Distributed Transaction Processing and The Camelot System

Alfred Z. Spector
January 1987
CMU-CS-87-100

ABSTRACT

This paper describes distributed transaction processing, a technique used for simplifying the construction of reliable distributed systems. After introducing transaction processing, the paper presents models describing the structure of distributed systems, the transactional computations on them, and the layered software architecture that suppports those computations. The software architecture model contains five layers, including an intermediate layer that provides a common set of useful functions for supporting the highly reliable operation of system services, such as data management, file management, and mail. The functions of this layer can be realized in what is termed a *distributed transaction facility*. The paper then describes one such facility - Camelot. Camelot provides flexible and high performance commit supervision, disk management, and recovery mechanisms that are useful for inplementing a wide class of data types, including large databases. It runs on the Unix-compatible Mach operating system and uses the standard Arpanet IP communication protocols. Presently, Camelot runs on RT PC's and Vaxes, but it should also run on other computers including shared-memory multiprocessors.

This work was supported by the Defense Advanced Research Projects Agency, ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520, and the IBM Corporation.

This paper is to appear as a chapter of *Distributed Operating Systems: Theory and Practice*, Yakup Paker,ed., Springer-Verlag, 1987.

Table of Contents

List of Figures

1. Introduction

Distributed computing systems are potentially reliable, because their components are redundant and fail independently. Additionally, distributed computing systems potentially offer very high throughput for applications that can use loosely-coupled multiprocessing. The major challenge is to realize these potentials efficiently, and without incurring intolerable penalties in complexity or cost. Consequently, there has been great interest in general-purpose techniques and practices for simplifying the construction of efficient and robust distributed systems.

This paper discusses a technique based on *distributed transactions* and describes a *distributed transaction facility* called Camelot that supports their use. Section 2 defines the concept of a transaction and provides a brief historical perspective on its development. Section 3 then presents three models useful for describing distributed systems that support transaction processing. Section 4 adds concreteness to this discussion by describing the goals, implementation bases, functions, structure, and status of the Camelot distributed transaction facility. This paper ends with Section 5, which describes some important challenges in the area of distributed transaction processing.

2. Distributed Transactions

The database literature typically defines a transaction as a collection of operations bracketed by two markers: *BeginTransaction* and *EndTransaction*. A transaction has three special properties:

- Either all or none of a transaction's operations are performed. This property is called *failure atomicity*.
- If a transaction completes successfully, the results of its operations will never be lost, except in the event of catastrophes, which can be made arbitrarily improbable. This property is called *permanence*.
- Transactions are allowed to execute concurrently, but the results will be the same as if the transactions executed serially in some order. This property is usually called *serializability*.

Transactions lessen the burden on application programmers by simplifying the treatment of failures and concurrency. Failure atomicity ensures that if a transaction is interrupted by a failure, its partial results will undone. Permanence ensures that updates performed by completed transactions are not lost. Serializability ensures that other concurrently executing transactions cannot observe inconsistencies. Programmers are therefore free to cause temporary inconsistencies during the execution of a transaction knowing that their partial modifications will never be visible.

Transactions were initially developed for centralized data management systems to aid in maintaining application-dependent *consistency constraints* on stored data. Transactions are a very useful abstraction, because these consistency constraints must typically be maintained despite failures and without

unnecessarily restricting the concurrent processing of application requests. System R is an example of an early database system with good support for transactions [Astrahan et al. 76].

Extending the concept of transactions for use on distributed systems was not difficult: Distributed transactions permit operations on multiple objects that are scattered across multiple nodes. Though this concept is easily described, efficient implementations have been difficult to achieve. However, there are database systems that support distributed transactions. Berkeley's Ingres, CCA's SDD-1, and IBM's R* [Stonebraker and Neuhold 77, Rothnie et al. 80, Williams et al. 81, Lindsay et al. 84] are three early examples, and new products have recently been announced by purveyors of commercial database software.

Supporting distributed processing is only one of the possible extensions to the transaction concept. Researchers, starting in the late 1970's, also looked for ways to increase the domain of processing in which distributed transactions could be used. One of their goals was to facilitate the construction of *highly available* applications — that is, applications that continue to work despite failures. Increasing the flexibility of transactions required extensions to the transaction model, as well as new algorithms and architectures to improved their implementation.

Perhaps, the most fundamental addition to the transaction model was the notion that programmers should be permitted invoke operations on user-defined abstract objects, rather than being restricted to using objects of a predefined type, such as relational database. *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. This notion of object is similar to the notions of classes in Simula [Dahl and Hoare 72] and packages in ADA [Department of Defense 82].

Gray and Lomet discussed the extended use of the transaction model in two relatively early papers [Gray 81, Lomet 77]. In other important work, Moss and Reed at MIT developed an important addition to the transaction model by defining what it means to have transactions *nested* within other transactions [Reed 78, Moss 81]. Briefly, nested transactions permit a transaction to spawn children which may run in parallel, but are synchronized so that the transaction system still exhibits serializability. Nested transactions also are permitted to abort without causing the loss of the entire parent transaction's work. Allchin, Herlihy, Schwarz, Weihl, and I also wrote about many other issues in the synchronization, recovery and overall structuring of abstract objects for transaction systems [Allchin 83, Herlihy 85, Schwarz 84, Schwarz and Spector 84, Spector and Schwarz 83, Weihl 84].

Related implementation work has focused on developing what we call a *distributed transaction facility* and compatible linguistic constructs for accessing it. We define a distributed transaction facility as a distributed collection of components that supports the execution of transactions and the implementation of shared abstract data objects on which operations can be performed. Although there is room for diversity in the exact functions of a distributed transaction facility, it must make it easy to begin and end transactions, to call operations on objects from within transactions, and to implement abstract objects that have correct synchronization and recovery properties. By providing a common set of synchronization and recovery mechanisms, such a facility can lead to implementations of separate abstractions that permit their combined use. For example, a computer aided design system might be built using a transactional database management system and a transactional file system.

Among the most well-known implementation work, Liskov and her group at MIT developed the Argus language and runtime system [Liskov 82, Liskov and Scheifler 83, Liskov 84]. At Georgia Tech, a group has worked on developing the Clouds operating system with support for distributed transactions [Allchin and McKendry 83]. Tandem's TMF is an example of a commercial system supporting distributed transaction processing [Helland 85]. Recent work on transactional file systems, for example at UCLA, can also support reliable and available distributed applications [Weinstein et al. 85].[1]

At Carnegie Mellon University, my colleagues and I developed TABS, which is a prototype distributed transaction facility [Spector et al. 85a, Spector et al. 85b]. We are are now developing the production-oriented Camelot (Carnegie Mellon Low Overhead Transaction) facility [Spector et al 86, CMU 87], which is described below. To reduce further the amount of effort required to construct reliable distributed systems, Herlihy and Wing and their colleagues are developing the Avalon language facilities — a collection of constructs that will use Camelot and provide linguistic support for reliable applications [Herlihy and Wing 86]. Avalon comprises runtime libraries and extensions to C++, Common Lisp, and ADA, which together automatically generate the necessary calls on lower-level components [Herlihy and Wing 86]. This work on Camelot and Avalon is based upon our collective experience with TABS and Argus.

To summarize, the goal of this research and development is to simplify the construction of reliable distributed applications that access shared objects by providing facilities for supporting distributed transactions and appropriate interfaces to them. The transaction facilities need to be efficient, easily managed, flexible enough to support a wide collection of abstract types including large databases, and, preferably, useful on heterogeneous hardware in large network environments. The following sections should help concretize these notions.

---

[1] This section contains but a partial list of related work. Due apology is made to researchers whose work has been omitted.

3. Three Models

This section presents three broadly applicable models that are useful for describing distributed systems that support transaction processing: The *system model* describes underlying assumptions about the hardware, storage, and network. The *computation model* describes system operation as a collection of distributed transactions. The *architecture model* describes a five-tiered software organization for a distributed system supporting transactions.

3.1. System Model

There is substantial agreement on the underlying system model for distributed processing, which contains processing nodes and a communication network as Figure 3-1 illustrates. Processing nodes are fail-fast and include uniprocessors or shared memory multiprocessors of many types. Processing nodes are generally assumed to have independent failure modes.

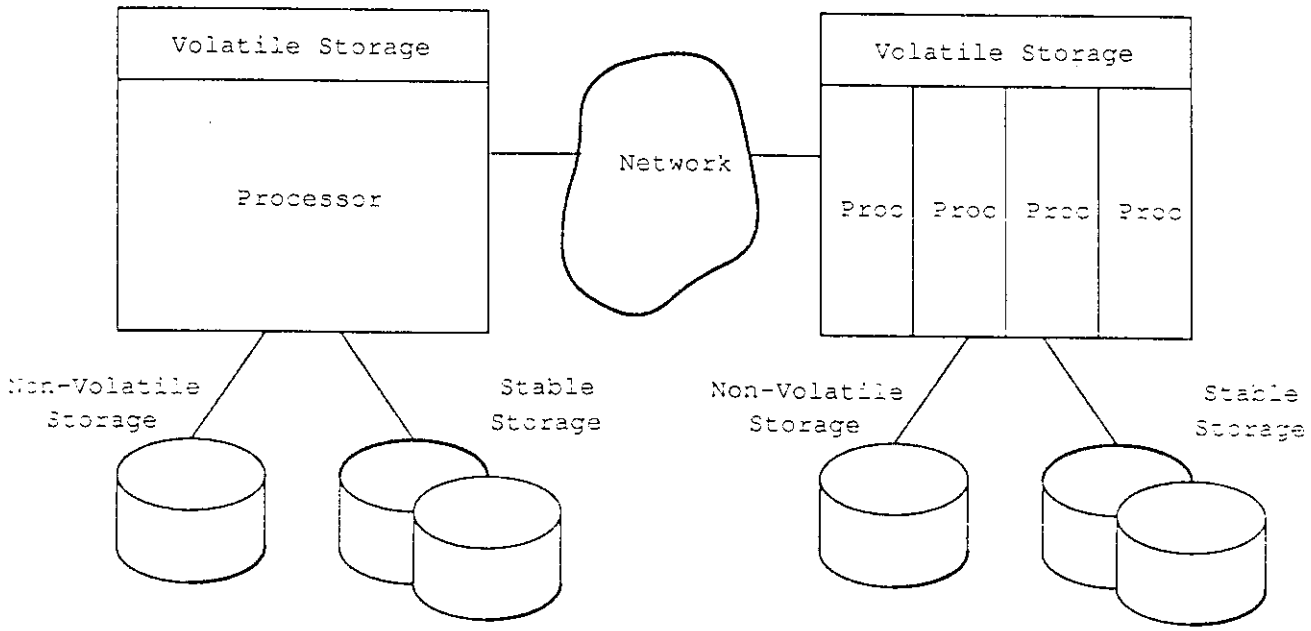

Figure 3-1: System Model

This figure shows the components of the system model. Stable storage and non-volatile storage, though pictured as disks, could be implemented with other media, such as battery backed-up memory.

Storage on processing nodes comprises 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. Stable storage can be implemented using two non-volatile storage units on a node or using a network service [Daniels et al. 86] as described in Section 4.2.3.

The system model's communication network provides datagram-oriented, internetworked OSI Level 3 functions [Zimmermann 82] such as the Arpanet IP protocol [Postel 82]. In other words, the network comprises both local and long-haul components and permits processes to send datagrams having a fixed maximum size. Some local area networks may specially support multicast or broadcast, and the network protocols are assumed to support these features for reasons of efficiency. Because applications using the system may need high availability, communication networks should have sufficient redundancy to render network partitions unlikely. Network partitions can nonetheless occur, so higher levels of the system must take measures to protect themselves against the erroneous computations or inconsistencies that could result.

3.2. Computation Model

The computation model comprises applications that perform processing by executing operations on user-defined data objects within distributed transactions. A transaction either *commits*, in which case all its updates appear to be atomically made, or it *aborts*, in which case no changes appear at all. Transactions may be nested as described above.

Data objects may be distributed across the network and are encapsulated within protection domains that (1) export only operations that make-up the defined interface and (2) guarantee that the invoker has sufficient access rights. Data objects may be nested. This computation model applies to many systems, including R*, Argus, TABS, and Camelot.

3.3. Architecture Model

The architectural model describes how processing on a node is organized; that is, it describes how to realize the computation model on the system model. It is structured in five logical levels, as shown in Figure 3-2. As one might hope, few calls proceed from lower levels to upper levels. (The levels referred to in this model are distinct from the OSI levels, and subsume functions in OSI levels 4 to 7.)

At the base in Level 1 is the operating system kernel that implements processes, local synchronization, and local communication. Example kernels include the V and Accent kernels [Cheriton 84, Rashid and

Robertson 81]. Level 2, the subject of a recent paper [Spector 86], provides session and datagram-based inter-node communication using the network support as defined in the system model. The Mach operating system provides functions in both Levels 1 and 2 [Accetta et al. 86, Rashid 86].

Above the communication level is the distributed transaction facility, Level 3. Level 3 supports failure atomicity and permanence, and was described in Section 2. The distributed transaction facility builds upon the process, synchronization, and communication facilities of Levels 1 and 2.

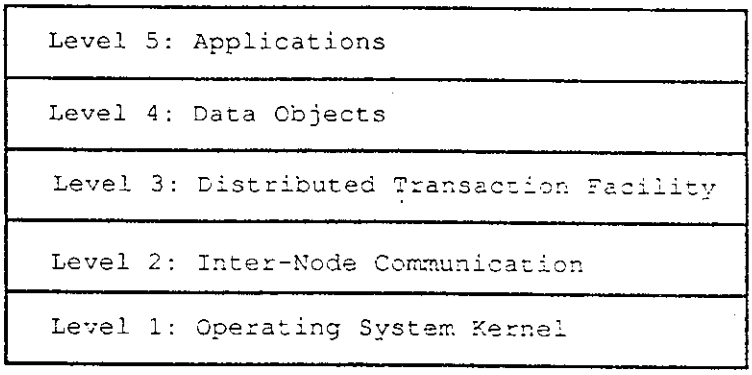

Figure 3-2: Five Level Architecture Model

This figure illustrates the five system levels. The kernel level provides processes and inter-process communication. The communication level provides inter-node communication. The distributed transaction facility provides complete support for transaction processing on distributed objects. Shared data objects are maintained in Level 4 using system-supplied library routines or the runtime runtime support of a programming language. The applications that use the shared data objects are in Level 5.

Abstract objects may be grouped into a subsystem, and there may be multiple subsystems within the Data Object Level (Level 4). The abstract objects, such as database managers or mail systems, use the distributed transaction facility so that they may be accessed within transactions.

In Level 5, applications use the distributed transaction facility to begin, commit, and abort transactions and to execute operations on objects implemented within Level 4. Example applications include a banking terminal system and an interactive interface to a database manager.

The language support used by Levels 4 and 5 does not fit entirely within any level. Rather it naturally consists of one or more translators, which are external to this model, and runtime support that may reside in Levels 4 and 5. Of course, the language makes calls on facilities provided by the three lowest levels. For example, synchronization is typically implemented by runtime support within Level 4 objects, while the

coordination of transaction commits is handled in Level 3, and the transmission of inter-node messages is handled in Level 2.

This architecture, which provides a single distributed transaction facility, has two benefits over traditional architectures that may combine Levels 3, 4, and 5: First, because many of the components that support transactions are standardized and moved lower into the system hierarchy, there is the potential to implement them more efficiently. Second, the architecture provides a common notion of transactions and data objects for all objects and applications in the system. As mentioned in Section 2, this permits more uniform access to data. It allows an application, for example, to update transactionally a relational database containing indexing information, a file containing image data, and a hierarchical database containing performance records.

Having characterized the system, computational, and architectural structures of distributed transaction processing, it is now appropriate to examine a particular implementation — the Camelot distributed transaction facility.

4. Camelot

Camelot provides flexible and efficient support for distributed transactions on a wide variety of user-defined objects such as databases, files, message queues, and I/O objects. Clients of the Camelot facility encapsulate objects within server processes, which then execute operations in response to remote procedure calls. Camelot's features include the following:

- **Compatibility with standard operating systems.** Camelot runs on Mach, a Berkeley 4.3 Unix™-compatible operating system [Rashid 86]. Mach's Unix-compatibility makes Camelot easier to use and ensures that good program development tools are available. Mach's support for shared memory, message passing, and multiprocessors makes Camelot more efficient and flexible.

- **Compatibility with Arpanet protocols.** Camelot uses datagrams and Mach messages, both of which are built on the standard Arpanet IP network layer [Postel 82]. This facilitates large distributed processing experiments.

- **Machine-independent implementation.** Camelot is intended to run on all the uniprocessors and multiprocessors that Mach supports. For example, Camelot is developed on IBM RT PC's, but tested frequently on DEC MicroVaxes to ensure that no machine dependencies have been added.

- **Powerful functions.** Camelot supports functions that are sufficient for many different abstract types. For example, Camelot supports both blocking and non-blocking commit protocols, nested transactions as in Argus, and permits shared, recoverable objects to be accessed in virtual memory. (Section 4.2 describes Camelot's functions in more detail.)

- **Efficient implementation.** Camelot is designed to reduce the overhead of executing transactions. For example, shared memory reduces the use of message passing; multiple threads of control increases parallelism; and a common log reduces the number of synchronous stable storage writes. (Section 4.3 describes Camelot's implementation in more detail.)

- **Careful software engineering and documentation.** Camelot is being coded in C in conformance with careful coding standards [Thompson 86]. This increases Camelot's portability and maintainability and reduces the likelihood of bugs. The internal and external system interfaces are specified in the Camelot Interface Specification [Spector et al 86], which is then processed to generate Camelot code. A manual based on the specification is nearly complete [CMU 87].

Figure 4-1 shows the relationship of Camelot to Avalon and Mach and describes how the components fit into the architecture model.

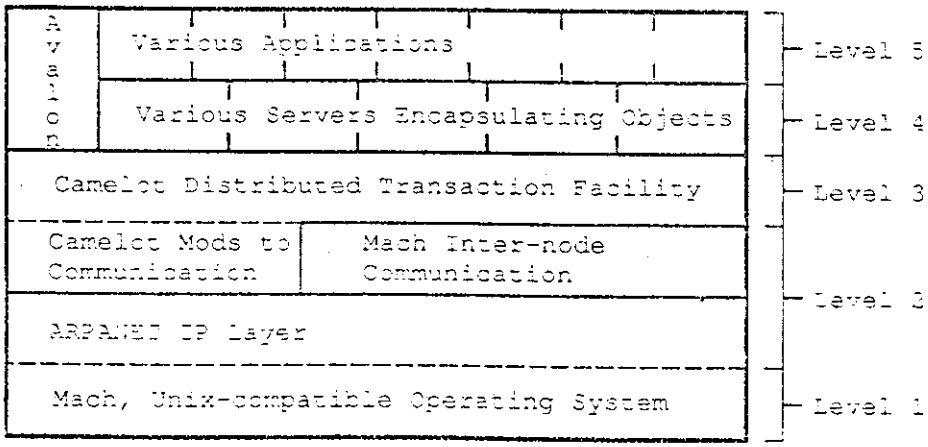

Figure 4-1: Relationship of Camelot to Other System Layers

Mach executes on uniprocessor and multiprocessor hardware and supports the functions of Level 1 of the architecture model. Level 2, or the inter-node communication level, is logically layer on top of Mach. Camelot supports distributed transaction processing (Level 3 functions) and includes several specialized additions to the communication layer. Avalon provides linguistic support for accessing Camelot and Mach, and is not a numbered level of the architecture model. Users define servers encapsulating objects (Level 4) and applications (Level 5) that use those objects. Examples of servers are mail repositories, distributed file systems, and database managers.

4.1. Background on Major Camelot Implementation Techniques

This section identifies many of the algorithms and paradigms that Camelot uses for supporting distributed transactions on abstract objects. The section focuses on issues in abstract objects, distribution, and transaction processing.

4.1.1. Abstract Objects

Many models exist for implementing the abstract objects of Level 4 in the architecture model. In one model, objects are encapsulated in protected subsystems and accessed by protected procedure calls or capability mechanisms [Saltzer 74, Fabry 74]. Camelot 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 send a response message containing the result. Servers that encapsulate data objects are called *Data Servers* in Camelot, *Resource Managers* in R* and *Guardians* in Argus.

Message transmission mechanisms and server organizations differ among implementations based upon the client/server model. In these aspects, Camelot is substantially influenced by the Mach operating system on which it was developed [Rashid 86]. Mach provides heavyweight processes with 32-bit virtual address spaces and supports messages addressed 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 84] 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 Camelot 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 specified in a Pascal-like fashion. Its outputs are client and server stubs that pack and unpack 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 need not.

One implementation approach for servers 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. Another approach is to allocate independently schedulable threads of control that share access to data. With this approach, a server is a class of related processes — in the Simula sense of the word "class." This technique is supported by Mach and used in Camelot [Cooper 86]. 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 can be 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 ports, in response to name lookup requests. Name servers can cooperate with each other to provide transparent naming across a network.

4.1.2. Distribution

Replicated and partitioned distributed objects within Level 4 of the architecture model are feasible to implement using 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 Camelot 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 Mach 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 implements the mapping between the local port used by the sender and the corresponding remote port belonging to the target process, providing transparent communication.

4.1.3. Transactions

Camelot provides features to support all the standard notions of transactions, but does not require that objects enforce serializability, failure atomicity, or permanence. Transactions are permitted to interfere with each other and to show the effects of failure — if this is useful. That is, Camelot provides basic facilities for supporting many different types of objects and lets implementors choose how they want to use them. (A programming language, such as Avalon, can also restrict the ways in which the Camelot facilities are used.)

Many techniques exist for synchronizing the execution of transactions. Locking, optimistic, timestamp, and many hybrid schemes are frequently discussed; many of these are surveyed by Bernstein and Goodman [Bernstein and Goodman 81]. We have chosen to support two compatible types of synchronization in Camelot: locking and hybrid atomicity [Date 83, Herlihy 85]. Hybrid atomicity has features of both timestamps and locking, and requires a Lamport clock facility [Lamport 78] from Camelot and substantial support from Avalon [Herlihy and Wing 86]. Since both types of synchronization are implemented primarily by servers (within Level 4), implementations can be tailored to provide the highest concurrency. For example, 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].

Both locking and hybrid atomicity may delay transaction execution, 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, Camelot Release 1, like many other systems, relies on time-outs, which are explicitly set by system users [Tandem 82].

Recovery in Camelot 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]. It assumes the storage hierarchy defined in the system model, above.

In recovery techniques based upon write-ahead logging, stable storage contains an append-only sequence of records. These records may contain a redo component, that permits the effects of committed transactions to be redone and possibly an undo component that permits the effects of aborted transactions to be undone. 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 commit time, and contiguous allocation of objects on secondary storage [Gray et al. 81, Traiger 82, Reuter 84]. All objects in Camelot Release 1 use one of two co-existing write-ahead logging techniques and share a common log.

The simpler technique is called *old value/new value* logging, in which the undo and redo portions of a log record contain the old and new values of an object's representation. During recovery after node or server crashes, 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 *new value* logging, in which log records contain only a redo component. New value logging requires less log space but results in increased paging for long running transactions. This is because pages cannot be written back to their home location until a transaction completes. Camelot assumes that the invoker of a transaction will know if the transaction will be short and will specify that new value logging should be used.

Release 2 of Camelot will also provide another write-ahead log-based recovery technique 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. Operation based recovery can permit a greater degree of concurrency than the value based recovery, and may require less log space to be written. Its disadvantage is complexity.

The Camelot recovery algorithms are similar to other previously published write-ahead log-based algorithms [Gray 78, Lindsay et al. 79], in particular those of Schwarz [Schwarz 84]. However, they have been extended to support aborts of nested transactions, new value recovery, and the logging of arbitrary regions of memory.

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. Camelot also periodically forces certain pages in volatile storage to non-volatile storage and may abort long running to lessen the amount of log that must be rapidly accessible. To reduce the cost of recovering from disk failures, Camelot infrequently *dumps* 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, Eppinger and Spector 85]. Camelot integrates virtual memory management with recovery and ensures that the necessary log records are written to stable storage before pages are written to their home location.

The most important component of a transaction facility not yet discussed is the one that oversees initiating, commiting, and aborting transactions. Commit algorithms vary in their efficiency and robustness [Lindsay et al. 79, Dwork and Skeen 83]. Camelot's algorithms are both instances of star-shaped star-shaped commit protocols, in which the initiator of the outermost top-level transaction serves as coordinator for all nodes in the transaction. One algorithm is based on the standard 2-phase commit protocol. It is efficient, but has failure modes in which nodes participating in a distributed transaction must restrict access to some data until other nodes recover from a crash or a network partition is eliminated. Camelot also provides a hybrid protocol, which is a cross between 3-phase and byzantine commit protocols, to reduce the likelihood that access to data will be blocked.

The increased interest in building nested abstractions using transactions has led to the investigation and implementation of facilities for supporting nesting. These facilities limit the concurrency anomalies that can occur within a single transaction that has multiple threads of control and permit portions of a transaction to abort independently. Camelot supports nested transactions as in Argus.

4.1.4. Summary of Implementation Background

The major points of this development can be tersely summarized: Camelot 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. Inter-node communication uses both sessions and datagrams. Inter-transaction synchronization is done via locking or hybrid atomicity, with time-outs used to resolve deadlock (in Release 1). Write-ahead logging is the basis of recovery and transaction commit is done via either a blocking or non-blocking commit protocol. Camelot supports the Argus nested transaction model.

4.2. Camelot Functions

As mentioned, the most basic building blocks for reliable distributed applications are provided by Mach, its communication facilities, and the Matchmaker RPC stub generator [Cooper 86, Jones et al. 85]. These building blocks include processes, threads of control within processes, shared memory between processes, and message passing.

Camelot provides functions for system configuration, recovery, disk management, transaction management, deadlock detection, and reliability/performance evaluation. Simple lock-based synchronization mechanisms are provided in the Camelot libraries. (Avalon's runtime support is required to use Hybrid Atomicity.) These functions are specified in the Camelot Interface Specification and Guide to the Camelot System [Spector et al 86, CMU 87]. Certain more advanced functions will be added to Camelot for Release 2.

4.2.1. Configuration Management

Camelot supports the dynamic allocation and deallocation of both new data servers and the recoverable storage in which data servers store long-lived objects. Camelot maintains configuration data so that it can restart the appropriate data servers after a crash and reattach them to their recoverable storage. These configuration data are stored in recoverable storage and updated transactionally.

4.2.2. Disk Management

Camelot provides data servers with up to 2^{48} bytes of recoverable storage. With the cooperation of Mach, Camelot permits data servers to map that storage into their address space, though data servers must call Camelot to remap their address space when they overflow 32-bit addresses. To simplify the allocation of contiguous regions of disk space, Camelot assumes that all allocation and deallocation requests are coarse (e.g., in megabytes). Data servers are responsible for doing their own microscopic storage management.

So that operations on data in recoverable storage can be undone or redone after failures, Camelot provides data servers with logging services for recording modifications to objects. Camelot automatically coordinates paging of recoverable storage to maintain the write-ahead log invariant [Eppinger and Spector 85].

4.2.3. Recovery Management

Camelot's recovery functions include transaction abort, and server, node, and media-failure recovery. To support these functions, Camelot Release 1 provides the two forms of write-ahead value logging mentioned above.

Camelot writes log data to locally duplexed storage or to storage that is replicated on a collection of dedicated network log servers [Daniels et al. 86]. In some environments, the use of a shared network logging facility could have survivability, operational, performance, and cost advantages. Survivability is likely to be better for a replicated logging facility because it can tolerate the destruction of one or more entire processing nodes. Operational advantages accrue because it is easier to manage high volumes of log data at a small number of logging nodes, rather than at all transaction processing nodes. Performance might be better because shared facilities can have faster hardware than could be afforded

for each processing node. Finally, providing a shared network logging facility would be less costly than dedicating duplexed disks to each processing node, particularly in workstation environments.

4.2.4. Transaction Management

Camelot provides facilities for beginning new top-level and nested transactions and for committing and aborting them. Two options exist for commit: *Blocking* commit may result in data that remains locked until a coordinator is restarted or a network is repaired. *Non-blocking* commit, though more expensive in the normal case, reduces the likelihood that a node's data will remain locked until another node or network partition is repaired. In addition to these standard transaction management functions, Camelot provides an inquiry facility for determining the status of a transaction. The Camelot library and Avalon use this to support lock inheritance.

4.2.5. The Camelot Library

The Camelot library comprises routines and macros that allow a user to implement data servers and applications.[2] For servers, it provides a common message handling framework and standard processing functions for system messages. Thus, the task of writing a server is reduced to writing procedures for the operations supported by the server.

The library provides several categories of support routines to facilitate the task of writing these procedures. Transaction control routines provide the ability to initiate and abort nested and top level transactions. Data manipulation routines permit the creation and modification of static recoverable objects. Locking routines maintain the serializability of transactions. (Lock inheritance among families of subtransactions is handled automatically.) Critical sections control concurrent access to local objects. A macro facilitates remote procedure calls to other servers.

Applications use a subset of the library facilities. In particular, they use the transaction control routines and server access macro.

4.2.6. Deadlock Detection

Clients of Camelot Release 1 must depend on time-out to detect deadlocks. Release 2 will incorporate a deadlock detector and export interfaces for servers to report their local knowledge of wait-for graphs. We anticipate that implementing deadlock detection for arbitrary abstract types in a large network environment like the Arpanet will be difficult.

---

[2]The functions of the Camelot library are subsumed by Avalon's more ambitious linguistic support.

Figure 4-3 shows the seven processes in Release 1 of Camelot: master control, disk manager, communication manager, recovery manager, transaction manager, node server, and node configuration application.[3]

- **Master Control.** This process restarts Camelot after a node failure.

- **Disk Manager.** The disk manager allocates and deallocates recoverable storage, accepts and writes log records locally, and enforces the write-ahead log invariant. For log records that are to be written to the distributed logging service, the disk manager works with dedicated servers on the network. Additionally, the disk manager writes pages to/from the disk when Mach needs to service page faults on recoverable storage or to clean primary memory. Finally, it performs checkpoints to limit the amount of work during recovery and works closely with the recovery manager when failures are being processed.

- **Communication Manager.** The communication manager forwards inter-node Mach messages, and provides the logical and physical clock services. In addition, it knows the format of messages and keeps a list of all the nodes that are involved in a particular transaction. This information is provided to the transaction manager for use during commit or abort processing. Finally, the communication manager provides a name service that creates communication channels to named servers. (The transaction manager and distributed logging service use IP datagrams, thereby bypassing the communication manager.)

- **Recovery Manager.** The recovery manager is responsible for transaction abort, server recovery, node recovery, and media-failure recovery. Server and node recovery respectively require one and two backward passes over the log.

- **Transaction Manager.** The transaction manager coordinates the initiation, commit, and abort of local and distributed transactions. It fully supports nested transactions.

- **Node Server.** The node server is the repository of configuration data necessary for restarting the node. It stores its data in recoverable storage and is recovered before other servers.

- **Node Configuration Application.** The node configuration application permits Camelot's human users to update data in the node server and to crash and restart servers.

The organization of Camelot is similar to that of TABS and R*. Structurally, Camelot differs from TABS in the use of threads, shared memory interfaces, and the combination of logging and disk management in the same process. Many low-level algorithms and protocols have also been changed to improve performance and provide added functions. Camelot differs from R* in its greater use of message passing and support for common recovery facilities for servers. Of course, the functions of the two systems are quite different; transactions in R* are intended primarily to support a particular relational database system.

---

[3]Camelot Release 2 will use additional processes to support deadlock detection and reliability and performance evaluation.

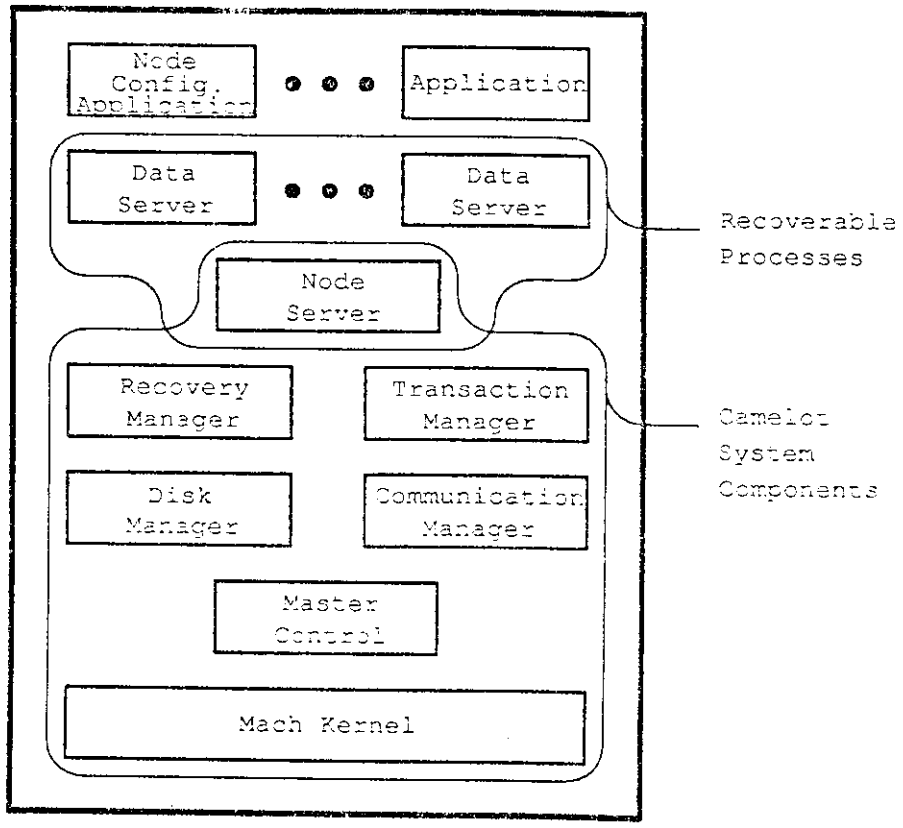

Figure 4-3: Processes in Camelot Release 1

This figure shows the Mach kernel and the processes that are needed to execute distributed transactions. The node server is both a part of Camelot, and a Camelot data server because it is the repository of essential configuration data. Other data servers and applications use the facilities of Camelot and Mach. The node configuration application permits users to exercise control over a node's configuration.

4.4. Discussion

As of January 1987, Camelot Release 1 was still being coded though enough (about 25,00 lines of C) was functioning to commit and abort local transactions. Though many pieces were still missing (e.g., support for stable storage and distribution), Avalon developers could begin their implementation work. Before we add to the basic set of Camelot Release 1 functions, others will be encouraged to port abstractions to Camelot and provide feedback on its functionality and performance.

Performance is a very important system goal. Experience with TABS and very preliminary performance numbers make us believe that we will be able to execute roughly 20 non-paging write

transactions/second on an RT PC or MicroVax workstation. Perhaps, it is worthwhile to summarize why the Camelot/Mach combination should have performance that even database implementors will like:

- Mach's support for multiple threads of control per process permit efficient server organizations and the use of multiprocessors.

- Disk I/O should be efficient, because Camelot allocates recoverable storage contiguously on disk, and because Mach permits it to be mapped into a server's memory. Also, servers that know disk I/O patterns, such as database managers, can influence the page replacement algorithms by providing hints for prefetching or prewriting.

- Recovery adds little overhead to normal processing because Camelot uses write-ahead logging with a common log. Though Camelot Release 1 has only value-logging, operation-logging will be provided in Release 2.

- Camelot has an efficient, datagram-based, two-phase commit protocol in addition to its non-blocking commit protocol. Even without delaying commits to reduce log forces ("group commit"), transactions require only one log force per node per transaction. Camelot requires just three datagrams per node per transaction in its star-shaped commit protocol, because final acknowledgments are piggy-backed on future communication. Camelot also has the usual optimizations for read-only transactions.

- Camelot does not implement the synchronization needed to preserve serializability. This synchronization is left to servers (and/or Avalon), which can apply semantic knowledge to provide higher concurrency or to reduce locking overhead.

5. Challenges

Many commercial transaction processing applications already use distributed transactions. Many more algorithms and applications will benefit from them when general-purpose, high performance transaction facilities are available. For example, there have been promising applications built on both TABS and Argus that could be very useful in production environments. Also, there are a plethora of unimplemented distributed replication techniques that depend upon transactions to maintain invariants on the underlying replicas.

The challenges lie in constructing facilities that have high performance, yet are easy to use and operate within familiar computing environments. The performance challenge is clear due to the obvious complexity of implementing commit protocols, stable storage, media recovery, etc. There are also concurrency control bottlenecks that could become problematical. However, there are many good algorithms to use, and the structure of distributed transaction processing seems sufficiently well understood to permit clean implementations.

Ease of use and operational challenges are equally important. For many applications, a distributed transaction facility must run on all the nodes of a large distributed system. Thus, it should be installed

along with the operating system and require minimal, if any, operator intervention. Taking dumps, managing log space, reconfiguring nodes, and adding and removing servers should be nearly automatic or at least easy to do. Programmers should also find it straightforward to use the facility. The Argus and Avalon languages, or carefully defined library support such as that in the Camelot library can substantially reduce programming complexity, but it remains to be seen how successful these efforts will be for large systems.

Camelot is intended to meet many of these challenges and to help demonstrate that transaction facilities can be sufficiently efficient and easy to use for a wide range of distributed programs. However, there are additional challenges not addressed by Camelot.

Support for heterogeneous networks and operating systems would add much to the utility of transaction processing. Even with all the standardization efforts that are underway, it would still desirable to support distributed transactions running on multiple types of systems and networks: for example, it would be useful if Camelot/Mach workstations could participate in transactions with existing data on 370/MVS mainframes. Technically, this is possible, but it is difficult to perform the needed protocol translations efficiently.

Perhaps the most important challenge is to construct the transactional abstract objects that are needed to make distributed transaction facilities really useful: mail systems, specialized databases, file systems, window managers, and queuing systems. While the availability of transactions make these objects easier to implement, they are still complex — particularly if they require replication. Reconciling the transactional semantics of new objects with the non-transactional objects that they replace is also difficult. (This could be a particularly tough problem with the Unix file system.)

In spite of these challenges, distributed transaction facilities should become more prevalent. Ongoing work in the research and commercial spheres, aided by ever-faster hardware will continue to improve their performance and usability.

Acknowledgments

I thank Jeff Eppinger, who thoroughly read and critiqued this paper, and my colleagues on the Camelot and Avalon Projects for their contributions to the systems that I have described.

References

[Accetta et al. 86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer Usenix*. July, 1986.

[Allchin 83] James E. Allchin. *An Architecture for Reliable Distributed Systems*. PhD thesis, Georgia Institute of Technology, September, 1983.

[Allchin and McKendry 83] James E. Allchin, Martin S. McKendry. *Facilities for Supporting Atomicity in Operating Systems*. Technical Report GIT-CS-83/1, Georgia Institute of Technology, January, 1983.

[Astrahan et al. 76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: A Relational Approach to Database Management. *ACM Transactions on Database Systems* 1(2), June, 1976.

[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.

[Bloch 86] Joshua J. Bloch. A Practical, Efficient Approach to Replication of Abstract Data Objects. November, 1986. Carnegie Mellon Thesis Proposal.

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

[CMU 87] *The Guide to the Camelot Distributed Transaction Facility: Release 1* Pittsburgh, PA, 1987. Under development.

[Cooper 86] Eric C. Cooper. C Threads. June, 1986. Carnegie Mellon Internal Memo.

[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 et al. 86] Dean S. Daniels, Alfred Z. Spector, Dean Thompson. *Distributed Logging for Transaction Processing*. Technical Report CMU-CS-86-106, Carnegie-Mellon University, June, 1986.

[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, December, 1985.

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

[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 81] James N. Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the Very Large Database Conference*, pages 144-154. September, 1981.

[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.

[Helland 85] Pat Helland. Transaction Monitoring Facility. *Database Engineering* 8(2):9-18, June, 1985.

[Herlihy 85] Maurice P. Herlihy. *Availability vs. atomicity: concurrency control for replicated data.* Technical Report CMU-CS-85-108, Carnegie-Mellon University, February, 1985.

[Herlihy and Wing 86] M. P. Herlihy, J. M. Wing. *Avalon: Language Support for Reliable Distributed Systems.* Technical Report CMU-CS-86-167, Carnegie Mellon University, November, 1986.

[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.

[Lamport 78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 21(7):558-565, July, 1978.

[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.

[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 83] Barbara H. Liskov, Robert W. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems* 5(3):381-404, July, 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.

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

[Postel 82] Jonathan B. Postel. Internetwork Protocol Approaches. In Paul E. Green, Jr. (editor), *Computer Network Architectures and Protocols*, chapter 18, pages 511-526.Plenum Press, 1982.

[Rashid 86] Richard F. Rashid. Threads of a New System. *Unix Review* 4(8):37-49, August, 1986.

[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.

[Rothnie et al. 80] J. B. Rothnie Jr., P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. Reeve, D. W. Shipman, and E. Wong. Introduction to a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems* 5(1):1-17, March, 1980.

[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 86] Alfred Z. Spector. Communication Support in Operating Systems for Distributed Transactions. In *Proc. IBM European Networking Institute 1986.* July, 1986. Also available as Technical Report CMU-CS-86-165, Carnegie-Mellon University, November 1986.

[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 86] Alfred Z. Spector, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, Dean S. Thompson. The Camelot Interface Specification. September, 1986. Camelot Working Memo 2.

[Spector et al. 85a] 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.

[Spector et al. 85b] Alfred Z. Spector, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Randy Pausch. Distributed Transactions for Reliable Systems. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 127-146. ACM, December, 1985. Also available in *Concurrency Control and Reliability in Distributed Systems*, Van Nostrand Reinhold Company, New York, and as Technical Report CMU-CS-85-117, Carnegie-Mellon University, September 1985.

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

[Stonebraker and Neuhold 77] M. Stonebraker and E. Neuhold. A Distributed Data Base Version of INGRES. In *Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 19-36. Lawrence Berkeley Lab, University of California, Berkeley, California, May, 1977.

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

[Thompson 86] Dean Thompson. Coding Standards for Camelot. June, 1986. Camelot Working Memo 1.

[Traiger 82] Irving L. Traiger. Virtual Memory Management for Database Systems. *Operating Systems Review* 16(4):26-48, October, 1982. Also available as 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 84] William E. Weihl. *Specification and Implementation of Atomic Data Types*. PhD thesis, Massachusetts Institute of Technology, March, 1984.

[Weinstein et al. 85] Matthew J. Weinstein, Thomas W. Page, Jr., Brian K. Livezey, Gerald J. Popek. Transactions and Synchronization in a Distributed Operating System. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 115-126. ACM, December, 1985.

[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.

[Zimmermann 82] Hubert Zimmermann. A Standard Network Model. In Paul E. Green, Jr. (editor), *Computer Network Architectures and Protocols*, chapter 2, pages 33-54.Plenum Press, 1982.