CMU-CS-86-166

The Camelot Project

Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels,
Richard P. Draves, Dan Duchamp, Jeffrey L. Eppinger,
Sherri G. Menees, Dean S. Thompson

November 1986

Abstract

Camelot provides flexible and high performance transaction management, disk management, and recovery mechanisms that are useful for implementing a wide class of abstract data types, including large databases. To ensure that Camelot is accessible outside of the Carnegie Mellon environment, Camelot runs on the Unix-compatible Mach operating system and uses the standard Arpanet IP communication protocol. Camelot is being coded on RT PC's, is being frequently tested on MicroVaxes, and it will also run on various shared-memory multiprocessors. This paper describes Camelot's functions and internal structure.

Technical Report CMU-CS-86-166

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 appears in *Database Engineering* 9(4), December 1986.

1. Introduction

Distributed transactions are an important technique for simplifying the construction of reliable and available distributed applications. The failure atomicity, permanence, and serializability properties provided by transactions lessen the attention a programmer must pay to concurrency and failures [Gray 80, Spector and Schwarz 83]. Overall, transactions make it easier to maintain the consistency of distributed objects.

Many commercial transaction processing applications already use distributed transactions, for example, on Tandem's TMF [Helland 85]. We believe there are many more algorithms and applications that will benefit from transactions as soon as there is a widespread, general-purpose, and high performance transaction facility to support them. For example, there are a plethora of unimplemented distributed replication techniques that depend upon transactions to maintain invariants on the underlying replicas.

A few projects have developed systems that support distributed transaction processing on abstract objects. Argus, Clouds, and TABS [Liskov and Scheifler 83, Allchin and McKendry 83, Spector et al. 85, Spector 85] are a few examples. These systems permit users to define new objects and to use them together within transactions. While the interfaces, functions, and implementation techniques of Argus, Clouds, and TABS are quite different, the projects' goals have been the same: to provide a common transactional basis for many abstractions with the ultimate goal of simplifying the construction of reliable distributed applications.

Building on the experience of these and other projects, we have designed and are now implementing an improved distributed transaction facility, called Camelot (Carnegie Mellon Low Overhead Transaction Facility). 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 system encapsulate objects within *data server* processes, which execute operations in response to remote procedure calls. Other attributes of Camelot include the following:

- **Compatibility with standard operating systems.** Camelot runs on Mach, a Berkeley 4.3 Unix^{TM}-compatible operating system [Accetta et al. 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 will facilitate large distributed processing experiments.

Machine-independent implementation. Camelot is intended to run on all the uniprocessors and multiprocessors that Mach will support. We develop Camelot on IBM RT PC's, but we frequently test it on DEC MicroVaxes and anticipate running it on multiprocessors such as the Encore and Sequent machines.

- **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 a scheme for supporting recoverable objects that are accessed in virtual memory. (Section 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 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 user manual based on the specification will be written.

To reduce further the amount of effort required to construct reliable distributed systems, a companion project is developing a set of language facilities, called Avalon, which provide linguistic support for reliable applications [Herlihy and Wing 86]. Avalon encompasses extensions to C++, Common Lisp, and ADA and automatically generates necessary calls on Camelot. Figure 1-1 shows the relationship of Camelot to Avalon and Mach.

One goal of the Camelot Project is certainly the development of Camelot; that is, a system of sufficient quality, performance, and generality to support not only our own, but others' development of reliable distributed applications. In building Camelot, we hope to demonstrate conclusively that general purpose transaction facilities are efficient enough to be useful in many domains. However, we are also developing new algorithms and techniques that may be useful outside of Camelot. These include an enhanced non-blocking commit protocol, a replicated logging service, and a facility for testing distributed applications. We also expect to learn much from evaluating Camelot's performance, particularly with respect to the performance speed-up on multiprocessors.

2. Camelot Functions

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

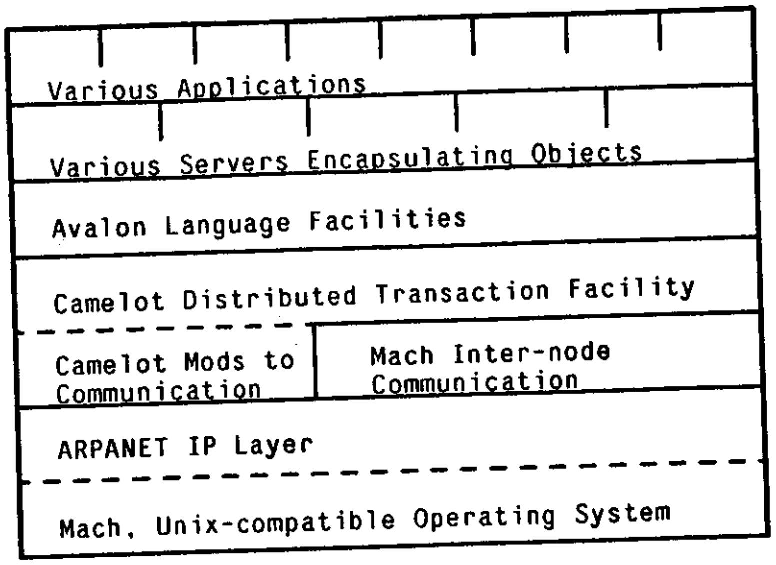
Various Applications
Various Servers Encapsulating Objects
Avalon Language Facilities
Camelot Distributed Transaction Facility
Camelot Mods to Communication
Mach Inter-node Communication
ARPANET IP Layer
Mach, Unix-compatible Operating System

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

Mach executes on uniprocessor and multiprocessor hardware. Inter-node communication is logically layered on top of Mach. Camelot provides support for transaction processing, including certain additions to the communication layer. Avalon provides linguistic support for accessing Camelot and Mach. Users define servers encapsulating objects and applications that use those objects. Examples of servers are mail repositories, distributed file system components, and database managers.

shared memory between processes, and message passing.

Camelot provides functions for system configuration, recovery, disk management, transaction management, deadlock detection, and reliability/performance evaluation[1]. Most of these functions are specified in the Camelot Interface Specification and are part of Camelot Release 1. Certain more advanced functions will be added to Camelot for Release 2.

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.

[1]Synchronization mechanisms for preserving serializability are distributed among data servers; Camelot supports servers that perform either locking or hybrid atomicity [Herlihy 85]. This synchronization is commonly implemented with the assistance of Avalon's runtime support.

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 space 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].

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 two forms of write-ahead value logging; one form in which only new values are written to the log, and a second form in which both old values and new values are written. New value logging requires less log space, but results in increased paging for long running transactions. This is because pages can not be written back to their home location until a transaction commits. Camelot assumes that the invoker of a top-level transaction knows the approximate length of his transaction and specifies the type of logging accordingly.

Camelot's two logging protocols are based on the old value/new value recovery technique used in TABS [Spector 85] and described by 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.

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.

Release 2 of Camelot will support an operation (or transition) logging technique in which type implementors can log non-idempotent undo and redo operations. This type of logging increases the feasible concurrency for some types and reduces the amount of log space that they require.

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. Data servers and Avalon need this to support lock inheritance.

2.5. Support for Data Servers

The Camelot library packages all system interfaces and provides a simple locking mechanism. It also contains routines that perform the generic processing required of all data servers. This processing includes participating in two-phase commit, handling undo and redo requests generated after failures, responding to abort exceptions, and the like. The functions of this library are subsumed by Avalon's more ambitious linguistic support.

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.

2.7. Reliability and Performance Evaluation

Camelot Release 2 will contain a facility for capturing performance data, generating and distributing workloads, and inserting (simulated) faults. These capabilities will help us analyze, tune, and validate Camelot and benefit Camelot's clients as they analyze their distributed algorithms. The information returned by the facility could also be used to provide feedback for applications that dynamically tune themselves. We believe that, when properly designed, a reliability and performance evaluation facility will prove as essential for building large distributed applications as source-level debuggers are essential for traditional programming.

The reliability and performance evaluation facility has three parts. The first captures performance data and permits clients to gauge critical performance metrics, such as the number of messages, page faults, deadlocks, and transactions/second. Certain information is application-independent, but other useful information depends on the nature of the application. Therefore, the performance evaluation facility will be extensible and capture application-specific data from higher level components. Once information is obtained from various nodes on the system, the facility combines and presents it to system implementors or feeds it back to applications for use in dynamic tuning.

The second part of the performance and reliability evaluation facility permits the distribution of applications (or workloads) on the system. When many nodes are involved in a workload, this task can be very difficult unless it is possible to specify the nodes and workloads from a single node. We have built a prototype facility of this type for TABS, and we will extend it for use on Camelot.

The third part permits simulated faults to be inserted according to a pre-specified distribution. This is crucial for understanding the behavior of a system in the presence of faults. For example the low-level communication software may be instructed to lose or reorder datagrams with a pre-specified probability. Or, a pair of nodes could greatly raise network utilization to probe the effects of contention.

2.8. Miscellaneous Functions

Camelot provides both a logical clock [Lamport 78] and a synchronized real-time clock. These clocks are useful, for example, to support hybrid atomicity [Herlihy 85] and replication using optimistic timestamps [Bloch 86]. Camelot also extends the Mach naming service to support multiple servers with the same name. This is useful to support replicated objects.

3. Camelot Implementation

The major functions of Camelot and their logical relationship is illustrated in Figure 3-1. Disk management and recovery management are at the base of Camelot's functions. Both activities are local to a particular node, except that recovery may require communication with the network logging service. Deadlock detection and transaction management are distributed activities that assume underlying disk management and node recovery facilities. Communication protocols and reliability and performance evaluation are implemented within many levels of the system. The library support for data servers rests on top of these functions.

All of Camelot except the library routines is implemented by a collection of Mach processes, which run on every node. Each of these processes is responsible for supporting a particular collection of functions. Processes use threads of control internally to permit parallelism.

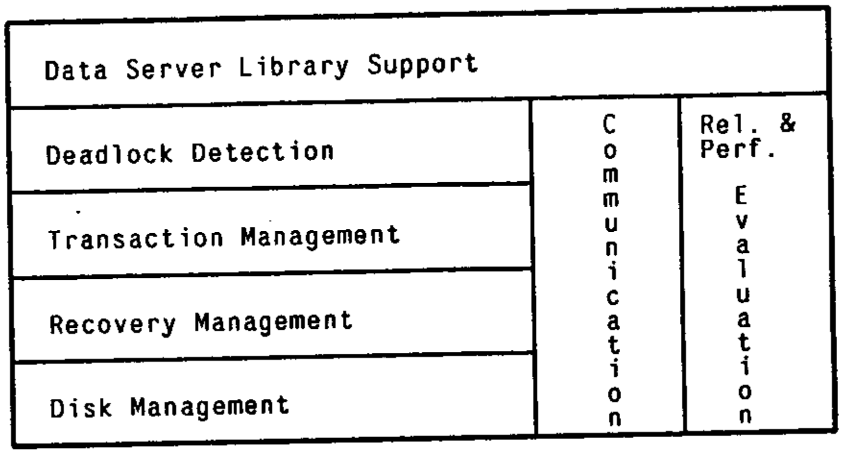

Figure 3-1: Logical Components of Camelot
This figure describes the logical structure of Camelot. Camelot is logically hierarchical, except that communication and reliability and performance evaluation functions span multiple levels.

Thus, calls to Camelot (e.g., to begin or commit a transaction), must be directed to a particular Camelot process. Some frequently called functions such as log writes are invoked by writing to memory queues that are shared between a data server and a Camelot process. Other functions are invoked using messages that are generated by Matchmaker.

Figure 3-2 shows the seven processes in Release 1 of Camelot[2]: master control, disk manager, communication manager, recovery manager, transaction manager, and node server, and node configuration application.

- **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.

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

- **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.

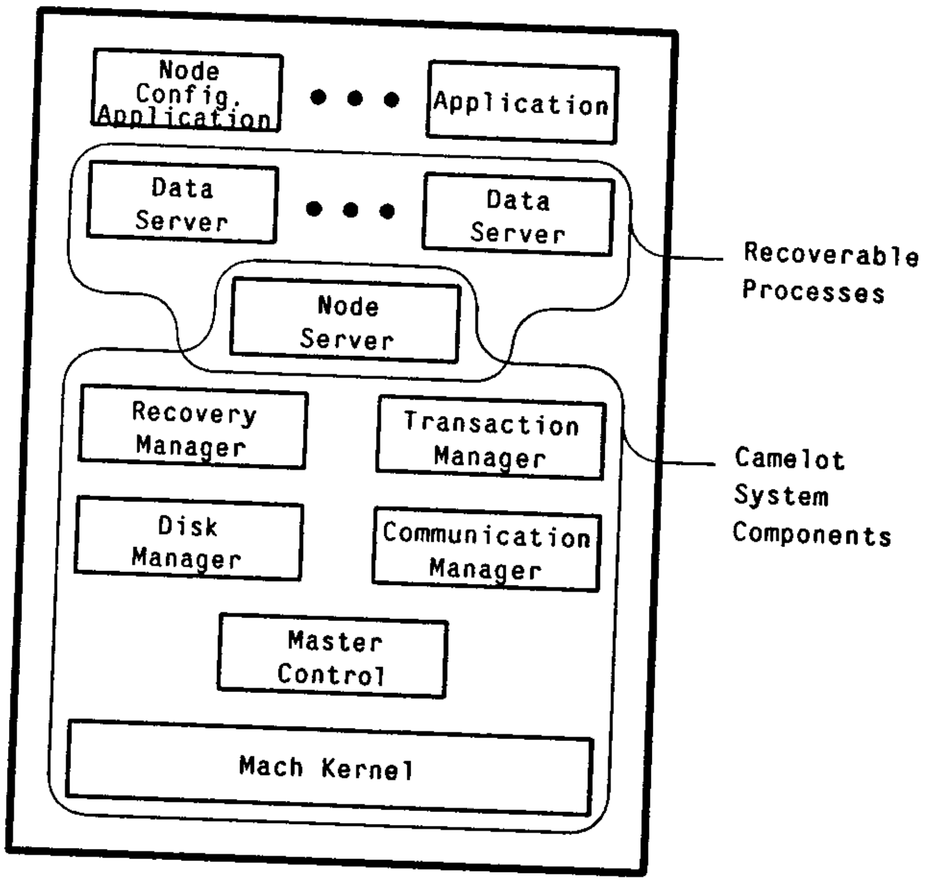

Figure 3-2: Processes in Camelot Release 1
This figure shows the Mach kernel and the processes that are needed to execute distributed transactions. The node server is both a part of Camelot, and a Camelot data server because it is the repository of essential configuration data. Other data servers and applications use the facilities of Camelot and Mach. The node configuration application permits users to exercise control over a node's configuration.

- **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* [Spector 85, Lindsay et al. 84]. 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; R*'s transactions are intended primarily to support a particular relational database system.

4. Discussion

As of December 1986, Camelot 1 was still being coded though enough (about 20,000 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 begin adding to the basic set of Camelot 1 functions, we will encourage others to port abstractions to Camelot, so that we can get 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. Shared memory between processes permits efficient inter-process synchronization.
- 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.

Today, we would guess that Camelot's initial bottlenecks will be low-level disk code and the remaining message passing. For example, though the frequent calls by servers to Camelot are asynchronous and via shared memory, all operations on servers are invoked via message using the RPC stub generator. To further reduce message passing overhead, we might have to substitute a form of protected procedure call. This should not change Camelot very much since all inter-process communication is already expressed with procedure call syntax.

In the course of our implementation and the subsequent performance evaluation, we expect to learn much about large reliable distributed systems. Once Camelot is functioning, we plan to perform extensive experimentation on multiprocessors and distributed systems with a large number of nodes. In particular, we will measure the actual availability and performance of various replication techniques.

Our overall goal remains to demonstrate that transaction facilities can be sufficiently general and efficient to support a wide range of distributed programs. We are getting closer to achieving this goal, but much work remains.

Acknowledgments

Thanks to our colleagues Maurice Herlihy and Jeannette Wing for their advice, particularly as it relates to support for Avalon.

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 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.

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

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

[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.

[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.

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

[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.

[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.

[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 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.

[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.

[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.

[Spector 85] Alfred Z. Spector. The TABS Project. *Database Engineering* 8(2):19-25, June, 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 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. 85] 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.

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