Distributed Transactions for Reliable Systems

Alfred Z. Spector, Dean Daniels, Daniel Duchamp,
Jeffrey L. Eppinger, Randy Pausch

Department of Computer Science
Carnegie-Mellon University

Abstract

Facilities that support distributed transactions on user-defined types can be implemented efficiently and can simplify the construction of reliable distributed programs. To demonstrate these points, this paper describes a prototype transaction facility, called TABS, that supports objects, transparent communication, synchronization, recovery, and transaction management. Various objects that use the facilities of TABS are exemplified and the performance of the system is discussed in detail. The paper concludes that the prototype provides useful facilities, and that it would be feasible to build a high performance implementation based on its ideas.

This work was supported by IBM and the Defense Advanced Research Projects Agency, ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and by graduate fellowships from the National Science Foundation and the Office of Naval Research.

1. Introduction

General purpose facilities that support distributed transactions are feasible to implement and useful in simplifying the construction of reliable distributed applications. To justify this assertion, this paper describes the design, implementation, use, and performance of TABS [Spector et al. 85], a prototype facility that supports transactions on user-defined abstract objects. We attempt to generalize from our experiences with the prototype, particularly in the sections on the usage and performance of TABS.

We define a *distributed transaction facility* as a distributed collection of components that supports not only such standard abstractions as processes and inter-process communication, but also the execution of transactions and the implementation of objects on which operations can be performed. Although there is room for diversity in its exact functions, a distributed transaction facility must make it easy to initiate and commit transactions, to call operations on objects from within transactions, and to implement abstract types that have correct synchronization and recovery properties.

Transactions provide three properties that should make them useful in a variety of distributed applications [Lomet 77, Liskov 82, Spector and Schwarz 83]. Synchronization properties, such as serializability, guarantee that concurrent readers and writers of data do not interfere with each other. Failure atomicity simplifies the maintenance of invariants on data by ensuring that updates are not partially done. Permanence provides programmers the luxury of knowing that only catastrophic failures will corrupt or erase previously made updates.

Certainly, these properties of transactions are useful in database applications [Gray 78, Date 83]. Database applications are typically characterized by the need for absolute data integrity, permanent updates, and careful synchronization between processes that access large quantities of shared data. When considering the application of transactions to other domains such as the construction of distributed operating systems and real time systems, there are questions pertaining to what transaction facilities should be provided, how they should

be implemented to achieve adequate performance, and where they should be used. For example, a typical question is whether the recovery and synchronization techniques that are suitable for database systems have sufficient performance and flexibility to support transactions on user-defined shared abstract types in other applications. Quite a few research projects in addition to our own are considering these issues [Liskov and Scheifler 82, Allchin and McKendry 83, Birman et al. 83, Diel et al. 84, Jensen and Pleszkoch 84].

The next section surveys the underlying models and techniques on which this research is based and provides necessary background into the function, implementation, and use of transaction facilities. The reader who is expert in distributed transaction processing may be able to skip most of this section and read only the summary in Section 2.1.4. Following this survey, Section 3 describes the interface and implementation of TABS.

Section 4 shows how the TABS prototype is used to support various abstract data types including arrays, queues, directories, replicated directories and reliable terminal displays. Although these objects do not constitute user-level applications, they represent rather important building blocks. The primary goal of this section is to show how the TABS interface is used and thereby highlight its strengths and weaknesses.

Section 5 describes the performance of the TABS prototype on a variety of benchmarks, both in terms of execution time and in terms of primitive operations. This performance evaluation permits us to predict the effect of changes to the system (e.g., combining certain TABS processes or reduced message passing times) and conclude that high performance general purpose transaction facilities based on the ideas of TABS are feasible. Section 6 contains a brief comparison of TABS with two important related systems, R* [Williams et al. 81] and Argus [Liskov et al. 83]. Section 7 contains the conclusions of this research project and directions for future work.

2. Background

This section surveys the research and development that has influenced this work and identifies many of the algorithms and paradigms that we have used. The discussion is divided into two parts. The first discusses the fundamental issues in *implementing* distributed transactions on abstract objects focusing on the objects themselves, distribution, and transaction processing. The second part discusses the *use* of distributed transactions.

2.1. Distributed Transactions on Abstract Objects

2.1.1. Abstract Objects

Abstract objects are data or input/output devices, having distinct names, on which collections of *operations* have been defined. Access to an object is permitted only by these operations. A queue object having operations such as **Enqueue**, **Dequeue**, **EmptyQueue** is a typical data object, and a CRT display having operations such as **WriteLine**, and **ReadLine** is a typical I/O object. Objects vary in their lifetimes and their implementation. The notion of object presented here is similar to the *class* construct of Simula [Dahl and Hoare 72], *packages* in ADA [Department of Defense 82], and the abstract objects supported by operating systems such as Hydra [Wulf et al. 74]. The operating system work has tended to emphasize authorization — an issue not addressed here.

Many models exist for implementing abstract objects that are shared by multiple processes. In one model, objects are encapsulated in protected subsystems and accessed by protected procedure calls or capability mechanisms [Saltzer 74, Fabry 74]. TABS uses another model, called the *client/server* model, as a basis for implementing abstract objects [Watson 81]. Servers encapsulate one or more data objects. They accept request messages that specify operations and a specific object. To implement operations, they read or modify data they directly control and invoke operations on other servers. After an operation is performed, servers typically send a response message containing the result. Servers that encapsulate data objects are called *Data Servers* in TABS, *Resource Managers* in R* [Lindsay et al. 84], and *Guardians* in Argus [Liskov et al. 83].

Message transmission mechanisms and server organizations differ among implementations based upon the client/server model. In these aspects, TABS is substantially influenced by the Accent operating system kernel on which it was developed [Rashid and Robertson 81]. Accent provides heavyweight processes with 32-bit virtual address spaces and supports messages that are arbitrarily long vectors of typed information, addressed to *ports*. Many processes may have send rights to a port, but only one has receive rights. Send rights and receive rights can be transmitted in messages along with ordinary data. Large quantities of data are efficiently conveyed between processes on the same machine via copy-on-write mapping into the address space of the recipient process. This message model differs from that of Unix 4.2 [Joy et al. 83] and the V Kernel [Cheriton 84a] in that messages are typed sequences of data which can contain port capabilities, and that large messages can be transmitted with nearly constant overhead.

The programming effort associated with packing and unpacking messages is reduced in TABS through the use of a remote procedure call facility called *Matchmaker* [Jones et al. 85]. (We use the term *remote procedure call* to apply to both intra-node and inter-node communication.) Matchmaker's input is a syntactic definition of procedure headers and its outputs are client and server stubs that pack data into messages, unpack data from messages, and dispatch to the appropriate procedures on the server side.

Servers that never wait while processing an operation can be organized as a loop that receives a request message, dispatches to execute the operation, and sends a response message. Unfortunately, servers may wait for many reasons: to synchronize with other operations, to execute a remote operation or system call, or to page-fault. For such servers, there must be multiple threads of control within a server, or else the server will pause or deadlock when it needn't.

One implementation approach for servers is to allocate independently schedulable processes that share access to data. With this approach, a server is a class of related processes — in the Simula sense of the word "class." An alternative approach is to have multiple lightweight processes within a single server process. Page-faults still cause all lightweight processes to be suspended, but a lightweight process switch can occur when a server would otherwise wait. Although this approach does not permit servers to exploit the parallelism of a multiprocessor, it was easy to implement on Accent, and TABS uses it. The topic of server organization has been clearly discussed by Liskov and Herlihy [Liskov and Herlihy 83].

Before leaving the topic of abstract objects, it is necessary to discuss how objects are named. Certainly, a port to a server and a *logical object identifier* that distinguishes between the various objects implemented by that server are sufficient to name an object. The dissemination of these names can be done in many ways. A common method is for servers to register objects with a well known server process on their node, often called a *name server*, and for the name server to return one or more [port, logical object identifier] pairs in response to name lookup requests. Name servers can cooperate with each other to provide transparent naming across a network.

2.1.2. Distribution

Replicated and partitioned distributed objects are feasible to implement in the client/server model. For example, there may be many servers that can respond identically to operations on a replicated object. However, servers must contain the replication or partitioning logic. The TABS project hypothesizes that the availability of transaction support substantially simplifies the maintenance of distributed and replicated objects.

Transparent inter-node message passing can simplify access to remote servers. In the Accent environment, inter-node communication is achieved by interposing a pair of processes, called Communication Managers, between the sender of a message and its intended recipient on a remote node [Rashid and Robertson 81]. The Communication Manager supplies the sender with a local port to use for messages addressed to the remote process. Together with its counterpart at the remote node, the Communication Manager manages the network and implements the mapping between the local port used by the sender and the corresponding remote port belonging to the target process.

There has been considerable research aimed at providing high-performance inter-process communication mechanisms. Local and inter-node message facilities can be optimized with the use of simplified protocols, machine registers, microcode, and careful coding [Nelson 81, Spector 82, Birrell and Nelson 84, Cheriton 84b]. The TABS Project assumes that high performance communication systems can be constructed, but it has not invested the effort to build one for the prototype. However, TABS has been careful to use datagrams for communication during transaction commit; more costly communication based on sessions is used only for the remote procedure calls that implement operations on remote data objects. R* also uses both datagram and session-based communication [Lindsay et al. 84].

2.1.3. Transactions

Although the concept of a transaction has been defined precisely in the literature [Eswaran et al. 76, Gray 80], TABS does not require that objects enforce serializability, failure atomicity, or permanence. Certainly, support exists for the standard notions, but transactions are permitted to interfere with each other and to show the effects of failure — if this is useful. In other words, TABS provides basic facilities for supporting many different type of objects and lets the implementors choose how they want to use them.

Many techniques exist for synchronizing the execution of transactions. Locking, optimistic, timestamp, and many hybrid schemes are frequently discussed; these are surveyed by Bernstein and Goodman [Bernstein and Goodman 81]. TABS has chosen to use locking [Date 83]. To obtain synchronized access to an object, a transaction must first obtain a *lock* on all or part of it. A lock is granted unless another transaction already holds an incompatible lock.

TABS chose to use locking for two reasons. First, locking is an efficient synchronization mechanism that has been used successfully in many commercial data management systems. Second, because servers implement locking locally, they can tailor their locking mechanism to provide better performance. With *type-specific* locking, implementors can obtain increased concurrency by defining type-specific lock modes and lock protocols [Korth 83, Schwarz and Spector 84, Schwarz 84]. Type-specific locking requires use of a specialized compatibility relation to determine whether a lock may be acquired by a particular transaction.

Locking restricts the flow of information between transactions by delaying operations on shared data, even if that delay leads to a deadlock. Some systems implement local and distributed deadlock detectors that identify and break cycles of waiting transactions [Obermarck 82, Lindsay et al. 84]. However, TABS, like many other systems, currently relies on time-outs, which are explicitly set by system users [Tandem 82].

Recovery in TABS is based upon *write-ahead logging*, rather than *shadow paging* [Lorie 77, Gray 78, Lindsay et al. 79, Gray et al. 81, Lampson 81, Haerder and Reuter 83, Schwarz 84]. To discuss write-ahead logging, it is first necessary to discuss the three-tiered storage model on which it depends. Storage consists of volatile storage — where portions of objects reside when they are being accessed, non-volatile storage — where objects reside when they have not been accessed recently, and stable storage — memory that is assumed to retain information despite failures. The contents of volatile storage are lost after a system crash, and the contents of non-volatile storage are lost with lower frequency, but always in a detectable way.

In recovery techniques based upon logging, stable storage contains an append-only sequence of records. Many of these records contain an undo component that permits the effects of aborted transactions to be undone, and a redo component, that permits the effects of committed transactions to be redone. Updates to data objects are made by modifying a representation of the object residing in volatile storage and by spooling one or more records to the log. Logging is called "write-ahead" because log records must be safely stored (forced) to stable storage before transactions commit, and before the volatile representation of an object is copied to non-volatile storage. Because of this strategy, there are log records in stable storage for all the changes that have been made to non-volatile storage, and for all committed transactions. Thus, the log can be used to recover from aborted transactions, system crashes and non-volatile storage failures.

The advantages of write-ahead logging over other schemes have been discussed elsewhere and include the potential for increased concurrency, reduced I/O activity at transaction commit time, and contiguous allocation of objects on secondary storage [Gray et al. 81, Traiger 82, Reuter 84]. All objects in TABS use one of two co-existing write-ahead logging techniques and share a common log.

The simpler technique is *value* logging, in which the undo and redo portions of a log record contain the old and new values of at most one page of an object's representation. During recovery processing, objects are reset to their most recently committed values during a one pass scan that begins at the last log record written and proceeds backward. If this value logging algorithm is used, only one transaction at a time may modify any individually logged component of an object that is to be failure atomic and permanent.

The other technique is called *operation* (or *transition*) logging. With it, data servers write log records containing the names of operations and enough information to invoke them. Operations are redone or undone, as necessary, during recovery processing to restore the correct state of objects. An important feature of this algorithm is that operations on multi-page objects can be recorded in one log record. The operation-based recovery algorithm also permits a greater degree of concurrency than the value based recovery algorithm and may require less log space. However, it is more complex, and it requires three passes over the log during crash recovery, instead of the single pass needed for the value-based algorithm. The TABS recovery algorithms are similar to other previously published write-ahead log-based algorithms [Gray 78, Lindsay et al. 79], and are fully described by Schwarz [Schwarz 84].

Both value and operation logging algorithms require that periodic system *checkpoints* be taken. Checkpoints serve to reduce the amount of log data that must be available for crash recovery and shorten the time to recover after a crash [Haerder and Reuter 83]. At checkpoint time, a list of the pages currently in volatile storage and the status of currently active transactions are written to the log. Some systems also force certain pages to non-volatile storage and abort transactions that have been running for a long time. To reduce the cost of recovering from disk failures, systems infrequently *dump* the contents of non-volatile storage into an off-line archive.

Recently, researchers have begun to discuss high performance recovery implementations that integrate virtual memory management with the recovery subsystem and use higher performance stable storage devices [Traiger 82, Banatre et al. 83, Stonebraker 84, Diel et al. 84]. Section 3 discusses how virtual memory management and recovery are integrated in TABS.

The most important component of a transaction facility not yet discussed is the one that commits and aborts transactions. Commit algorithms vary in their efficiency and robustness [Lindsay et al. 79, Dwork and Skeen 83]. TABS uses a tree-structured variant of the 2-phase commit protocol, in which each node serves as coordinator for the nodes that are its children. Though 2-phase commit is simple and efficient, it does have failure modes in which nodes participating in a distributed transaction must restrict access to some data until other nodes recover from a crash. TABS could use one of the other commit algorithms that do not have this deficiency.

As a final point in the implementation of transactions, the increased interest in building nested abstractions using transactions has led to the investigation and implementation of facilities for supporting nesting [Reed 78, Moss 81, Liskov et al. 83]. These facilities limit the concurrency anomalies that can occur within a single transaction that has simultaneous threads of control, and they permit portions of a transaction to abort independently.

TABS has a limited subtransaction facility, which was very easy to implement. It can be characterized by its synchronization and commit policies. With respect to synchronization, a subtransaction behaves as a completely separate transaction. This provides protection between simultaneous threads of control, but may cause intra-transaction deadlock if two subtransactions update the same data. With respect to commit, a subtransaction

is not committed until its top-level parent transaction commits, but a subtransaction can abort without causing its parent transaction to abort. Subtransactions that can abort independently permit their parent to tolerate the failure of some operations.

2.1.4. Summary of Implementation Issues

The major points of this development can be tersely summarized: TABS supports transactions on abstract objects. Objects are implemented within server processes, and operations on objects are invoked via messages with a remote procedure call facility to reduce the programming effort of packing, unpacking, and dispatching. Inter-node communication uses both sessions and datagrams. Inter-transaction synchronization is done via locking, with time-outs used to resolve deadlock. Write-ahead logging is the basis of recovery and transaction commit is done via the tree structured two-phase commit protocol. A limited subtransaction model is implemented.

2.2. Use of Transactions

Currently, transactions are primarily used to support the hierarchical, relational, and networked abstract types used in database systems. Date surveys these abstract types and describes some aspects of their implementation [Date 83]. The literature contains many descriptions of more general types, and there are some implementations of these. For example, Lomet, Weihl and Liskov, and Schwarz and Spector have written about buffer, directory, queue, and mailbox types [Lomet 77, Weihl and Liskov 83, Schwarz and Spector 84], and there have been a few experimental transactional file systems, e.g., one described by Paxton [Paxton 79].

The properties provided by these transactional types simplify abstractions that are built on them. For example, the invariants needed for the replicated objects described by Gifford, Bloch et al., and Herlihy [Gifford 79, Bloch et al. 84, Herlihy 84] are easier to maintain. The availability of distributed transactions make it easier to generate R*'s query execution plans [Daniels 82]. The integrity guarantees of a mail system, such as one sketched by Liskov, are also simplified. More collections of abstract types, combined into larger and more diverse applications, will undoubtedly be developed as general purpose transactions facilities become more prevalent. (See Section 4 for a discussion of abstract types that we have built.)

3. An Experimental Design - The TABS Prototype

The TABS Prototype is implemented in Pascal on a collection of networked Perq workstations [Perq Systems Corporation 84] running a modified version of the Accent operating system. At each node, there is one instance of the TABS facilities and one or more user-programmed data servers and/or applications. Data servers are programmed with the aid of system supplied libraries for doing synchronization and recovery, and for performing a data server's role during two-phase commit. Applications initiate transactions and call data servers to perform operations on objects. The library interfaces to TABS are described in detail in Section 3.1.

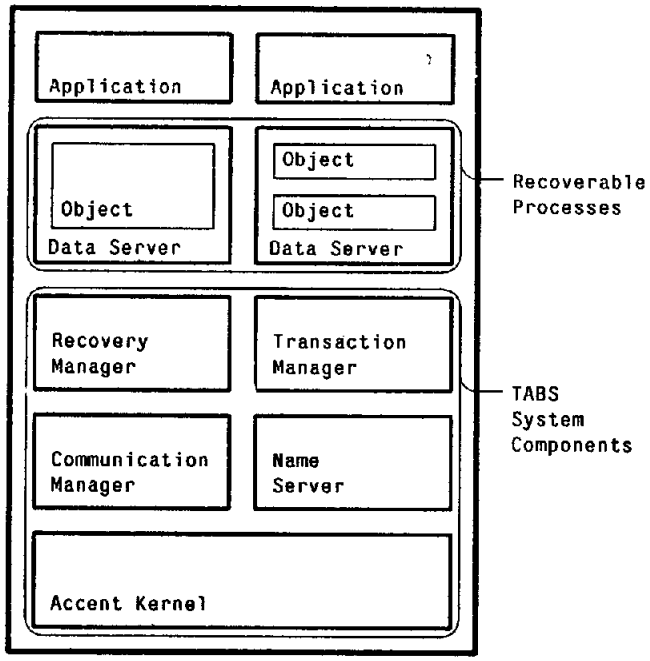

Figure 3-1: The Basic Components of a TABS Node

The TABS facilities are made up of four processes that run on Accent (see Figure 3-1). The processes, called Name Server, Communication Manager, Recovery Manager, and Transaction Manager, perform name dissemination, network communication, recovery and log management, and transaction management, respectively. Section 3.2 briefly describes the implementation of these TABS processes and our modifications to Accent.

TABS began to operate in the Fall of 1984, and all the facilities described in this paper are operational with one exception. Operation-based recovery and the necessary type-specific locking is not supported in the TABS libraries, though the operation-based algorithm has been tested and integrated with the value-based algorithm. The system contains about 51,000 lines of Matchmaker, Pasmac macro language [Lansky 80], and Pascal sources. This count includes one data server and application that we use in testing releases, but it does not include the changes we have made to the Accent kernel.

3.1. The Tabs Programming Interface

The interface to TABS is provided by three libraries. The server library, used only by data servers, supports shared/exclusive

Routine	Purpose
InitServer(ServerID)	Startup
ReadPermanentData(DiskAddress) returns (VirtualAddress, DataSize)	Startup
RecoverServer	Startup
AcceptRequests(DispatchFunction)	Startup
CreateObjectID(VirtualAddress, length) returns (ObjectID)	Address Arithmetic
ConvertObjectIDtoVirtualAddress(ObjectID) returns (VirtualAddress)	Address Arithmetic
LockObject(ObjectID, LockMode)	Locking
ConditionallyLockObject(ObjectID, LockMode) returns (Boolean)	Locking
IsObjectLocked(ObjectID) returns (Boolean)	Locking
PinObject(ObjectID)	Paging Control
UnPinObject(ObjectID)	Paging Control
UnPinAllObjects	Paging Control
PinAndBuffer(ObjectID)	Paging Control, Logging
LogAndUnPin(ObjectID)	Logging, Paging Control
LockAndMark(ObjectID, LockMode)	Locking
PinAndBufferMarkedObjects	Paging Control, Logging
LogAndUnPinMarkedObjects	Paging Control, Logging
ExecuteTransaction(TransactionProcedure)	Transaction Management

Table 3-1: The Complete TABS Server Library

This table summarizes the library routines used by data servers. The routine names have been made as explicit as possible; a description of their function may be found in the accompanying prose. The types of the parameters and return values are shown as well as the general purpose of each routine. Routines used both by data servers and applications are shown in Tables 3-2 and 3-3.

locking, value logging, and miscellaneous utilities. The transaction management library provides routines for controlling the execution of transactions. The name server library provides access to TABS name dissemination services. The use of many routines from these libraries is illustrated in Section 4.

3.1.1. The Server Library

The functions that make up the server library fall into six broad categories. These categories are listed beside the procedure headers of the library routines in Table 3-1.

Four procedures are used to initialize the data server. **InitServer** initializes server library data structures, and **ReadPermanentData** maps the data server's recoverable data into virtual memory. (See Section 3.2.1.) **RecoverServer** accepts the log records that the Recovery Manager reads from the log. This procedure understands the format of the log records written by the server library routines during forward processing, and calls the server library's undo/redo code to restore the data to a transaction-consistent state. Once the virtual memory copy of the recoverable data is consistent, the data server calls **AcceptRequests**. This routine takes a procedure argument that dispatches on operation request messages.

Since a programmer works with virtual addresses but the log manager works with disk addresses contained in ObjectIDs, data servers must do address translation. The routines **CreateObjectID** and **ConvertObjectIDtoVirtualAddress** perform these conversions.

Three routines support locking. **LockObject** attempts to acquire a lock, and waits if the lock is not available. **ConditionallyLockObject** also attempts to acquire a lock, but it returns immediately if the lock is unavailable. **IsObjectLocked** returns true if and only if a lock is set. All unlocking is done automatically by the server library at commit or abort time.

Paging control operations prevent the kernel from paging an object to secondary storage. They are used to ensure that an object's permanent representation is not changed before all modifications to it have been logged. **PinObject** prevents the kernel from paging an object to secondary storage until **UnPinObject** or **UnPinAllObjects** is called.

The paging control operations are usually performed as side effects of logging routines. **PinAndBuffer** pins the specified object and then copies the existing (old) value of the object into a buffer in anticipation of a modification. After the modification is made, **LogAndUnPin** sends the (buffered) old value and the existing (new) value to the Recovery Manager and unpins the object.

The checkpoint protocol requires that data servers not wait (e.g., for a lock) while objects are pinned. One approach to meeting this requirement is to set all locks before any modifications are performed. The server library facilitates this by providing three routines: **LockAndMark** locks the specified object and enqueues a reference to the object on a "to be modified" queue. **PinAndBufferMarkedObjects** pins every object on the queue and copies each object's current (old) value into buffers. **LogAndUnPinMarkedObjects** sends to the Recovery Manager the (buffered) old value and existing (new) value for each object on the queue When all the old and new values are logged, **LogAndUnPinMarkedObjects** unpins all the objects and deletes the queue.

The remaining routine, **ExecuteTransaction**, takes a procedure argument and executes that procedure within a new top-level transaction.

Lightweight processes use a coroutine mechanism embedded within every data server. The server library treats each incoming request as a separate coroutine invocation. A coroutine switch is performed only when an operation waits, e.g., for a lock or for starting a transaction. The server library contains additional code that automates a data server's participation in transaction commit, abort, and checkpoint.

3.1.2. The Transaction Management Library

The routines in the transaction management library provide a standard interface to transaction management functions (see Table 3-2). **BeginTransaction** creates a *subtransaction* of the specified transaction. To create a new top-level transaction, a special null **TransactionID** is given as the argument. **EndTransaction** and **AbortTransaction** initiate commit and abort of the specified transaction, respectively. The **TransactionIsAborted** exception is raised in the application process if the specified transaction has been aborted by some other process.

Routine
BeginTransaction(TransactionID) returns(NewTransactionID)
EndTransaction(TransactionID) returns(Boolean)
AbortTransaction(TransactionID)
TransactionIsAborted(TransactionID) [*exception*]

Table 3-2: The TABS Transaction Management Library

3.1.3. The Name Server Library

The abstractions represented by data servers are permanent entities that must persist despite node failures, even though the ports through which they are accessed change. The TABS Name Server implements an interface that allows a single name to be mapped to one or more **<port, LogicalObjectIdentifier>** pairs. A data server has the option of servicing operation requests for several objects on the same port, and independent data server processes can together implement replicated objects. The most important routines in the Name Server library are summarized in Table 3-3.

Routine
Register(Name, Type, Port, ObjectID)
DeRegister(Name, Port, ObjectID)
LookUp(Name, NodeName, DesiredNumberOfPortIDs, MaxWait) returns(ArrayOfPortIDPairs, ReturnNumberOfPortIDs)

Table 3-3: The TABS Name Server Library

3.2. Implementation of TABS

Most of the operations TABS libraries provide to data servers and applications are implemented by the TABS System components and the Accent kernel. The modifications to the kernel and the TABS system components are summarized in this section and described in more detail in a recent paper [Spector et al. 85].

3.2.1. The Accent Kernel

The failure atomic and/or permanent data stored by data servers are stored in disk files that are mapped into virtual memory. These files are called *recoverable segments*. When mapped into memory, the kernel's paging system updates a recoverable segment directly instead of updating paging storage [Eppinger and Spector 85].

To support the write-ahead log algorithms used by TABS, the kernel sends three types of messages to the Recovery Manager. The first message indicates that a page frame that is backed by a recoverable segment has been modified for the first time. The second message indicates that the kernel wants to copy a modified page back to its recoverable segment. The kernel does not write the page until it receives a message from the Recovery Manager indicating that all log records that apply to this page

have been written to non-volatile storage. The third and final message indicates whether the contents of a page frame have been successfully copied to a recoverable segment.

In addition to the special messages that support the write-ahead log algorithms, the Accent Kernel also implements the paging control primitives of the server library.

A final modification to Accent has been made to support the TABS operation logging recovery algorithm. This algorithm requires that the kernel atomically write a sequence number each time it copies a page of a recoverable segment to non-volatile storage. This sequence number (currently, 39 bits) is stored in header space that is available on a Perq disk sector. The Recovery Manager sends the sequence number to the kernel in the message that indicates that the page can be written to disk. During crash recovery, the Recovery Manager sends a request to the kernel when it wishes to read a page's sequence number.

3.2.2. Recovery Manager

The Recovery Manager coordinates access to the log. The log should be on *stable storage*; but, because of our Perq hardware restrictions (only one disk), the non-volatile storage used for the log is not stable. Hence, we do not consider disk failures in this work.

The Recovery Manager writes log records in response to messages sent by data servers, the Transaction Manager, and the Accent kernel. Log records written in response to kernel messages help to identify (at recovery time) the pages that were in memory at crash time. All log records are written into a volatile buffer until the buffer fills or until the buffer is forced to non-volatile storage by either the write-ahead-log or commit protocols. Upon transaction abort, the recovery manager follows the backward chain of log records that were written by the transaction and sends messages to the servers instructing them to undo their effects.

After a node crash, the Recovery Manager scans the log one or more times. It directly interprets the recovery log records, but it must pass transaction management records back to the Transaction Manager. The Recovery Manager then queries the Transaction Manager to discover the state of the transaction. Based on this information, the Recovery Manager gives each data server instructions to redo or undo previously performed operations. In this way the Recovery Manager assures that objects in recoverable segments reflect only the operations of committed and prepared transactions.

The last function of the Recovery Manager is to coordinate checkpoints. After a crash, the Recovery Manager must read the portion of the log written after the last checkpoint. Depending on the contents of the checkpoint record, earlier sections of the log may also be read, but the most recent checkpoint record contains enough information to determine when crash recovery will be complete. In our system, checkpoints are performed at intervals determined by the transaction manager or when the system is close to running out of log space. In the latter instance, the Recovery Manager runs a reclamation algorithm that attempts to reclaim log space. Log reclamation may force pages back to disk before they would otherwise be written.

3.2.3. Transaction Manager

The Transaction Manager's major responsibilities are implementing commit protocols and allocating globally unique transaction identifiers. Application processes and data servers send the Transaction Manager messages to begin a transaction, to attempt to commit a transaction, or to force a transaction to be aborted. The tree-structured two-phase commit protocol used by Transaction Manager is based on a spanning tree where a node A is a parent of another node B if and only if A were the first node to invoke an operation on behalf of the transaction on B. The information about a node's relation to the nodes directly above and below it in the spanning tree is kept by its Communication Manager.

There are two messages that processes send to inform the Transaction Manager of the progress of a transaction. The first is sent by a data server the first time it is asked to perform an operation on behalf of a particular transaction; doing so enables the Transaction Manager to know which servers it must inform when the transaction is being terminated. The other message is sent by the Communication Manager the first time an inter-node message is sent or received on behalf of a particular transaction. This message indicates that there are remote sites that have servers active on behalf of the given transaction. At this point, the Transaction Manager becomes aware that remote sites are involved in the transaction, but it cannot yet identify these sites. The complete site list is obtained from the Communication Manager during commit processing.

The existence of subtransactions in the TABS model does not complicate transaction management. The same messages that are used to inform the Transaction Manager about top-level transactions are used for subtransactions. The only regard in which transaction processing differs is that subtransactions can be aborted without requiring the parent transaction to abort. Subtransactions, however, may not be committed before their parents. When a parent transaction commits or aborts, its subtransactions are committed or aborted as well.

3.2.4. Communication Manager

The Communication Manager is the only process that has access to the network. It implements three forms of network communication: datagrams for the distributed two-phase commit; reliable *session* communication for implementing remote procedure calls; and broadcasting for name lookup by the Name Server.

For session communication, two Communication Managers cooperate to provide at-most-once, ordered delivery of arbitrary-sized messages. The Communication Manager detects permanent communication failures and, thereby, aids in the detection of remote node crashes. The Communication Manager also scans any transaction identifiers included in messages and is responsible for constructing the local portion of the spanning tree that the Transaction Manager uses during two-phase commit. In particular, the Communication Manager records the node's parent, whether the transaction was initiated by a remote node, and the list of all the node's children. It also records a small amount of additional information that is used for detecting some types of node crashes.

3.2.5. Name Server

In TABS, the Name Server process on each node maintains a mapping of object names to one or more <**port, logical-object-identifier**> pairs for all the objects managed by data servers on that node. Whenever the Name Server is asked about a name it does not recognize, it broadcasts a name lookup request to all other Name Servers. If the broadcast is successful, the Communication Managers on the local and the remote machine automatically establish a session between the requesting node and the data server implementing the named object.

4. The TABS Prototype In Use

This section presents five of the data servers we have implemented with the TABS prototype: The integer array server, the weak queue server, the IO server, the B-tree server, and the replicated directory object. The integer array server, B-tree server, and replicated directory object all preserve the serializability, failure atomicity, and permanence of the transactions that invoke them. The IO server provides a permanent, non-failure atomic object, and the weak queue server provides a permanent, failure atomic object that is not serializable.

4.1. The Integer Array Server

The integer array server maintains an array of (one word) integers, and provides the following abstract operations:

```
FUNCTION  GetCell(cellNum: integer):
                  integer;

PROCEDURE SetCell(cellNum: integer;
                  value:   integer);
```

The two operations supported by the integer array server are simple enough that the best description is the Pascal code that implements one of them. Note that the virtual address of a cell is obtained by adding the proper offset to the base of the recoverable segment.

```
FUNCTION  SetCell(arrayPort:port;   { for RPC }
                  transaction:tid;
                  cellNum:integer;
                  value:integer): GeneralReturn;

{ SetCell sets array[cellNum] to contain 'value' }

VAR
   obj: ObjectID; { object for the cell }
   size: integer; { the size of a cell }

BEGIN

IF (cellNum >= 1) AND (cellNum <= maxCell) THEN
        BEGIN
        size := WordSize(integer);
        obj := CreateObjectID(baseOfArray +
                    (cellNum-1) * size, size );
        LockObject(obj, Write);
        PinAndBuffer(obj);
        obj.ptr^ := value; { do the assignment }
        LogAndUnPin(obj);
        SetCell := Success;
        END
ELSE SetCell := IndexOutOfRange;

END;
```

The implementation of **GetCell** is very similar, and the combined code for both operations requires 50 lines of Pascal. The balance of the 140 lines of code in the integer array server perform module imports and initialization. The integer array server is a very straightforward data server; it uses only the two-phase locking, value logging techniques found in many transaction-based systems. The data servers described below take more advantage of the flexibility of TABS.

4.2. The Weak Queue Server

The weak queue server provides access to a weak queue, sometimes called a semi-queue [Weihl and Liskov 83, Schwarz and Spector 84]. In a weak queue, items in the queue are not guaranteed to be dequeued strictly in the order that they were enqueued. Relaxing the strict FIFO nature of the queue allows greater concurrency while retaining failure atomicity. The weak queue server provides the following abstract operations:

```
PROCEDURE Enqueue(data: integer);
FUNCTION  Dequeue: integer;
FUNCTION  IsQueueEmpty: boolean;
```

The queue is implemented as an array of individually lockable elements, with head and tail pointers bounding the currently used section of the array. Because gaps may exist in the range between the head and tail pointers, each element in the array contains both its contents and an extra boolean, **InUse**, indicating whether that element actually contains a value that is currently stored in the queue. **Enqueue** and **Dequeue** set and clear this **InUse** bit, and if they abort, this bit is restored along with the previous contents of the element. The head pointer is a permanent, failure atomic object. The tail pointer can be recomputed after crashes by examining the head pointer and **InUse** bits, so it is kept in volatile storage.

To add a new item to the queue, **Enqueue** places the item in the element below the tail pointer, sets that element's **InUse** bit to true, and sets the tail pointer to the new element. If the **Enqueue** later aborts, this will leave a gap in the array when the **InUse** bit is reset to false. Because the tail pointer is not locked, the weak queue server relies on the monitor semantics of TABS coroutines to ensure that only a single transaction at a time can update the tail pointer.

Dequeue is more complex, because elements in the array may not be legally dequeued for either of two reasons: If an element is locked, another operation is still manipulating it; If an element's **InUse** bit is False, the **Enqueue** of that element aborted, or the element has already been successfully removed. **Dequeue** scans elements starting at the head pointer, using the **IsObjectLocked** primitive, and then testing the **InUse** bit. When an unlocked element whose **InUse** bit is True is found, **Dequeue** locks it and returns its contents.

Enqueue and **Dequeue** both read the head pointer to check for a full queue. **Dequeue** does not alter the head pointer because this would restrict concurrency. The head pointer must eventually be moved, however, or the queue will fill. Abstractly, one imagines a "garbage collection" operation that gets randomly invoked and moves the head pointer past any elements that are not locked, and whose **InUse** bits are False. The current implementation does the garbage collection as a side effect of **Enqueue**.

The weak queue server is 380 lines of Pascal code. Its design prompted the addition of the **ConditionallyLockObject** and **IsObjectLocked** primitives to the server library. Much of the work that went into creating the weak queue server dealt with mapping the logical operations on the queue into manipulations of data with the value logging mechanisms. For this reason, we believe that certain abstract data types are more suited to operation logging than value logging.

4.3. The Input/Output Server

The IO server extends the domain of TABS to include the bitmap display by restoring the screen contents after a failure, and by giving the user a comfortable model of transaction-based input/output. The current implementation uses character input/output in a standard typescript fashion. Recovering the screen is straightforward; TABS runs within a window manager that provides overlapping, rectangular windows. Restoring the screen requires keeping track of a window's contents and location in a recoverable segment.[1]

Providing a good user model of transaction-based IO is more complex. Writing to a terminal is often cited as the canonical non-recoverable action. An obvious approach is to buffer all output and only display it if the transaction commits, but this technique fails for conversational transactions. The IO server displays all output as it occurs, in a style that indicates the current state of the transaction that performed the output. While a transaction is in progress, the output is displayed in gray, to indicate its tentative nature. If the transaction commits, the output is redrawn in black, to indicate that the operation really occurred. If the transaction aborts, lines are drawn through the output. This is preferable to making the output disappear, which is disconcerting to the user. Users know that an operation has not really happened until its output is displayed in black.

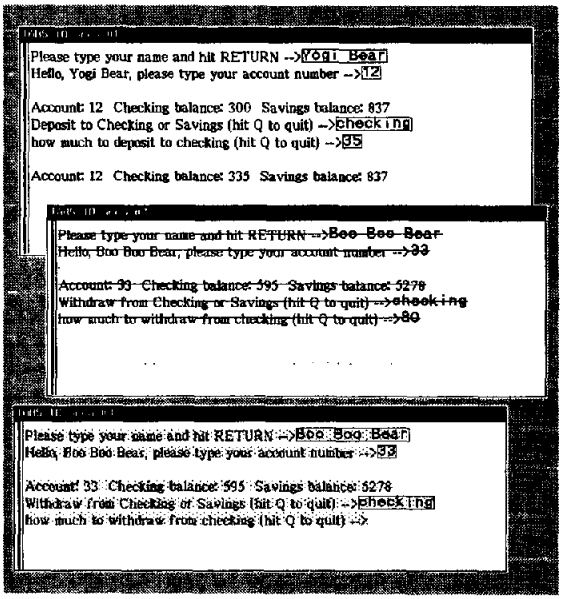

Figure 4-1: Sample Display Screen

This is an actual snapshot of the current IO server running a trivial bank implementation. This example exhibits the IO server; the bank application also uses the integer array server to store its information.

In area one, the user successfully deposited 35 dollars to a checking account. The user knew that the action had occurred (committed), because its output was displayed in black. In area two, the user attempted to withdraw 80 dollars from a checking account, but the node failed during the transaction, causing it to abort. The IO server restored the screen when the system became available, and the user is currently trying again in area three, where the transaction is still in progress. The rectangles drawn around user input indicate that the characters have been read by the application.

Multiple input/output areas are maintained on the screen, to allow for concurrent interaction with the user. The abstract operations are:

```
FUNCTION  ObtainIOarea:     ioAreaID;
PROCEDURE DestroyIOarea    (ioArea: ioAreaID);
PROCEDURE WriteToArea      (ioArea: ioAreaID;
                            data:   String);
PROCEDURE WriteInToArea    (ioArea: ioAreaID;
                            data:String);
FUNCTION  ReadCharFromArea(ioArea: ioAreaID):
                            Char;
FUNCTION  ReadLineFromArea(ioArea: ioAreaID):
                            String;
```

[1] The easiest way to test whether windows are restored to the correct location is to mark display screens with grease pencils. This leads to research on chemicals that remove grease pencil markings from display screens ...

To display output even after a client transaction later aborts, the IO server maintains permanent, non-failure atomic data in an array of characters for each area. Rather than having the client transaction modify this array, the IO server uses **ExecuteTransaction** to invoke a new top-level transaction to write the data for each operation. If the client transaction aborts, the characters stored via the **ExecuteTransaction** will not be altered.

In order to display the output of a transaction, the IO server needs to determine the status (aborted, committed, or in progress) of the transaction. The Transaction Manager cannot provide this facility, because doing so would require retaining an infinite amount of log data. When a transaction establishes ownership of an area, the IO server uses **ExecuteTransaction** to write *aborted* into a state object in the data structure for the area. The IO server then has the client transaction lock the state object and set it to contain *committed*. This causes an old value/new value pair of *aborted/committed* to be written in the log for the client transaction. The IO server can now determine the transaction's current state by using the **IsObjectLocked** primitive. If the state object is locked, the client transaction is still in progress. If the object is no longer locked, then the transaction has finished. If the state object contains *aborted*, the transaction aborted, and the object was reset by the recovery mechanisms. Otherwise, the object contains *committed*, and the IO server knows that the transaction must have committed.

The implementation is 2500 lines of Pascal code, and, like the weak queue server, uses the ability to test if an object is currently locked. The IO server also provides an example of a data server that needs to invoke transactions of its own in order to process requests. The IO server is interesting because it extends the domain of the transaction model.

4.4. The B-Tree Server

The B-tree server maintains arbitrary collections of directory entries in B-trees, and is being used in an implementation of replicated directories. The B-tree server provides the standard operations on multi-key directories: add, delete, modify, etc. Indices on non-primary keys are implemented as separate B-trees, each of which points to the primary key B-tree's leaves which contain the data.

Because the B-tree server dynamically allocates storage within the recoverable segment, it was necessary to create a recoverable storage allocator. If a transaction uses an operation that allocates storage, and the transaction later aborts, the memory is made available for re-use. The B-tree server maintains a separate storage pool for each size object that it allocates, and allocates blocks from the pool using techniques similar to the weak queue server. This technique works for fixed sized blocks, but cannot be used for variable size block allocation, which will be implemented in future data servers using operation-based logging.

The B-tree server was originally implemented as a Pascal program running outside the TABS environment. By using the **LockAndMark**, **PinAndBufferMarkedObjects**, and **LogAndUnPinMarkedObjects** primitives, we were able to use most of the existing code intact. These routines allowed us to avoid bracketing every assignment in the original program with **PinAndBuffer** and **LogAndUnPin** calls, in order to avoid having data pinned when requesting other locks. The total modifications, including initialization and storage allocation changes, increased the size of the B-tree server from 4500 to 5000 lines of Pascal code.

4.5. A Replicated Directory Object

The replicated directory object provides an abstraction identical to a conventional directory but stores its data in multiple *directory representative servers* on different nodes. The replicated directory uses our variation of Gifford's weighted voting algorithm for global coordination [Gifford 79, Daniels and Spector 83, Bloch et al. 84]. Each of the directory representative servers uses a B-tree server to actually store the data, and requires another 2700 lines of code to perform localized functions for the voting algorithm. The interface to client programs is provided by a module that does global coordination of the voting, and is implemented as 1100 lines of code that are linked in with the client program.

The replicated directory object demonstrates many of the facilities of the TABS prototype: Aborting transactions that use the replicated directory requires recovery on multiple nodes, and committing transactions requires the global coordination protocols for multiple node commit. Our tests so far involve 3 nodes, which permits one node to fail and have the data remain available.

4.6. Evaluation of Data Servers

The data servers that have been created cover a good range of the design space, although we have currently restricted our data servers to use standard read/write locking and value logging. Most of the advantages of the system are easy to overlook because they involve simply having a system in the first place. For example, recovery, synchronization, and communication mechanisms exist as tools that are relatively easy to use. Moreover, these tools are not mechanically imposed, which has made it possible to to add new primitives easily, and to build several data servers that use these tools in novel ways.

These flexible tools underscore our major claim: Many interesting data servers are difficult, if not impossible, to build using traditional read/write locking. In support of this claim, we note that all the data servers except the integer array server required the addition of primitives to circumvent the locking mechanism, and that even with these additions, the implementors were required to use unnecessarily complex algorithms

and/or unprotected reads of data. We intend to explore the type-specific locking capability of TABS with future data servers.

Our second claim is: Value logging is inconvenient for non-array implementations. The implementors of the weak queue server, the IO server, and the B-tree server storage allocator initially sketched simple designs that used operation-based logging. The eventual implementations were complicated by the use of value logging. The use of operation-logging, type-specific locking, and value logging where appropriate will provide a rich environment for re-implementing existing data servers, and creating new ones.

5. Analysis of TABS Performance

This section presents analytical and experimental evidence supporting our hypothesis that it is possible to implement efficient general purpose facilities that support distributed transactions. This evaluation permits us to describe the performance and limitations of the current implementation, and it permits us to predict how well TABS would work if it used more efficient underlying primitives and were more tightly integrated. The analysis presented in this paper is an application of a performance evaluation methodology for predicting the cost of transaction execution (latency and resource utilization) under conditions of no load [Spector and Daniels 85][2].

The section continues by describing a collection of benchmarks and characterizing them in terms of the repeated execution of primitive operations. We use both this characterization and an empirical performance study to describe the performance of TABS and to predict how it would perform if improvements were made.

5.1. A Microscopic Approach to Transaction System Performance Evaluation

The performance of a commercial transaction processing system can be described macroscopically by its performance on standard work loads [Anonymous et al. 85]. This approach is not sufficient for our evaluation of TABS for two reasons. First, the work loads encountered by a general purpose facility supporting abstract types are not easily characterizable. Second, throughput rates or latencies, by themselves, do not lead to an understanding of how individual algorithms and implementation decisions have affected system performance. Hence, we need to understand more microscopic effects to critique our system and to predict the effects of algorithmic or structural changes.

To describe TABS performance, we have chosen to measure the performance of a collection of benchmarks from which it is possible to deduce the performance of other transactions. To illustrate this deduction process, consider two benchmarks: one is a read-only transaction that performs one remote read operation on data in primary memory, and the other is a similar transaction that performs five remote read operations on data in primary memory. From these two transactions, it is possible to deduce the amount of time to perform an incremental non-paging, remote read operation. The benchmarks, which are described below, are as simple as possible consistent with their forming a basis for estimating the performance of other transactions.

The execution times of benchmarks, while useful for predicting the performance of other transactions, do not explain how transaction performance changes as a function of algorithmic or underlying system changes. Nor do the execution times of benchmarks shed light on the resources that they use. To provide this additional information requires a more complex analysis. The analysis that we propose is based on the notion that each benchmark is substantially made up of the repetitious execution of a collection of primitive operations, such as disk reads or inter-node datagrams. These primitives have counterparts in all transaction systems and collectively account for much of the execution time of a transaction.

The primitive operations we use are the following:

- **Data Server Calls** in TABS are remote procedure calls between applications and data servers on a single node. Servers instantiate a coroutine for each call. We measure the time for the Data Server Call primitive by measuring the time for a TABS application process to call a null procedure in a TABS data server process.

- **Inter-Node Data Server Calls** are implemented and measured analogously to single node Data Server Calls. These calls use sessions implemented by the Communication Manager.

- **Datagrams** are used for inter-node transaction management messages.

- Accent inter-process messages are used for communication between TABS applications, data servers, and TABS system processes on one node. Because message performance depends on the size of messages and on the method by which data is transferred from one process to another, three different message types are counted:

    - A **Small Contiguous Message.** Small messages typically contain less than 100 bytes, but in all cases have less than 500 bytes.

    - A **Large Contiguous Message.** We use 1100 bytes for the average size of these messages.

    - A **Pointer Message** containing a pointer to data that is transmitted by copy-on-write remapping of processes' virtual memory.

[2] This methodology does not address the effects of concurrent transaction execution on performance, even though TABS fully supports the necessary synchronization.

Primitive	Average Time
Data Server Call	26.1
Inter-Node Data Server Call	89.
Datagram	25.
Small Contiguous Message	3.0
Large Contiguous Message	4.4
Pointer Message	18.3
Random Access Paged I/O	32.
Sequential Read	18.
Stable Storage Write	79.

Table 5-1: Primitive Operation Times (in milliseconds)

These primitive times are used to predict system time in Table 5-4.

- In TABS, all disk reads and writes, other than those for the log, are performed by Accent as part of its demand paging of virtual memory. Pages are 512 bytes. In Accent, random access reads and writes take about the same time, so we report only one (combined) **Random Access Paged I/O** primitive. Because our Perq's have only a single disk, log writing breaks up sequential access disk writes, so sequential access writes do not occur. **Sequential Reads** do occur in our benchmarks, and this primitive is also reported separately.
- The **Stable Storage Write** primitive is the elapsed time required for the Recovery Manager to force a page of log data to non-volatile storage.

The costs of the primitives were estimated by repeatedly calling the appropriate Accent and TABS functions. For example, we determined demand-paged I/O costs by instrumenting a program that repeatedly read (or wrote) individual pages in a large array that is mapped into virtual memory. This experiment measures the average cost of a read (or a read/write pair). Table 5-1 shows the measured performance of the primitive operations on a Perq T2 computer [Perq Systems Corporation 84]. The Data Server Call primitive time is high due to an inefficient implementation of coroutines. As background, we note that the speed of a Perq executing Pascal is approximately 20 to 25 percent the speed of a Vax-11/780 executing C [Fitzgerald and Rashid 86].

After determining the appropriate primitives and measuring their performance, the next step in our analysis is to define a set of benchmarks and to express the latency of each benchmark as a function of primitive operation times. The benchmarks are among the simplest that can be designed to produce the desired system behavior. There are four dimensions of system behavior that the benchmarks exercise. First, some benchmarks are read-only while others modify data. Second, benchmarks either cause no page faults, cause random page faults, or read pages sequentially. Third, benchmarks either perform a single data

Benchmark	Data Server Calls	Remote Data Server Calls	Small Local Msg	Large Local Msg	Sequential Page Reads	Random Page I/O
1 Local Read, No Paging	1		4			
5 Local Read, No Paging	5		4			
1 Local Read, Seq. Paging	1		4		1	
1 Local Read, Random Paging	1		4			.86
1 Local Write, No Paging	1		6	1		
5 Local Write, No Paging	5		14	5		
1 Local Write, Seq. Paging	1		10	1		2
1 Lcl Rd, 1 Rem Rd, No Paging	1	1	8			
1 Lcl Rd, 5 Rem Rd, No Paging	1	5	8			
1 Lcl Rd, 1 Rem Rd, Seq. Paging	1	1	8		2	
1 Lcl Wr, 1 Rem Wr, No Paging	1	1	12	2		
1 Lcl Wr, 1 Rem Wr, Seq. Paging	1	1	20	2		4
1 Lcl Rd, 1 Rem Rd, 1 Rem Rd, NP	1	2	11			
1 Lcl Wr, 1 Rem Wr, 1 Rem Wr, NP	1	2	17	3		

Table 5-2: Pre-Commit Primitive Counts

This table shows the number of primitive operations each benchmark is expected to perform before starting commit. The primitive operations are listed in Table 5-1. The number .86 is the measured number of page I/O's per transaction. Blank entries denote zero values.

server operation on each node or perform multiple data server operations on one of the nodes in the benchmark. Finally, benchmarks perform operations on one, two, or three nodes.

There are four read-only benchmarks in which both the application and data server process are located on the same machine. The simplest is a transaction that reads an identical element of a recoverable array of integers. The second is similar, but is a transaction that reads the same array element five times. This permits the determination of the costs of individual read operations on data servers. The third is similar to the first but is modified to measure the performance of the demand paging of recoverable data. It is a transaction that reads an element from successive pages of a large array. This array is 5000 pages, which is more than three times the available physical memory on a Perq with TABS running. The final test reads random elements from the array and demonstrates the effect of random I/O times on TABS performance.

The performance of the system for transactions that modify recoverable data is measured by benchmarks that write the array instead of reading it. Because there is only one disk on our system, there should be no significant difference between the random-access case and the sequential-access case because of the intervening seeks required by paging writes. Hence, we include only a sequential paging test.

To study the performance effects of inter-node communications, there are similar benchmarks that use two data servers, one on the same node as the application and one on a remote node. Read tests have one local non-paging read and one remote non-paging read; one local non-paging read and five remote non-paging reads; and one local sequential paging read and one remote sequential paging read. Two additional tests measure two node write transactions: one test with one local non-paging write and one remote non-paging write, and one test with one local paging write and one remote paging write. These remote write tests reflect the cost of the more complex two-phase commit protocol. We do not include a benchmark that measures 5-write operations remotely, as this can be deduced from other benchmark times.

To show how the cost of transaction commit increases as a function of the number of nodes, benchmarks that read or write the same cell on three nodes are included. The performance of these benchmarks must be adjusted for the number of operations to show the incremental commit cost directly.

The time in each benchmark attributable to primitive operations can be expressed as a function of primitive operation times. Potentially, this analysis involves complicated stochastic models, but our benchmarks have a simple approximate analysis. In our transaction model, all operations prior to commitment execute sequentially. Hence the pre-commit latency of a transaction that is due to the execution of primitive operations is a sum of the primitive operation times weighted by the numbers of primitive operations performed. The benchmarks are deterministic in steady state, so determining the primitive counts is fairly easy. For the random read benchmark, it is simpler to count page reads during the test than to measure the available buffer memory and estimate what fraction of references will be to pages in the buffer. These formulas are reported in Table 5-2, the *Pre-Commit Primitive Count Table*.

The latency of the commit portion of a transaction is sequential in the local case, but involves parallel processing in the distributed case. For each type of transaction commit protocol, we estimate the execution path of longest duration through the distributed system. This path is used as the basis of the benchmark counts that are incorporated in Table 5-3, the *Commit Primitive Count Table*. Because different transactions use the same commit protocol, there are fewer entries in this table than in the Pre-Commit Benchmark Count Table. Commit times for the three node benchmarks are longer than commit

Protocol	Remote Datagram Msg	Small Local Msg	Large Local Msg	Local Pointer Msg	Stable Storage Writes
1 Node, Read Only		5			
1 Node, Write		8	1		1
2 Node, Read Only	2	11	1		1
2 Node, Write	4	17	5	1	4
3 Node, Read Only	2.5	11	1		1
3 Node, Write	5	17	5	1	4

Table 5-3: Commit Primitive Counts

This table shows the number of primitive operations in the longest estimated execution path for various commit protocols. The one-half datagram time in the 3 Node, Read Only case is an approximation for the time required to immediately send a "Prepare" datagram to the second remote node. The 3 Node, Write case contains 2 one-half datagram times, because there is also a "Commit" datagram that is sent to the second remote node. Blank entries denote zero values.

Benchmark	Sys Time Predicted by Primitives	Measured TABS Proc Time	Measured Elapsed Time	Improved TABS Architecture	New Primitive Times
1 Local Read, No Paging	53	41	110	107	67
5 Local Read, No Paging	157	41	217	213	80
1 Local Read, Seq. Paging	71	41	126	123	75
1 Local Read, Random Paging	81	41	140	137	98
1 Local Write, No Paging	156	83	247	228	136
5 Local Write, No Paging	302	119	467	424	225
1 Local Write, Seq. Paging	232	104	371	345	249
1 Lcl Rd, 1 Rem Rd, No Page	306	223	469	459	228
1 Lcl Rd, 5 Rem Rd, No Page	662	368	829	819	268
1 Lcl Rd, 1 Rem Rd, Seq. Page	341	226	514	504	257
1 Lcl Wr, 1 Rem Wr, No Page	697	407	989	775	442
1 Lcl Wr, 1 Rem Wr, Seq. Page	864	441	1125	873	539
1 Lcl Rd, 1 Rem Rd, 1 Rem Rd, NP	416	381	621	611	282
1 Lcl Wr, 1 Rem Wr, 1 Rem Wr, NP	831	670	1200	968	534

Table 5-4: Benchmark Times (in milliseconds)

This table shows predicted, average measured, and projected improved times for the benchmarks. The **System Time Predicted by Primitives** is computed by summing the primitive operation times for each benchmark from Tables 5-1, 5-2, and 5-3. **Measured TABS Process Time** is the sum of TABS system process times on all nodes. **Measured Elapsed Time** is the *average* measured time of the benchmark over a long run including all points except starting and ending transients. For the single node tests, **Predicted System Time** plus **Measured TABS Process Time** should approximately yield **Measured Elapsed Time**, as they do. As described in the text, the TABS architecture could be improved and the primitive times reduced. The **Improved TABS Architecture** column shows projection of elapsed times based on algorithmic and structural changes to TABS. The **New Primitive Times** column shows how the times in the preceding column would improve if primitive operations times were as in Table 5-5. Multi-node write tests used one or two Perq 2 computers, which have average disk seek times about 15 milliseconds slower than the Perq T2 used for the primitive time measurements of Table 5-1. The **System Time Predicted by Primitives** for these tests have been increased by 30 or 45 milliseconds. Projected times always assume the use of Perq T2 disks.

times for the two node benchmarks by a one-half datagram time for read-only transactions and by 2 one-half datagram times for update transactions. This is due to the estimated cost of sending datagrams in parallel to different nodes.

5.2. The Performance of TABS

The sum of the primitive operation times in Table 5-1, as weighted by the counts of Tables 5-2 and 5-3, accounts for a significant portion of the latency of each benchmark. This sum is shown in the first column of Table 5-4, labeled "System Time Predicted by Primitives."

Benchmarks times were measured by counting the number of transactions executed in 20 or 30 second time intervals and averaging these rates over 20 to 30 minutes of testing. Transients at the beginning and ending of tests were discarded. The column labeled "Measured Elapsed Time" in Table 5-4 show the average elapsed time for each benchmark. The column labeled "Measured TABS Process Time" reports the sum of average measured CPU time of the TABS Communication, Recovery, and Transaction Manager Processes on all nodes in the test.

Rather than reiterating numbers in the tables, we instead present more details about the performance of the system. We account for the latency of a local, single operation, non-paging read transaction. We also show where the additional time is spent in a single node, non-paging write transaction. Finally, we show how to reconcile System Time Predicted by Primitives, Measured TABS Process Time, and Measured Elapsed Time for two-node transactions. This discussion uses execution time data for individual processes, which are not included in Table 5-4.

The measured elapsed time for processing a transaction that performs a single node, non-paging read operation is 110 msec. This is 57 msec greater than predicted by primitive operations alone. Of this additional time, 41 msec is accounted for by TABS system processes: 36 msec in the Transaction Manager and 5 msec in the Recovery Manager. (TABS system process times remain constant in all local read-only transactions.) By a

complex deduction, we determined that the application and data server require about 3 msec and 4 msec, respectively, to initiate and commit a transaction. Our analysis does not account for the remaining 9 msec.

The difference in measured times between the simplest read and simplest update transactions is 137 msec, of which 78 msec is the time for the Stable Storage Write. The data server uses an additional 5 msec to do a write, rather than a read. This time is used to format and send log data to the Recovery Manager. The Recovery Manager uses an extra 10 msec to spool this data to the log. The more complex commit protocol for an update-transaction requires an additional 8 msec in the Recovery Manager, 24 msec in the Transaction Manager, and 4 msec in the data server. Together, these times with the additional message primitives executed (see Tables 5-2 and 5-3) sum to 155 milliseconds. This is 18 milliseconds more than measured, which may be partially due to double counting some Recovery Manager time included in the Stable Storage Write time.

Two-node distributed transactions involve little parallel execution, so we might expect System Time Predicted by Primitives plus Measured TABS Process Time to equal Measured Elapsed Time. This is not true, however, because communication time is counted in both the Measured TABS Process Time and the System Time Predicted by Primitives. If the Communication Manager time were subtracted, the sum of the remaining TABS Process time and Predicted times is within 4 percent of elapsed time for read transactions and within 10 percent for write transactions. Three node transactions involve considerable parallel processing during commit so this simple reconciliation is not applicable.

5.3. Improving TABS performance

In this section, we use the primitive operation analysis to project the performance of different implementations of TABS. Two projections are given here. The first projection is based on the measured times of primitive operations reported above, but assumes feasible architectural and implementation changes to TABS. The second projection is based on the first, but also assumes new primitive operation times, which are described and justified below. In neither case are we counting on a faster processor or better compiler; thus, projected times are higher than measured TABS Process Time except for benchmarks having parallelism or high communication costs.

For the first projection, labeled "Improved TABS Architecture" in Table 5-4, we assume that the Recovery Manager and Transaction Manager processes are merged with the Accent kernel. This eliminates message passing between these three components, and also allows one prepare message sent from a data server to the modified kernel to perform the function of two messages in the current implementation. We have previous experience with the integration of functions implemented by separate processes into the kernel and believe that this is a simple process. Additionally, we assume optimized commit algorithms that eliminate unnecessary messages and permit some of the processing for commit of distributed write transactions to occur in parallel with the execution of succeeding transactions. The projections based on these changes are derived by reducing the measured elapsed times by the times for primitive operations that would not be performed. Remote write transactions show the biggest performance increase, because of the elimination of considerable commit processing from the critical execution path of the transaction.

The second performance projection, labeled "New Primitive Times" in Table 5-4 is derived from the "Improved TABS Architecture" projections by setting the primitive operation times to those given in Table 5-5. The costs of these new primitives are based on our estimates of the applicability to the Perq/Accent environment of published techniques for efficient implementation of these primitives. Accent random I/O times already approach the performance of the disk, so we do not assume any improvement here, though we hypothesize a small improvement in sequential read time.

Primitive	Average Time
Data Server Call	2.5
Inter-Node Data Server Call	9.
Datagram	2.0
Small Contiguous Message	1.0
Large Contiguous Message	1.25
Pointer Message	15.
Random Access Paged I/O	32.
Sequential Read	10.
Stable Storage Write	32.

Table 5-5: Achievable Primitive Operation Times (in milliseconds)
This table shows primitive times achievable by tuning software and adding disks.

Intra-processor message times have been reported as low as 0.77 msec on hardware that is (roughly) similar in performance to the Perq [Cheriton 84b]. However, Accent processes have completely separate virtual address spaces and context switching times are greater for Accent than for other operating systems, and so we chose times of 1.0 and 1.25 msec for our projections. The implementation of pointer messages is fairly complex and we therefore assume only small improvement. Careful implementation or the use of lazy evaluation should substantially eliminate to high costs of coroutine allocation in the Data Server Call primitive.

Considerable work has been devoted to efficient inter-processor message passing [Birrell and Nelson 84, Spector 82].

We feel that times of 9 msec for remote data server calls, and 2 msec for datagram messages allow reasonable overheads compared with times reported for similar hardware [Nelson 81].

If the existence of small (disk track size) quantities of zero latency stable storage (e.g. battery backup CMOS primary memory) and dedicated logging disks are assumed, then log writing costs could approach main memory copy costs. However, to lend more credence to our projections we estimate that log writing can be performed for the same cost as paged disk writes. This estimate assumes dedicated logging disks and offline archival of the log.

With these improvements, the projected performance of local transactions range from 67 msec for non-paging, read-only transactions to 249 msec for paging write transactions. The performance of multi-node benchmark transactions range from 228 msec to 539 msec. Of course, these numbers could be reduced further by improving the code in the TABS system components and by using a faster CPU. TABS system process times dominate the costs in these projections, and their execution time would decrease on a faster CPU.

6. Relationship to R* and Argus

TABS is similar in many ways to R* and Argus [Williams et al. 81, Lindsay et al. 84, Liskov 84, Liskov et al. 83]. R* is a distributed database management system, developed at IBM San Jose Research, that supports transactions on relational database servers. Argus is a programming language, developed at the MIT Laboratory for Computer Science, that supports transactions and user-defined types on which they can operate.

The transaction facility of R* is implemented by a combination of the underlying operating system, CICS [IBM Corporation 78], and a component called TM*. This logically unified facility permits servers to register themselves and their operations when they are ready to receive requests, and performs routing of operation requests to local servers. The facility also issues transaction identifiers, oversees transaction commitment and aborting, and does deadlock detection.

Servers in R* have two types of interfaces. The first type includes operations specific to a server. The second type includes operations required for transaction management, deadlock detection/resolution, and remote access by other servers. In R*, requests are never directly issued to remote servers. Instead, they are passed to local servers, which then interact with remote ones.

Broadly, TABS is very similar to R* in that both systems make available transaction facilities for applications and servers. However, they differ in many ways. For example, TABS, its applications, and its servers are implemented as a collection of processes that communicate via messages, rather than via the protected procedure calls, which R* uses. Another major difference is that remote servers in TABS can be directly invoked in a transparent way. Also, TABS servers retain little context between operations and use a common log and recovery algorithms provided by the system; servers in R* must utilize the same context for each operation within a transaction, and each server must provide for its own recovery. Some of these differences are relatively minor, but some affect performance or usability. For example, the common log and transparent inter-node communication provide efficiency and flexibility respectively; but, on the other hand, protected procedure calls on the IBM 370 are very fast.

Internally, Argus contains many facilities that are analogous to those of TABS and R*, but it has the more ambitious goal of making those facilities very easy to use. Some objects can be implemented without the type implementor having to consider synchronization or recovery issues. However, types needing highly concurrent access require explicit attention paid to synchronization and recovery. For these high concurrency types, synchronization and recovery are done with the aid of a specialized object, called a *mutex*, rather than via explicit locking and logging.

Argus is certainly easier to use than TABS for constructing simple objects. However, it is difficult to compare the amount of work needed to use mutex objects versus that of explicitly setting locks and writing log records. We have not considered the performance differences between the approaches.

7. Conclusions

Our use of TABS has convinced us that its facilities for supporting transactions and data servers are useful for both local and distributed abstractions. Specialized distributed database systems, file systems, mail systems, spoolers, editors, etc. could be based on the implementation techniques that our existing servers use. In our view, the use of location-transparent operation invocation, locking within data servers, write-ahead logging with a common log, and the implementation of permanent objects in virtual memory were good design choices. We must give due credit to the Accent kernel, which implements many of the facilities that TABS uses or provides, and which has proven invaluable for supporting distributed computation.

Because TABS uses nearly the minimum number of expensive primitive operations such as disk I/O's, log writes, inter-node messages, and datagrams, TABS performance is sufficient for many applications in an interactive workstation environment. Transactions considerably more complex than the benchmarks of Section 5 take less than a few seconds of elapsed time. For example, our analysis indicates that about two seconds are required for a local transaction that invokes five operations, each of which updates two pages that are not in memory. The same transaction would require about one-half second if the data were

in main memory. If the operations were performed on one or more remote nodes, these transactions would take only about one second longer.

Certainly, TABS can be substantially improved. To simplify programming simple data servers, the calls to TABS synchronization and recovery facilities should be hidden in a language run-time system, such as that of Argus. For more complex servers that need greater flexibility, the server library should provide a better set of primitives, including some for operation logging and type-specific locking. Thought should also be given to providing better debugging support for data servers.

Functionally, TABS should be extended to permit the recovery of a single server without the recovery of the entire node. In addition, TABS should use stable storage for the log and support media recovery. Finally, TABS should probably have a more complete subtransaction model, particularly for the implementation of replicated objects.

In its implementation, TABS loses performance because of the division of the Recovery Manager, Transaction Manager, and Accent into separate processes. The TABS coroutine, logging, and inter-node communication facilities need re-implementation or tuning. If these changes were made and TABS used more modern hardware, one would expect transaction times that are four to ten times faster than the currently measured ones.

We are continuing to enhance the system and study its use. For example, we plan to empirically compare the relative merits of value and operation logging. We are also continuing to investigate architectures and algorithms that will provide increased transaction throughput. In addition, we would like to develop a performance methodology for measuring and predicting throughput. Though much work remains, our experiences to date have convinced us that general purpose distributed transaction facilities are feasible and useful for a wide variety of systems.

Acknowledgments

Jacob Butcher, Charles Fineman, Sherri Menees, and Peter Schwarz made major design and programming efforts; their work was essential to the underlying TABS prototype and the data servers that use it. Maxwell Berenson constructed the distributed performance monitoring system that made it possible to get accurate performance measurements of distributed transactions. Sherri Menees provided editorial assistance for this paper, and Maurice Herlihy, David Nichols, and Rick Rashid provided helpful comments.

References

[Allchin and McKendry 83] J. E. Allchin, M.S. McKendry. Synchronization and Recovery of Actions. In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 31-44. ACM, August, 1983.

[Anonymous et al. 85] Anonymous, et al. A Measure of Transaction Processing Power. *Datamation* 31(7), April, 1985. Also available as Technical Report TR 85.2, Tandem Corporation, Cupertino, California, January 1985.

[Banatre et al. 83] J. P. Banatre, M. Banatre, F. Ployette. Construction of a Distributed System Supporting Atomic Transactions. In *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems*. IEEE, October, 1983.

[Bernstein and Goodman 81] Philip A. Bernstein, Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys* 13(2):185-221, June, 1981.

[Birman et al. 83] K. P. Birman, D. Skeen, A. El Abbadi, W.C. Dietrich, T. Raeuchle. *Isis: An Environment for Constructing Fault-Tolerant Distributed Systems*. Technical Report 83-552, Cornell University, 1983.

[Birrell and Nelson 84] Andrew D. Birrell, Bruce J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems* 2(1):39-59, February, 1984.

[Bloch et al. 84] Joshua J. Bloch, Dean S. Daniels, Alfred Z. Spector. *Weighted Voting for Directories: A Comprehensive Study*. Technical Report CMU-CS-84-114, Carnegie-Mellon University, April, 1984.

[Cheriton 84a] David R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software* 1(2):186-213, April, 1984.

[Cheriton 84b] David R. Cheriton. An Experiment using Registers for Fast Message-Based Interprocess Communication. *Operating Systems Review* 18(4):12-20, October, 1984.

[Dahl and Hoare 72] O.J. Dahl, C. A. R. Hoare. Hierarchical Program Structures. In C. A. R. Hoare (editor), *A.P.I.C. Studies in Data Processing*. Volume 8: *Structured Programming*, chapter 3, pages 175-220.Academic Press, London and New York, 1972.

[Daniels 82] Dean S. Daniels. Query Compilation in a Distributed Database System. Master's thesis, Massachusetts Institute of Technology, March, 1982.

[Daniels and Spector 83] Dean S. Daniels, Alfred Z. Spector. An Algorithm for Replicated Directories. In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 104-113. ACM, August, 1983.

[Date 83] C. J. Date. *The System Programming Series: An Introduction to Database Systems Volume 2*. Addison-Wesley, Reading, MA, 1983.

[Department of Defense 82] *Reference Manual for the Ada Programming Language* July 1982 edition, Department of Defense, Ada Joint Program Office, Washington, DC, 1982.

[Diel et al. 84] Hans Diel, Gerald Kreissig, Norbet Lenz, Michael Scheible, Bernd Schoener. Data Management Facilities of an Operating System Kernel. In *Sigmod '84*, pages 58-69. June, 1984.

[Dwork and Skeen 83] Cynthia Dwork, Dale Skeen. The Inherent Cost of Nonblocking Commitment. In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 1-11. ACM, August, 1983.

[Eppinger and Spector 85] Jeffrey L. Eppinger, Alfred Z. Spector. *Virtual Memory Management for Recoverable Objects in the TABS Prototype*. Technical Report CMU-CS-85-163, Carnegie-Mellon University, October, 1985. Forthcoming.

[Eswaran et al. 76] K. P. Eswaran, James N. Gray, Raymond A. Lorie, Irving L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM* 19(11):624-633, November, 1976.

[Fabry 74] R. S. Fabry. Capability-Based Addressing. *Communications of the ACM* 17(7):403-411, July, 1974.

[Fitzgerald and Rashid 86] Robert P. Fitzgerald, Richard F. Rashid. The Integration of Virtual Memory Management and Interprocess Communication in Accent. *ACM Transactions on Computer Systems* 4(2), May, 1986. To be presented at the Tenth Symposium on Operating System Principles, Orcas Island, Washington, December, 1985.

[Gifford 79] David K. Gifford . Weighted Voting for Replicated Data. In *Proceedings of the Seventh Symposium on Operating System Principles*, pages 150-162. ACM, December, 1979.

[Gray 78] James N. Gray. Notes on Database Operating Systems. In R. Bayer, R. M. Graham, G. Seegmuller (editors), *Lecture Notes in Computer Science*. Volume 60: *Operating Systems - An Advanced Course*, pages 393-481.Springer-Verlag, 1978. Also available as Technical Report RJ2188, IBM Research Laboratory, San Jose, California, 1978.

[Gray 80] James N. Gray. *A Transaction Model*. Technical Report RJ2895, IBM Research Laboratory, San Jose, California, August, 1980.

[Gray et al. 81] James N. Gray, et al. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys* 13(2):223-242, June, 1981.

[Haerder and Reuter 83] Theo Haerder, Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys* 15(4):287-318, December, 1983.

[Herlihy 84] Maurice P. Herlihy. *General Quorum Consensus: A Replication Method for Abstract Data Types*. Technical Report CMU-CS-84-164, Carnegie-Mellon University, December, 1984.

[IBM Corporation 78] *Customer Information Control System/Virtual Storage, Introduction to Program Logic* SC33-0067-1 edition, IBM Corporation, 1978.

[Jensen and Pleszkoch 84] E. D. Jensen, N. Pleszkoch. ArchOS: A Physically Dispersed Operating System. *IEEE Distributed Processing Technical Committee Newsletter* , June, 1984.

[Jones et al. 85] Michael B. Jones, Richard F. Rashid, Mary R. Thompson. Matchmaker: An Interface Specification Language for Distributed Processing. In *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages*, pages 225-235. ACM, January, 1985.

[Joy et al. 83] William Joy, Eric Cooper, Robert Fabry, Samuel Leffler, Kirk McKusick, David Mosher. *4.2 BSD System Interface Overview*. Technical Report CSRG TR/5, University of California Berkeley, July, 1983.

[Korth 83] Henry F. Korth. Locking Primitives in a Database System. *Journal of the ACM* 30(1):55-79, January, 1983.

[Lampson 81] Butler W. Lampson. Atomic Transactions. In G. Goos and J. Hartmanis (editors), *Lecture Notes in Computer Science*. Volume 105: *Distributed Systems - Architecture and Implementation: An Advanced Course*, chapter 11, , pages 246-265.Springer-Verlag, 1981.

[Lansky 80] Amy L. Lansky. *Pasmac -- A Macro Processor for Pascal*. Technical Report CSL-TN-174, Stanford University Computer Systems Laboratory, April, 1980.

[Lindsay et al. 79] Bruce G. Lindsay, et al. *Notes on Distributed Databases*. Technical Report RJ2571, IBM Research Laboratory, San Jose, California, July, 1979. Also appears in Droffen and Poole (editors), *Distributed Databases*, Cambridge University Press, 1980.

[Lindsay et al. 84] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, Robert A. Yost. Computation and Communication in R*: A Distributed Database Manager. *ACM Transactions on Computer Systems* 2(1):24-38, February, 1984.

[Liskov 82] Barbara Liskov. On Linguistic Support for Distributed Programs. *IEEE Transactions on Software Engineering* SE-8(3):203-210, May, 1982.

[Liskov 84] Barbara Liskov. *Overview of the Argus Language and System*. Programming Methodology Group Memo 40, Massachusetts Institute of Technology Laboratory for Computer Science, February, 1984.

[Liskov and Herlihy 83] Barbara Liskov, Maurice Herlihy. Issues in Process and Communication Structure for Distributed Programs. In *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems*. October, 1983.

[Liskov and Scheifler 82] Barbara Liskov, Robert Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. In *Proceedings of the Ninth Annual Symposium on the Principles of Programming Languages*, pages 7-19. ACM, January, 1982.

[Liskov et al. 83] B. Liskov, M. Herlihy, P. Johnson, G. Leavent, R. Scheifler, W. Weihl. *Preliminary Argus Reference Manual*. Programming Methodology Group Memo 39, Massachusetts Institute of Technology Laboratory for Computer Science, October, 1983.

[Lomet 77] David B. Lomet. Process Structuring, Synchronization, and Recovery Using Atomic Actions. *ACM SIGPLAN Notices* 12(3), March, 1977.

[Lorie 77] Raymond A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems* 2(1):91-104, March, 1977.

[Moss 81] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, April, 1981.

[Nelson 81] Bruce Jay Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, May, 1981. Available as Technical Report CMU-CS-81-119a, Carnegie-Mellon University.

[Obermarck 82] Ron Obermarck. Distributed Deadlock Detection Algorithm. *ACM Transactions on Database Systems* 7(2):187-208, June, 1982.

[Paxton 79] William H. Paxton. A Client-Based Transaction System to Maintain Data Integrity. In *Proceedings of the Seventh Symposium on Operating System Principles*, pages 18-23. ACM, December, 1979.

[Perq Systems Corporation 84] *Perq System Overview* March 1984 edition, Perq Systems Corporation, Pittsburgh, Pennsylvania, 1984.

[Rashid and Robertson 81] Richard Rashid, George Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the Eighth Symposium on Operating System Principles*, pages 64-75. ACM, December, 1981.

[Reed 78] David P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Massachusetts Institute of Technology, September, 1978.

[Reuter 84] Andreas Reuter. Performance Analysis of Recovery Techniques. *ACM Transactions on Database Systems* 9(4):526-559, December, 1984.

[Saltzer 74] Jerome H. Saltzer. Protection and the Control of Information in Multics. *Communications of the ACM* 17(7), July, 1974.

[Schwarz 84] Peter M. Schwarz. *Transactions on Typed Objects*. PhD thesis, Carnegie-Mellon University, December, 1984. Available as Technical Report CMU-CS-84-166, Carnegie-Mellon University.

[Schwarz and Spector 84] Peter M. Schwarz, Alfred Z. Spector. Synchronizing Shared Abstract Types. *ACM Transactions on Computer Systems* 2(3):223-250, August, 1984. Also available as Technical Report CMU-CS-83-163, Carnegie-Mellon University, November 1983.

[Spector 82] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM* 25(4):246-260, April, 1982.

[Spector and Daniels 85] Alfred Z. Spector, Dean S. Daniels. Performance Evaluation of Distributed Transaction Facilities. September, 1985.Presented at the Workshop on High Performance Transaction Processing, Asilomar, September, 1985.

[Spector and Schwarz 83] Alfred Z. Spector, Peter M. Schwarz. Transactions: A Construct for Reliable Distributed Computing. *Operating Systems Review* 17(2):18-35, April, 1983. Also available as Technical Report CMU-CS-82-143, Carnegie-Mellon University, January 1983.

[Spector et al. 85] Alfred Z. Spector, Jacob Butcher, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman, Abdelsalam Heddaya, Peter M. Schwarz. Support for Distributed Transactions in the TABS Prototype. *IEEE Transactions on Software Engineering* SE-11(6):520-530, June, 1985. Also available in Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems, Silver Springs, Maryland, IEEE, October, 1984 and as Technical Report CMU-CS-84-132, Carnegie-Mellon University, July, 1984.

[Stonebraker 84] Michael Stonebraker. Virtual Memory Transaction Management. *Operating Systems Review* 18(2):8-16, April, 1984.

[Tandem 82] *ENCOMPASS Distributed Data Management System* Tandem Computers, Inc., Cupertino, California, 1982.

[Traiger 82] Irving L. Traiger. *Virtual Memory Management for Database Systems*. Technical Report RJ3489, IBM Research Laboratory, San Jose, California, May, 1982.

[Watson 81] R.W. Watson. Distributed system architecture model. In B.W. Lampson (editors), *Lecture Notes in Computer Science*. Volume 105: *Distributed Systems - Architecture and Implementation: An Advanced Course*, chapter 2, , pages 10-43.Springer-Verlag, 1981.

[Weihl and Liskov 83] W. Weihl, B. Liskov. Specification and Implementation of Resilient, Atomic Data Types. In *Symposium on Programming Language Issues in Software Systems*. June, 1983.

[Williams et al. 81] R. Williams, et al. *R*: An Overview of the Architecture*. IBM Research Report RJ3325, IBM Research Laboratory, San Jose, California, December, 1981.

[Wulf et al. 74] W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack. HYDRA: The Kernel of a Muiltiprocessor Operating System. *Communications of the ACM* 17(6):337-345, June, 1974.