CUCS-456-89

Analysis of Transaction Management Performance

Dan Duchamp

Computer Science Department
Columbia University
New York, NY 10027

Abstract

There is currently much interest in incorporating transactions into both operating systems and general-purpose programming languages. This paper provides a detailed examination of the design and performance of the transaction manager of the Camelot system. Camelot is a transaction facility that provides a rich model of transactions intended to support a wide variety of general-purpose applications. The transaction manager's principal function is to execute the protocols that ensure atomicity.

The conclusions of this study are: a simple optimization to two-phase commit reduces logging activity of distributed transactions; non-blocking commit is practical for some applications; multithreaded design improves throughput provided that log batching is used; multicasting reduces the variance of distributed commit protocols in a LAN environment; and the performance of transaction mechanisms such as Camelot depend heavily upon kernel performance.

1 Introduction

The semantics of a transaction — atomic, serializable, permanent — suggest that it should be a good programming construct for fault-tolerant (distributed) programs that operate on long-lived data. This thesis has recently been explored by incorporating transactions into both operating systems [25][16] and general-purpose programming languages [21][29]. This paper examines the design and performance of the transaction manager (TranMan) of the Camelot system [2], which is a service, usable by other services at any level of abstraction, that provides transactions as a technique for synchronization, recovery, and fault-tolerance.

The transaction manager is responsible for implementing the most basic calls of the transaction interface (begin-transaction, commit-transaction, and abort-transaction), so its performance is an issue central to the use of transactions as a programming tool.

In Camelot, transactions can be arbitrarily nested and distributed. This permits programs to be written more naturally, but makes several aspects of transaction management more difficult. The demands that influence the design of the transaction manager are:

- Function: transactions will be used in widely different ways, and the transaction manager should have mechanisms to cope with this. Two specific mechanisms incorporated into the Camelot transaction manager are nested transactions and non-blocking commitment.
- Performance: transactions should be cheap enough to be used as if they were an operating system primitive. Even short transactions must experience little overhead, and the transaction manager must cope with high traffic. Multicast is used to reduce the overhead of committing large distributed transactions.

This work examines some notable aspects of the Camelot transaction manager, most of which are in some way performance-oriented. These aspects and the conclusions about their performance are:

1. An optimization to the well-known two-phase commit protocol is quite effective in decreasing logging activity of distributed transactions.
2. A non-blocking commit protocol (one that can survive any single failure, including partition) although inherently slower than two-phase commit, is practical for some applications.
3. A multithreaded transaction manager improves throughput provided that log batching exists.
4. Multicast reduces the variance of distributed protocols in an extended-LAN environment.
5. Camelot is operating-system-intensive, and so is heavily dependent upon kernel performance.

2 Camelot Overview

Camelot runs on the Mach operating system [1], which is a communication-oriented extension of Berkeley UNIX 4.3. Extensions include the ability to send typed messages to ports of local processes, read-write sharing of arbitrary regions of memory (among processes that are related as ancestor and descendant), provision for multi-threaded processes, and an **external pager interface** that allows an extra-kernel process to provide virtual memory management for arbitrary regions of memory. Additionally, the Mach implementation is kernelized and has been reorganized (i.e., with the addition of locking) to permit it to run on multiprocessors. Camelot makes extensive use of all of the new Mach functions. Message passing is used for most inter-process communication, shared memory for the rest. Threads are used for parallelism within all processes. The external pager facility provides Camelot with the control over pageout needed to implement the write-ahead log protocol.

To use Camelot, someone who possesses a database that he wishes to make publicly available writes a **data server** process that controls the database and allows access to client **application** processes. Any computer on which a data server runs must also run a single instance of each of four processes that comprise the implementation of Camelot. An application initiates a transaction by getting a transaction identifier from the transaction manager and then performs data manipulation operations by making synchronous inter-process procedure calls to any number of data servers, local or remote. Every operation must explicitly list the transaction identifier as one of its arguments. While processing a request, a data server may in turn call other data servers. Eventually, the application orders the transaction manager to either commit or abort.

Each data server "manages" one or more objects which are instances of some abstract data types. Object management consists of doing storage layout and concurrency control, and implementing the operations advertised in the interface. The first time a server processes an operation on behalf of a transaction, it must first ask the local transaction manager whether it may **join** the transaction. Joining allows a transaction manager to keep track of which local servers are participating in the transaction. A server must serialize access to its data by locking.

While servers are responsible for ensuring serializability, atomicity and permanence are implemented by Camelot using a common stable-storage log. Besides the transaction management process, the other portions of Camelot are:

- **Disk Manager Process.** The disk manager is a virtual-memory buffer manager that protects the disk copy of servers' data segments by cooperating with servers and with Mach (via the external pager interface) to implement the write-ahead log protocol. Also, it is the only process that can write into the log.
- **Communication Manager Process (or ComMan).** The communication manager has two functions. First, it forwards inter-site messages from applications to servers and back again. While doing so it spies on the contents, keeping track of which transactions are traveling to which sites. This information is needed by the transaction manager for executing the commit and abort protocols. Second, it acts as a name service.
- **Recovery Process.** After a failure (of server, site, or disk) or an abort, the recovery process reads the log and instructs servers how to undo or redo updates of interrupted transactions.
- **Runtime Library.** A library of routines exists to aid those programming servers or applications. Among other things, routines exist to facilitate multi-thread concurrent execution, to implement shared/exclusive mode locking, and to change control flow in the event of abort.

A complete illustration of the control flow of a simple transaction is given in Figure 1.

3 Transaction Manager Overview

The transaction manager is essentially a protocol processor; most calls from applications or servers invoke one protocol or another. As explained in Section 3.1, hooks in the inter-site communication mechanism allow the TranMan to know which other sites any particular transaction has spread to, a necessary condition for executing the distributed protocols. These protocols include the two varieties of distributed commitment protocols described in Sections 3.2 and 3.3, as well as several others [10].

3.1 Inter-site Communication

Mach allows messages to be sent only between two threads on a single site. Therefore, a forwarding agent is needed to pass a message from one thread at one site to a second at another site. The Mach network message server ("NetMsgServer") is such a forwarding agent, and a name service as well. A client wishing to locate some service presents the NetMsgServer with a string naming the desired service and gets a port in return. The port is one endpoint of a reliable connection made between a client process and a server process. The client then invokes RPCs along this connection.

The communication manager is used by Camelot applications and data servers just as a non-Camelot program uses the NetMsgServer. The ComMan in turn uses the NetMsgServer to provide the same services with the same interface, but it also includes special provisions for transaction management. The presence of the communication manager changes the RPC path from

client-NetMsgServer-network-NetMsgServer-server

to

client-ComMan-NetMsgServer-network-NetMsgServer-ComMan-server.

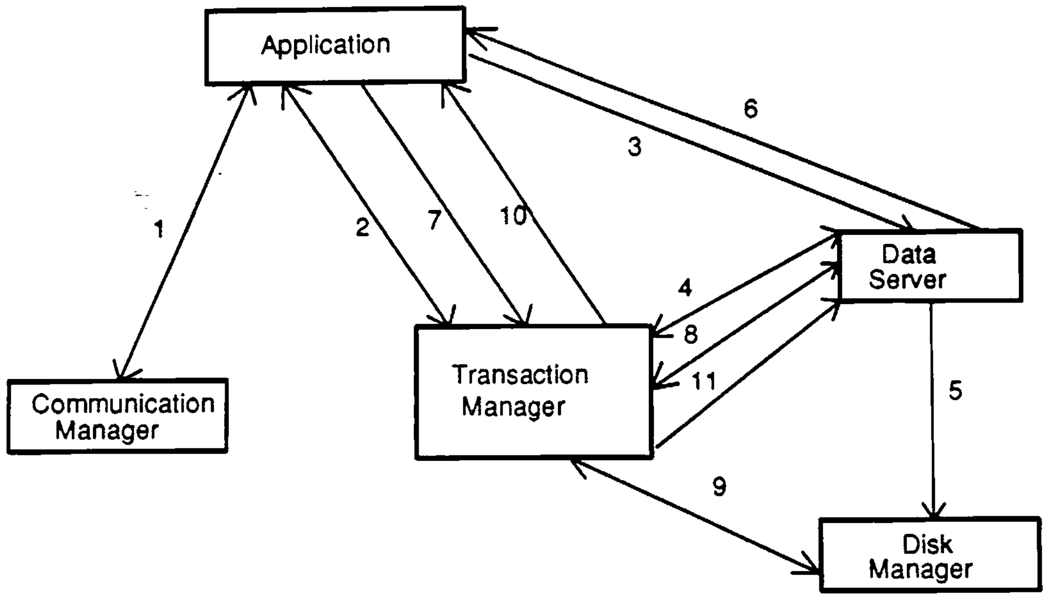

Figure 1: Execution of a Transaction
Lines with arrows at both ends are synchronous calls. The events of the transaction are:

1. Application uses the ComMan as a name server, getting a port to the data server.
2. Application begins a transaction by getting a transaction identifier from TranMan.
3. Application sends a message requesting service.
4. Server notifies TranMan that it is taking part in the transaction.
5. Server sets the appropriate lock(s), pins the required pages in memory, and does the update. Before replying, it reports both the old and new value of the object to the disk manager. This record is logged as late as possible. In the best (and typical) case, only one log write is needed to commit the transaction.
6. Server completes the operation and replies to the Application.
7. Application tells the transaction manager to try to commit the transaction.
8. TranMan asks the Server whether it is willing to commit. The Server says that it is.
9. TranMan writes a record into the log, indicating that the transaction is committed.
10. TranMan responds to the Application, saying that the transaction is committed.
11. TranMan tells the Server to drop the locks held by the transaction.

Messages containing transaction identifiers are specially marked, and the communication manager is aware of their format. When a response message leaves one site, the communication manager at the sending site intercepts the message and before forwarding it adds to it the list of sites used to generate the response. This list is removed from the message by the communication manager at the destination site and merged with lists sent in previous responses. If every operation responds, the site that begins a transaction will eventually learn the identity of all other participating sites; these participants will be the subordinates during commitment. If some operation fails to respond, the site that invoked it should eventually initiate the abort protocol [7], which can operate with incomplete knowledge about which sites are involved.

3.2 Two-phase Commitment

Commitment of distributed transactions is usually done using the well-known two-phase commit protocol. Camelot's version incorporates the improvements of the Presumed Abort variation (described in [23]), and is further optimized as described in [9]. The effect of the optimization is that subordinate update sites make one fewer log force per transaction. The subordinate drops its locks *before* writing a commit record.

In the unoptimized protocol, a subordinate writes its own commit record to indicate that the transaction is committed and therefore that locks may be dropped. The optimized protocol uses the commit record *at the coordinator* to indicate the same fact. So the coordinator must not forget about the transaction before the subordinate writes its own commit record; hence, the commit acknowledgement cannot be sent until the subordinate's commit record is written.

The optimization has two advantages. First, throughput at the subordinate is improved because fewer log forces are required. The amount of improvement is dependent upon the fraction of transactions that require distributed commitment. Second, locks are retained at the subordinate for a slightly shorter time; this factor is important only if the transaction is very short. Throughput is improved at no cost to latency.

3.3 Non-blocking Commitment

The two-phase commitment protocol has one significant drawback. For a period of time all of the information needed to make the decision about whether to commit or abort is located at the coordinator and nowhere else. If a subordinate loses contact with the coordinator during this **window of vulnerability**, then it must remain prepared until the failure is repaired and communication with the coordinator is reestablished. Until then, the subordinate continues to hold its write locks for the transaction, and is said to be **blocked**.

The Camelot transaction manager incorporates a new "non-blocking" commitment protocol [8] that allows at least some sites to commit or abort in spite of any single site crash or network partition. (The type of commitment protocol to execute — two-phase versus non-blocking — is specified as an argument to the commit-transaction call.) The protocol is correct despite the occurrence of any number of failures, although all sites may block if there are two or more failures. It is impossible to do better [26]. When there is no failure, the protocol requires three phases of message exchange between the coordinator and the subordinates and requires each site to force two log records. Read-only transactions are optimized so that a read-only subordinate typically writes no log records and exchanges only one round of messages with the coordinator, just as with two-phase commit.

The protocol makes five changes to two-phase commit:

1. The prepare message contains the list of sites involved in the transaction, as well as quorum sizes needed in the "replication phase" explained in item 3 below.
2. The subordinates do not wait forever for the commit/abort notice, but instead timeout and become coordinators. The transaction can be finished by a new coordinator that was once a subordinate. Having several simultaneous coordinators is possible, but is not a problem.
3. An extra phase (called the **replication phase**) exists between the two standard phases. During this phase, the coordinator collects the information that it will use to make the commit/abort decision, and replicates it at some number of subordinates. The subordinates write the information in the log, just as they write a prepare record and a commit record. The coordinator is not allowed to make the decision until the information has been sufficiently widely replicated to exclude the other outcome. This is the well-known quorum consensus method [13]. The atomic action that marks the commitment point of the protocol is the writing of a log record that forms a commit quorum.
4. No transaction manager **forgets** (i.e., expunges its data structures) about a transaction until all sites have committed or aborted.
5. The coordinator prepares before sending the prepare message.

The first two changes are quite intuitive. The first simply gives subordinates the ability to communicate after the loss of the coordinator. The second tells how they communicate: by having a subordinate become a coordinator and tolerating the presence of multiple coordinators, there is no need to elect a new coordinator. The third change prevents a partition from causing incorrect operation by ensuring that no site can commit or abort until it is certain the other outcome is excluded. The fourth change prevents a site from joining both types of quorums for the same transaction. The last change merely increases the chances of committing during the failure.

3.4 Multi-Threading

The transaction manager is multithreaded to permit true parallelism when running on multiprocessors, and to improve throughput by permitting threads to run even when some others are performing long, synchronous operations such as forcing a log record. The approach to handling multiple threads is the following:

- Create a pool of threads when the process starts and increase the number as needed. Never destroy a thread.
- Use locks to protect critical regions that manipulate primary data structures. Avoid sharing message and log buffers by having a set for each thread.
- When executing a distributed protocol, do not "tie" any thread to any particular function or transaction. Instead, have every thread wait for any type of input, process the input, and resume waiting.

The transaction manager uses the primitives of the C-Threads package [6] for creating threads and managing their access to global data. C-Threads defines data types for threads, locks, and condition variables, and provides routines for manipulating all of these types. Threads are preemptively scheduled by Mach. Condition signaling is done by sending a message from one thread to another. Locks are purely exclusive and a thread waits for a lock by spinning. The method for indicating whether a lock is held is unsophisticated: a global integer is either 0 or 1; therefore, a thread can deadlock with itself by requesting a lock which it already holds. A second locking package, called "rw-lock," built on top of C-Threads is also useful: rw-lock provides read/write locks, and uses condition variables for waiting, resulting in considerable CPU savings if a thread must wait for a lock for an extended period. The programmer must choose whether to use a lock that spins or one that sleeps.

Camelot's current suite of applications mostly execute small non-nested transactions serially. It is rare for there to be concurrent requests for transaction management service from the same transaction or even from different nested transactions within the same nesting family. Accordingly, locking is designed to permit concurrency only among different transaction families. The principal data structure is a hash table of family descriptors, each with an attached hash table of transaction descriptors. Each family descriptor is protected by its own lock. Only uncommon combinations of operations theoretically suffer contention. These combinations are mostly cases where many parallel nested transactions are simultaneously doing the same thing: all joining, all committing, and so on. Each of these operations is fast, so in fact contention is unlikely. The method for deadlock avoidance is classic: there is a defined hierarchy of locks, and when a thread is to hold several locks simultaneously it must obtain the locks in the defined order.

3.5 Log Batching

If the log is implemented as a disk, then a transaction facility cannot do more than about 30 log writes per second. To provide throughput rates greater than 30 TPS requires writing log records that indicate the commitment of many transactions, a technique which is called **log batching** or "group commit" [12][17]. It sacrifices latency in order to increase throughput, and is essential for any system that hopes for high throughput and uses disks for the log. Camelot batches log records within the disk manager, which is the single point of access to the log.

4 Performance Evaluation

This section examines the performance of the various features explained above. The operating system was a black box: it was not possible to make measurements of actions happening inside. So some conclusions are drawn based on deduction rather than direct measurement.

Most measurements reported here were made in late 1988 using Camelot version 0.98(71) and Mach version 2.0 on one or more IBM RT PCs, model 125, a 2-MIP machine. The network was a 4Mbs IBM token ring without gateways. Table 1 lists the results of several simple benchmark programs in order to give a feel for the performance of the RT, of Mach, and of common calls of the C runtime library.

4.1 Inter-site Communication

The breakdown of Camelot RPC latency was determined as follows:

1. Measure the time used to perform 1000 RPCs (28.5 sec).
2. Divide by 1000 (yielding 28.5 ms per call).

BENCHMARK DESCRIPTION	TIME
Procedure call, 32-byte arg.	12.0 us
Data copy, bcopy()	8.4 us + 180us/KB
Kernel call, getpid()	149 us
Copy data in/out of kernel	35 us + copy time
Local IPC, 8-byte in-line	1.5 ms
Remote IPC, 8-byte in-line	19.1 ms
Context switch, swtch()	137 us
Raw disk write, 1 track	26.8 ms

Table 1: Benchmarks of PC-RT and Mach

The meaning of Unix jargon is:

- bcopy() — Library routine for fast byte copy.
- getpid() — Get the process id; fastest kernel call.
- swtch() — Invoke the scheduler.

3. Measure the time attributable to the basic Mach NetMsgServer-to-NetMsgServer RPC mechanism (19.1 ms).
4. Compute the extra time attributable to IPC between ComMan and NetMsgServer (2 * 1.5 ms = 3 ms).
5. Measure ComMan CPU (3.2 ms per call at each site). CPU time was measured at both sites using the Unix process status command.
6. Compare. Miraculously, there is no extra or missing time:

$$19.1 + 3 + 3.2 + 3.2 = 28.5$$

This test was run many times, with stable results.

The very high processing time within communication managers is due to unusually inefficient coding, and is not an unavoidable cost. However, it is clear that the Mach RPC mechanism is quite slow compared to other implementations running on similar hardware [3][28] and that interposing an extra process into the RPC path increases latency even more.

Mach's RPC time is as high as it is because:

1. Messages are not simply collections of bytes, but are typed and may contain ports and out-of-line data segments. The message must be scanned. Ports must be translated into network identifiers. Out-of-line data — passed lazily by the message system — must finally be transferred across address spaces.
2. The design philosophy to restrict the function of the kernel to simple data transport, giving rise to the NetMsgServer process.

The same design philosophy restricts the NetMsgServer to performing basic connection maintenance and port translation. Systems like Camelot that require further communication support are forced to build it into other processes.

4.2 Two-phase Commitment

For judging the latency of the normal case of a commitment protocol, two events are important: the moment at which all locks have been dropped, and the moment when the synchronous commit-transaction call returns. The **critical path** of a commitment protocol is the shortest sequence of actions that must be done sequentially before all locks are dropped and the call returns. The shortest sequence of actions before (only) the call returns is the **completion path**. In Camelot, the critical path is always longer than the completion path.

Commitment protocols are amenable to "static" (non-empirical) analysis [5][14] because serial and parallel portions are clearly separated. Assuming that identical parallel operations proceed perfectly in parallel and have constant service time, the length of the critical path is simply that of the serial portion plus the time of the slowest of each group of parallel operations. The length of either path can be evaluated approximately by adding the latencies of the major actions (or **primitives**) along the path. The evaluation is approximate because minor costs (such as CPU time spent within processes) are ignored, and because the assumptions are somewhat inaccurate. These sources of error tend to produce an underestimate of the true latency. Nonetheless, having a breakdown of the critical path into primitives is significant because it constitutes a more fundamental measure of the cost of the protocol than does a simple measured time. Furthermore, a "formula" stated in terms of primitive costs can be used to predict latency in case either the cost of the primitives or the protocol's use of them should change. There is a tension between the accuracy and the portability of a formula; a detailed latency accounting (such as [3]) is likely to be system-dependent. The primitives that dominate the latency of a commitment protocol are log forces and inter-site datagrams used to communicate among transaction managers. [1]

Throughput is another important measure of performance. The general technique for increasing throughput is **batching**: having a single execution of a primitive serve several transactions, as with group commit. Batching improves throughput, but increases latency, and so is not always desirable. Message batching (piggybacking) could be used to decrease the number of inter-TranMan messages used per commitment. Camelot batches only those messages that are not in the critical path.

Both empirical and non-empirical answers are provided for these questions:

1. What is the latency of distributed commitment?
2. What is the effect of the "read-only optimization"? (Sites that are read-only do not prepare and are omitted from the second phase.)
3. What is the effect of the delayed-commit optimization of Section 3.2?
4. What is the effect of only delaying the commit-ack message (but forcing the subordinate's commit record)? This represents a dissection of the delayed-commit optimization.
5. What are the formulas for the completion and critical paths, stated in terms of primitives, and how accurate are they?

The basic experiment consisted of executing a minimal distributed transaction on a coordinator and on 1, 2, and 3 subordinate sites. The "minimal transaction" performed one small operation at a single server at each site. A minimal transaction is used in order to more easily divide the latency of a transaction into two portions attributable to operation processing (not of interest) and transaction management. The reason for doing one operation at each site as opposed to doing no operations at all is to exercise all portions of transaction management, including that associated with the communication manager. Transaction management is defined to include all messages used by applications and data server to communicate with Camelot, and messages used among the Camelot processes. For a minimal transaction, everything but the operation call from application to server is charged to transaction management.

Each basic experiment was run four times, varying the type of operation and the implementation of the protocol. The four variations were:

1. Write operation, subordinate commit record *not* forced and commit-ack piggybacked. This is the most optimized possible protocol, as described in Section 3.2.
2. Write operation, subordinate commit record forced and commit-ack *not* piggybacked. This is a completely unoptimized implementation of two-phase commit.
3. Write operation, subordinate commit record forced and commit-ack *not* piggybacked. This protocol is intermediate between the previous two.
4. Read operation.

The first three experiments serve to establish the time to commit a minimal update transaction, and to determine the value of the optimization of Section 3.2. The fourth experiment establishes the time to commit a minimal read transaction.

The empirical results are displayed in Figure 2. The time attributable to transaction management alone is a derived number, but it should be quite reliable because the application performs its update operations in sequence and because no other activity is in progress while the updates are performed. Transaction management cost is derived by subtracting the time due to operation calls. The cost of a local operation is 3.5ms (3ms

[1]ComMan does not provide message transport for the transaction manager. In order to process distributed protocols as quickly as possible, transaction managers on different sites communicate using datagrams. A transaction manager is responsible for implementing mechanisms such as timeout/retry and duplicate detection.

for the operation IPC and 0.5ms for locking and data access). The cost of each remote operation is 29.0ms (28.5ms for the operation RPC and 0.5ms for locking and data access). So for an N-subordinate transaction the number of milliseconds to subtract is 3.5 + 29N.

Two important facts are evident: variance goes up quickly as the number of subordinates goes up, and the assumption that "parallel" operations proceed in parallel is far from satisfied. The time attributed to transaction management should be constant for any non-zero number of subordinates. It is not, but seems to rise (although not smoothly) as the number of subordinates rises. The observation of variance rising with network load is true in this and all subsequent experiments.

Unfortunately, the accurate timing tools needed to measure and analyze the components of the delay were not available. The likeliest source of congestion is the coordinator. It is doubtful that there was much variance in subordinate processing of phase one. All were similarly equipped and loaded, and all performed the same on tests involving either only local transactions or RPCs. The token ring on which the experiments were performed is a single continuous ring, and so queueing at a gateway can be ruled out. One known cause of the rising times is the "cycle time" for sending datagrams. A send takes 1.7ms, meaning for example that the third prepare message is sent about 3.4ms after the first. This alone is a small effect.

A surprising result is that multicasting messages from coordinator to subordinates reduces variance substantially, suggesting that much of the variance is created by the coordinator's repeated sends and not by its repeated receives. This may be due to operating system scheduling policies.

More can be said about the latency and variance increases in the unoptimized protocol. First, there is more network activity due to the commit-ack message not being piggybacked. Second, there was lock contention. The application used in the experiment locked and updated the same data element during every transaction. The operation of the second transaction arrived at the data element before the first transaction could drop the lock, so the operation of the second transaction waited. After the commit-transaction call of the first transaction returns to the application, the time needed for the remote operation of the second transaction to arrive at the remote data element is approximately:

> 1.5ms (begin-transaction) + 5ms (local join-transaction and operation) + 29ms/2 (time for remote operation to arrive) = 21ms

Meanwhile, the time for the first transaction to drop the lock is:

> 10ms (commit datagram) + 15ms (commit log force) + 1ms (remote drop-locks call) = 26ms

By this simple analysis, the second operation waits approximately 5ms. However, these activities are interleaved at the coordinator, so the delay could be much longer.

PRIMITIVE	TIME (ms)
Local in-line IPC	1.5
Local in-line IPC to server	3
Local out-of-line IPC	5.5
Local one-way inline message	1
Remote RPC	29
Log force	15
Datagram	10
Get lock	0.5
Drop lock	0.5
Data access: read	negligible
Data access: write	negligible

Table 2: Latency of Camelot Primitives

The result of static analysis on simple transactions is presented in Table 3 using latencies of primitives reported in Table 2. The static analysis contained in Table 3 accounts for 24.5 of the 31 milliseconds of the local update experiment, for 99.5 of the 110 milliseconds of the 1-subordinate update experiment, and for 9.5 of the 13 milliseconds of the local read experiment. The missing time is accounted for by CPU time in various processes, and by error associated with the primitive latencies and with the method.

Based on the small sample in Figure 3, the method of static analysis seems reasonably reliable. As expected, the addition of primitive latencies provides an underestimate of the measured time. In addition, the method seems less accurate with smaller transactions, possibly because inaccuracy in the latencies of primitives is more directly reflected in the predicted total times.

4.3 Non-blocking Commitment

To commit a single transaction as quickly as possible with the non-blocking protocol, it is necessary for both the coordinator and each subordinate to force two log records. The forces take place "in parallel" at the subordinate sites, so the critical path through the protocol consists of 4 log forces and 5 messages. This compares to 2 and 3, respectively, for two-phase commit. It is the replication phase that accounts for extra log forces in the non-blocking protocol. The ratios of the dominant primitives are 4/2 and 5/3, which implies that the critical path of the non-blocking protocol is about twice the length of that of two-phase commit. The optimality work of Dwork and Skeen [11] suggests that the 2-to-1 ratio is to be expected. The length of the completion path is one datagram shorter for both protocols.

The questions asked for non-blocking commit are:

1. What is the speed of the non-blocking protocol relative to two-phase commit? Comparisons are made against the optimized implementation of two-phase commit explained in Section 3.3.
2. What is its speed in absolute terms?
3. What is the effect of the read-only optimization?
4. How accurate is static analysis?

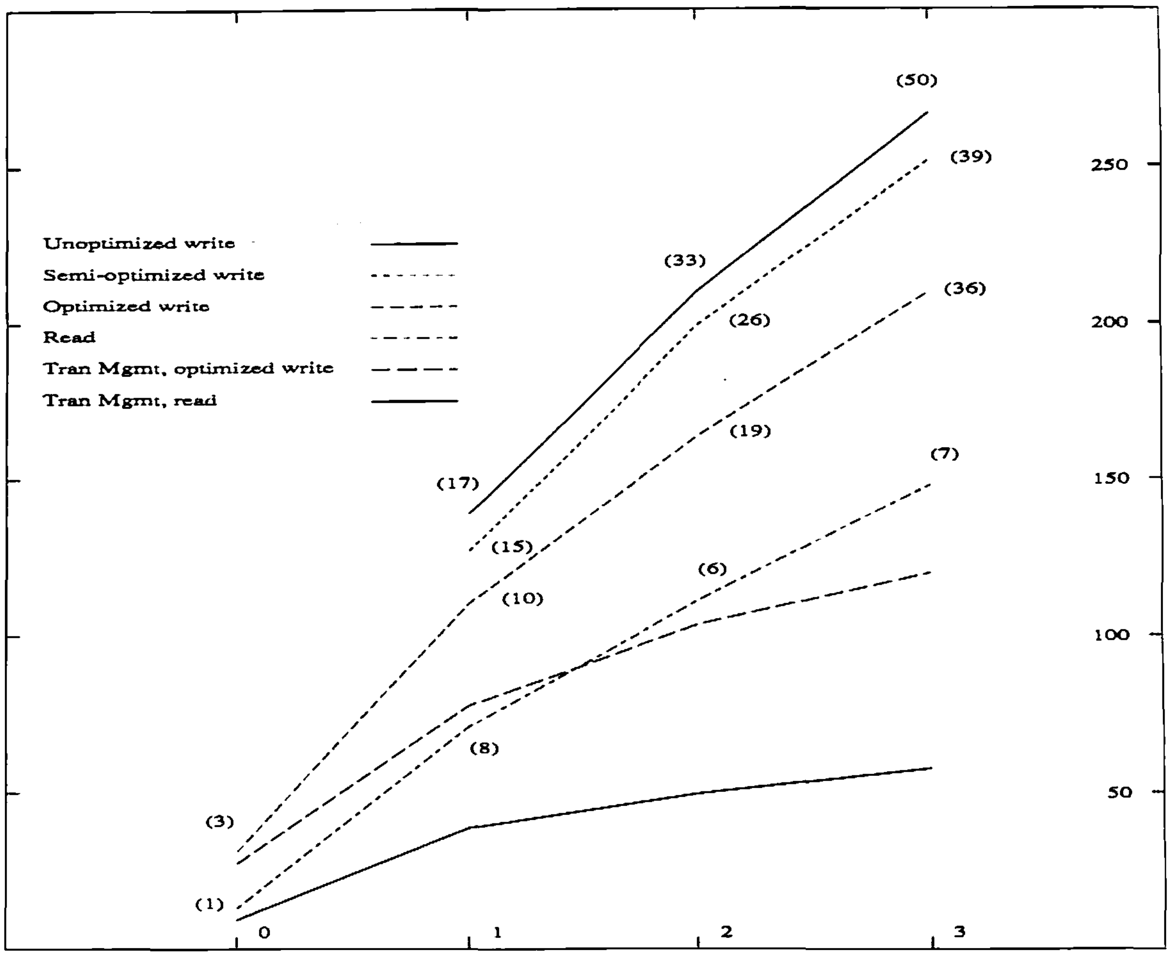

Figure 2: Latency of Transactions, Two-phase Commit (subordinates vs. ms)
Standard deviation of measured times indicated in parentheses.

The basic experiment was the same as that reported in Section 4.2: a minimal distributed transaction on a coordinator and on 1, 2, and 3 subordinate sites. Each basic experiment was run twice, for read and write operations. The results are shown in Figure 3.

Static analysis of a 1-subordinate update transaction gives a completion path consisting of 4 log forces, 4 datagrams, 1 remote operation, and 20ms of local transaction management messages. Using the primitive times listed in Table 2, the sum of these primitives provides an underestimate of 150ms, yet times as low as 145ms were measured. As seen above, all numbers go up swiftly as the size of the transactions increases.

Static analysis of a 1-subordinate read transaction gives a completion path consisting of only 2 datagrams, 1 remote operation, and 20ms of local transaction management messages. This represents only 70ms, which is quite far from the measured number of 101ms. Unlike in other tests, transaction management cost grows more slowly as more sites become involved. Variance remains high, since distributed read-only transactions consist mostly of inter-site messages.

The cost of non-blocking commitment relative to two-phase commitment seems somewhat less than twice as high, a result that is in line with the ratios computed statically. In absolute terms, the protocol executes in a fraction of a second in the Camelot/Mach environment. In order for the latency of the commitment protocol to be negligible (say, less than 5%), non-blocking commitment should be used with transactions that last longer than a few seconds. This implies that non-blocking commitment is suitable for transactions used in application programming, but not in system programming.

4.4 Multi-Threading

A number of tests measured the extent to which the transaction manager is able to increase its throughput as its load increases. The basic experiment consisted of having different numbers of application/server pairs execute minimal transactions. Separate pairs of applications and servers were used to ensure that operation processing was not a bottleneck. Instead, the system components experiencing the greatest load were Mach and Camelot. For read transactions, the transaction

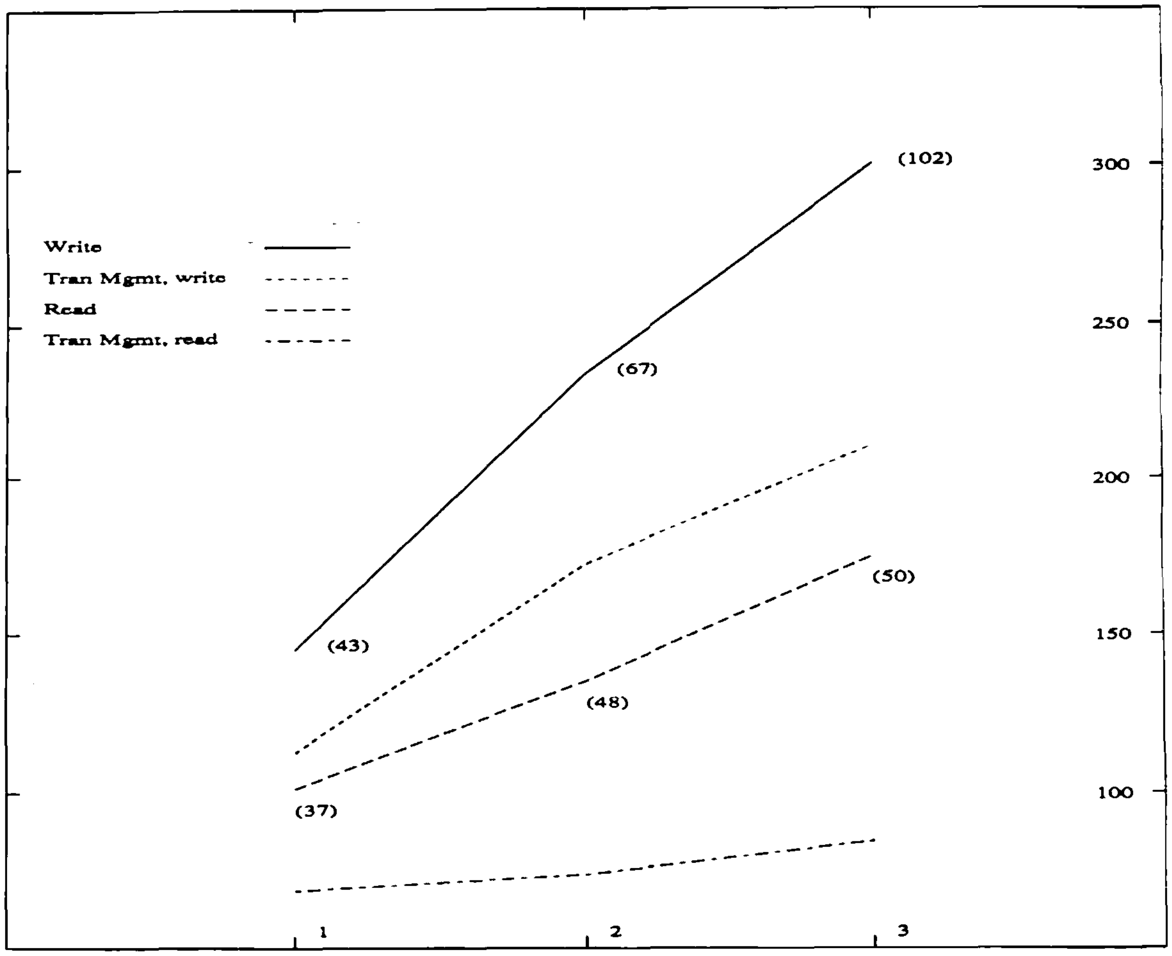

Figure 3: Latency of Transactions, Non-blocking Commit (subordinates vs. ms)
Standard deviation of measured times indicated in parentheses.

manager and the message system are the only components that receive substantial load. For update transactions, the logger (disk process) also receives high traffic. The technique was to increase the number of application/server pairs until saturation, when throughput ceased to increase. Because small transactions are message-intensive, the load on the operating system was considerably higher than the load on any part of Camelot. This is an unavoidable consequence of implementing the transaction manager as a service reachable only via IPC.

The important questions are:

1. Is parallelism ever needed? That is, is the transaction manager ever a bottleneck in transaction processing?
2. If so, can significant throughout increase be attained by multithreading?
3. When throughput peaks, what points are the bottlenecks?

The basic experiment was performed on a 4-way symmetric shared memory VAX multiprocessor (employing 1-MIP model 8200 CPUs). The number of threads within the transaction manager was limited to a fixed number, and was a parameter of experimentation. The values used were 1, 5, and 20. These three numbers were chosen as being clearly insufficient, barely sufficient, and easily sufficient, respectively, to handle the offered load. In each configuration, the load was increased until saturation; that is, until throughput decreased or leveled. The results are presented in Figures 4 and 5.

From the numbers for read-only transactions, it is apparent that a single transaction management thread can accommodate more than 1 client but not more than 2. With more than two clients the experiment becomes "TranMan-bound." It is not operating-system-bound, because the same test with 5 and 20 TranMan threads yields somewhat better results.

In update tests, the logger is the bottleneck. This is seen most obviously in comparing the numbers gathered with and without group commit enabled. It is also suggested more indirectly by the fact that, for a given number of TranMan threads, there is greater throughput increase for read tests than for update tests (52%

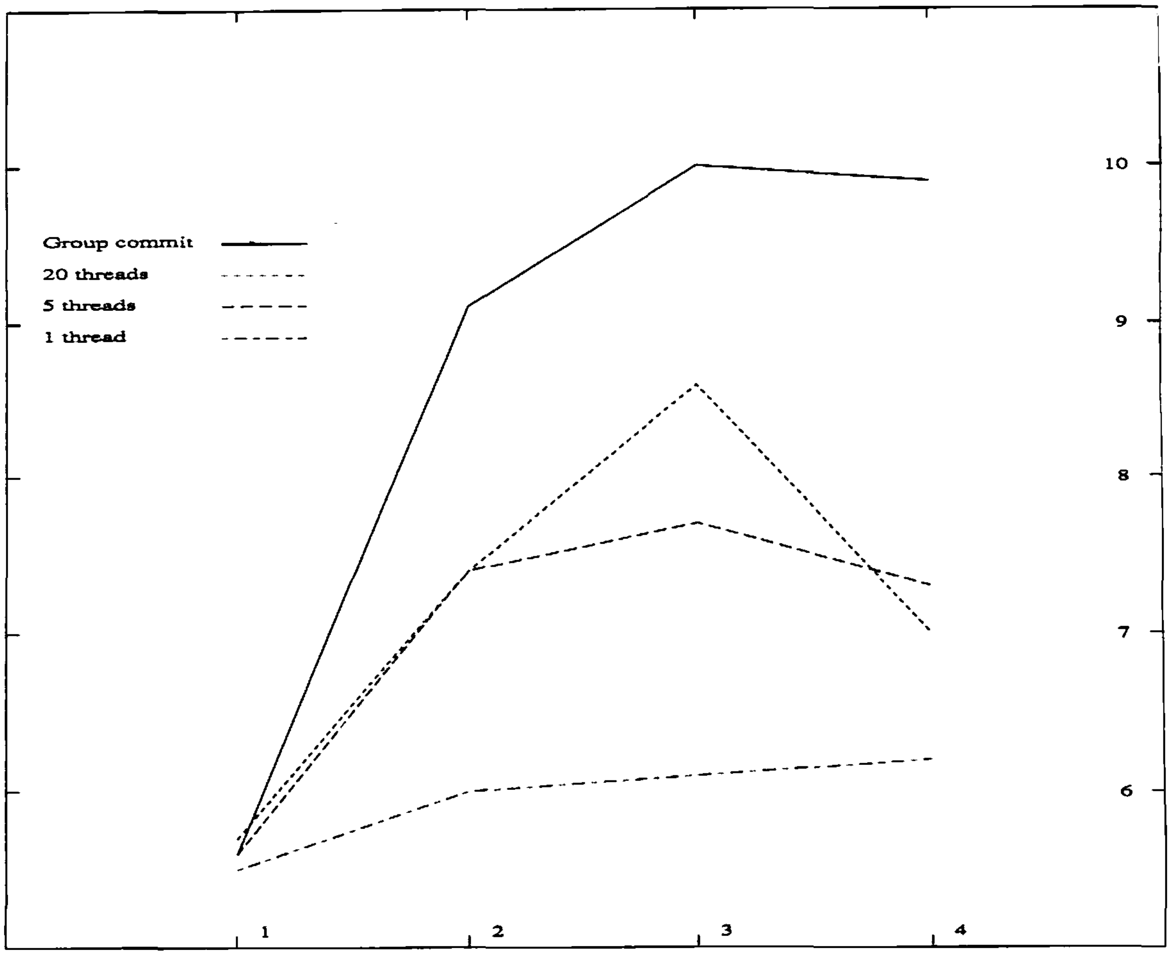

Figure 4: Update Transaction Throughput (Appl./server pairs vs. TPS)

vs. 32% from 1 to 2, and 12% vs. 4% from 2 to 3). The difference between a read and an update test is essentially only the log force (see Table 3). The numbers for the 20-thread tests are roughly the same as those for the 5-thread tests. The lack of improvement is a further piece of indirect evidence that logging, and not transaction management, is the bottleneck of Camelot in update tests.

4.5 Camelot Process Structure and Mach Primitives

One notable aspect of the Camelot design is the large number of processes used to implement the system. Advantages of having transaction management bundled in a separate process are two-fold: those associated with having transaction management code outside servers, and improved performance in some cases. Performance improvements result from sending fewer inter-site messages and forcing fewer log records when a site has multiple servers involved in a transaction. Instead, the transaction manager acts as a forwarding agent for messages and a gathering point for log writes. Having transaction management code in a separate address space is safer and also makes it easier to change transaction management software. Inevitably, however, transaction management performance depends upon Mach primitives. This section contains a short qualitative evaluation of IPC and thread-switching performance in Mach version 2.0.

Mach messages are not as fast as those of some other systems because of the generality of the message model. It would be helpful to be able to quickly send small data with restricted semantics, as in Quicksilver and V [4]. Mach makes more sophisticated scheduling decisions than either Quicksilver or V, and has internal locking which leads to more and bigger data structures. Also, being ready for the possibility of page fault before kernel data copy-in consumes time on kernel entry. Complications related to maintaining Unix compatibility include having to check for signals on kernel exit. Unlike Quicksilver, Mach can pause within the kernel, meaning that a kernel stack must exist and be managed. The concern for portability also impacts performance, since assembly-language solutions cannot be used for key operations such as scheduling and message-passing tricks such as passing data in registers. Likewise, "low-

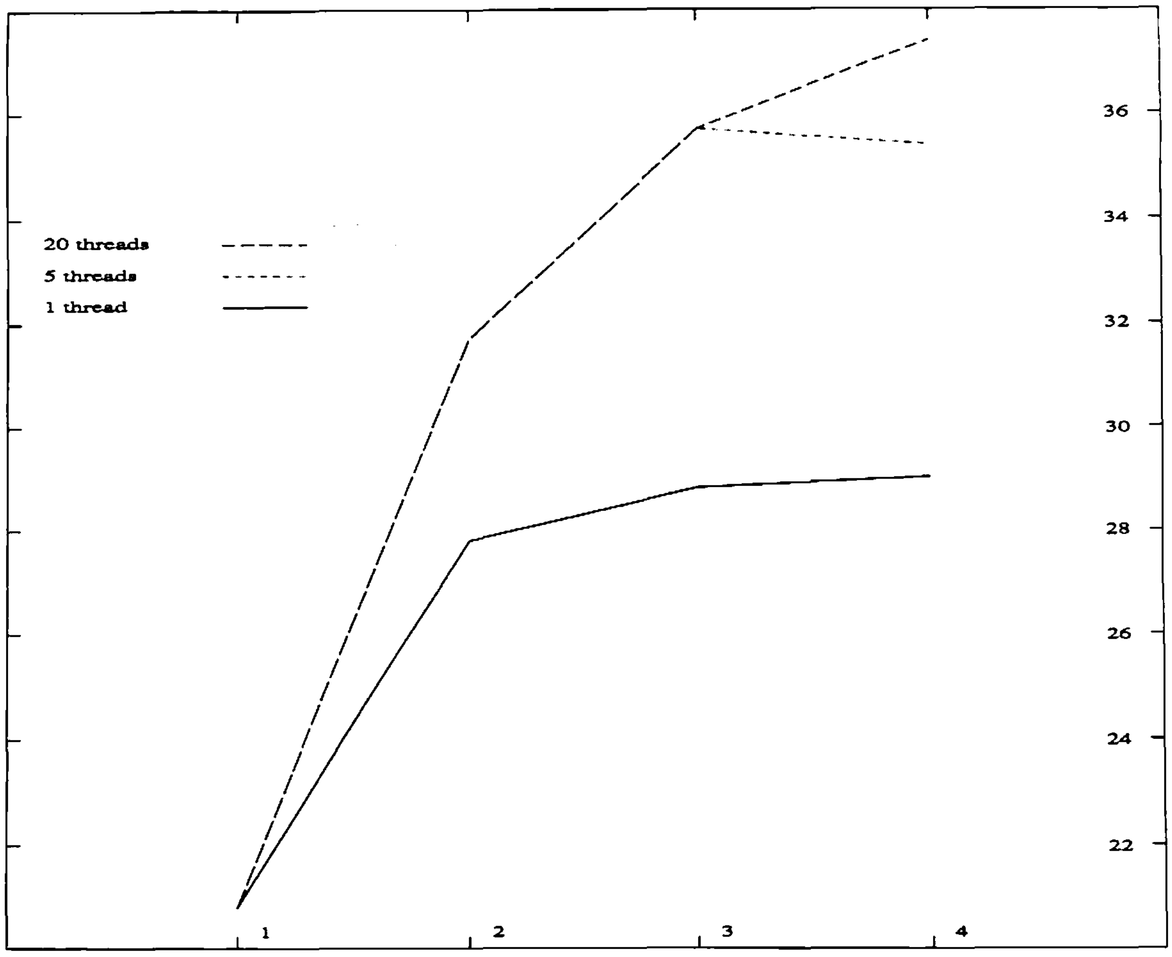

Figure 5: Read Transaction Throughput (Appl./server pairs vs. TPS)

level hacking" cannot be used. For example, Quicksilver tunes source code so as to eliminate procedure calls during kernel entry and exit.

Factors affecting the speed of switching between threads include the use of sophisticated scheduling and the fact that the version of Mach on which these tests were run had only a single run queue on one "master" processor. Another major factor is that the heavyweight VAX "load/store context" instruction, intended for task switching, is used to implement thread switch as well.

In summary, a multi-process, multi-threaded transaction facility built on top of a message-passing kernel is essentially an extension of the operating system since transaction overhead consists mostly of operating system primitives. If performance is a dominating concern then either the operating system should be very carefully tuned or the transaction facility should be reimplemented inside the kernel. The latter strategy has the effect of converting some inter-process messages into intra-kernel procedure calls.

5 Related Work

In terms of transaction management, the systems most related to Camelot are Argus and Quicksilver. Camelot has taken certain techniques, especially those related to communication support, from another IBM research system, R* [20][24].

Camelot and Argus have nearly the same transaction model; these two projects represent the two implementations of Moss-model nested transactions. The only major difference is that in Argus a transaction can make changes at only one site. Diffusion must be done within a nested transaction. Argus has paid close attention to the performance of their implementation of two-phase commit [22], but apparently has not incorporated the optimization of Section 3.2. There is no non-blocking protocol despite the fact that transaction management is built into servers, and therefore failures are more likely.

Quicksilver is an entire system, consisting of a kernel and several service processes defined similarly to those in Camelot. While transaction management is outside the kernel, the notion of a TID is known to

CAMELOT PRIMITIVES	LOCAL READ	LOCAL WRITE	2-SITE WRITE
Begin-transaction	1.5	1.5	1.5
Local operation IPC	3	3	3
Local join-transaction	1.5	1.5	1.5
Get lock \ data access	0.5	0.5	0.5
Notice of diffusion	---	---	1
Notice of arrival	---	---	1
Remote operation RPC	---	---	29
Remote join-transaction	---	---	1.5
Get lock \ data access	---	---	0.5
Commit-transaction	1.5	1.5	1.5
Get subordinates from ComMan	---	---	5.5
Local server prepare	1.5	1.5	1.5
Prepare datagram	---	---	10
Remote server prepare	---	---	1.5
Force log: prepare	---	---	15
Prepare response datagram	---	---	10
Force log: commit	---	15	15
Predicted total	9.5	24.5	99.5
Measured total	13	31	110
Predicted tran. mgmt.	6	21	66.5
Measured tran. mgmt.	9.5	27.5	77.5
Critical path formula	0	1LF	3DG+2LF
Local drop locks msg.	---	1	1
Commit datagram	---	---	10
Remote drop locks msg.	---	---	1
Commit-ack datagram	---	---	10
Tell ComMan to forget	---	---	1

Table 3: Latency Breakdown
The upper portion of the table lists, in approximate order, the events on the critical path and their latencies. The middle portion compares static and empirical analyses. There, "DG" denotes a datagram, while "LF" denotes a log force. The lower portion lists other operations that must happen, but which are not in the critical path.

the kernel, and the IPC system keeps track of the (local) spreading of transactions. As in Camelot, a communication manager keeps track of diffusion to remote sites. Having the IPC system understand transactions means that TIDs are hidden from clients, which is a distinct advantage. Quicksilver does not support nested transactions, although an extension is planned. Its implementation of two-phase commit is hierarchical and does not include the optimization described in Section 3.2, but does include many variations intended to support special classes of servers. There is no general non-blocking protocol, but Quicksilver does have two useful techniques that address the blocking problem:

- A two-site transaction uses "coordinator migration." Coordinator migration allows the subordinate and the coordinator to switch roles. Coordinator migration merely makes the more reliable site act as coordinator.
- Transactions involving more than two sites may use "coordinator replication." In essence, the coordinator nominates one of the subordinates to serve as a co-coordinator. Each co-coordinator coordinates commitment of approximately half the sites. If either co-coordinator fails, then the other takes over as coordinator for all sites. Coordinator replication shortens the window of vulnerability.

The motivation for a non-blocking commitment protocol has existed for some time, and many protocols have been proposed. Most previous protocols do not correctly handle partitions [15][19][26]. The notable exception is Skeen's "quorum-based" protocol [27], part of which resembles the non-blocking protocol of Section 3.3. The improvements incorporated in this work [8] are:

- A complete and correct specification, including when and how to forget. Most specifications of commitment protocols end once one site reaches the outcome. A protocol that allows a site to forget too soon can be wrong.
- A design optimization: the special handling of the read-only case.
- Performance optimizations: reduction in the number of log forces and provision for piggybacking later messages.
- A proof that the protocol survives any single failure.
- A complete implementation and evaluation.

TMF uses broadcast and memory-based (rather than log-based) replication to provide very fast non-blocking commitment for a different failure model in which more than a single failure represents disaster. A practical approach to blocking is the "heuristic commit" feature of LU 6.2 [18], which allows a blocked transaction to be resolved either by an operator or by a program. While not guaranteeing correctness, this approach does not slow down commitment in the regular case.

6 Conclusions

The requirements of transaction management exercise substantial influence on the design of whatever mechanism is used to provide communication between processes on different sites. Accordingly, it is important for a transaction facility to have "hooks" at appropriate points in the communication mechanism.

The critical path of two-phase commit can be optimized so that update transactions need contain only two log writes (both forces) and two inter-site messages. Non-blocking commit can be done with a three-phase protocol, including two log forces at each site and five messages in the critical path of an update transaction. Read-only sites need never participate in the notify phase, and often need not participate in either the replication or notify phases. A transaction that is completely read-only has the same critical path performance as in two-phase commitment.

Because of its inherently higher cost, non-blocking commitment is not suitable for all applications. Its main uses are for applications that regularly run large transactions, for transactions executed at sites spanning a wide area, and for systems that, unlike Camelot, have each server act as a transaction manager. Multicast communication for coordinator to subordinates does not reduce commit latency, but does reduce variance.

A multi-threaded design can prevent the transaction manager from being the bottleneck of a transaction system. However, the utility of a multithreaded transaction manager is determined by whether group commit is turned on.

7 Acknowledgments

Jim Gray's comments greatly improved both the content and presentation of this paper.

This work was supported by IBM and the Defense Advanced Research Projects Agency, ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsoring agencies or of the United States Government.

References

[1] M. Accetta. et. al.
Mach: A New Kernel Foundation for Unix Development.
In *Proc. of Summer Usenix*, pages 93–112, July 1986.

[2] G. Bruell A. Z. Spector, R. Pausch.
Camelot, a Flexible, Distributed Transaction Processing System.
In *Thirty-third IEEE Computer Society Intl. Conf. (COMPCON)*, pages 432–437, March 1988.

[3] M. Burrows M. Schroeder.
Performance of Firefly RPC.
Technical Report 43, Digital Systems Research Center, April 1989.

[4] D. R. Cheriton.
The V Distributed System.
Comm. ACM, 31(3):314–333, March 1988.

[5] D. Daniels and A. Spector.
Performance Evaluation of Distributed Transaction Facilities.
In *Proc. of the Intl. Wkshp. on High Performance Transaction Systems*, page 10. ACM, September 1985.

[6] R. P. Draves and E. C. Cooper.
C Threads.
Technical Report CMU-CS-88-154, Carnegie Mellon University, June 1988.

[7] D. Duchamp.
An Abort Protocol for Nested Distributed Transactions.
Technical Report CUCS-459-89, Columbia University, September 1989.

[8] D. Duchamp.
A Non-blocking Commitment Protocol.
Technical Report CUCS-458-89, Columbia University, September 1989.

[9] D. Duchamp.
Protocols for Distributed and Nested Transactions.
In *Proc. Unix Transaction Processing Wkshp.*, pages 45–53, May 1989.

[10] D. Duchamp.
Transaction Management.
PhD thesis, Carnegie Mellon Univ., December 1988.
Available as Technical Report CMU-CS-88-192, Carnegie-Mellon University.

[11] C. Dwork and D. Skeen.
The Inherent Cost of Nonblocking Commit.
In *Proc. 2nd Ann. Symp. on Principles of Distributed Computing*, pages 1–11. ACM, August 1983.

[12] D. Gawlick and D. Kinkade.
Varieties of Concurrency Control in IMS/VS Fast Path.
Database Engineering, 8(2):63–70. June 1985.

[13] D. K. Gifford.
Weighted Voting for Replicated Data.
In *Proc. Seventh Symp. on Operating System Principles*, pages 150–162. ACM, December 1979.

[14] J. N. Gray.
A View of Database System Performance Measures.
In *Proc. 1987 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 3–5, May 1987.

[15] M. Hammer and D. Shipman.
Reliability Mechanisms for SDD-1.
ACM Trans. on Database Systems, 5(4):431–466, December 1980.

[16] P. Helland.

The Transaction Monitoring Facility (TMF).
Database Engineering, 8(2):71–78. June 1985.

[17] P. Helland. et. al.
Group Commit Timers and High Volume Transaction Systems.
In *Proc. of the Second Intl. Wkshp. on High Performance Transaction Systems*, page 24, September 1987.

[18] IBM Corporation.
Systems Network Architecture. Format and Protocols Reference Manual: Architecture Logic for LU Type 6.2. sc30-3269-3 edition, December 1985.

[19] G. Le Lann.
A Distributed System for Real-time Transaction Processing.
Computer, 14(2):43–48, February 1981.

[20] B. Lindsay et. al.
Computation and Communication in R*: A Distributed Database Manager.
ACM Trans. on Computer Systems, 2(1):24–38, February 1984.

[21] B. Liskov et. al.
Argus Reference Manual.
Technical Report MIT/LCS/TR-400, MIT, November 1987.

[22] B. H. Liskov B. M. Oki and R. W. Scheifler.
Reliable Object Storage to Support Atomic Actions.
In *Proc. Tenth Symp. on Operating System Principles*, pages 147–159. ACM, December 1985.

[23] C. Mohan and B. Lindsay.
Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions.
In *Proc. Second Ann. Symp. on Principles of Distributed Computing*, pages 76–88. ACM, August 1983.

[24] R. Obermarck C. Mohan and B. Lindsay.
Transaction Management in the R* Distributed Data Base Management System.
ACM Trans. on Database Systems, 11(4):378–396, December 1986.

[25] W. Sawdon R. Haskin, Y. Malachi and G. Chan.
Recovery Management in Quicksilver.
ACM Trans. on Computer Systems, 6(1):82–108, February 1988.

[26] D. Skeen.
Crash Recovery in a Distributed Database System.
PhD thesis, Univ. of California, Berkeley, May 1982.

[27] D. Skeen.
A Quorum-based Commit Protocol.
In *Proc. Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 69–80, February 1982.

[28] A. W. Tanenbaum R. van Renessee and H. van Staveren.
Peformance of the World's Fastest Distributed Operating System.
ACM SIGOPS Operating System Review, 22(4):25–34, October 1988.

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