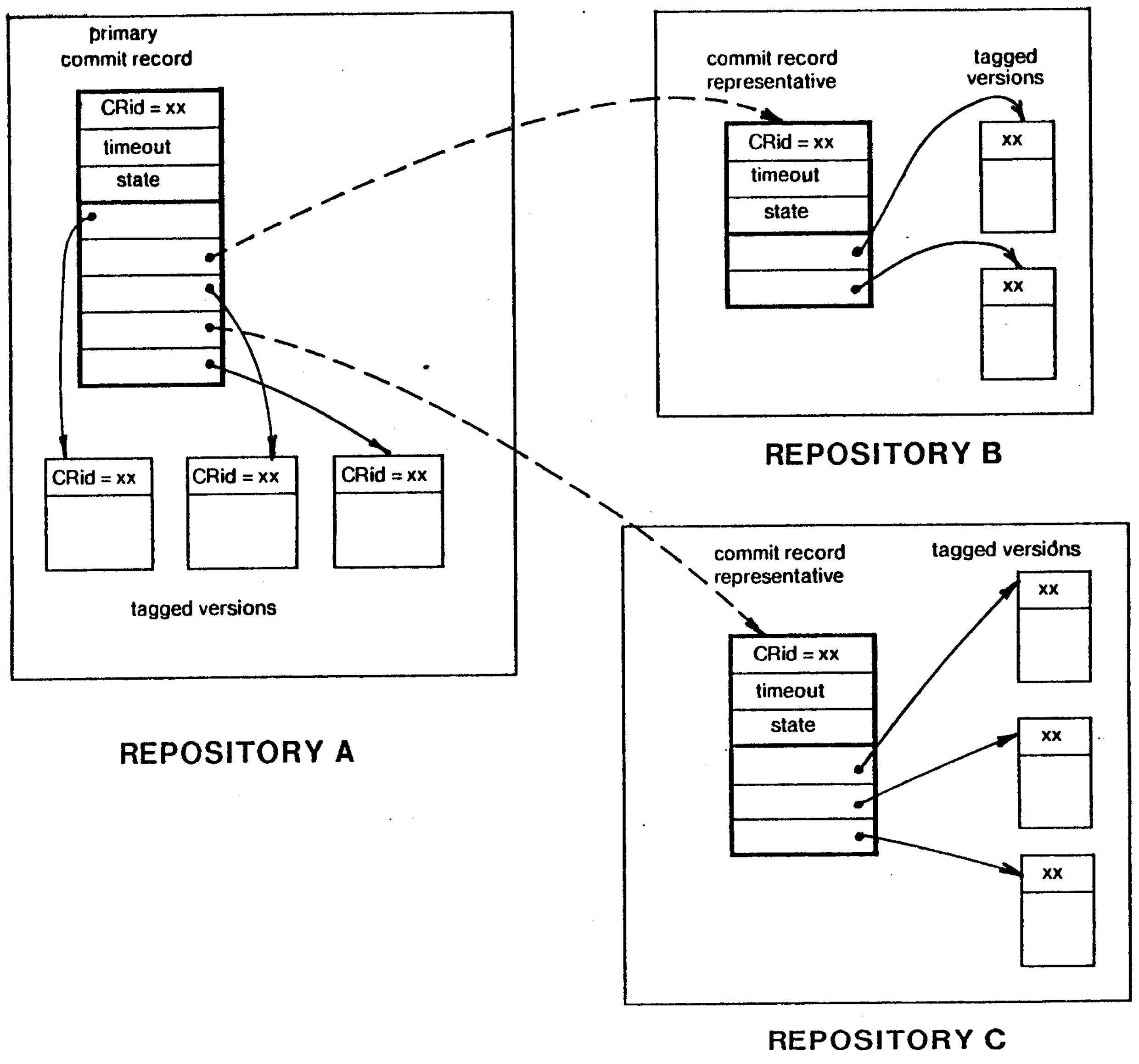

**Figure 2:** Implementation of a multi-site possibility with commit record representatives