# Transactional Distributed Shared Memory

**Andrew B. Hastings**

July 1992

CMU-CS-92-167

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania

*Submitted to Carnegie Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.*

This work was supported by IBM and by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title *Research on Parallel Computing* issued by DARPA/CMO under Contract MDA972-90-C-0035, ARPA Order No. 7330.

Carnegie Mellon

School of Computer Science

**DOCTORAL THESIS**
**in the field of**
**Computer Science**

*Transactional Distributed Shared Memory*

**ANDREW HASTINGS**

Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

| | |
|---|---|
| **ACCEPTED:** | |
| [illegible] MAJOR PROFESSOR | 7/21/92 DATE |
| [illegible] DEAN | 8/21/92 DATE |
| **APPROVED:** | |
| [illegible] PROVOST | 26 August 1992 DATE |

# Abstract

Atomic transactions have proven to be an important technique for constructing reliable applications. Traditionally, transactions have been extended to distributed environments through the use of *function shipping*, a technique in which message passing or remote procedure calls are used to invoke computational requests on remote nodes. Recently, the *data sharing* approach to constructing distributed applications has received attention in the form of distributed file systems and distributed shared virtual memory. Applying the data sharing approach to transactions produces transactional distributed shared memory (TDSM) which yields benefits for a certain class of distributed application.

The union of transactions and distributed shared memory offers synergies in transaction recovery, concurrency control, and coherency control, but introduces challenges in transaction recovery. In this dissertation, I describe the design of a system that provides TDSM in the form of distributed recoverable virtual memory. Using the external pager interface of the Mach operating system, I implemented a prototype based on the Camelot distributed transaction facility. I analyze the prototype and its performance, offer techniques for improving the design of future TDSM systems, and characterize the applications for which TDSM is useful.

## Acknowledgments

My advisor, Alfred Spector, deserves my deepest thanks. Despite his many commitments at CMU and later at Transarc Corporation, he always found time for me. As a researcher, teacher, editor, manager, entrepeneur, psychologist, and friend, he has no equal in my experience.

I am grateful to Rick Rashid, Eric Cooper, and Marvin Theimer for returning to CMU to serve on my thesis committee. Rick is my second advisor; his advice on writing style improved the presentation tremendously, and made the dissertation much easier to write. Marvin deserves special thanks: he read the first draft of every chapter, and his detailed comments were invaluable.

The members of the Camelot project are to be congratulated for developing the transaction system that was the platform for my work. The Camelot project consisted of Joshua Bloch, Dean Daniels, Richard Draves, Dan Duchamp, Jeffrey Eppinger, Elliot Jaffe, Toshihiko Kato, George Michaels, Lily Mummert, Randy Pausch, Peter Stout, and Dean Thompson. I would especially like to thank Peter Stout for technical discussions, and Josh Bloch for proving that it was possible to finish after Alfred left CMU. I also wish to acknowledge Jeff Eppinger for permitting me to use in Chapter 2 some of the background prose from his Ph.D. dissertation.

I am indebted to Joe Barrera for implementing the External Memory Manager library, saving me many months of effort. Joe, Daniel Julin, and Mike Young were very willing to explain Mach-related topics and to fix bugs when necessary. I thank the numerous others who made CMU Computer Science an interesting and productive place to work.

My officemates provided a stable, supportive environment during my five years at CMU. I thank Alan Christiansen, Puneet Kumar, Alicia Perez, and Manuela Veloso for being so understanding. I also wish to thank my roommates over the years for their companionship: Gilles Dowek, Alison Ford, Daniella Gerstein, Spiro Michaylov, and Tom Ruschak. Spiro and Tom have been very good friends, and I remember fondly the time I spent with each of them.

I'd like to thank all of my friends who helped to put life and graduate school in perspective. My friends from Cray Research, including Clark Piepho, Jim Harrell, and Peter Hill, reminded me what a Real Job is like. I especially appreciate the opportunity Jim gave me to escape from CMU when I wanted some time away. I'm thankful for the fellowship I found at several Presbyterian churches in Pittsburgh. And I am very grateful for the friendship of Lissie Crock, Lesley Kromer, and Margie Martini.

Above all, I thank my family, including my brother and sister. I could not have finished this dissertation without the support and encouragement of my parents. For their unfailing love and devotion, I dedicate this dissertation to my mother and father.

# Table of Contents

1. Introduction 1
   - 1.1. Problem Description 1
   - 1.2. Programming models 2
     - 1.2.1. Client/server Model 3
     - 1.2.2. Transactions 3
     - 1.2.3. Replication and Partitioning 4
     - 1.2.4. Caching 4
     - 1.2.5. Distributed Shared Memory 5
   - 1.3. Purpose of the Dissertation 6
   - 1.4. Examples 7
   - 1.5. Synergies and Challenges 10
   - 1.6. Outline 11

2. Background 13
   - 2.1. Distributed Computing 13
   - 2.2. Transactions 14
     - 2.2.1. Transaction management 16
     - 2.2.2. Recovery management 17
     - 2.2.3. Buffer management 18
     - 2.2.4. Log management 19
     - 2.2.5. Concurrency management 20
     - 2.2.6. Communication management 21
     - 2.2.7. Configuration management 21
   - 2.3. Data Sharing 21
     - 2.3.1. Directory methods 22
     - 2.3.2. Snoopy methods 24
     - 2.3.3. Other methods 25
   - 2.4. Distributed Concurrency Control 26
   - 2.5. Integrated Technologies for TDSM 28
     - 2.5.1. Shared disks 28
     - 2.5.2. Hsu and Tam 31

3. Camelot and Mach Environment 35
   - 3.1. Camelot 35
     - 3.1.1. Camelot library 36
     - 3.1.2. Disk Manager 38
       - 3.1.2.1. Data structures 38
       - 3.1.2.2. Algorithms 39
     - 3.1.3. Recovery Manager 40
     - 3.1.4. Communication Manager 41
     - 3.1.5. Transaction Manager 41
     - 3.1.6. Node Server/Node Configuration Application 41

3.2. Mach 42
4. Design 45
4.1. Goals 45
4.2. Architecture 46
4.2.1. Camelot and Mach 47
4.2.2. Log management 49
4.2.3. Transaction management 49
4.2.4. Recovery management 50
4.2.5. Buffer management 52
4.2.6. Concurrency management 53
4.2.7. Communication management 54
4.2.8. Configuration management 54
4.3. Discussion 55
4.4. Example 58
4.5. Summary 63
5. Implementation 65
5.1. Concurrency Management 65
5.1.1. Programmer interface changes 67
5.1.2. Cache control 67
5.1.3. Failures 68
5.1.4. Lock Manager/library interface 69
5.1.5. Data structures 70
5.2. Buffer Management 70
5.2.1. External Memory Manager 71
5.2.2. Disk Manager 71
5.2.2.1. Server record 72
5.2.2.2. Segment record 72
5.2.2.3. Grid 73
5.2.2.4. Internal concurrency control 74
5.2.3. Remote Execution Manager 74
5.2.4. Algorithms 75
5.2.4.1. Forward Processing 75
5.2.4.2. Coherency control 76
5.2.4.3. Segment activation 77
5.2.4.4. Paging 77
5.2.4.5. Hot Pages 78
5.2.4.6. Preflush 79
5.2.4.7. Down servers 80
5.2.4.8. Server startup 81
5.2.4.9. Server termination 82
5.2.4.10. System shutdown 82
5.3. Recovery Management 82
5.3.1. Old-value/new-value abort 83
5.3.2. New-value-only abort 84
5.3.3. Segment and server recovery 84
5.4. Configuration Management 85
5.5. Communication Management 87
5.6. Log Management 88
5.7. Transaction Management 88

5.8. Summary 88
6. Performance 91
6.1. Experimental Environment 91
6.2. Primitives 92
6.3. Operation Costs 94
6.3.1. Experimental parameters 95
6.3.2. Non-paging tests 99
6.3.2.1. Non-paging reads 99
6.3.2.2. Non-paging writes 101
6.3.3. Paging tests 103
6.3.4. Multi-server tests 106
6.3.4.1. Multi-server reads 107
6.3.4.2. Multi-server writes 108
6.3.5. Summary 110
6.4. Throughput 111
6.5. Function Shipping vs. Data Sharing 116
6.6. Summary 118
7. Analysis 119
7.1. Performance Model 119
7.1.1. Parameters of the model 120
7.1.2. Actual parameters 120
7.1.3. Scenario 1 123
7.1.4. Scenario 2 125
7.1.5. Conclusion 126
7.2. Throughput vs. Cache Hit Ratio 127
7.3. Performance Improvements 130
7.3.1. Optimizations 130
7.3.2. Architectural changes 133
7.4. Deficiencies of the System 135
7.4.1. Recovery 135
7.4.2. Configurability 136
7.4.3. Security 136
7.4.4. Availability 137
7.4.5. Incremental Growth 138
7.5. Summary 138
8. Conclusions 139
8.1. Motivation 139
8.2. Contributions 141
8.3. Future work 143
8.4. Conclusion 144
Appendix A. Performance Model 145
A.1. Parameters 145
A.2. Operation Model 146
A.3. Modeled results 148
Appendix B. Interfaces 153
B.1. DH Interface 153
B.2. DN Interface 154

B.3. DR Interface 155
B.4. DS Interface 157
B.5. DX Interface 158
B.6. HS Interface 158
B.7. NA Interface 160
B.8. ND Interface 161
B.9. RD Interface 162
B.10. SH Interface 163
B.11. XD Interface 165
References 167

# List of Figures

| | | |
|---|---|---|
| Figure 2-1: | Data sharing with shared disks | 28 |
| Figure 3-1: | Camelot architecture | 36 |
| Figure 3-2: | The grid | 39 |
| Figure 3-3: | External Memory Manager | 44 |
| Figure 4-1: | TDSM architecture | 48 |
| Figure 4-2: | False sharing | 57 |
| Figure 4-3: | Flow of log records and paging requests | 58 |
| Figure 4-4: | Messages for BEGIN_TRANSACTION, LOCK, and MODIFY | 59 |
| Figure 4-5: | Messages for END_TRANSACTION | 60 |
| Figure 4-6: | Messages for LOCK and MODIFY | 61 |
| Figure 4-7: | Messages for abort | 62 |
| Figure 5-1: | Camelot/TDSM architecture | 66 |
| Figure 5-2: | Segment and server records | 73 |
| Figure 5-3: | Mapping of offsets | 79 |
| Figure 6-1: | Test configuration 1 | 97 |
| Figure 6-2: | Test configuration 2 | 97 |
| Figure 6-3: | Test configuration 3 | 97 |
| Figure 6-4: | Test configuration 4 | 97 |
| Figure 6-5: | Test configuration 5 | 97 |
| Figure 6-6: | Test configuration 6 | 97 |
| Figure 6-7: | Test configuration 7 | 97 |
| Figure 6-8: | Test configuration 8 | 97 |
| Figure 6-9: | ET1 transaction | 112 |
| Figure 6-10: | ET1 "local" configuration | 113 |
| Figure 6-11: | ET1 "remote" configuration | 113 |
| Figure 6-12: | Data sharing configuration | 116 |
| Figure 6-13: | Function shipping configuration | 116 |
| Figure 7-1: | Throughput vs. cache hit ratio, actual parameters | 127 |
| Figure 7-2: | Throughput vs. cache hit ratio, Scenario 1 | 128 |
| Figure 7-3: | Throughput vs. cache hit ratio, Scenario 2 | 128 |

# List of Tables

Table 5-1: Lines of code 89
Table 6-1: Representative IPC times 93
Table 6-2: Non-paging read times 100
Table 6-3: Non-paging write times 102
Table 6-4: Paging times 104
Table 6-5: Multi-server read times 107
Table 6-6: Multi-server write times 109
Table 6-7: Summary of performance measurements 111
Table 6-8: Function shipping vs. data sharing 117
Table 7-1: Performance model parameters 121
Table 7-2: Modeled ET1 performance (actual parameters) 122
Table 7-3: Modeled ET1 performance (Scenario 1) 124
Table 7-4: Modeled ET1 performance (Scenario 2) 126
Table 7-5: Effects of combining concurrency and coherency control 131
Table 7-6: Effects of "log optimization" 132
Table 7-7: Effects of omitting writeback of modified page 132
Table A-1: Performance model parameters 146
Table A-2: Performance model summary 146
Table A-3: Modeled performance (actual parameters) 149
Table A-4: Modeled performance (Scenario 1) 150
Table A-5: Modeled performance (Scenario 2) 151

# Chapter 1
# Introduction

Transaction processing systems provide failure atomicity, permanence, and serializability guarantees for distributed systems. These guarantees simplify the application programmer's task by reducing the attention that must be paid to concurrency and failures.

Distributed shared memory provides a simple model for application programmers by giving the illusion of a single, global address space for all processes participating in a particular distributed application. This illusion simplifies the application programmer's task by eliminating the need for explicit communication operations to obtain data.

My thesis is that the combination of transactions and distributed shared memory is feasible and useful for a certain class of distributed application, especially those applications which have concurrent access to data that must not be corrupted, and need caching to provide adequate performance. In this dissertation, I motivate this idea and discuss the design and implementation of a prototype system providing transactional distributed shared memory (TDSM). I evaluate the implementation, reflect on the design, and suggest a direction for future designs.

## 1.1. Problem Description

Imagine an application developed for a centralized system that must be adapted for use on a distributed system. The programmer making such an adaptation faces a number of difficulties. Instead of a few processors running on a single clock, a distributed system has many processors operating independently. Which processor should operate on the data? How and when does a processor communicate its results to other processors? To the two-level memory hierarchy of main memory and disk storage, a distributed system adds a third level of network access to main memory and disks attached to other nodes. Where should the data be stored? How and when is data transferred between nodes of the distributed system?

A distributed program may need to meet several requirements [Spector 89a].

- Ease of programming. Programmers are accustomed to centralized system programming models. A programming model for a distributed system that extends existing well-known models can simplify the programmer's task.
- Incremental growth. It should be possible to increase the storage or processing capacity of the system by adding nodes; algorithms used by the distributed program should not stop working or perform poorly as the number of nodes increases.

- Multiple access to data. Users may wish to access data from any node in the distributed system. Updates may occur simultaneously on many nodes.
- Availability. The program should continue to run even if one or more nodes crash. Applications may require that all, or only a subset, of the data be available after a crash.
- Data integrity. Data should not be lost when messages are lost, or when nodes or processes crash. Application consistency constraints should be maintained even if data is distributed among several nodes that may fail or concurrently update the data.
- Security. An application may need to restrict the operations that particular users may perform on various subsets of the data.
- Performance. An application may need to meet response time and throughput requirements, even though processes exchange many messages to meet data integrity and availability requirements.

Concurrent updates complicate the task of maintaining data integrity. Distributed systems exacerbate the problem of concurrent updates by increasing concurrency and increasing the possible delay between reads and writes. The problem that concurrency poses is that independent updates can become interleaved and introduce an inconsistent state. (See [Gray 78] for examples.)

Updates in the presence of failures complicate the task of maintaining data integrity. Failures arise when messages are lost, re-ordered, or corrupted, or when individual nodes and processes crash. The problem that failures pose is that an update may be partially performed, introducing an inconsistent state. (See [Davidson 89] for a discussion.)

The application programmer could be overwhelmed by the complexity of these issues. The next section describes various programming models that can reduce the complexity of the programmer's task.

## 1.2. Programming models

Several programming models aid in meeting application requirements on a distributed system. The key dichotomy in this section is between function shipping (the client/server model), and data sharing (e.g., distributed shared memory). Independent of this dichotomy are transactions, replication and partitioning, and caching. In most current systems, transactions are associated with function shipping, while caching, replication, and partitioning are part of efficient implementations of data sharing.

### 1.2.1. Client/server Model

The *client/server model* offers a straightforward approach to constructing distributed applications by extending the familiar notion of procedure call. Data may be distributed among multiple nodes in the distributed system. To access data, a client makes a *remote procedure call* by sending a message to a server on the node where the data resides (possibly the same node as the client). The server examines the message to identify the procedure and its arguments, calls the appropriate procedure, and sends a message containing the results to the client. Because the client's message contains a request for the server to perform a particular function, the client/server model is also known as *function shipping*. The server can meet security requirements by refusing to perform some operations for a given client.

The client/server model does not directly address availability or data integrity requirements. Remote procedure calls are a relatively expensive mechanism for accessing remote data because the concept does not include any automatic caching. Thus, performance may suffer if remote procedure calls overwhelm the carrying capacity of the network or the processing capacity of a given server.

### 1.2.2. Transactions

*Transactions* make it easier to meet data integrity requirements in the face of concurrency and failures. A *transaction* is a sequence of actions grouped into a unit. If the application's data is in a consistent state, a transaction must transform the data into a new consistent state. (The data may be transformed into a inconsistent state temporarily while a transaction is in progress as long as consistency is restored by the end of the transaction.) If each transaction executes as an atomic, indivisible unit, then data will never be left in an inconsistent state.

Transaction systems address the concurrency problem by preventing transactions from observing each other's partial updates. Transactions may execute concurrently, but locking or timestamp schemes order the accesses to shared data so that each transaction sees only the values stored by previously completed transactions.

Transaction systems address the failure problem by ensuring that the sequence of actions within a transaction succeed or fail as a unit. Transaction systems may undo or redo the actions of partially completed transactions to simulate atomic execution in the presence of failures. A transaction *commits* if it runs to completion; if it fails before completion, any changes it makes are undone, and it *aborts*.

Formally, a transaction has three properties: failure atomicity, permanence, and serializability. *Failure atomicity* ensures that either all of the operations within the transaction complete successfully, or none of them do (partial updates are undone). *Permanence* ensures that the effects of a committed transaction are not lost due to failures. *Serializability* states that there is a serial sequence of transactions that produces the same results as a given concurrent execution of a set of transactions. Thus, concurrently executing transactions cannot observe inconsistent states.

### 1.2.3. Replication and Partitioning

Replication and partitioning can aid in meeting availability, growth, and performance requirements. If there is more data than one node can store, if there are more client requests than one server can process, or if the application requires better availability than one node can provide, the data must be distributed among several nodes instead of being stored on just one. To increase storage capacity, the data may be *partitioned* with no overlaps among the nodes. To improve availability, the data may be *replicated* on a set of nodes, so that other nodes may provide the data if one node goes down. (Partitioning can also improve availability in that part of the data is still available even if one node crashes.) To increase processing capacity, either replication or partitioning may be used, although a poor choice of partitions may still overload a node if the data stored there is accessed too frequently. With replication, a client may contact any server that contains a replica of the data, but extra communication is necessary to maintain a consistent state among the replicas.

Optimizing performance using replication and partitioning can be tricky. When the data is replicated, reading the data is inexpensive since a client may choose the closest or least-lightly-loaded server. Updates are expensive because updates must eventually be propagated to every replica. The cost of updates may be reduced by delaying the propagation until the data is read, but this increases the cost of reads, may reduce availability, and must be managed carefully to avoid inconsistencies. Thus, replication provides good performance when access to data is primarily read-only. When the data is partitioned, there is little cost difference between updates and reads. However, there is a cost difference associated with the location of the data. A data reference is less expensive when the data is local to the client's node. A data reference is more expensive when the client must contact a remote node. Thus, partitioning provides good performance when there is high *locality of reference* (i.e., most references are local rather than remote). If the set of data that is frequently referenced changes over time, then good performance can be maintained only if partitioning changes with it.

### 1.2.4. Caching

Application performance can be improved through the use of caching. A *cache* reduces communication costs by remembering previous requests and corresponding replies. If a client wishes to make a request that matches a previous request, the cache may be able to replay the corresponding reply, eliminating the communication and processing necessary to repeat the request to the appropriate server. Offsetting this savings is extra communication and processing needed to remove stale data from the cache: the cache should not replay a previous reply if it knows that the server would return a different reply to the repeated request. A cache is *coherent* if it does not return stale data to the client.

A general-purpose cache of requests and replies is difficult for the communication system to provide because it has little knowledge of the internal semantics of requests and replies, and thus does not know when it is appropriate to delete stale data from the cache. Application programmers can build application-specific caches with knowledge of request/reply semantics, but the cache coherency problem is complicated enough to be the subject of current research, and many applications may need to be changed as new solutions become available.

### 1.2.5. Distributed Shared Memory

Distributed shared memory can make it easier for the application programmer to obtain good performance. Shared memory provides processes with a shared address space; processes sharing an address space may access shared information directly without explicit communication operations such as messages or remote procedure calls. *Distributed shared memory* (DSM) provides a shared address space via software and/or hardware to processes on different nodes that do not physically share memory. A system with virtual memory hardware can use software to provide distributed shared virtual memory.

A well-designed distributed shared memory system can improve application performance through the use of caching. Caching at the memory system level can benefit all applications that use shared memory, and advances in cache coherency algorithms need be implemented in only one system rather than multiple applications. A distributed shared memory cache may be simpler than application-specific caches since it need recognize only four requests: read, write, lock, and unlock. Since read and write requests are built into the hardware, the distributed shared memory cache can often make use of hardware assists (such as virtual memory) to improve performance. Distributed shared memory will not necessarily benefit all applications, however, since it may transfer more data than is needed by the application, and the cost of keeping the cached data consistent may be too large relative to the cost of the processing performed on the data.

Distributed shared memory is an example of data shipping, or the data sharing model. In *data sharing* data is sent to the node where it is to be processed under the assumption that the data will be used there again. Data sharing should be contrasted with the client/server model, or function shipping, in which the function request is sent to the node where the data resides. Data sharing and function shipping are functionally equivalent since each can be expressed in terms of the other. Essentially, data sharing is a specialized implementation of a few remote procedure calls (in the case of distributed shared memory, read, write, lock, and unlock). The messages underlying function shipping can be implemented via message queues that are data-shared. The difference between the two approaches is one of emphasis: function shipping focuses on the flow of control, whereas data sharing focuses on the flow of data.

Data sharing offers the performance benefits of both replication and partitioning. When data is read at a node, it is stored in the cache at that node. Thus, the data sharing model automatically replicates data that is read-only. When data is updated at a node, it is stored at that node and becomes stale in the caches of all other nodes. Thus, the data sharing model automatically partitions data that is frequently updated at one node and not accessed at other nodes. As the frequently-referenced set of data changes over time, partitioning changes with it.

## 1.3. Purpose of the Dissertation

This dissertation explores a new model for building distributed applications: transactional distributed shared memory. There are several reasons for exploring this model.

The client/server model, as provided by systems such as Sun RPC [Sun 88], NCS [Kong et al 90], and MIG [Jones et al 85], has several limitations. Concurrency and failures are hard to deal with, and distributing the data to obtain good performance can be difficult.

Data sharing has been used widely in distributed file systems (such as NFS [Sun 86], AFS [Satyanarayanan et al 85], and DCE [OSF 92]), and is a popular research topic in the form of distributed shared memory [Forin et al 88, Stumm and Zhou 90, Bisiani et al 89, Black et al 89, Li and Hudak 86, Fleisch 88, Cheriton 86, Ramachandran and Khalidi 88].

Among the many systems that have demonstrated the value of transactions are CICS [Kageyama 89], Tuxedo [Unix System Laboratories 92], Camelot [Eppinger et al 91], and Encina [Eppinger and Dietzen 92]. CICS, for most IBM operating systems, is one of the most popular database/data communications control systems. It provides concurrency control and recovery of local databases, and communications facilities for accessing remote databases. Tuxedo, for the UNIX System V environment, has two components. Tuxedo System/T provides facilities to define and manage distributed transaction processing services, including two-phase commit. Tuxedo System/D is a local database that provides concurrency control, logging, backup, and recovery for atomic transactions. Camelot, for the Mach operating system, provides facilities for implementing transactions that may include operations on recoverable virtual memory, remote procedure calls, or nested transactions. Encina, for UNIX and other operating systems, comprises two tiers. The Encina toolkit provides logging, concurrency control, and two-phase commit of distributed transactions. Other Encina products provide management functions, queue and record storage, and interoperation with non-Encina systems.

Transactional distributed shared memory is an interesting topic to investigate since it is the natural research area based on the combination of two interesting topics: transactions and data sharing. Combining data sharing and transactions offers the benefits of both, and the combined system provides opportunities for optimizing performance. For example, the failure atomicity property of transactions may be provided in part by writing a log containing all of the changes made to a data structure. This log write could also serve to update the contents of the caches in a distributed shared memory system, eliminating the need for additional messages to keep caches coherent.

TDSM will not benefit all distributed applications. Transactions may not benefit applications that can tolerate weaker consistency guarantees but not the extra overhead of transactions. Distributed shared memory does not improve the performance of applications if the distributed shared memory system transfers more data than is needed by the application. If the cost of transferring the data is not amortized over repeated accesses, it may be less expensive to ship function requests to the original location of the data. Also, application-specific security restrictions are harder to provide with distributed shared memory. For example, given a set of salaries, the client/server model can easily restrict a particular user to viewing the sum of the salaries but not the individual salaries. Since distributed shared memory can only restrict requests it knows about (read and write), a user must be able to read individual salaries in order to view the sum that is computed from those salaries.

This dissertation makes several contributions. It demonstrates that TDSM is feasible to implement and analyzes a prototype implementation. This analysis is used to show how the combination of transactions and distributed shared memory can be optimized for good performance. The analysis is also used to identify the characteristics of applications for which TDSM is suitable.

## 1.4. Examples

Some examples illustrate the utility of transactional distributed shared memory. In general, applications that can benefit from TDSM have these characteristics:

- There is concurrent access to data by multiple clients.
- The data is important and must not be corrupted.
- To provide adequate performance, some combination of replication, partitioning, and caching is needed.
- Locality of reference may change over time.

A classic application that involves problems of concurrency and failures is an airline database. Reservations are entered into the database by travel agents and reservations agents, and may refer to several flights. Reservations are indexed by passenger name, flight number, and departure date. A flight database records the flight schedule, the cities and aircraft involved, and the seating capacities. For each flight on each date there is a record of the reservations for that flight. Reservations are updated as requested by passengers, as flight schedules are changed, and as flights are flown. Locality of reference arises as the data for a particular passenger or a particular flight is referenced. If two agents concurrently attempt to reserve a seat, the database should not lose either reservation and should not reserve more seats than are available. If a processor fails while storing a reservation, the database should not allow the indices to become inconsistent.

Transactional distributed shared memory supports a solution to the problems of failures and concurrency in the airline database, and provides good performance. To handle the volume of reservations, the data is distributed across several processors. Distributed shared memory automatically partitions the data among nodes to match the locality of reference, and migrates the data as references move. Transactions prevent failures or concurrency from creating inconsistencies during updates.

An approach that has been used to construct the airline database is to store the data on a large, centralized server that can be queried and updated from remote terminals [Gifford and Spector 84]. The disadvantage of this approach is that the centralized server becomes a performance bottleneck as the number of clients increases. Higher throughput can be obtained by splitting the database among several systems. Function shipping and caching can keep the data consistent among systems. But TDSM offers a simpler approach for migrating the airline database to a distributed environment, since the centralized server can run on a TDSM system with few or no changes.

Another example is an authentication database for a distributed operating system. The authentication database typically provides these operations: add user, delete user, authenticate user, and change password. Because the same data is used to authenticate users on all nodes in the distributed system, the database should be replicated to improve availability. When a user password is changed, the new password should supersede the old password on all nodes. Failures and replication make this task more difficult: one of the replicas could fail to receive the updated password. Concurrent updates could make the replicas inconsistent, if a system administrator attempts to change a user password at the same time as the user. Since a given user often uses just a subset of the nodes in the distributed system, the data for the user should be located near the subset, although this should not prevent the user from migrating to a different subset of nodes.

Transactional distributed shared memory is useful for constructing this authentication database. The user data can be stored in a shared memory hash table, with the obvious implementations of add, delete, lookup, and modify. (For protection, the shared memory hash table should be accessible only by a privileged server on each node, which accepts remote procedure calls to perform the requested operations.) Distributed shared memory provides automatic replication for availability and partitioning for locality. Transactions ensure that the replicas are consistent by preventing concurrent updates, and backing out partial updates (with notification to the user) when there is a failure.

Sun Microsystems used an alternate approach to construct the Yellow Pages, a distributed authentication database [Sun 86]. The user database originates as a text file, which a special program transforms into a sparse file accessible via hashing by the dbm library. The original text file resides on a node known as the master server; the sparse file resides on the master server and several slave servers. To add or delete a user, the system administrator edits the original text file on the master server, and then invokes a procedure which rebuilds the sparse file and copies the entire file to each of the slave servers. To change a user password, the user makes a remote procedure call to a process running on the master server, which goes through approximately the same steps of editing the text file and propagating the sparse file. The sparse file is also propagated hourly to compensate for updates missed by any servers. To authenticate a user, the login program locates the nearest server via broadcast, and makes a remote procedure call to that server. The Sun Yellow Pages does not guarantee consistency in the presence of failures and concurrency. A slave server may use an old version of the database if it misses an update due to failure. Multiple servers may respond to the login broadcast, so a user may have to supply a different password depending on which server responds. The Sun Yellow Pages does not take advantage of locality of reference and may have performance problems with frequent updates, since all updates must go through a single server and are then propagated to all servers.

An application similar to the authentication database (and another application for which the Sun Yellow Pages has been used) is a name service for hosts in a network. The name service must translate a host name to an address, and vice versa. Hosts are frequently introduced and removed from the networks. Addresses of deleted hosts may be reused, and the address of any host may change if the host moves from one network to another. In a network the size of the Internet, the data should be partitioned to reduce storage requirements at individual sites, and to increase site autonomomy. The data should be replicated to improve availability. At a given site, only a subset of the host names and addresses are translated, but the subset changes as users come and go at the site. Failures and concurrent updates are common, and can cause temporary inconsistencies if updates fail to reach replicas or if updates reach replicas in a different order.

Transactional distributed shared memory can be used to construct this name service. Two shared memory hash tables perform the mapping from name to address and vice versa. Distributed shared memory automatically replicates and partitions the hash tables to match the locality of reference. (In the Internet, message latencies can be very large, so caching is essential.) Transactions eliminate inconsistencies due to failures or concurrency during updates.

The Internet Domain Name service [Mockapetris 83a, Mockapetris 83b, Mockapetris 86] offers a more complicated approach to constructing a host name service by using a hierarchy of servers. On each node, a local cache manager retains recently obtained translations, and associates a timeout with each translation. When the timeout expires, the translation is discarded from the cache. If a requested translation cannot be found in the cache, the local cache manager consults a root server, which returns the names of several authoritative servers for the given class of name or address. The local cache manager contacts each authoritative server in turn until it obtains the requested translation and associated timeout, which it then stores in its cache. An update can be made at one of the authoritative servers, but does not appear everywhere until timeouts cause previously cached translations to be discarded. Users of the Internet Domain Service are apparently willing to tolerate the temporary inconsistencies that timeouts allow due to failures and concurrency. Users also tolerate temporary service interruptions when cached translations have expired, and all authoritative servers are down or unreachable.

## 1.5. Synergies and Challenges

The combination of transactions with distributed shared memory offers some synergies; that is, the implementation of the two together may offer better performance than the implementation of the two separately.

For example, a common technique used to ensure serializability in transaction systems is locking. Before writing, a process must obtain a write lock, and all other processes must release their locks. A similar protocol is often used to ensure cache coherency: before writing an object, a cache must invalidate all other copies of the object. Since a correct transaction must always lock an object before writing it, a TDSM system could transfer locks and cacheable objects together, instead of using separate protocols for each.

Another possible synergy is to combine transaction logging with the cache coherency algorithm. To ensure failure atomicity and permanence, a common technique used in transaction systems is to write all changes made by a transaction in a log. This log write could also serve to update the caches in a distributed shared memory system, eliminating the need for separate messages to keep caches coherent.

The major challenge in designing a TDSM system is transaction recovery. To simulate atomic execution, a transaction system may need to undo the modifications a partially completed transaction has made to an object in distributed shared memory. However, the DSM object may have migrated from the node where the modification was originally made, so the transaction system must somehow locate both the object and a description of the modification in order to abort the transaction.

To guarantee permanence, a transaction system may need to redo the modifications made by a committed transaction if the modified object is cached on a node that fails. The transaction system must either locate the modified object on another node, or locate an older version of the object, and redo the modifications made by the committed transaction. In a complicated distributed system with many transactions concurrently modifying many objects in distributed shared memory, the task of reliably locating the appropriate version of an object, and/or locating a description of the modifications made by a particular transaction can be very difficult.

The key issue in designing a recovery algorithm for TDSM is the tradeoff between availability and performance. For good performance, objects in DSM must remain cached at the site of use, and modifications to those objects should be recorded in as few places as possible. For high availability, objects in DSM, as well as records of modifications to those objects, should be stored in as many places as possible.

## 1.6. Outline

Chapter 2 provides a overview of distributed transactions and data sharing. Chapter 3 discusses the Camelot and Mach environment in which a prototype transactional distributed shared memory (TDSM) system was implemented. Chapter 4 describes the design and Chapter 5 describes the implementation of this TDSM system. Chapter 6 reports the performance of the implementation. Chapter 7 analyzes the system and its performance, and offers directions for the design of future TDSM systems. Chapter 8 concludes by summarizing the contributions of this work.

# Chapter 2
# Background

Chapter 1 introduced transactions and distributed shared memory as programming models that assist in meeting application requirements on a distributed system. This chapter continues that introduction by addressing the issues involved in implementing transactions and distributed shared memory.

Section 2.1 sets the stage with an overview of distributed computing. Section 2.2 discusses support for transactions, and Section 2.3 describes solutions to the cache coherency problem of data sharing that is central to distributed shared memory. Section 2.4 outlines techniques for distributed concurrency control in conjunction with data sharing. Section 2.5 shows how the technologies can be integrated to produce transactional distributed shared memory (TDSM).

## 2.1. Distributed Computing

In a distributed computing environment, processes need to communicate with each other. In this dissertation it is assumed that processes communicate via *messages*: variable-sized collections of data. Messages are sufficiently general that they may be implemented on top of other inter-process communication abstractions such as datagrams, streams, or sockets. A process may send a message to another process on any node as long as it has a handle (a port, process id, channel, or some other form of name) for the other process. Messages should be delivered, uncorrupted and in order for each (sender, receiver) pair.

*Remote procedure call* (RPC) simplifies the task of constructing, sending, receiving, and interpreting messages through use of the familiar procedure call paradigm. The sender makes a local procedure call to a *stub* routine, which packages the parameters into a message and transmits the message to the recipient. The recipient's service routine unpackages the parameters, calls the appropriate procedure, and awaits the results. The results are similarly packaged and returned to the sender's stub in a reply message.

RPC leads naturally to the *client/server* model. Data is distributed among nodes in the distributed system; access to a each data set is encapsulated in a server. To access data, a client makes a remote procedure call to the appropriate server, which performs the requested function and returns the results.

A server is typically structured in a loop. At the top of the loop, the server awaits a request message. Once the server receives a message, it examines the message to determine the type of request, and calls the appropriate processing routine. When the routine returns, the server sends a reply message and returns to the top of the loop. If the processing routine must delay for some reason, it must save enough state to allow the request to be continued, return to the top of the server loop, and wait for the state to be restored at a later point in time. Because saving and restoring state is cumbersome and error-prone, and to better support multiprocessors, many systems support multiple *threads* of control within an address space.

A distributed application usually pays more attention to failures than a non-distributed program. Because it has more pieces, a distributed environment offers more opportunities for failure, but it also offers more work-arounds for failure since redundant resources are often available. Also, many failures may be temporary due to congestion or resources being temporarily overloaded. For this reason a distributed application may include algorithms for retrying operations that fail, perhaps slightly altering the parameters each try.

Security is harder to obtain on a distributed system, because the kernel on remote systems cannot be trusted, even if the kernel on the local system is trusted. Trusted servers must run on physically secure machines. Because most networks allow eavesdropping, authentication and secure messages must be encrypted.

## 2.2. Transactions

Transactions make it easier to meet data integrity requirements in the face of concurrency and failures. A *transaction* is a sequence of actions grouped into a unit. Transactions provide three properties:

- **Failure atomicity** ensures that if a transaction is interrupted by a failure, any partially completed work is undone.
- **Permanence** ensures that the effects of a committed transaction are not lost due to failures.
- **Serializability** ensures that concurrently executing transactions cannot observe inconsistent states.

If the application's data is in a consistent state, a transaction must transform the data into a new consistent state. (The data may be transformed into a inconsistent state temporarily while a transaction is in progress as long as consistency is restored by the end of the transaction.) If each transaction executes as an atomic, indivisible unit, then data will never be left in an inconsistent state. The *ACID* properties refer to the combination of this *consistency* property with *atomicity*, *isolation* (serializability), and *durability* (permanence).

A transaction *commits* if it runs to completion; otherwise, it *aborts*, and any partial computations are undone. A transaction that performs operations on objects on different nodes in a distributed system is said to be a *distributed transaction*. Transaction systems must take extra care to ensure that all nodes involved in a distributed transaction agree on the transaction's outcome.

Transactions may be *nested* to better support parallelism and limit the effects of failures [Moss 81, Reed 78]. An outermost or *top-level* transaction can initiate multiple, nested transactions which may execute in parallel with each other; the parent is suspended until the *subtransactions* commit or abort. (A top-level transaction with all of its descendants is called a transaction *family*.) A nested transaction may obtain locks that are held by an ancestor, but not a sibling. When a subtransaction terminates, all of its locks are returned to and held by its parent. The effects of a nested transaction are made permanent only when its top-level transaction commits. If a nested transaction aborts, all of its work and the work of its children is undone, and its parent is notified. The parent may then continue processing or abort itself.

Transaction support may be provided by the operating system, by libraries that execute in each process, by a layer in between the operating system and other processes, or by a combination of these techniques. Regardless of where transactional support is provided, this section discusses the implementation of transactions as a single logical layer.

The transaction support layer assists its clients in dealing with failures. Certain failures are assumed to be masked by the underlying hardware and operating system. Most communication failures (corruption, duplication, out-of-order delivery, message loss) are handled by the underlying system through use of checksums, sequence numbers, retransmission, etc.; only network partitions are not masked. Processor failures are recoverable, but they must be detected and the processor must raise an exception or halt.

Storage failures are usually not masked, and failures are detected by checksums. Storage is divided into three classes:

- **Volatile storage** is the main memory of the machine, where objects are buffered as they are accessed. The contents of volatile storage are lost if the system crashes.
- **Non-volatile storage** is where objects reside when they have not been accessed recently. Magnetic disks are usually used for non-volatile storage. The contents of non-volatile storage are lost much less frequently, and always in a detectable way.
- **Stable storage** maintains information despite system crashes and power failures. Stable storage is typically provided in the form of mirrored disks.

The transaction support layer must manage these three classes of storage carefully in order to maintain the failure atomicity, serializability, and permanence properties of transactions. Stable storage is the key abstraction enabling the permanence property.

The transaction support layer helps its clients cope with the failures that are not masked by the hardware or operating system. There are seven major functions that the transaction support layer must provide [Spector 89b]:

- **Transaction management** to coordinate the completion of transactions that span multiple processes.

- **Recovery management** which must restore the proper state after a failure; i.e., backing out partial updates (to ensure failure atomicity) or repeating updates (to ensure permanence).
- **Communication management** tracks the spread of transactions from node to node.
- **Configuration management** stores the configuration of the transaction facility.
- **Concurrency management** coordinates the execution of concurrent transactions to ensure serializability.
- **Buffer management** controls the transfer of recoverable objects between volatile and non-volatile storage.
- **Log management** responsible for recording data in stable storage as directed by the other components.

The components do not perform their work in isolation, but interact with each other as described in subsequent subsections.

### 2.2.1. Transaction management

The transaction management function determines whether a given transaction has committed or aborted and notifies participants of the transaction's outcome. Any participant in a transaction should be able to abort the transaction at any time. Because of its knowledge of transaction outcomes, the transaction management function may be consulted by the concurrency management function to determine whether a lock held by a subtransaction may be inherited by another transaction in the same family.

Because many processes may participate in a transaction, a protocol is needed to achieve consensus on the transaction's outcome. A few of the most common protocols are described below. In these protocols, one participant is selected as the *coordinator*, and the other participants become *subordinates*. To ensure permanence, each participant writes records in a stable storage log.

- **Two-phase commit**. In the first phase, the coordinator asks each of the subordinates to prepare to commit the transaction, and requests a vote from each subordinate. If a subordinate votes to commit, it gives up its right to abort the transaction, and must await word of the transaction's outcome from the coordinator in the second phase. Each subordinate that votes to commit the transaction writes a record of its vote to its log, and forces the log to stable storage. If any subordinate votes no, or if the coordinator does not receive all the votes, the transaction aborts, and the coordinator notifies each subordinate.

  Otherwise, the coordinator writes a commit record to the log, and forces the log to stable storage. It then notifies all subordinates that the transaction committed. If the coordinator is unable to contact a subordinate, the subordinate will remain in the prepared state, retaining control over any objects it has modified. No other transaction will be able to access the objects until the coordinator successfully contacts the subordinate.

  The two-phase commit protocol can be used when the participants are separate nodes, or individual processes within a given node. If the processes within a given node share a common log, the protocol can be optimized by forcing the log to stable storage only once per node.

- **Non-blocking commit.** The non-blocking commit protocol [LeLann 81, Duchamp 89] is a modification of two-phase commit to guarantee that at least one site will not block in the event of a single failure. The prepare message is changed to include a list of the nodes involved in the transaction. While awaiting commit/abort notification, a subordinate may time out the coordinator, become become coordinator itself and finish the transaction. An additional phase is inserted into the middle of the protocol during which subordinates learn how other subordinates voted.

- **One-phase commit.** A transaction that involves only one process can be committed unilaterally by that process with a single log force. As an optimization, the log force can be omitted for a *lazy* commit [Mummert et al 91]. In this case, permanence of effects is not guaranteed until another transaction has later committed using another protocol.

### 2.2.2. Recovery management

The recovery component restores data to a consistent state after a failure, as directed by the transaction management component or by the configuration management component. Recovery is supported for four types of failures:

- **Transaction failure** is when one of the [processes] participating in a transaction decides to abort the transaction. All of the transaction's effects at all of the participating [processes] must be undone. In case of a communications failure, the system can abort transactions whose messages cannot get through.

- **Server failure** is when a [process] crashes due to an unanticipated condition, such as a shortage of resources or a transient software error. All of the active transactions in which the [process] is participating are aborted. When the [process] restarts, the effects of all aborted transactions will be undone, and the effects of all committed transactions will be preserved.

- **Node failure** is when a processor crashes due to a hardware error or a software failure such as the kernel running out of resources. All of the active transactions in which the node is participating are aborted. When the node restarts, the effects of all aborted transactions will be undone, and the effects of all committed transactions will be preserved. This is logically the same as all of the [processes] on a node crashing, except that the node's transaction facility also crashes.

- **Media failure** is when some of the node's non-volatile storage is damaged. The contents of this storage must be restored.

[Eppinger 89, p. 16]

Haerder and Reuter present a taxonomy based on four criteria for classifying recovery techniques [Haerder and Reuter 83].

- **Propagation.** A transaction may modify data in several locations. Is each modification propagated immediately to nonvolatile storage, or are the modifications delayed and propagated atomically as a unit? The first choice may require some modifications to be undone if the transaction aborts. The second choice may require additional space to store both old and new versions of the data.

- **Buffer handling.** Data is usually buffered in main memory before being written to nonvolatile storage. When buffer space becomes full, the system may need to write modified data to disk before the completion of the transaction which made the modification. This data may be written to the home location of the data, or to a temporary location. The first choice complicates recovery of the data after a crash; the second choice requires additional nonvolatile storage.

- **End-of-transaction processing.** Is the buffered data forced to nonvolatile storage at the end of the transaction? If the data is not forced, some additional information must be logged to allow the transaction to be redone after a crash.
- **Checkpointing.** A checkpoint is used to limit the amount of work that the recovery component must do to recover from a failure. To ensure consistency, the system may temporarily halt initiation of new transactions, and await the completion of active transactions before creating a checkpoint. Systems that cannot afford to halt forward processing instead create fuzzy checkpoints that are more complicated for the recovery algorithm to handle.

Because the algorithms for recovery ("backward processing") are closely intertwined with algorithms for forward processing, the next subsection presents examples of specific recovery techniques with buffer management techniques.

### 2.2.3. Buffer management[1]

The buffer management component is responsible for coordinating the transfer of data between volatile and non-volatile storage. It must cooperate with the recovery component to ensure that the failure atomicity and permanence properties of transactions are guaranteed.

An *intentions list* [Lampson 81] can be used to guarantee failure atomicity and permanence:

1. All the changes that a transaction wants to make to data objects are stored in a list.
2. The list is written to non-volatile storage.
3. The transaction management component determines whether the transaction has committed or aborted.
4. If the transaction did commit, change the objects in non-volatile storage.
5. Finally, delete the list.

The list must be carefully written to non-volatile storage so it can be recognized that the list is complete. By deferring updates to objects until the transaction is committed, the transaction is aborted simply by deleting the list. In case of node or server failure, the system restarts and the list is consulted. If the list does not exist or is incomplete, the transaction aborts and the list is deleted. Otherwise, the list is complete, and the system finishes making the updates described in the list and then deletes the list.

*Shadow paging* [Gray et al. 81] is another way to provide failure atomicity and permanence. Non-volatile storage is organized as a tree (e.g., by logical address or as a hierarchical file system). The transaction makes changes to vertices of the tree by writing new vertices in unused locations in non-volatile storage. Changes are incorporated into the tree by writing new versions

[1]The text of this subsection and Subsection 2.2.4 is adapted from [Eppinger 89].

of parent vertices in unused locations. Changes to ancestor vertices continue until the least common ancestor vertex of all the changed vertices is changed. This vertex is then updated in place after consulting the transaction management component. This last update is called an *atomic pointer swap*. In the event of node or server failure, the system restarts. A transaction that has not yet done the atomic pointer swap is aborted. A transaction that has done the atomic pointer swap committed.

A *write-ahead log* [Peterson and Strickland 83, Schwarz 84] is similar to an intentions list with many optimizations. Write-ahead logging uses an append-only *log*, structured as a sequence of variable-length records. Updates to a data object are made by modifying a copy of the object cached in volatile storage and by spooling one or more records to the log. These records contain an *undo* component that permits the effects of aborted transactions to be undone, and a *redo* component that permits the effects of committed transactions to be redone. Write-ahead logging permits an *update-in-place* strategy: when a cached block is copied back to non-volatile storage, it is copied back to the location from which it was previously read. Special care must be taken when copying blocks of a modified object back to non-volatile storage; the blocks cannot be copied back to non-volatile storage until all spooled log records pertaining to those blocks have been written to the log.

In addition to records describing changes to non-volatile storage, records indicating that transactions commit and abort are written to the log. When the system restarts after a crash, the log is consulted. Modifications made by transactions for which no commit record exists are undone. Modifications made by committed transactions are redone. Although write-ahead logging is more complex, it offers several performance advantages over the other techniques. It does not scatter data all over non-volatile storage; it allows multiple transactions to execute simultaneously using a common log; and system restart using a write-ahead log can be done with as little as one scan of the log.

Write-ahead logging also allows changes to non-volatile storage to be buffered. This allows expensive updates to non-volatile storage to be grouped together and amortized. All accesses to non-volatile storage are done via volatile primary memory. By caching blocks of non-volatile storage in volatile storage, the number of non-volatile reads and writes can be reduced. The blocks used in the volatile cache are called the buffer pool, which is managed by the buffer manager. The buffer manager must coordinate the transfer of blocks between volatile and non-volatile storage with log writes to enforce the write-ahead log invariant.

### 2.2.4. Log management

Generally, there are two ways of describing updates in log records: logically or physically. Logical or *operation* logging places a description of the logical operation being performed into the log record. The description may describe how to redo the change, or how to redo and undo the change. When physical logging is used, one or two bit patterns are stored in the log record.

*Old-value/new-value* logging includes two bit patterns: a before image and an after image. *New-value* logging includes only one bit pattern, an after image. Hybrid approaches are also possible: for example, operation logging for undo, and new-value logging for redo. The choice of which values to write in the log can affect the buffer management strategy. New-value-only logging requires that modified blocks not be written back to non-volatile storage until after the transaction commits.

A separate log can be maintained for each process on a node, but if the appropriate interface is provided, a common log can be used by all processes on the same node. All the log records spooled by processes on the node can be written to the log together, reducing the number of expensive log writes. As mentioned previously, the commit protocol can also be optimized if there is a common log.

### 2.2.5. Concurrency management

There are four major techniques used for concurrency control in transaction systems [Bernstein and Goodman 82]:

- **Two-phase locking.** A transaction must obtain a lock in the appropriate mode on each object it wishes to access. If another transaction already holds the lock in a conflicting mode, the requesting transaction must wait. After a transaction releases a lock, it may not obtain any more locks. (Thus, there is a growing phase as locks are obtained by the transaction, and a shrinking phase as locks are released.) Timeouts, deadlock avoidance, or deadlock detection are used to prevent transactions from waiting forever.

  With *read/write* locking, a transaction must obtain a read (shared mode) lock before reading an object, and a write (exclusive mode) lock before writing an object. This allows multiple transactions to read an object concurrently, but only one transaction to write the object.

  *Type-specific* locking [Korth 83, Schwarz and Spector 84] offers the possibility of increased concurrency on abstract data types when operation logging is used. For example, a counter might support an increment mode lock which conflicts with read or write locks, but is compatible with another increment lock. Clearly, the order in which two increments occur does not affect the final result.

- **Timestamp ordering.** Each transaction is assigned a timestamp as it is created. Each access by a transaction to an object is marked by the transaction's timestamp. But the transaction is not allowed to access an object (i.e., the transaction is aborted) if the object has been accessed by a transaction with a later timestamp.[2] Thus, all accesses to objects occur in timestamp order.

- **Serialization graphs.** The system builds a graph of dependencies between transactions (a *serialization graph*) as the transactions execute. (Transaction $A$ depends on transaction $B$ if $A$ reads an object that was written by $B$.) The system aborts a transaction before allowing it to access an object if the access would cause a cycle in the serialization graph.

[2]To avoid aborts, the system may attempt to delay transactions with later timestamps.

- **Certifiers.** No checking for conflicts between transactions is done until the transaction enters the prepared state. When a transaction enters the prepared state, the system uses one of the above techniques (usually locking) to determine whether any of the transaction's accesses conflicted with another transaction. If a conflict is found, the transaction is aborted. This technique is usually called *optimistic* concurrency control.

### 2.2.6. Communication management

The task of the communication management component varies, depending on what services are provided by the underlying system. It may provide a name service that creates communication channels to named processes. It may provide logical clock services. It may forward messages between nodes, handling retransmission as necessary.

In a system based on the client/server model, the communication management component extends the remote procedure call concept to the context of transactions. A *transactional RPC* is an RPC made within the scope of a transaction. The communication manager may spy on transactional RPCs to learn which processes are involved in a particular transaction. During commit or abort processing, it presents a list of processes to the transaction management component.

### 2.2.7. Configuration management

The configuration management component provides the memory of the transaction system about itself. The configuration management component stores information about recoverable objects and the processes that may access them. A user interface allows authorized users to create, delete, start, restart, or shutdown objects or processes. The configuration management component controls orderly shutdown of the system, and restarts the system after a crash.

## 2.3. Data Sharing

In *data sharing*, data is sent to the node where it is to be processed under the assumption that the data will be used there again. Data sharing should be contrasted with the client/server model, or function shipping, in which the function request is sent to the node where the data resides. Data sharing offers two performance advantages over function shipping through the use of caching:

- Data that is read-only is automatically replicated at each node where it is read.
- Data that is frequently updated at a single node is automatically partitioned at that node.

Unfortunately, the caching of data sharing leads to a difficult problem: how to keep data cached at multiple nodes consistent, or the *cache coherence* problem.

Solutions to the cache coherence problem have appeared in several domains. A limited form of the coherence problem appears in any (single processor) system that places a cache between the processor and memory, since the cache must be kept coherent with memory. The more general problem occurs in multiprocessors where each processor has its own cache. Another domain is that of non-uniform memory access time (NUMA) multiprocessors, in which shared data may be replicated in several local memories, or migrated from one local memory to another to improve performance. Closely related to this domain is the area of distributed shared memory, where a non-uniform memory access machine is simulated by a set of hosts connected by a network.

Two basic approaches have been used to address the coherence problem. In *directory* techniques, information about where data is replicated is kept by a centralized manager, which must be involved each time the data is replicated or migrated. In *snoopy* techniques, information is broadcast to all processors when data is replicated or migrated. In the multiprocessor cache domain, both approaches have been used. Because of the poor scalability of broadcast methods, distributed shared memory systems rely almost exclusively on directory techniques.

Basic to any approach is the consistency model that it assumes. Most work in the area has been based on the uniprocessor memory model, in which a load from a given location always returns the most recent value stored there. Recently, consistency models that relax this constraint have been receiving attention due to their promise of improved concurrency. These models are addressed briefly in Subsection 2.3.3.

### 2.3.1. Directory methods

Directory methods are so named because they maintain a central or distributed directory indicating which processors have a copy of each datum. In general, a number of processors may have a read-only copy of a given datum, but only one processor is allowed to have a writable copy of the datum.

To obtain exclusive access to a block of data to service a write request, a processor must contact the manager for the block. If the block was shared (read-only) by several processors, the manager contacts each one to discard (*invalidate*) the block. The manager then returns a copy of the block to the requesting processor. If the block was held exclusively by a single processor, the manager contacts that processor, which either returns the block to the manager for forwarding to the requester, or sends the block to the requester directly.

Li and Hudak [Li and Hudak 86] propose a taxonomy of directory methods for the distributed shared memory domain based on the location of the directory manager. The manager may be *centralized* (e.g., main memory as is done in most multiprocessor caches), or *distributed* (i.e., each processor manages a subset of the data). A distributed manager may be assigned a *fixed* subset of the data, or the assignment may vary *dynamically*. (In a dynamic scheme, the processor which last wrote the data is usually the manager.) To locate the current manager in a dynamic distributed directory management scheme, a processor may resort to broadcasts, or it may have to follow a chain of forwarding pointers from the last known manager.

Agarwal et al [Agarwal et al 88] propose a taxonomy of directory methods for multiprocessor caches based on two criteria: the number of indices *i* used to indicate which caches contain a copy of the data, and whether broadcast is ever used to invalidate cached data. If broadcast is not allowed, then no more than *i* caches may contain the data; otherwise, broadcast is used only when more than *i* caches contain the data. Clearly, broadcast must be allowed when *i* is zero.

Many distributed shared memory systems use a fixed distributed manager. Examples of these systems include Li and Schaefer's hypercube implementation [Li and Schaefer 89] and Mach's **netmemoryserver** [Forin et al 88]. Mirage [Fleisch and Popek 89a] uses a fixed distributed manager, but introduces a time window Δ to reduce thrashing when several processors are contending for a page. If an invalidation request is received before the page's Δ has expired, the request is rejected, and must be retried later.

Clouds [Ramachandran and Khalidi 88] provides a distributed shared memory using a variation of the fixed distributed manager technique. A Clouds process explicitly locks a segment before accessing it, and unlocks the segment when it is through. Clouds uses this concurrency control information to implement coherence: when a process unlocks a segment, the segment is *discarded* and the manager is notified, eliminating the need for subsequent invalidation messages from the manager when another process wishes exclusive access. In addition to coherent read and write access described above, Clouds allows "weak read" access (which returns a copy of the segment without waiting for any writers to return the most up-to-date copy of the segment), and a form of exclusive access which requires invalidation and cannot be used in conjunction with any of the other forms of access.

Gray and Cheriton [Gray and Cheriton 89] propose a timer-based mechanism called *leases* for maintaining coherency of cached files in the V system. Each cached datum has an associated lease term. During the lease term, a processor may read the cached datum with impunity, and the manager will not allow any processor to write the datum. After the lease expires, a processor must contact the manager to extend the lease before reading the datum again. To write a datum, a processor must forward the write request to the manager, which will either contact all the leaseholders to terminate their leases prematurely, or wait for the leases to expire, and then perform the write.

In a NUMA architecture, the time to access a given location depends on whether the location resides in memory local to the processor, or in a remote memory. Since remote memory access times are significantly larger than local memory access times, it is often desirable to replicate read-only data, or migrate read-write data to the processor doing the writing. Replication is desirable when the data is seldom written. Migration is desirable when there is considerable locality of reference. Two recent NUMA systems apparently use a centralized directory, accessible by any processor, to maintain coherency of replicated pages.

- Bolosky et al [Bolosky et al 89] use the standard invalidation technique when a processor wishes exclusive access to a page. After a page has been migrated more than a certain number of times, however, the page is frozen at its present location, and becomes ineligible for subsequent replication or migration. All subsequent accesses must be performed via the hardware remote memory access mechanism.
- In PLATINUM [Cox and Fowler 89], a timer is used to determine when to freeze a page. The timer is started when the page is invalidated. If the page is invalidated a second time before the timer expires, the page is frozen. A second timer with a much larger interval unfreezes the page, to allow for varying access patterns.

Dias et al [Dias et al 87, Dias et al 89] describe a centralized manager using a slightly different algorithm. Each time a cached block is accessed, a processor must contact the manager to see if the cached block is still valid. The manager returns a yes/no reply, and makes a note that the processor is using the block. When a processor wishes exclusive access to a block, it first obtains an exclusive lock for the block (thus insuring that no other processor is actively using the block). The processor writes the block back to secondary storage and releases the lock when it has finished, and notifies the manager. The manager updates its internal data structures to indicate that no other processor has a valid copy of the block.

## 2.3.2. Snoopy methods

Several snoopy methods have been implemented in multiprocessor caches where each cache monitors traffic on the system bus to update the state of the blocks it has cached. No single cache or memory has complete information about where a block is cached. Since main memory holds a copy of all blocks, it must participate in the coherence protocol, but most protocols treat memory as a special case. In *write-through* techniques, a cache write is immediately propagated to main memory. To reduce bus traffic, *write-back* techniques delay the update of main memory until the modified block is flushed from the cache.

Write-through with invalidate is used in several commercial multiprocessors [Agarwal et al 88]. Each time a block is written to a cache, the store is sent to main memory, and all of the other caches invalidate their copy of the block. If a different processor accesses the block, a cache miss occurs, and the block is loaded from main memory.

Archibald and Baer [Archibald and Baer 86] describe several snoopy techniques that use write-back. Briefly, these are:

- **Write-once.** The first time a block is written, it is copied back to main memory, and all other caches invalidate their copies of the block. Subsequent writes by the same processor modify only its associated cache; the block is written back to memory when it is flushed from the cache. A cache miss may be satisfied by memory or by another cache.

- **Synapse.** When a processor writes a block, its cache broadcasts an invalidation message to all other caches. Subsequent writes by the processor occur only in cache. A cache miss is always satisfied by memory, but the reply from the memory may be delayed while the cache holding a modified copy of the block writes it back to memory.
- **Berkeley.** Writes are treated as in the Synapse method. A cache miss, however, may be satisfied by the cache holding a modified version of the block, and no writeback to memory occurs in this case.
- **Illinois.** This method modifies the Berkeley method by including a simultaneous writeback to memory when a cache miss is satisfied by another cache holding a dirty copy of the block.
- **Firefly.** Unlike the previous methods, the Firefly technique allows several caches to have a modified copy of a given block (although all copies are guaranteed to be identical). Cache misses may be satisfied by another cache or memory; a special bus line indicates whether the block is present in more than one cache. When a processor writes a block, its associated cache will broadcast the modification to all other caches and memory if the block is present in any other cache. If the processor has an exclusive copy of the block, no broadcast takes place.
- **Dragon.** Like the Firefly technique, Dragon allows multiple writers of a given block. However, writes when a block is shared are broadcast only to the other caches and not to memory. A writeback occurs later when the block is flushed from the cache.

### 2.3.3. Other methods

Bus-based multiprocessor architectures are usually limited to a small number of processors. Several researchers have proposed a hierarchy of caches to remove this limitation. Cheriton et al [Cheriton et al 89] describe a hierarchy using a centralized directory on each level. Ramachandran and Mohindra [Ramachandran and Mohindra 88] propose a suite of hierarchical protocols based on several snoopy methods: write-once, Berkeley, Illinois, Firefly, and Dragon.

Hsu and Tam [Hsu and Tam 88] propose a method for coherency control of distributed recoverable virtual memory that is similar to the Berkeley cache coherency protocol. A fixed number of *shares* are issued for each block. To obtain shared (read) access to a block, a processor broadcasts a request for the block, and receives a certain number of shares. To obtain exclusive (write) access to a block, a processor broadcasts a request, and waits until it receives all of the shares in reply. The protocol includes a facility for regenerating shares when a processor crashes or is otherwise inaccessible.

Bisiani et al [Bisiani et al 89] simulate the use of timestamps to provide coherence in a distributed memory machine. Each write operation is marked with a timestamp provided by a central clock, and is propagated to all other copies of the datum, where the operation may be buffered or processed immediately based on its timestamp. Reads that do not require coherency take place with no delay. The system may delay a coherent read until the (dynamically estimated) propagation delay has expired.

All of the previously mentioned techniques, except Bisiani et al, rely on a consistency model known as *sequential consistency*, after Lamport's definition [Lamport 79]:

[A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Sequential consistency requires that the loads and stores performed by a parallel program return the same results as some interleaving of the execution of the parallel processes on a sequential machine.

Other consistency models that have been proposed, in order of decreasing constraints, include:

- **Processor consistency** [Goodman 89] requires that stores issued from a processor must be observed in the same order than they are issued. However, the order in which stores from two processors occur, as observed by themselves or a third processor, may differ.
- **Weak consistency** [Dubois et al 86] guarantees that memory is consistent only at specific synchronization points. All previous accesses must be completed at the beginning and end of each critical section. Within a critical section, the ordering of loads and stores by different processors may occur in any order.
- **Release consistency** [Gharachorloo et al 90] extends weak consistency by classifying synchronization points into *acquire* (lock) and *release* (unlock) accesses. Release consistency ensures that all previous shared data updates are performed before a release of a synchronization variable is observed by any processor. However, accesses following a release need not be delayed for the release to complete.
- **Entry consistency** [Bershad and Zekauskas 91] extends release consistency by tying access to data within a critical section to the guard variable that must be acquired to enter the critical section, and noting whether the guard variable is acquired in exclusive or non-exclusive mode.

## 2.4. Distributed Concurrency Control

Local filesystems often provide locking primitives for concurrency control. Several distributed filesystems have extended these primitives for distributed operation.

- Sprite [Welch 90] implements UNIX flock() advisory locks, with all lock requests forwarded from the client to the fileserver. In the case of a conflict, the fileserver saves the client's process id, and replies immediately with an error indication. When the conflict is resolved, the server notifies the client, which then retries the request.
- The DCE DFS [Bottos 92] also implements UNIX locks via a token scheme. Read and write lock tokens covering a byte range of the file can be cached at each node. As long as a node holds the appropriate tokens, it can grant lock requests without contacting the fileserver. The fileserver asks for the release of a token when another node requests a token that conflicts with the outstanding token.
- The V system [Cheriton 87, Cheriton 88] provides both block- and file-level locks at the fileserver. File locking may be specified on open requests. Block locking may be done explicitly via lock and unlock calls, or implicitly on each read and write request. Blocks are locked automatically when caching is used.

- Digital's VAXcluster system under VMS includes a shared filesystem and locking primitives that operate on a hierarchical lock name space [Snaman and Thiel 87]. Lock management is separate from the filesystem, and may be used to synchronize access to any shared resource in the VAXcluster. Management of the lock name space is partitioned between nodes via a directory scheme; a separate connection manager notifies all lock managers when a node enters or leaves the cluster so that the lock name space may be repartitioned. When a process first requests a lock on a resource, the name of the resource is hashed to determine which lock manager contains the directory for the resource. This lock manager then forwards the request to the manager for the resource. (The first node to request the resource becomes its manager. When all locks on the resource are released, no node is manager.) The lock interface includes an asynchronous notification mechanism that allows a VMS process to be notified when a second process requests a lock held by the first.

Hastings [Hastings 90] implemented a distributed lock manager for the Camelot distributed transaction facility. Each node has its own lock manager which manages the portion of the lock name space that is (statically) assigned to the node. Locks are requested by transactions, but cached by servers so that a subsequent request by another transaction in the same server may be processed by the server without contacting the lock manager. A call-back to the server is used by the lock manager when a cached lock is requested by a different server.

Concurrency control and coherency control are seldom discussed in isolation. Early work in the distributed shared memory domain did not provide special techniques for synchronization, but relied on standard multiprocessor synchronization techniques such as test-and-set instructions and spin locks [Li 89, Forin et al 88, Fleisch and Popek 89b]. More recent work uses separate messages or RPCs to provide semaphores for synchronization [Li and Schaefer 89, Cheriton 88]. The proposed multiprocessor VMP-MC [Cheriton et al 89] includes separate lock and unlock bus operations with a queue of waiters to eliminate spin-waiting.

As noted previously, Clouds utilizes locking to implement its coherency control algorithm [Ramachandran et al 89]. When a process locks a segment locally in preparation for accessing it, the kernel will request the segment from the manager. When the process unlocks the segment, the kernel discards it. Clouds also provides separate P and V semaphore operations. These operations are forwarded to the manager of the appropriate segment. The manager performs the operation without transferring the segment containing the semaphore to the requester.

Dias et al [Dias et al 87, Dias et al 89] also rely on locking to implement their coherency control algorithm, but use a centralized lock manager. Lock requests may be combined with coherency control requests, since the lock manager processes both types of requests.

## 2.5. Integrated Technologies for TDSM

The previous sections have discussed how to implement transactions, and how to implement distributed shared memory. The question remains, how can transactional distributed shared memory be implemented?

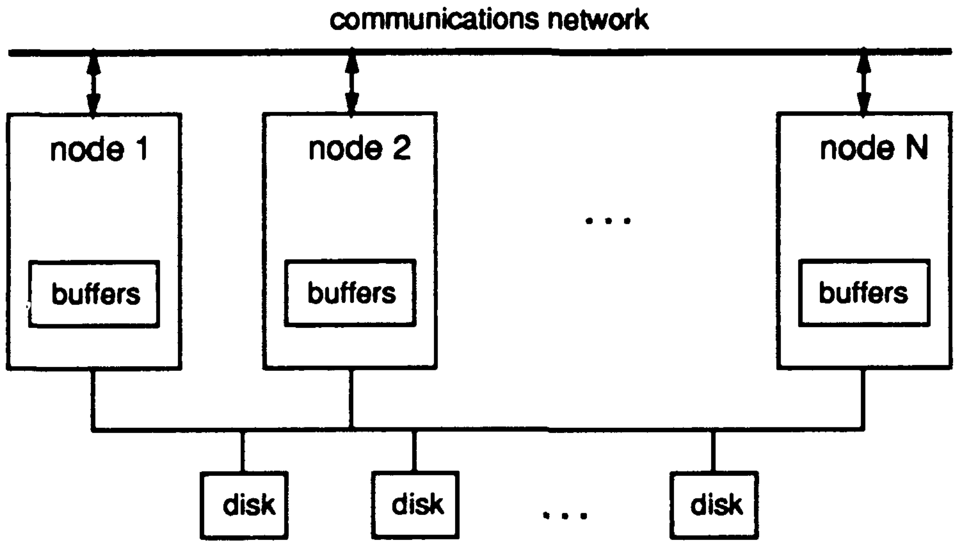


**Figure 2-1:** Data sharing with shared disks

Related work on this question falls into two categories. Several researchers have investigated the question in the context of a closely-coupled distributed system in which all nodes are directly connected to each disk. (See Figure 2-1.) The interconnection network is used to coordinate access to the disks, and may be used to transfer blocks between nodes.

The second category of related work is in the context of general purpose distributed systems, as used for the system described in this dissertation. However, unlike the system described in this dissertation, the related work in this category was not implemented, and the design was validated solely via modeling and simulation.

### 2.5.1. Shared disks

Most large mainframe transaction processing systems are multiprocessor-based. One way of coupling multiple processors is via main memory; another is via a network as described in this dissertation. A third approach, taken in IMS and other commercial systems, is via shared disks. Dias et al [Dias et al 87] cite the following advantages of this data sharing approach over function shipping:

- Since every node can access the data, availability of the system as a whole can be improved.
- Commit protocols are simpler.

- Load balancing is easier.
- Migration from a single system is much simpler, since the database does not need to be partitioned.

Yu et al [Yu et al 85] describe a system based on IMS data sharing with distributed lock management. Concurrency control is at the record level for read access, and at the block level for update access. Coherency control is based on broadcast invalidation. Invalidation requests are queued and transferred with lock requests at regular intervals. When a transaction completes, its update locks are held until the associated invalidation requests have been acknowledged.

In [Yu et al 87], several possible performance problems and design issues associated with data sharing systems are noted:

- **Data contention.** A data sharing system supports more concurrent transactions than a single system, so contention for the same data may be a problem.
- **Concurrency control overhead.** When data is shared, locks must be shared, which may lead to a greater overhead to obtain a lock.
- **Lock granularity.** A coarser lock granularity means fewer lock requests (and less locking overhead) at the possible cost of higher contention.
- **Data obsolescence.** Transaction systems buffer data to reduce I/O rates. When a node updates shared data, all other buffered copies of the data must be invalidated, leading to higher I/O rates.
- **Coherency control overhead.** The system incurs additional overhead to generate and process buffer invalidation requests.

In [Yu et al 86], some of these problems are addressed through the use of transaction routing: by directing an incoming transaction to the node where the data it needs is already buffered, the number of buffer invalidations (and associated lock overhead) can be reduced.

The primary focus of the work of Yu and Dias et al in shared disk data sharing systems is in coherency and concurrency control, comparing performance to function shipping systems, and evaluating the effects of different locking schemes. They do not address the transaction management, recovery management, or log management issues raised in this chapter.

Rahm [Rahm 89] concentrates on recovery (and associated transaction management, log management, and buffer management issues) in the shared disk environment. He assumes locking for concurrency control, physical logging, and update-in-place for buffer management.

Among the issues to consider when designing node recovery are:

- Lock tables describing which node is holding each lock may be lost in a crash.
- Coherency control tables describing which node is currently holding a block may similarly be lost.
- Blocks may be in use on nodes that are still running.
- Blocks may be transferred between nodes via the shared disk, or more quickly via the network.

According to Rahm, most existing shared disk data sharing systems force modified blocks to disk at the end of each transaction. This strategy simplifies node recovery, since no modifications ever need to be redone, and also simplifies coherence control, because the disk always contains a transaction-consistent copy of each block. However, Rahm advocates the alternative strategy of not forcing modified blocks at the end of transaction; instead, only log records are forced to disk. This strategy offers higher performance by reducing I/O rates and I/O waiting time.

The granularity of logging is another issue. If entire blocks are logged, recovery is simplified, since the correct block is immediately available. However, because block-level logging generates enormous amounts of log data, most systems use record-level logging. This complicates node recovery, since a block may migrate through several nodes, each of them modifying a different record, without the block being forced to disk. If the node holding the block crashes, modifications performed by preceding nodes may have to be redone. Or, if a preceding node writes a modification to disk, the transaction which made the modification aborts, and the final node holding the block crashes, the aborted modification may have to be undone. In general, redoing and undoing these intermediate modifications requires access to the appropriate log records in chronological order (e.g., via a global log).

To reduce the number of messages, coherency control may be tied to concurrency control: a record is transferred when its corresponding lock is obtained. Unfortunately, this optimization causes the granularity of locks to become an issue. If several records are stored in the same block, then multiple nodes could be modifying (different records in) the same block simultaneously. This means that several partially updated blocks must be merged before writing a block to disk. To avoid this problem, most existing shared disk data sharing systems support only block-level locking.

Given these issues, Rahm proposes the following architecture for TDSM in a shared disk environment:

- **Concurrency management** uses a distributed lock manager, with each node responsible for a fixed subset of the locks. (Essentially, responsibility for the database is partitioned among the nodes, although the entire database is accessible from each node.) The lock manager may allow multiple nodes to retain read authorization; this authorization is revoked when a transaction requests the lock in write mode.
- **Buffer management** relies on the lock manager to maintain cache coherency. When a node requests a lock, the lock manager's reply includes a copy of the corresponding block (if the node's cached copy is invalid or non-existent). When a node releases a lock, it sends any modified block directly to the lock manager's node. If the lock manager's node runs out of buffer space, it writes the modified block to disk, and will tell subsequent requesters to read the updated block from disk.
- **Log management** uses new-value logging. Since the lock manager's node receives all modified blocks, log records are also recorded at the lock manager's node. Thus, the global log is partitioned among the nodes in exactly the same manner as locks are partitioned.

- **Transaction management** uses a special two-phase commit protocol. In the first phase, log data is forced to the local log, and locally controlled locks are released. In the second phase, remote data is processed: (1) Retain redo log records in a separate buffer. (2) Send modified blocks and log records with unlock request to lock manager node. (3) When lock manager acknowledges request, clear separate redo buffer. The lock manager node writes out the log records it has received before acknowledging the request.

- **Recovery management.** Transaction and server failure are handled locally, since log records are retained on the node where the transaction runs. When a node fails, the surviving nodes must handle recovery for the partition assigned to the failed node. The shared disk environment simplifies this task, since the log for the failed node still can be read by any surviving node.

  One node is chosen to run the recovery algorithm for the failed partition. It broadcasts a message that halts all activity in the failed partition, and causes all nodes to forward lock information and buffered log records for the failed partition to the recovery node, which appends the log records to the failed node's log. Each node discards cached blocks for the failed partition, unless the node holds a valid read authorization. The recovery node scans the failed node's log to determine which transactions failed to commit, and which blocks are missing committed modifications. It processes redo records on the appropriate blocks for committed transactions, and writes out the corrected blocks. The recovery node also examines the failed node's log to discover if any remote modifications that were committed must be retransmitted to other nodes.

  Finally, the failed node's partition is reassigned to another node(s), and processing in the partition may continue. When the failed node restarts, processing must again halt while the responsibility for the partition is reassigned.

Rahm's design is quite similar to the architecture described in this dissertation, although it was developed independently and in a different environment. Rahm's architecture benefits from the shared disk environment in that (1) the lock manager node may tell a requesting node to read a block directly from disk, and (2) the log from a failed node is still accessible on disk by the surviving nodes. Rahm does not appear to have implemented his design, nor does he include a performance evaluation. Rahm does not directly address communication management or configuration management.

### 2.5.2. Hsu and Tam

Hsu and Tam have proposed several designs for TDSM in the distributed system environment.

In [Hsu and Tam 88], they propose a design that requires a reliable broadcast mechanism which supports failure atomicity, message synchronization, a system-wide logical clock, and detection of node failure. The design does not handle the problem of network partition.

- **Buffer management.** Coherency is maintained via a *share* mechanism. A fixed number of shares is associated with each page. To read a page, a node broadcasts a request to become a shared owner. Exactly one existing owner will reply, conveying a number of shares in the page to the requester. To write a page, a node broadcasts a request to become an exclusive owner. All of the existing owners must reply and convey all of the shares to the requester. If the requester does not have a current copy of the page, an update (containing either the entire contents of the page, or a series of incremental updates) is transmitted along with the shares. No two nodes can attempt to become exclusive owner simultaneously because of the timestamps provided by the reliable broadcast mechanism. If a node is unable to collect all of the shares within a given time limit, it enters a *reformation* phase in which it asks all operational nodes to give it the authority to regenerate missing shares.

- **Concurrency management** uses hardware assists to provide locking via faults. Each read or write access causes hardware lock bits and transaction id registers to be compared. If this check fails, a fault into the lock manager is generated. The lock manager checks the ownership of the page, and invokes the coherency control mechanism if necessary. The coherency control algorithm will not grant shares to a node requesting shared ownership if the page is write-locked. It will not grant shares to a node requesting exclusive ownership if the page is read- or write-locked.
- **Log management** is apparently based on physical logging of entire pages. Sites are grouped into predefined backup groups. Each node in a backup group monitors broadcasts by other group members, so that updates are replicated on each node in the group.
- **Transaction management** depends on reliable broadcast. To commit a transaction, a node must first broadcast the changes made by the transaction to all members of its backup group. Then it performs local commit processing: writing dirty pages and a commit record to nonvolatile storage, and finally dropping locks held by the transaction. The commitment point is the atomic broadcast to the backup group.
- **Recovery management.** Transaction and server recovery are apparently handled locally, relying on write-ahead logging. Also, before granting a write lock on a page, a before image of the page must be logged; apparently, this allows an update to be undone if the transaction aborts.

  Node failure is handled via the backup group mechanism. Sites are ranked within each backup group. Each node in a backup group monitors broadcasts to know when a group member becomes an exclusive owner. If an exclusive owner fails (as detected by the reliable broadcast mechanism), the highest-ranked backup node takes over as owner. Because a node broadcasts updates as part of its transaction commit protocol, each node in the backup group is guaranteed to have transaction-consistent data, and no further recovery action is necessary.
- **Communication management** must provide reliable broadcast (with failure atomicity, synchronization, detection of node failure, and total ordering via a system-wide logical clock) if not supported by the underlying system. All message traffic must be monitored to see if the message refers to a member of the node's backup group.
- **Configuration management** must allow backup groups to be defined and ordered, in addition to the tasks listed previously.

Hsu and Tam note several research issues with their design.

- **Garbage collection.** When should a node throw away pages it is not actively using? This seems to be part of the larger question (not answered by Hsu and Tam) of share management: how does a node decide how many shares to give to a requester? What does it do if it has only one share left, and it wishes to retain access to the block?

- **Granularity of locks and log records.** Logging is done on blocks of 128 bytes; locking uses hardware assists which restricts the granularity of locks to the hardware page size. In general, neither of these sizes may be optimal.
- **Locality management.** How can programmers structure their applications to provide the locality assumed to exist by the underlying system?

The decentralized nature of Hsu and Tam's design makes it appear very attractive. However, Hsu and Tam emphasize that the design is a straw man, and they do not evaluate its performance. This is the key difficulty: reliable broadcast does not scale. Although Hsu and Tam claim to eliminate distributed commit, essentially distributed commit has been forced down into the reliable broadcast protocol, which is invoked on every message. Another potential difficulty with the design is the backup group mechanism. It appears that only the owner node actually writes updates to the log. If the owner fails, the highest-ranked backup node takes over as owner, but it does not have access to the log.

In a later design, summarized in Tam's dissertation [Tam 91], Hsu and Tam dismiss the broadcast technique, and concentrate on reducing the time needed to perform node recovery.

- **Concurrency management.** Two algorithms for concurrency management are discussed. The 2PL-MC algorithm (very similar to the algorithm described in this dissertation) uses a distributed lock manager, with each node responsible for a fixed subset of the locks. A node may cache locks, so that it may grant a cached lock to a transaction without contacting the lock manager again.

  The 2PL* algorithm relies on the coherency control mechanism to ensure that a page being written is stored on only one node, while a page being read may be stored on many nodes. As long as a node has read or write permission for a page, it may grant locks in the corresponding mode for records stored on that page.

- **Buffer management** uses a fixed distributed manager to maintain coherency. The data is partitioned in the same manner as the locks. Each block has an *owner* node which may vary over time, and which is the only node allowed to write block. A *locator* node, which is fixed for each block, always knows which node is the current owner. To read a block, a node sends a message to the locator node, which forwards the request to the owner. If the owner is not writing, it returns a copy of the block to the requester. To write a block, a node sends a message to the locator. The locator notifies the owner, and records the new ownership of the block. The owner invalidates any existing readers, and sends the block to the new owner. Each time a block is transferred or invalidated, the action is logged by both the sender and the receiver. Each node also periodically checkpoints the information it has about block owners and readers.

  The database is assumed to fit in main memory, so blocks are never written to non-volatile storage except as part of logging.

- **Log management** uses new-value logging of entire blocks, to avoid the problem of merging partial updates from different nodes during recovery.
- **Recovery management.** Transaction and server recovery are apparently handled locally, relying on write-ahead logging.

  Node recovery depends on the logging and checkpointing of coherency actions. Apparently, while a failed node is down, no other node may obtain a block for which the failed node is owner or locator. Once the failed node restarts, it reads its latest checkpoint, and replays the log to restore its directory of block owners and readers.

Tam's dissertation does not address the transaction management, configuration management, or communication management issues raised in this chapter. It does not give transaction or server recovery algorithms, nor does it describe how a failed node restores its data to a transaction-consistent state after it restores its directory of block owners and readers. The architecture does not appear to have been implemented, although the performance of several algorithms is evaluated via modeling and simulation. The buffer management algorithm has been designed to allow fast node recovery, at some additional expense during forward processing.

Tam's dissertation also mentions a method for TDSM using optimistic concurrency control that is further described in [Bellew et al 90]. One node serves as the *validator* which must be contacted at the end of each transaction to obtain a transaction id and commit the transaction. Each of the other nodes acts as a *home* node, storing part of the database.

- **Concurrency management** uses optimistic concurrency control. Each page has a *timestamp* which is the id of the last transaction to update the page. A transaction is allowed to compute with whatever data is cached at the node where it is running. At the end of transaction, the node presents a list of timestamps for pages it has read, along with all the pages it has modified, to the validator. The validator compares the submitted timestamps with the most recent timestamps; if any of the submitted timestamps are out-of-date, the transaction must abort.
- **Transaction management** uses the validator to assign transaction ids. If the submitted timestamps are valid, the validator allocates a new transaction id, and stores it with each updated page.
- **Buffer management** relies on the validator to periodically broadcast updated pages. When a page is received by any node that has the page cached in main memory, the node updates the cached copy; otherwise, the page is discarded. When a node wishes to free a cached page that is not in use, it writes the page to disk if it is the home node for the page, otherwise it merely throws away the page.

  There are two cases to consider when a node wishes to access a page that it does not have cached in main memory. If the node is the home node for the page, it simply reads the page from its local disk. Otherwise, the node requests the page from the home node, which will locate the page in main memory or on disk, and return it along with the associated timestamp.

Log management, recovery management, communication management, and configuration management issues raised in this chapter are not addressed by this architecture; also note that in general, techniques using broadcast do not scale well. The integrity of updated pages is certainly a issue, since they may be lost when the validator or home node fails. Failure of a home node makes many pages unavailable, and failure of the validator may make the updated pages it holds unavailable. As with [Tam 91], the architecture was simulated and appears not to have been implemented. The simulation compares optimistic concurrency control to 2PL*. When locality of reference is high, the optimistic method offers lower throughput than 2PL* due to the cost of validating and broadcasting updates. When locality of reference is low, 2PL* offers lower throughput than the optimistic method due to the cost of obtain remote locks.

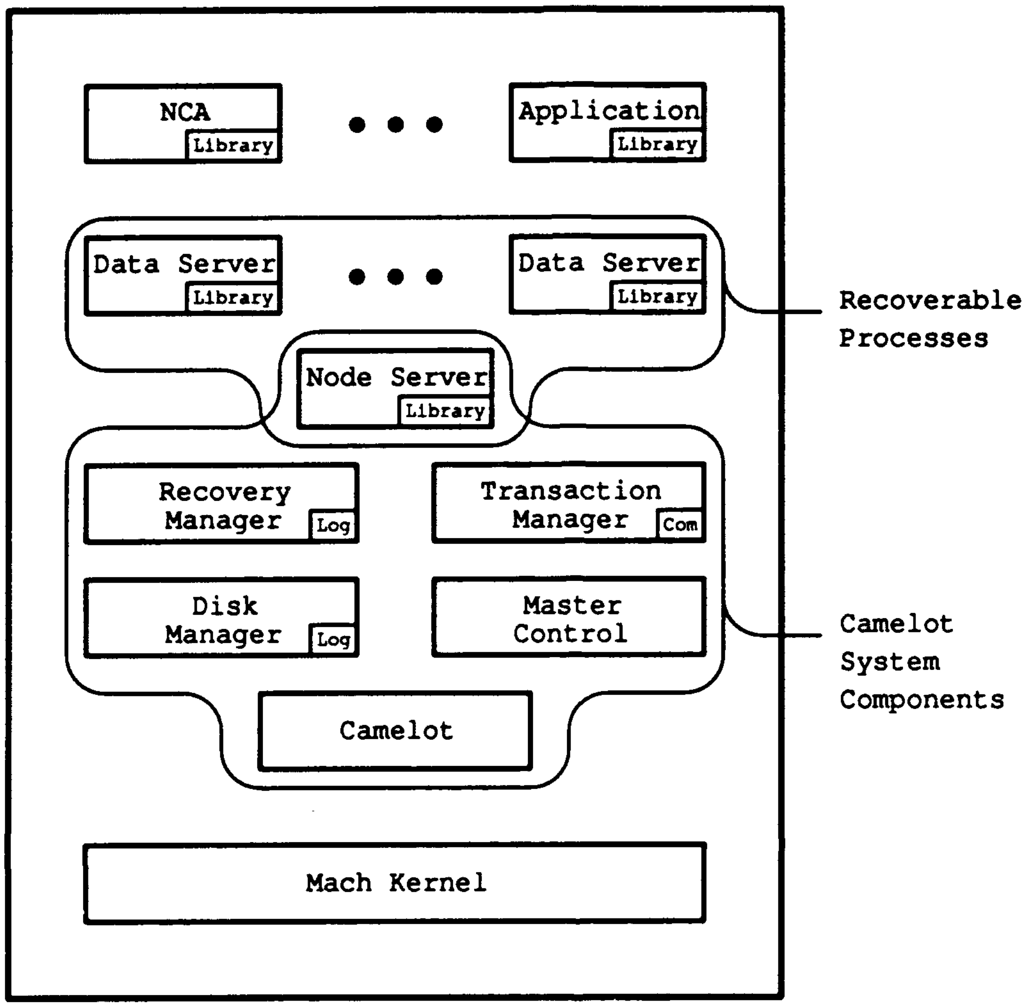
# Chapter 3
# Camelot and Mach Environment

This chapter, as a follow-up to the material presented in Chapter 2, introduces the Camelot distributed transaction facility and the Mach operating system on which Camelot runs. Camelot demonstrated that transactions could be efficiently layered on an operating system kernel as a general-purpose facility. Camelot is well-documented, and the source code is readily available for research use; thus, Camelot is a reasonable place from which to start work on transactional distributed shared memory. Camelot is described in detail in *Camelot and Avalon: a Distributed Transaction Facility* [Eppinger et al 91]; this chapter provides an overview of Camelot, with references to the appropriate chapters in [Eppinger et al 91].

## 3.1. Camelot

Camelot provides a framework for constructing programs that use distributed, atomic transactions in the client/server model. In Camelot, programs are divided into two classes. An *application* interacts with users, and may initiate and conclude transactions. Within a transaction, an application makes *server calls* (transactional RPCs) to Camelot servers in order to access persistent storage. A *data server* is started under the control of Camelot, and encapsulates all access to a unit of persistent storage (unique to each server) known as a *recoverable virtual memory (RVM) segment*. A server maps its recoverable virtual memory segments into its virtual address space, and may read and write RVM segments transactionally. If the enclosing transaction aborts, Camelot will undo any changes made by the transaction to the RVM segments. Once a transaction commits, Camelot guarantees the permanence of the transaction's changes. Within a transaction, both applications and servers may make many server calls to local or remote servers. Camelot ensures that all servers involved in a distributed transaction agree on the transaction's outcome.

Camelot is implemented on the UNIX-compatible Mach operating system, and is divided into several processes that communicate via Mach messages. The Camelot library provides a C language interface to Camelot services. Figure 3-1 shows the processes that run on each Camelot node. The Master Control Program and the Camelot startup process are uninteresting, but the other components are described in subsequent subsections.



Data servers and applications are written by Camelot users. Data servers maintain persistent data in recoverable virtual memory. Applications begin and end transactions that invoke operations on data servers via remote procedure calls. The Node Server is a distinguished data server used to store configuration information; the Node Configuration Application (NCA) provides a user interface. The Transaction Manager coordinates distributed agreement (ensuring that all participating nodes agree on the outcome of a transaction), and includes the Communication Manager, which tracks the spread of transactions from node to node. The Recovery Manager restores recoverable virtual memory to a transaction-consistent state after a failure, and includes routines to read the log. The Disk Manager is responsible for buffer management and writing the common log. The Camelot process initializes the system, and the Master Control Program coordinates the operation of the other components.

**Figure 3-1:** Camelot architecture

### 3.1.1. Camelot library

The Camelot library provides a high-level, C language interface to Camelot services [Bloch 91a]. The Camelot library is also primarily responsible for concurrency control.

An application written using the Camelot library might use code like this:

```
BEGIN_TRANSACTION
   ...
   SERVER_CALL("array_server",
               array_update(ARGS index, value));
   ...
END_TRANSACTION(status)
```

The BEGIN_TRANSACTION and END_TRANSACTION statements bracket a series of actions into a transaction. (The status variable indicates that the transaction committed or gives a coded reason for transaction abort.) The SERVER_CALL is a transactional RPC to the array_server; any actions performed by the array_server to process the RPC will take place in the context of the transaction begun by this application. The server is requested to perform an array_update operation with the given parameters.

The array_update operation in the array_server might use code like this:

```
BEGIN_RECOVERABLE_DECLARATIONS
  unsigned int array[SIZE];
END_RECOVERABLE_DECLARATIONS
...
  LOCK(&REC(array[index],
       LOCK_SPACE_PRIMARY,
       LOCK_MODE_WRITE);
  MODIFY(REC(array[index]), value);
```

The server must first declare the layout of its recoverable virtual memory via BEGIN_ and END_RECOVERABLE_DECLARATIONS. On receipt of the array_update RPC, the server may modify its RVM within the scope of the transaction initiated by the application.

The Camelot library requires the server programmer to use explicit LOCK operations. (Explicit locking allows use of logical locks, and gives the programmer added flexibility in choosing lock granularity.) Camelot supports multiple *lock spaces*, so that a package supporting a particular abstract data type may use a name space for locks that will not conflict with any other package. In the example above, the array_server obtains a write lock on the appropriate element of the array in the standard lock name space; the name of the lock is just the address of the element in the array.

After obtaining the write lock, the array_server is free to modify the corresponding item in recoverable storage. The MODIFY operation stores a new value in the array, and simultaneously generates log records that the Camelot system can use to undo the MODIFY should the enclosing transaction abort.

At the end of a transaction, the Camelot library automatically releases any locks held by the transaction. Thus, the library supports two-phase locking for serializability. Careful programmers may explicitly unlock locks before the end of transaction; however, Camelot does not guarantee serializability in this case.

To process a lock request, the Camelot library checks for conflicts with other transactions [Bloch 91b]. In most cases, the library has enough information to identify conflicts.

When transactions are deeply nested, however, the library may not be able to determine if two transactions with a common ancestor (in the same *transaction family*) are conflicting in their use of the lock, and the library asks the Transaction Manager to decide if there is a conflict.

## 3.1.2. Disk Manager

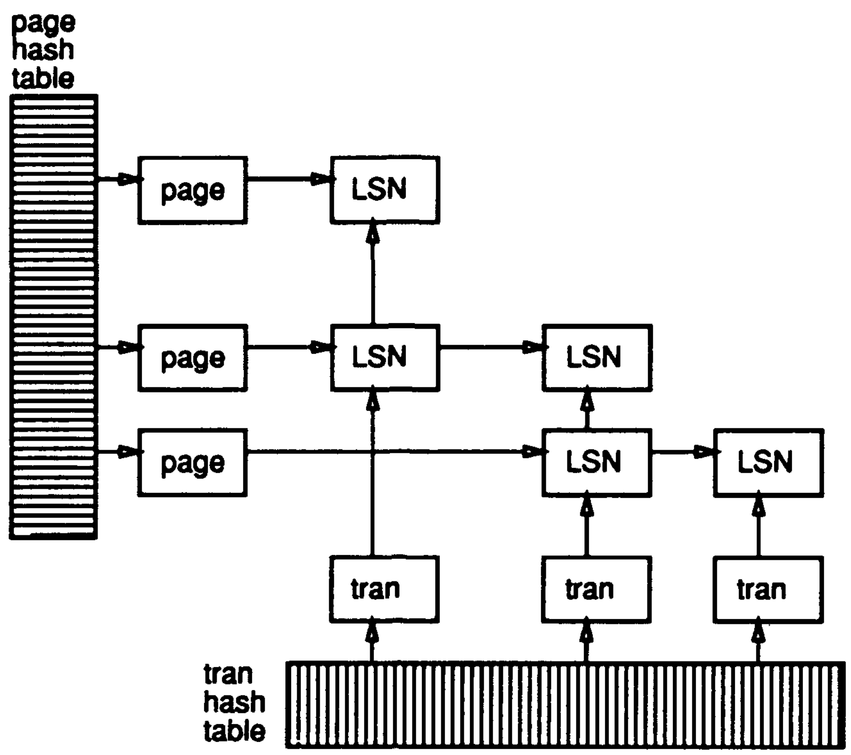
The Disk Manager reads and writes pages to/from the disk when Mach needs to service page faults on RVM or to clean primary memory. Camelot uses a common log for transaction management and recovery functions; the Disk Manager accepts and writes log records locally [Thompson and Jaffe 91], and coordinates log writes with paging writes to enforce the write-ahead log invariant. The Disk Manager allocates backing store for RVM segments, performs checkpoints to limit the amount of work during recovery, and works closely with the Recovery Manager when failures are being processed. The Disk Manager is multi-threaded to permit multiple I/O operations in parallel. The data server interface to recoverable storage is described in [Eppinger and Nichols 91], and the design of the Disk Manager is given in [Eppinger 91].

### 3.1.2.1. Data structures

The server record `s_record_t` contains data about an active server, including the server's state, information about the server's RVM segment, and Mach ports for contacting the server and the Mach kernel. Several hash tables allow the Disk Manager to locate a server record given a UNIX process id, a server id, a RVM segment id, or a Mach port.

As a server updates its recoverable storage, it spools log records describing the regions being updated and the values stored there. These log records are funneled through the Disk Manager to the common log. To spool log records efficiently, each server has its own private shared memory queue with the Disk Manager. The Camelot library sends asynchronous messages to the Disk Manager by storing them in this queue. When the queue becomes full, the library makes a synchronous RPC to the Disk Manager to process the queue. The Disk Manager may also process the queue at earlier times to enforce the write-ahead log invariant.

The most complicated data structure used by the Disk Manager is the grid, which is used to keep track of records in the log. The grid is a natural extension of the two hash tables required to do write-ahead logging: one hash table for the active pages, one for the active transactions. Keeping such careful track of log records is required to support new-value-only logging. Each log record is linked to both the page and transaction to which it refers. (See Figure 3-2.)



The LSN (log sequence number) uniquely identifies a log record. One hash table is used to find active pages; each log record is linked to the page it describes. Another hash table is used to locate active transactions; each log record is also linked to the transaction to which it belongs.

**Figure 3-2: The grid**

#### 3.1.2.2. Algorithms

Each time a server modifies a region in recoverable storage, it must use the *pin-update-log* protocol. Before modifying the region, the server pins the region in virtual memory. Pinning prevents the Disk Manager from writing the page(s) on which the region resides back to paging store. Next, the server updates the region in virtual memory by storing a new value. Finally, the server logs the modification, unpinning the region. The pin and log operations are simply entries the server makes in its shared memory queue.

The Disk Manager acts as an external pager for the RVM segment in each data server. (See Section 3.2 for a description of the external pager interface.) When the Disk Manager receives a `memory_object_data_write` message from the kernel, it checks that the server is using the pin-update-log protocol properly. If the page is not pinned, the Disk Manager forces the appropriate log records to stable storage, and then writes the page to disk. When the Disk Manager receives a `memory_object_data_request` message from the kernel, the Disk Manager locates the page on disk, and returns it to the kernel with a `memory_object_data_provided` message.

To support recovery, the Disk Manager responds to requests from the Recovery Manager for a list of all log records corresponding to a particular transaction or a particular page. To limit the amount of log data that must be processed during recovery, the Disk Manager periodically writes to the log a **checkpoint** record containing a list of active servers, active segments, active pages, and active transactions.

### 3.1.3. Recovery Manager

The Recovery Manager is responsible for transaction abort, server recovery, node recovery, and media-failure recovery [Thompson 91]. During normal processing, Camelot components and servers record in the log all modifications to recoverable storage, movements of pages between disk and memory, and outcomes of completed transactions. When a system component detects a failure, it notifies the Recovery Manager, which reads the log and undoes or redoes the effects of transactions as appropriate. The Recovery Manager also sends information from the log to servers and to the other Camelot components to allow them to restore their internal data structures. During recovery, additional records are written to the log describing what was undone or redone.

Camelot directly supports only value logging. When recoverable storage is updated, the new value, or *after-image*, of the modified region is written into a log record called a **modify record**. Depending on how the transaction was initiated, the old value, or *before-image*, of the region may also be written into the log. The Recovery Manager implements three distinct recovery algorithms: old-value/new-value abort, new-value-only abort, and server recovery. The abort algorithms are invoked by the Transaction Manager. The server recovery algorithm is invoked by the Disk Manager after a server crash or node failure.

To abort a transaction, the Recovery Manager reads the log backwards to extract the modifications made by the transaction. For a modification made by an old-value/new-value transaction, the log contains the old value to be restored; for a new-value-only transaction, the Recovery Manager must locate the old value in a log record written by a committed transaction or read the old value from its home location on disk. The Recovery Manager buffers up a request to undo the modification by restoring the old value. The requests are sorted by server id, and forwarded to the appropriate server when the buffer is full or there are no more requests to be buffered.

To recover a server, the Recovery Manager reads the log backwards to identify changes made by committed transactions that are not reflected in the disk copy of the page, and buffers up requests to redo these changes. It also identifies changes made by aborted transactions that were already written to the disk copy of the page, and buffers up requests to undo these changes. The buffered requests are sent to the server for processing after it has been restarted, but before it begins accepting RPCs from its clients.

### 3.1.4. Communication Manager

The Communication Manager provides a name service for Camelot servers and supports transactional RPCs [Stout 91]. The name service dispenses surrogate local ports for network ports so that all distributed transactional RPCs pass through the Communication Manager. Thus, the Communication Manager can keep a list of all the nodes involved in a particular transaction, and it supplies this list to the Transaction Manager for use during commit and abort processing.

### 3.1.5. Transaction Manager

The Transaction Manager coordinates the initiation, commit, and abort of local and distributed transactions [Mummert et al 91]. It fully supports nested transactions. Each transaction has a unique *transaction id* assigned by the Transaction Manager.

The primary protocol supported by the Transaction Manager is the two-phase commit protocol. To commit a local transaction, the Transaction Manager makes an RPC to each server involved in the transaction, requesting a vote for commit or abort; all servers must vote to commit in order for a transaction to commit. Once all votes are in, the Transaction Manager commits or aborts the transaction by forcing a record to the log. In the second phase of the protocol, the Transaction Manager notifies each server of the transaction's outcome.

To commit a distributed transaction, the Transaction Manager on the coordinator node (where the transaction was initiated) sends IP datagrams to the Transaction Manager on each subordinate node involved in the transaction. All Transaction Managers query their local servers. Each subordinate Transaction Manager forces a prepare record into its local log, and returns a single vote to the coordinator Transaction Manager. The coordinator Transaction Manager tallies the votes and forces a commit or abort record to its log. In the second phase of the protocol, the coordinator notifies each subordinate of the transaction's outcome.

Not described here is the non-blocking commit protocol supported by the Transaction Manager. The Transaction Manager also aborts transactions, and answers questions from the Camelot library about lock conflicts.

### 3.1.6. Node Server/Node Configuration Application

The Node Server is the repository of configuration data necessary for restarting the node [Thompson and Michaels 91]. To start a server, the Disk Manager needs to know what command line to execute, how much recoverable storage the server is allowed, and the name of the server and its recoverable segment. All of this data is maintained in recoverable storage by the Node Server and is recovered before other servers.

Backing store for recoverable storage is allocated in large fixed-size units called *chunks*. To allocate backing store, the Disk Manager makes an RPC to the Node Server. The Node Server runs a transaction to allocate the appropriate chunks, and returns the result to the Disk Manager.

The Node Configuration Application (NCA) permits Camelot's human users to update data in the node server and to crash and restart servers [Eppinger and Michaels 91].

## 3.2. Mach

Mach is a multiprocessor operating system that is binary compatible with 4.3 Berkeley UNIX. Mach provides the basic building blocks for distributed applications, including tasks, multiple threads of control within tasks, message passing, and shared virtual memory between tasks [Stout et al 91].

A *task* is a collection of system resources, including virtual memory and access rights to Mach ports. It is similar to a UNIX process. A *thread* is a unit of scheduling (i.e., a lightweight process) that executes within a task.

Interprocess communication in Mach is based on two abstractions: ports and messages. A *port* is a protected kernel object to which messages may be sent and queued until reception. A task may hold send and receive rights to a port. A *message* is an ordered collection of typed data, possibly including port rights or pointers to out-of-line data. The header of a message includes the port to which the message is sent, and optionally a port to which the receiver may reply.

Most Mach users never need to know how messages and headers are formatted; instead, they use the Mach Interface Generator (MIG) to automatically implement remote procedure calls [Pausch et al 91].

A special user-level task called the *netmsgserver* transparently extends interprocess communication across the network. The netmsgserver acts as a local representative for tasks on remote nodes. When a task sends a message to a port on a remote node, the message is actually delivered to the local netmsgserver. The local netmsgserver translates the destination port to a network address, converts the message to a format suitable for the network, and transmits the reformatted message to its counterpart on the destination node. The destination netmsgserver converts the network message into a Mach message. It re-sends the Mach message to the correct port on its local node.

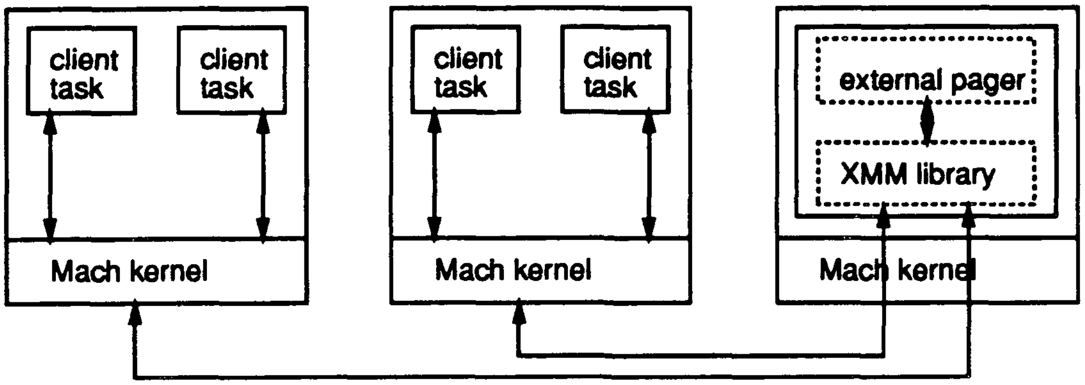
The netmsgserver also provides a port registration and lookup service. Using `netname_check_in`, a task can associate a name with send rights to a port it owns. Using `netname_look_up`, any other task can present the name and receive send rights to the port. `netname_look_up` also accepts a host name as a parameter, so that a task may look up a port on a remote host.

The Mach virtual memory design allows tasks to allocate/deallocate regions of virtual memory, specify the inheritance of regions of virtual memory (when a task forks), set the protection (read/write/execute) on regions of virtual memory, and specify a user-level task to handle paging for regions of virtual memory [Baron et al. 90]. It is this last feature, the *external pager interface*, which Camelot uses to implement recoverable virtual memory.

To use the external pager interface, a client task (e.g., a Camelot server) maps a *paging object* into its address space using `vm_map`. The paging object is represented by a port, obtained from the external pager task, to which the kernel will send external pager messages. When the client task takes a page fault on a page of its address space mapped to the paging object, the kernel sends a `memory_object_data_request` message to the port representing the paging object (i.e., to the external pager). The external pager receives the request, finds the data, and returns it to the kernel with the `memory_object_data_provided` call. The kernel puts the page into the client task's address space and resumes the thread. The external pager may ask the kernel to flush a page from the client's address space with the `memory_object_lock_request` call. In response to this call, or if the kernel wishes to free up physical memory, the kernel makes a `memory_object_data_write` call to the external pager.

The external pager interface makes it possible for a user-level task to implement distributed shared memory. The first implementation of this feature was the *netmemoryserver* [Forin et al 88]. As part of his research into kernel support for distributed memory multiprocessors, Barrera reimplemented the distributed shared memory functionality in a user library, the External Memory Manager, that can be linked with any external pager task [Barrera 92]. The External Memory Manager library allows an external pager to create a paging object and specify that it is to be used to provide distributed shared memory. The library interposes itself between the external pager and multiple client kernels. To the external pager, the External Memory Manager presents the illusion of a single client kernel. Client tasks on multiple nodes see the illusion of a single, shared paging object; every read of a byte in the shared object returns the most recent value stored there on any node. (See Figure 3-3.) The External Memory Manager library achieves this illusion by maintaining a directory for each page of those nodes with read or write permission for the page. (In Li's terminology, the XMM library is a fixed distributed manager.) The XMM library can support multiple paging objects, managing the coherency of each set of pages independently.

When a client kernel asks the XMM library for read access to a page, the XMM library checks its directory to see if any node has write access to the page. If no node has write access, the XMM library asks the external pager to supply a readable copy of the page, and returns it to the client kernel. But if some node has write access to the page, the XMM library first *invalidates* that copy, directing the node to return the modified page to the XMM library. The XMM library forwards the page to the external pager. Then the XMM library continues as above where no node has write access.



The External Memory Manager (XMM) library is an intermediary between an external pager and several Mach client kernels. It acts as a single client kernel to the external pager, and acts as an external pager to each of the actual client kernels, managing the migration of pages from node to node to achieve coherent distributed shared memory.

**Figure 3-3:** External Memory Manager

When a client kernel asks the XMM library for write access to a page, the XMM library checks its directory for readers and writers. It directs each reader or writer node to invalidate the page (readers simply discard the page, a writer writes back the modified page). Then the XMM library asks the external pager for a writable copy of the page, and returns it to the client kernel.

# Chapter 4
# Design

## 4.1. Goals

Chapter 2 outlined the numerous alternatives available in designing transaction processing and distributed shared memory systems. Choosing a particular design by selecting among the alternatives is best guided by having a set of goals in mind. The thesis of this dissertation is that transactional distributed shared memory (TDSM) is feasible and useful. To demonstrate the feasibility of TDSM, the design must be practical for a graduate student to implement in a reasonable time. To demonstrate the utility of TDSM, the design must provide enough functionality to allow the characteristics of TDSM to be evaluated.

These high-level goals can be further elaborated by considering their effect on how the design should meet distributed program requirements. The most important requirements are as follows.

- Ease of programming. As much as possible, the design should preserve an existing easy-to-use programmer interface.
- Data integrity. The design should maintain the failure atomicity, serializability, and permanence guarantees of transactions, even if an object in TDSM is concurrently updated on different nodes. The design should maintain these guarantees if messages are lost, or if processes or nodes crash.
- Multiple access to data. The design should allow concurrent access to multiple TDSM memory segments on multiple nodes; that is, one process may access a TDSM memory segment at the same time as some processes are accessing the same TDSM memory segment, and other processes are accessing other TDSM memory segments. However, a given process will not necessarily be able to access more than one TDSM memory segment.

Other distributed program requirements are addressed as follows.

- Performance. The design should utilize algorithms that provide reasonable performance and allow the performance of TDSM to be evaluated. It is not necessary to take advantage of all possible optimizations if these optimizations can be analytically evaluated.
- Security. It is acceptable for the design to provide the same security as an existing transaction processing facility. Access to a TDSM memory segment should be restricted to its creator.
- Availability. The design may restrict availability to protect data integrity in the presence of failures. When a node crashes, the design may make unavailable those parts of TDSM memory segments that have been modified by prepared transactions on the failed node.

- Incremental growth. The design should allow incremental increases in processing capacity by allowing additional nodes to access a TDSM memory segment at any time that the TDSM memory segment is available. (Of course, if individual objects in the TDSM segment are being read or written, the new node may have to wait for access to those objects in order to preserve serializability.)

## 4.2. Architecture

The architecture provides transactional distributed shared memory by allowing servers on multiple nodes to share a given TDSM memory segment. Each TDSM segment has a *home node* where the non-volatile storage for the segment is located, and which provides certain services for the segment. Nodes where servers read or write a TDSM segment are *using nodes* of the segment. A node may be the home node for many TDSM segments, and many nodes may act as home nodes for different segments. A node may be a home node or a using node or both for any number of segments. Each home node must have non-volatile storage for the blocks of TDSM segments; using nodes are not required to have any non-volatile storage. Nodes communicate via a network; no disks are shared between nodes.

To implement transactions, the TDSM architecture follows the structure outlined in Chapter 2: log management, recovery management, transaction management, communication management, configuration management, concurrency management, and buffer management. In brief, a transaction must log each modification it makes so that the recovery manager may back out partial updates (to ensure failure atomicity) or repeat updates (to ensure permanence). The transaction manager coordinates the completion of transactions that span multiple servers. The communication manager tracks the spread of transactions from node to node. The configuration manager stores the configuration of the transaction facility. The concurrency manager coordinates the execution of concurrent transactions to ensure serializability. The buffer manager controls the transfer of recoverable objects between volatile and non-volatile memory.

Many choices are possible when designing these functions. Generally, these choices are not specific to TDSM, but common to any transaction processing system. This architecture for TDSM leverages on the design choices of an existing transaction processing facility. For this architecture, the transaction processing facility and underlying operating system must provide:

- A mechanism for sending block-size messages between arbitrary processes. The message system must reliably deliver the messages in order, and the same mechanism must be used for local and remote recipients.
- Encapsulation of servers such that transaction services are provided (or can be provided) through the above message mechanism.
- A transaction commit protocol that does not necessarily support distributed transactions, but does use (or can be made to use) the aforementioned message mechanism.
- A clearly-defined interface for concurrency management that can be extended for sharing and network access; e.g., locks that can be distributed through communication with the home node.

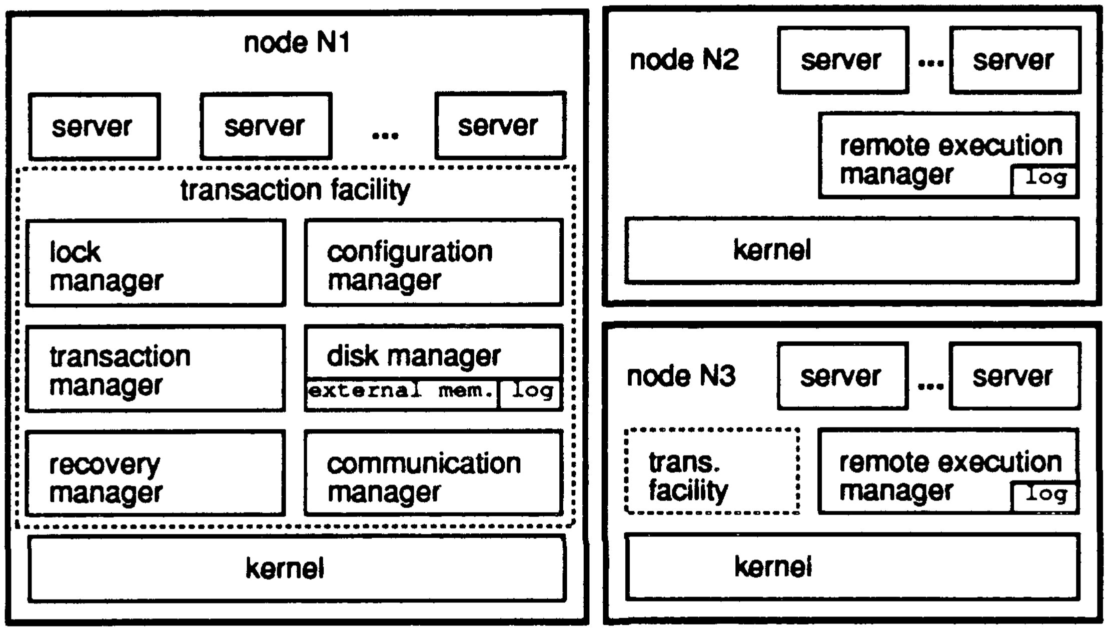
- A clearly-defined interface for buffer management that can be extended for sharing and network access. Instead of look'ng for a requested block on the local disk, the TDSM buffer manager contacts the home node. The home node locates the block on disk, or requests the return of the block from another buffer manager. The other buffer manager may have to wait until the requested block is released by a transaction before satisfying the request from the home node.
- A clearly-defined interface for recovery management that can be extended for sharing and network access. Intentions lists, shadow pages, or log records must be forwarded to the home node at the appropriate time to ensure failure atomicity and permanence. For availability reasons, the home node must be able to undo or redo actions.
- The ability to have a common log (intentions list, shadow pages) for all servers using a given TDSM segment.

### 4.2.1. Camelot and Mach

Rather than start from scratch, I use the Camelot transaction processing facility and the Mach External Memory Manager (XMM), described in Chapter 3, as a base for building the TDSM architecture. Other systems could be used as a base; I chose Camelot and Mach because they were readily available. In brief, Camelot offers the abstraction of a recoverable virtual memory segment that a Camelot server may read and update within the scope of a transaction, and the Camelot library provides an easy-to-use environment for building transactional applications. The XMM library offers a shared virtual memory coherency service to external pagers such as the one in Camelot.

The Camelot/TDSM architecture provides TDSM in the form of distributed recoverable virtual memory by allowing Camelot servers on multiple nodes to share a given recoverable virtual memory segment. Although parts of Camelot were designed to allow each server to access multiple RVM segments, the Camelot library (which maps the RVM segment at a fixed location in the server's virtual address space) and the Camelot Disk Manager (which handles page-in and page-out requests for the RVM segment) permit a given server to access exactly one RVM segment. The TDSM architecture does not remove this restriction from Camelot: to do so would require extensive changes to the programmer interface provided by the Camelot library. The focus of TDSM is to enable a set of servers to share a given memory segment, not to enable a given server to access multiple memory segments. Since Camelot allows a RVM segment to be larger than the virtual address space of a single process, one workaround for the single-segment-per-server restriction is to combine the multiple segments that a server wishes to access into a single segment. Another alternative for accessing multiple RVM segments is a hybrid of data sharing and function shipping: on a given using node, there are several servers, each accessing a different RVM segment, and communicating among themselves via local RPCs.

To ensure failure atomicity and permanence, Camelot uses write-ahead logging. The write-ahead log invariant requires that log records describing changes to a block be written to stable storage before that block can be written to non-volatile storage. Although write-ahead logging is more complicated than other techniques for providing failure atomicity and permanence, it offers several performance advantages [Spector 89b]. Also, since transaction commit protocols are frequently implemented by using a log, additional performance advantages are possible if the Transaction Manager shares a common log with the recovery algorithm. In general, Camelot's designers attempted to optimize forward processing at the possible expense of longer recovery times.



Nodes N1 and N3 run the transaction facility and may act as home nodes for several segments. If a server on node N1 or N3 accesses a segment stored there, then node N1 or N3 is also a using node. Nodes N2 and N3 run the Remote Execution Manager and may be using nodes of segments on other nodes.

**Figure 4-1:** TDSM architecture

Figure 4-1 illustrates how the functions that implement transactions are distributed between the home node and using nodes. On the home node, a set of processes known collectively as the transaction facility coordinate access to the stable storage common log and the non-volatile paging store for the segment. On using nodes (other than the home node), a process known as the Remote Execution Manager coordinates access to services on the home node. Subsequent subsections describe how TDSM functionality for a given RVM segment is provided by the transaction facility and the Remote Execution Manager.

### 4.2.2. Log management

The home node is responsible for the log. This decision simplifies other components by reducing the effort needed to locate the log, and ensures that the log records needed to perform recovery are available as long as the home node is available. It offers opportunities for reducing the number of separate messages that must be sent to the home node in that log records may be appended to messages for transaction management or buffer management.

Each using node ensures that the log records it generates for a segment are eventually forwarded to the home node. The Remote Execution Manager on the using node buffers log records to reduce latency. Log records must be forwarded when a transaction commits, at which time the log records must be forced to stable storage to ensure permanence. Log records must be forwarded when a page is written back to the home node or migrated to another using node; this allows the home node to recover from the failure of the first using node. Log records are forwarded when a transaction aborts, so the Recovery Manager has the information it needs to undo the effects of the transaction.[3] Finally, log records must be forwarded when the log record buffer becomes full.

All servers using a given TDSM segment must use a common log. (This is already done by Camelot, since Camelot uses a common log for all servers and Camelot components on a node.) Camelot uses physical logging. Operation logging could be used for TDSM, although operation logging for TDSM is complicated by the fact that undo and redo operations may be invoked on a different node than that which performed the original operation.

### 4.2.3. Transaction management

The home node provides transaction management services for the RVM segment. All servers using a given RVM segment write to a common log on the home node where the Transaction Manager resides. These servers rely on the home node for transaction management. The Transaction Manager uses the same mechanism to send messages to home node servers as it does to send messages to using node servers; indeed, the Transaction Manager does not know whether a server is local or remote. Because of this transparency for transaction management messages, and because the log for the servers is local to the Transaction Manager, the Transaction Manager need not worry about the location of the servers. Servers contact the Transaction Manager to begin server-based transactions and to join existing transactions; the Transaction Manager remembers which servers are involved in each transaction.

[3]If the using node crashes before forwarding the log records, the transaction (and all others running on the using node) still aborts. The Recovery Manager doesn't need the log records of the aborted transaction in this case, because either (1) the pages modified by the aborted transaction were never written back to the home node, and thus the Recovery Manager can simply use an older version of the pages from disk, or (2) the log records were forwarded earlier at the time the modified pages were written to the home node.

When a transaction aborts, the Transaction Manager uses its knowledge to notify the appropriate servers. The transaction may fail to contact a server if the server is on a different node. In this case, the server may become an *orphan* and attempt to continue processing; however, it will not be able to commit its changes, since to do so it would have to contact the Transaction Manager which already knows that the transaction has aborted. Other nodes may not be able to continue processing until the Transaction Manager has successfully contacted all servers.

When a transaction wishes to commit, the Transaction Manager asks each of the servers involved in the transaction to enter the prepared state and return a vote. If any server votes no, or if the Transaction Manager cannot contact a server, the transaction aborts as described above. Otherwise, the Transaction Manager writes a commit record to the log, and forces the log to stable storage. It then notifies all servers that the transaction committed. If the Transaction Manager is unable to contact a server, the server will remain in the prepared state, retaining any write locks it obtained. No other transaction in any server will be able to obtain these locks until the Transaction Manager successfully contacts the server. (If the Transaction Manager cannot contact the server because the using node is down, any pages for which the using node has write permission will be inaccessible, so inability to obtain locks is only part of the problem.)

With TDSM, the Transaction Manager does not contact any subordinate Transaction Managers to commit a transaction. The log for the servers is local to the home node, and there is no need for a subordinate Transaction Manager to record data in a separate log. Thus, the Transaction Manager does not use a traditional distributed commit algorithm, in which Transaction Managers on subordinate nodes must force data to the log when a transaction enters the prepared state. Instead, the home node Transaction Manager uses a single log force to commit a transaction, even though servers on several using nodes may be involved.

### 4.2.4. Recovery management

The home node manages recovery for the RVM segment. As with transaction management, the common log on the home node and the transparency of messages allows the Recovery Manager to perform its task for all servers without worrying about their location. The Recovery Manager sends messages to servers to recover from transaction failure. The Recovery Manager is notified whenever a server starts or terminates, so it may remember which servers are using a given RVM segment.

When a transaction aborts, the Recovery Manager must undo any changes to RVM that have been made by the aborted transaction. It does this by scanning the log, extracting records belonging to the aborted transaction, and sorting them by RVM segment. For each RVM segment, it selects one server to actually perform operations on the segment that undo the aborted transaction. (This server may or may not be the same server that made the original modifications.) If the selected server fails before completing the operations, the Recovery

Manager selects another server and continues. Any server using the RVM segment may perform the abort processing requested by the Recovery Manager, because all servers have access to the segment, and objects will remain locked until abort processing completes. The advantage of sending the requests to a single server is that the requests can be batched, reducing communication costs. The disadvantage of sending the requests to a single server is that if the Recovery Manager makes a poor choice, many pages will have to migrate to the server's node. Of course, Camelot assumes that recovery is infrequent, so this is not a major issue.

The Recovery Manager recovers from individual server failure by aborting all of the transactions that have not committed in the failed server (undoing their effects as above), and redoing the effects of committed transactions that are not yet reflected in non-volatile storage. Transactions in other servers using the same RVM segment are not directly affected by the failed server, although they may have to wait for locks and pages until recovery completes. If there are no active servers using the RVM segment, a special surrogate server must be started on the home node to perform undo and redo operations on behalf of the Recovery Manager. As soon as the Recovery Manager finishes its task, the surrogate server exits.

The actions taken by the Recovery Manager after a node failure are very different, depending on whether the failed node is the home node. If the home node fails, servers on other nodes may continue processing for a while in a limited manner, but they will eventually detect the failure of the home node and exit (implicitly aborting any transactions that have not yet committed). When the home node restarts, the Recovery Manager restores every RVM segment to a transaction-consistent state by starting a surrogate server on the home node for each segment, and sending undo and redo requests to the surrogates.

If the failed node is not the home node, then the Recovery Manager itself continues running. The Recovery Manager does not explicitly determine that a node has failed; instead, it performs multiple server recoveries as the TDSM system notices that each server on the failed node has failed.

When a using node fails, the log records it has buffered in volatile storage are lost. However, the loss of these buffered log records does not affect the correctness of any data in RVM, because the lost log records belong to transactions that the Recovery Manager is about to abort due to the node failure. (Recall that buffered log records must be forwarded to the home node before a transaction can commit.) For each page modified by uncommitted transactions on the failed node, one of the following cases will be true: either the modified page has been forwarded to the home node (due to migration or other pageout), or it has not. If the modified page has been forwarded to the home node, then the Recovery Manager will have log records describing the modifications made by uncommitted transactions, and it may use the log records to abort the transactions by undoing their modifications. (Recall that buffered log records must be forwarded to the home node when a pageout occurs.) If the modified page has not been forwarded to the home node, then the Recovery Manager will not have log records describing the modifications. However, in this case, there is nothing for the Recovery Manager to undo, since the copy of the page in non-volatile storage does not contain the modifications of the uncommitted transactions.

### 4.2.5. Buffer management

Responsibility for buffer management is split between the Disk Manager and External Memory Manager on the home node, and the operating system kernels on all using nodes. Each server maps the RVM segment into its virtual address space. Pages of the RVM segment are buffered in main memory by kernels on different nodes as part of their resident pool of virtual memory pages. Each kernel contacts the External Memory Manager on the home node to obtain non-resident pages, to write out modified pages, and to request write permission on resident pages. The External Memory Manager contacts a kernel to restrict read or write permission to ensure that pages are coherent. A page may migrate from node to node at any time whether or not a transaction is actively using the page; thus, an active transaction may be suspended if a page it needs is stolen for use by another node.

The External Memory Manager follows a single-writer, multiple-reader protocol to enforce page coherency. When a page is being read, it may be resident on many nodes. When a page is being written, it is resident only on the node that is writing. For each page, the External Memory Manager records a list of the nodes that have the page with read or write permission. When a node requests read permission for a page, the External Memory Manager supplies the permission immediately if there are no writers. If there is a writer, the External Memory Manager *invalidates* the page on the writing node, which is instructed to write back the updated page, and remove write permission. When a node requests write permission for a page, the External Memory Manager invalidates the page on any readers (which simply discard the page) or any writers (which write back updated pages).

The Disk Manager responds to requests from the External Memory Manager to copy pages to and from non-volatile storage. The Disk Manager must allocate non-volatile storage for RVM pages and must coordinate writes to non-volatile storage with log writes. The write-ahead log protocol requires that log records describing changes to a page be written to stable storage before that page can be written to non-volatile storage. Then, if the transaction that made the changes aborts, the log records will be available for the Recovery Manager to read, and it can undo the changes. Even if the home node crashes before writing the page to non-volatile storage, the Recovery Manager can reconstruct a correct copy of the page by scanning the log and redoing the effects of committed transactions.

The External Memory Manager maintains the coherency of pages between nodes; it is not involved when two servers on the same node share a page. Instead, the operating system kernel allows the servers to physically share the page where permitted by the virtual memory hardware. The External Memory Manager could maintain coherency between servers: if a server tries to access a page that is currently in use by another server on the same node, it would wait until the page could be migrated from server to server via the home node. The advantage of server-level coherency is increased isolation of servers, which is especially useful during debugging. The overwhelming disadvantage of server-level coherency compared to node-level coherency is significant communication cost where there was no cost.

The External Memory Manager is not required to use the single-writer, multiple-reader protocol to maintain coherency. However, this algorithm is straightforward to implement, offers good performance, and has been used in most distributed shared memory systems.

### 4.2.6. Concurrency management

As shown in Chapter 2, many forms of concurrency management are possible to ensure the serializability of transactions. Because it is well understood, straightforward to implement, and has been shown to provide good performance [Carey and Livny 88], two-phase locking is used by Camelot to maintain the serializability of transactions. A transaction must obtain a read lock before inspecting an object, and a write lock before modifying an object. Locks are released at the end of the transaction. The granularity of locks is chosen by the application programmer; the name of a lock is the address of the corresponding object. Several objects may reside in the same virtual memory page, and an object may span several pages. This gives the application programmer maximum freedom in allocating recoverable storage and in selecting the degree of concurrent access to objects. With this freedom comes the responsibility of ensuring that locks do not refer to overlapping objects, since the transaction facility does not know the size of the object corresponding to a lock.

For TDSM, responsibility for concurrency management is split between the Lock Manager on the home node, and all servers using the RVM segment. Each server caches the locks that it is actively using. Each server exchanges messages with the Lock Manager on the home node to obtain non-cached locks and to release cached locks. While a server has a lock cached, it may grant the lock to a transaction without further communication with the Lock Manager, as long as the transaction is requesting a mode that is a subset of the mode in which the lock is cached. (That is, if the lock is cached in write mode, a transaction may obtain the lock in read or write mode with no further communication. If the lock is cached in read mode, a transaction cannot obtain the lock in write mode until the server contacts the Lock Manager.)

The Lock Manager uses an algorithm similar to the External Memory Manager to enforce a single-writer, multiple-reader protocol for the caching of locks by servers. Each server enforces the single-writer, multiple-reader protocol for the acquisition of locks by transactions. The Lock Manager may request return of a previously cached lock when it is requested by another server. A lock may migrate only when it is not held by an active transaction. There is no deadlock detection; instead, a timer aborts long-running transactions.

A server may limit the number of locks it can cache. When this limit is exceeded, it may uncache locks that are not held by any active transaction. If all cached locks are held by active transactions, the server may request the Lock Manager to grant a lock to an individual transaction, instead of allowing the server to cache the lock. When the transaction commits, the server notifies the Lock Manager to release the lock.

When a server crashes, any of its locks may be obtained by other servers, except for those locks held by prepared transactions. If the Lock Manager crashes, servers retain their cached locks and can continue to grant them to transactions. When the Lock Manager restarts, it contacts each server to restore its knowledge of cached locks.

Concurrency management via distributed locks requires no changes to the programmer interface to Camelot. The implementation of the distributed Lock Manager preceded the detailed design of the TDSM architecture and provided evidence that such an architecture was feasible [Hastings 90].

## 4.2.7. Communication management

The home node manages communication for all servers using the RVM segment. In the function shipping model, the Communication Manager tracks the spread of transactions from node to node. In the data sharing model, transactions stay put while data spreads from node to node, so the Communication Manager on each node is reduced to providing a server name registration and lookup service. A server is registered only with the Communication Manager on the home node, and contacts the home node with any lookup requests. A Communication Manager may contact a Communication Manager on another node to satisfy lookup requests. From the perspective of the name service, servers on using nodes appear to be running on the home node. The advantage of this decision is that it simplifies configuration management, and interacts well with the other services provided by the home node.

A process that is not a server contacts the local Communication Manager with lookup requests. A using node that is not a home node does not have a local Communication Manager; in this case, a surrogate Communication Manager forwards all requests to a particular home node.

## 4.2.8. Configuration management

The home node manages the configuration of all servers that use the RVM segment. The Remote Execution Manager on each using node responds to requests from the home node to start and kill servers. The Remote Execution Manager also buffers log records for forwarding to the home node, and may act as a surrogate Communication Manager to forward name service requests to the home node.

A privileged user may add, delete, start, or kill servers, and modify the configuration of servers and RVM segments. When a server is added, the user must identify the RVM segment it is to use. If a new segment is specified, it is implicitly created under the user's ownership. If an existing segment is specified, it must be owned by the user. A RVM segment is implicitly destroyed when the last server configured to use the segment is deleted. The configuration of a server includes the name of the node on which the server is to run.

Using the home node for configuration management offers several benefits. Security for RVM segments is easily provided. Consistency of the configuration is easily guaranteed; it is not possible to configure a server to use a non-existent segment. A using node that is not a home node requires no non-volatile storage. The disadvantage of using the home node for configuration management is that it contributes to the reduction of using node autonomy. Also, it is more difficult to extend the design to allow a given server to use several RVM segments from different home nodes.

## 4.3. Discussion

Feasibility of implementation guided the selection of many alternatives within the architecture. The utility of the selected alternatives may be summarized by indicating the degree to which the alternatives meet distributed program requirements.

- Ease of programming. By utilizing the Camelot library, the design provides the same easy-to-use programmer interface. No syntax changes are necessary to allow a Camelot server to access shared RVM. The fact that a RVM segment is shared by several servers is encapsulated in the home node's configuration database.

- Data integrity. The design maintains all of the guarantees of transactions even with concurrent access on multiple nodes and in spite of message, server, or node failure. Camelot's algorithms for RVM transactions are easily extended to TDSM via the mechanism of a common log on the home node.

- Multiple access to data. Servers on many nodes can be configured to use the same RVM segment. The servers execute concurrently, and may read and write individual pages concurrently, with the restriction that, at the moment a page is being written, it is accessible only on the node that is writing.

  A more significant restriction is that a given server may access only one RVM segment. Although this restriction was originally dictated by the Camelot library, it simplifies the design of many transaction functions by guaranteeing that the log records for a transaction's activities in a single server may be found at a single node. If a server could access multiple RVM segments from different nodes, then the log records for each segment would have to be buffered separately, and forwarded to different logs. The Camelot library would have to note the node at which a transaction was initiated, and notify each home node's Transaction Manager as the transaction accesses an RVM segment from that node.

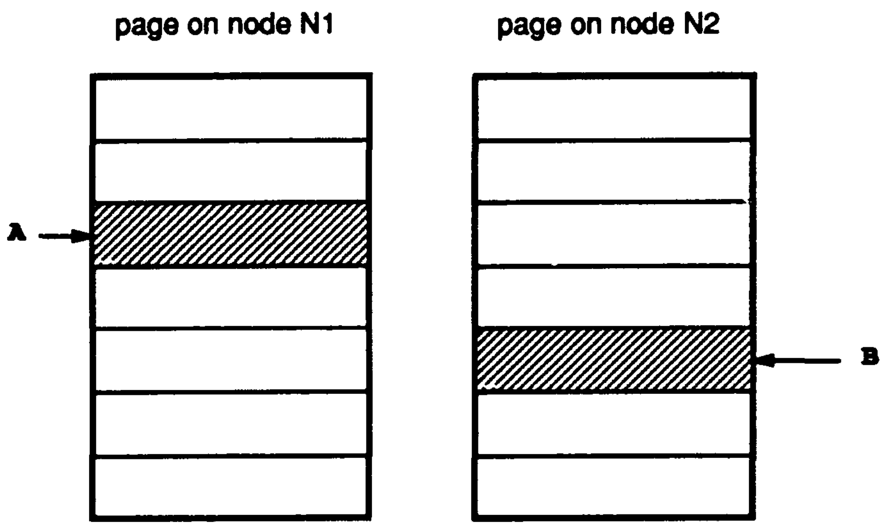
- Performance. The architecture attempts to minimize communication costs by buffering log records, batching undo and redo operations, caching locks, and keeping virtual memory pages resident. As will be shown in Chapter 7, it offers opportunities for further reducing communication costs by prefetching a page with any lock request for an object on that page, and by appending log records to pages being written back to the home node or to server votes during transaction commit. Using an existing transaction processing facility as a base allows the performance of TDSM to be compared to transactional RPC. Some alternative architectures that promise better performance are discussed in Chapter 2. However, to the best of my knowledge, none of these architectures has been implemented in a distributed computing environment.

- Security. Security relies on the authentication performed by the Camelot Node Server, which stores the configuration of servers and segments. A server must be configured before it may access a RVM segment. An unauthorized user may not configure a server to access a RVM segment created by another user.
- Availability. If a using node crashes, any pages for which it has write permission will be unavailable on other nodes. Locks obtained by transactions that have entered the prepared state on the crashed node will also be unavailable. If the home node crashes, processing on cached data may continue, but all transactions will abort when the home node restarts.
- Incremental growth. A server is restricted to one RVM segment, and the non-volatile storage for a segment is stored on one node, so the architecture does not aid incremental growth of storage capacity. Extending the programmer interface of Camelot to allow a given server to access multiple RVM segments could assist in incremental growth of storage capacity. Processing capacity can be increased by configuring servers on additional nodes up to the point that the home node becomes overloaded or the communications network becomes saturated.

In the architecture, there is no direct interaction between transaction management and coherency control: a page may migrate from node to node at any time without regard to transaction boundaries. This decision is necessitated by the possibility of *false sharing*, when two independent objects reside on the same virtual memory page. Suppose transaction 1 modifies A and transaction 2 modifies B independently. If the two transactions are on the same node, or if A and B are on different pages, neither transaction has to wait for the other. If the transactions are on different nodes, and A and B are on the same page, the single-writer, multiple-reader protocol will force one transaction to wait. If pages could not migrate until transaction commit time, transactions could deadlock waiting for each other's pages. Thus, the architecture does allow pages to migrate before transaction commit, and reduces the waiting time at the possible expense of thrashing if two transactions contend for the same page over a period of time.

To avoid page thrashing, application programmers should take care placing objects in the RVM segment so that false sharing does not occur. See [Bolosky et al 89] for a brief discussion of this issue in the context of a shared memory multiprocessor with non-uniform memory access times (NUMA).

Page migration independent of transaction boundaries is correct. Consider an example as illustrated in Figure 4-2. A transaction on node N1 locks and modifies object A, spooling log records describing that modification. Another transaction on node N2 wants to modify object B on the same page as A. Since A and B are different objects, the transaction on node N2 has no difficulty obtaining a lock for B. However, when it tries to modify B, it takes a page fault, and the



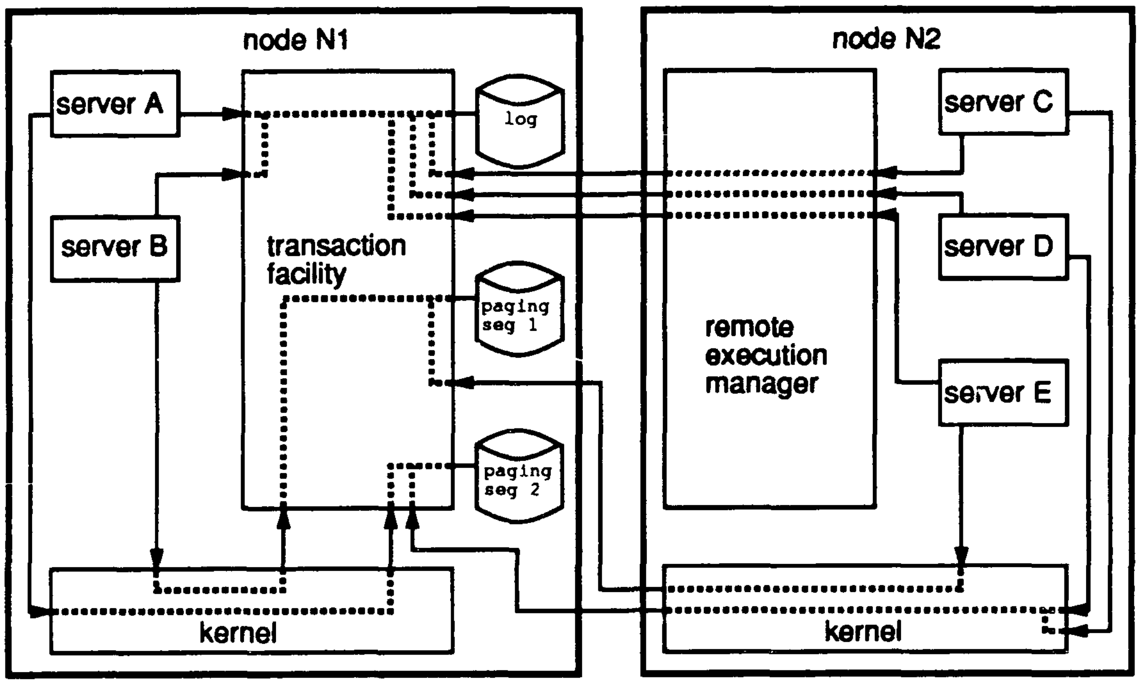
A transaction on node N1 modifies object A during the same period that a transaction on node N2 modifies object B on the same page.

**Figure 4-2:** False sharing

kernel on node N2 contacts the home node External Memory Manager. The home node asks node N1 for the page, and node N1 supplies the modified page, removing its own access to the page. Next, the home node asks node N1 for log records, and node N1 supplies log records describing the as yet uncommitted modification. Now the home node can forward the page to node N2's kernel, which returns from the page fault, and allows the transaction to modify B. If node N1 again tries to access the page, it will fault, and the page will migrate back to node N1 in a similar manner (and log records describing the modification to B will be forwarded to the home node before node N1 sees the modified page).

Note that serializability has been maintained, since both nodes obtain the appropriate locks before accessing shared data. If the transaction which modified A aborts, the home node can restore the page to the proper state, since it has log records describing the modification. The home node may even ask node N2 to undo the modification if it is holding the page, since the lock for A will be held by node N1 until the abort is complete.

Figure 4-3 illustrates the flow of log records and paging requests for two nodes. Node N1 is a home node for segments 1 and 2. Both nodes are using nodes for both segments. The transaction facility on node N1 enters log records for both segments into a common log. The transaction facility receives log records directly from servers A and B. On request, the Remote Execution Manager on node N2 forwards log records generated by server C, server D, or server E to node N1. Servers C and D physically share memory; the kernel on node N2 merges paging requests from these two servers into a single stream that it forwards to the transaction facility on node N1. The transaction facility on node N1 processes these requests for segment 2 in conjunction with requests it receives from server A via the kernel on node N1. The transaction facility also processes paging requests for segment 1 that it receives from server B (via the kernel on node N1) and from server E (via the kernel on node N2).



Log records flow to the transaction facility directly on the home node, or via the Remote Execution Manager on other nodes. Paging requests enter the transaction facility via the appropriate kernel. Servers B and E share segment 1. Servers A, C, and D share segment 2.

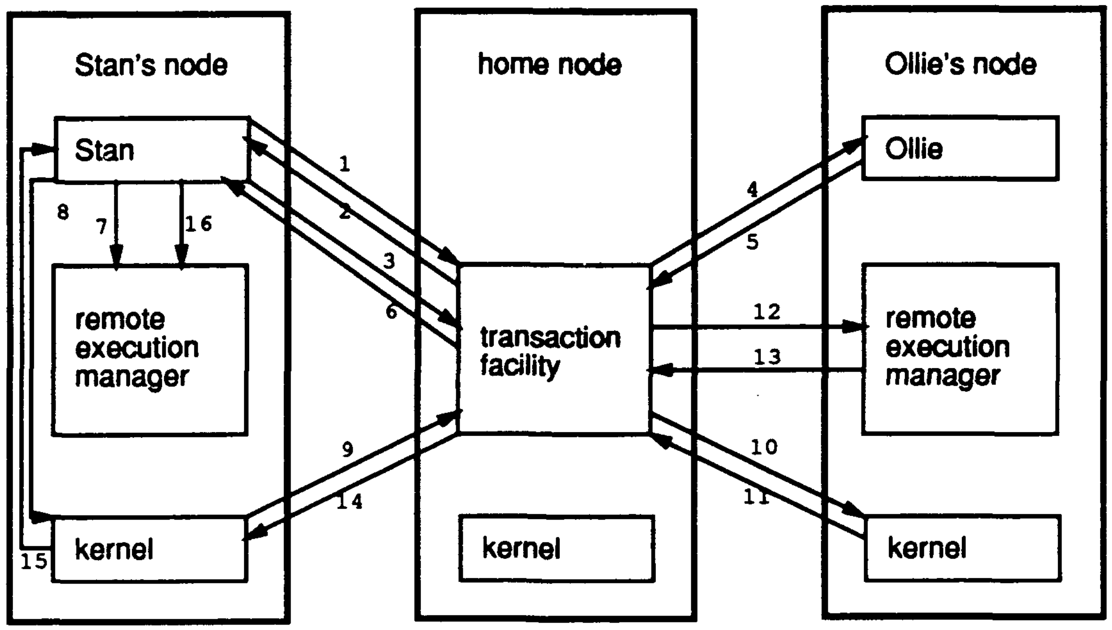
**Figure 4-3:** Flow of log records and paging requests

## 4.4. Example

To illustrate how the architecture works, consider the following example. Stan and Ollie are two servers on different nodes. They share a RVM segment whose home node is a third node. Ollie has cached the page of the RVM segment where `mydata` resides, as well as a lock for `mydata`. However, no transaction in Ollie is currently accessing `mydata`. Stan has neither the lock nor the page cached. Stan executes the following transaction:

```
BEGIN_TRANSACTION
  LOCK(&REC(mydata), LOCK_SPACE_SHARED, LOCK_MODE_WRITE);
  MODIFY(REC(mydata), value);
END_TRANSACTION(status)
```

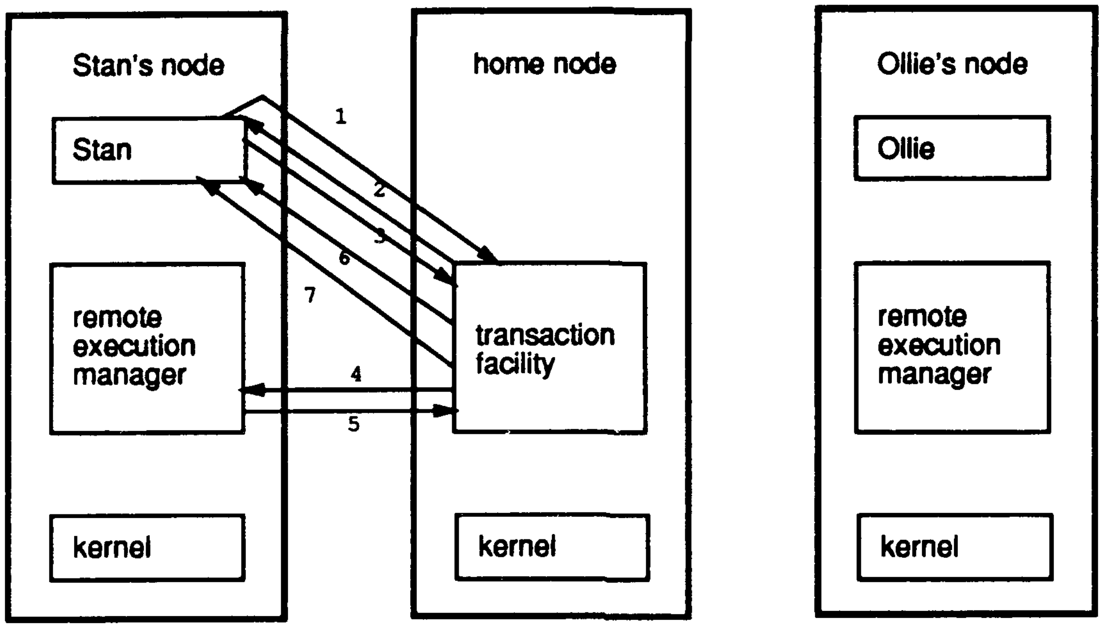
When the `BEGIN_TRANSACTION` statement is executed, Stan makes an RPC to the home node Transaction Manager requesting a new transaction identifier (`tid`). The Transaction Manager's reply provides several transaction identifiers, so that Stan's next few `BEGIN_TRANSACTION` statements may be executed without the need for another RPC. Next, Stan must obtain a write lock for the object that is to be modified. Stan makes an RPC to the home node Lock Manager to obtain the lock. The lock is currently cached by Ollie, so the Lock Manager makes an RPC to Ollie requesting its return. No transactions are holding the lock, so Ollie uncaches the lock and replies to the Lock Manager. The Lock Manager may then reply to Stan's request for the lock.



1 Begin transaction
2 Begin transaction reply with `tid`
3 Lock `mydata`
4 Uncache lock request
5 Uncache lock reply
6 Lock reply
7 Pin request
8 Page fault
9 Request page
10 Flush page
11 Page write
12 Log buffer request
13 Log buffer reply
14 Page provided
15 Return from page fault
16 Log request

**Figure 4-4:** Messages for BEGIN_TRANSACTION, LOCK, and MODIFY

Next, Stan must follow the pin-update-log protocol to modify RVM. (The *pin* step lets the buffer management function know that a modification is about to occur, and prevents the Disk Manager from writing the page to non-volatile storage. The *update* step actually changes data in virtual memory, and the *log* step generates a log record describing the modification, and unpins the page.) First, Stan sends a pin request to the Remote Execution Manager by storing the request in a region of memory that is shared by Stan and the Remote Execution Manager. Then, Stan attempts to store a new value in RVM. Because the page containing the object is not resident, Stan will take a page fault. The kernel determines that the page is managed by an external pager, and sends a request for the page to the External Memory Manager on the home node. The External Memory Manager knows that Ollie's node has the page in write mode, and asks Ollie's kernel to flush the page. Ollie's kernel sends the modified page back to the External Memory Manager, and removes the page from its memory. The External Memory Manager passes the page to the Disk Manager. The Disk Manager asks Ollie's Remote Execution Manager to forward any buffered log records. Ollie's Remote Execution Manager replies with an empty buffer, so the Disk Manager writes the page to non-volatile storage with no further ado. The External Memory Manager copies the page to Stan's kernel, which returns to Stan from the page fault. After storing the new value, Stan sends a log request to the Remote Execution Manager, again by buffering the request in a shared memory region. See Figure 4-4.

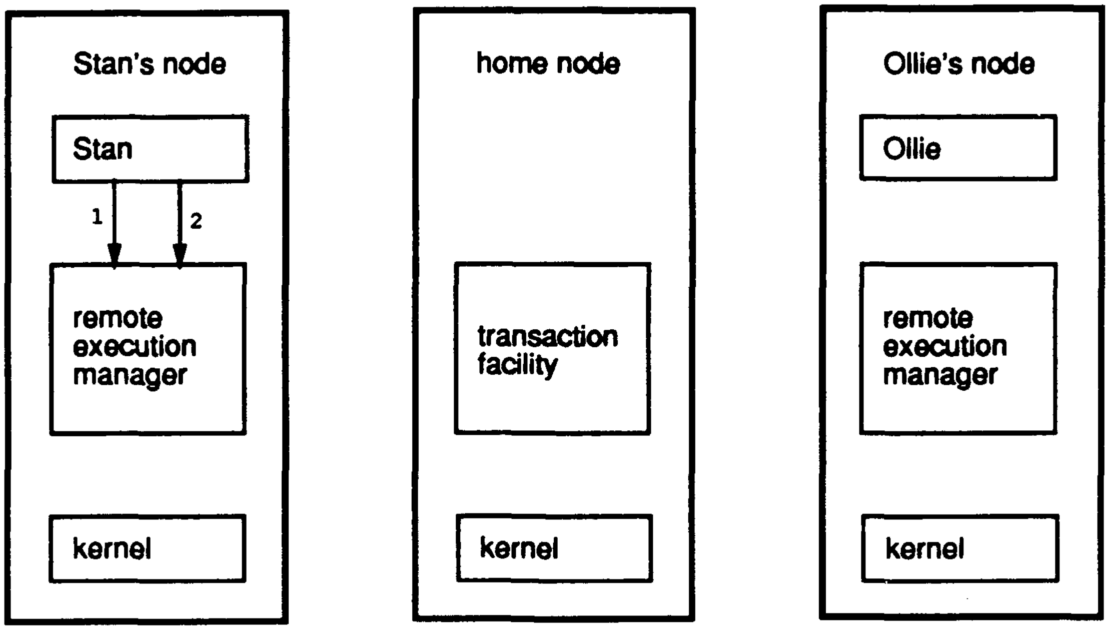


1 End transaction `tid` request
2 Vote request
3 Vote reply
4 Log buffer request
5 Log buffer reply
6 End transaction reply
7 Commit `tid` notification

**Figure 4-5:** Messages for END_TRANSACTION

When the `END_TRANSACTION` statement is executed, Stan makes an RPC to the home node Transaction Manager requesting that the transaction be committed. The Transaction Manager contacts Stan, the only server involved in the transaction, requesting a vote. If Stan votes to commit the transaction, the Transaction Manager asks the Disk Manager to force all log records to stable storage, and write a commit record. The Disk Manager makes an RPC to Stan's Remote Execution Manager to obtain the buffered pin and log requests, and writes them to the log, followed by a commit record. The Transaction Manager replies to Stan's request to commit, and sends a message notifying Stan that the transaction committed. See Figure 4-5.

If Stan's transaction commits, and no other node accesses `mydata`, then the lock and virtual memory page for `mydata` will remain cached on Stan's node. In this case, Stan can execute the same transaction again at a much lower cost. To begin the transaction, Stan can use a `tid` obtained from a previous RPC to the Transaction Manager. Since Stan still has the lock for `mydata` cached, the transaction can obtain the lock immediately. Stan again follows the pin-

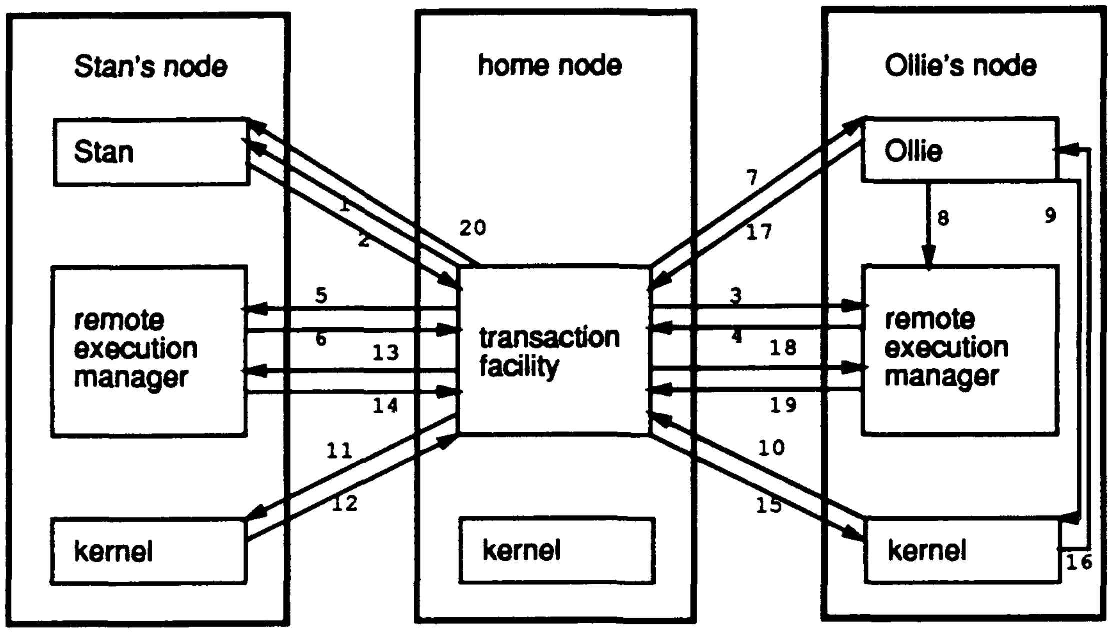


1 Pin request 2 Log request

**Figure 4-6:** Messages for LOCK and MODIFY

update-log protocol. The pin request is stored into a region memory shared with the Remote Execution Manager, Stan updates `mydata` *without* taking a page fault, and the log request is stored into the shared memory region. The transaction could update several other objects in the recoverable segment with no further communication as long as the locks and virtual memory pages are cached, and the shared memory queue of pin and log requests does not become full. When the `END_TRANSACTION` statement is executed, the transaction will commit at the cost of three RPCs and one asynchronous message. See Figure 4-6.

If some event causes Stan's transaction to abort before the `END_TRANSACTION` can complete, the following sequence could occur (if Stan and Ollie are still running). First, the home node Transaction Manager makes an RPC to Stan instructing Stan to suspend all activity on behalf of transaction `tid`. After Stan replies, the Recovery Manager asks the Disk Manager for a current copy of the log. The Disk Manager makes RPCs to both Remote Execution Managers to obtain any buffered log records, and returns the up-to-date log to the Recovery Manager. The Recovery Manager selects a server to reverse the modifications made by the transaction to the RVM segment. If Ollie is picked, the Recovery Manager makes an RPC to Ollie listing the modifications to be undone. To make the changes, Ollie does only the pin-update part of the pin-update-log protocol. Ollie does not need to lock the data being modified because the lock is still held by the transaction being aborted. Since Stan's node has the only copy of the page that Ollie wants to modify, Stan will page fault. The page fault is handled in the same manner as before. After Ollie makes the modification, Ollie replies to the Recovery Manager. The Recovery Manager notifies the Transaction Manager that restoration is completed. The



| | |
|---|---|
| 1 Suspend `tid` request | 11 Flush page |
| 2 Suspend reply | 12 Page write |
| 3 Log buffer request | 13 Log buffer request |
| 4 Log buffer reply | 14 Log buffer reply |
| 5 Log buffer request | 15 Page provided |
| 6 Log buffer reply | 16 Return from page fault |
| 7 Restore request | 17 Restore reply |
| 8 Pin request | 18 Log buffer request |
| 9 Page fault | 19 Log buffer reply |
| 10 Request page | 20 Abort `tid` notification |

**Figure 4-7:** Messages for abort

Transaction Manager asks the Disk Manager to write an abort record, and force log records to stable storage. The Disk Manager makes another RPC to Ollie's Remote Execution Manager to get the buffered pin request, and writes them to the log followed by an abort record. The Transaction Manager sends a message notifying Stan that the transaction aborted. See Figure 4-7.

If Stan crashes after the `MODIFY`, but before the `END_TRANSACTION`, then Stan's transaction must abort: When another server requests a lock for `mydata`, the home node Lock Manager tries to contact Stan and fail. After a timeout, it aborts all transactions in Stan, and then grants the lock to the requester. When another node tries to access the page where `mydata` resides (perhaps by request of the home node Recovery Manager which is aborting Stan's transactions), the node's kernel requests the page from the home node External Memory Manager. The External Memory Manager tries to contact Stan's kernel and fails; after a timeout, it invokes the server recovery algorithm of the Recovery Manager. To recover a server in an active segment, the Recovery Manager must undo the effects of any uncommitted transactions in the server, and redo the effects of any committed transactions that are not reflected in non-volatile storage. If Stan crashed before forwarding to the home node either the log records describing the modification or the modified page, then there is no work for the Recovery Manager to do: it can simply read the old version of the page from non-volatile storage. However, if Stan committed the transaction once, tried to run it again, and then crashed before committing the transaction a second time, the Recovery Manager must redo the modification made by the first transaction, since this modification is not reflected in the non-volatile (and now only existing) copy of the page. Or, if Stan crashed after forwarding to the home node both the modified page and log records describing the uncommitted modification, the Recovery Manager must undo the uncommitted modification that appears on the modified page.

## 4.5. Summary

One of the trickiest functions to provide when designing a TDSM system is recovery. The nature of data sharing systems requires some form of concurrency control and buffer management; thus these functions have appeared in the literature more frequently than recovery. Traditional approaches to recovery in transaction systems do not apply directly to TDSM because a single recoverable object may involve many nodes instead of just one:

- one or more nodes updating the object
- one or more nodes performing recovery actions on the object
- one or more nodes holding pieces of different versions of the object
- one or more nodes holding log records describing past updates to the object

The key design decision underlying the architecture is the concept of the home node. This decision simplifies the architecture by providing a simple, direct method for implementing transactions:

- Serializability is achieved via distributed locks. As soon as a server is initialized it knows which node to contact to obtain a distributed lock on behalf of a transaction.
- Permanence is achieved via non-volatile storage of the recoverable segment on the home node, and stable storage of log records on the home node. A server always knows where to obtain the most recent copy of a recoverable virtual memory page, and where to log the changes made by a transaction.
- Failure atomicity is achieved via stable storage of log records on the home node. The Recovery Manager on the home node has no difficulty finding all of the log records for the recoverable segment.

The home node concept is not necessarily the best choice for a TDSM architecture, but it meets the goal of feasibility and utility. Caching increases the autonomy of using nodes by allowing the bulk of the work within a transaction to proceed with no intervention by the home node until transaction commit. Caching also reduces the load on the home node; the number of messages increases linearly with the number of transactions, rather than linearly with the number of operations.

The primary disadvantage of the home node concept is its negative impact on availability. Failure of the home node prevents transactions on all using nodes from committing. Chapter 7 discusses other architectures that offer solutions to this problem.

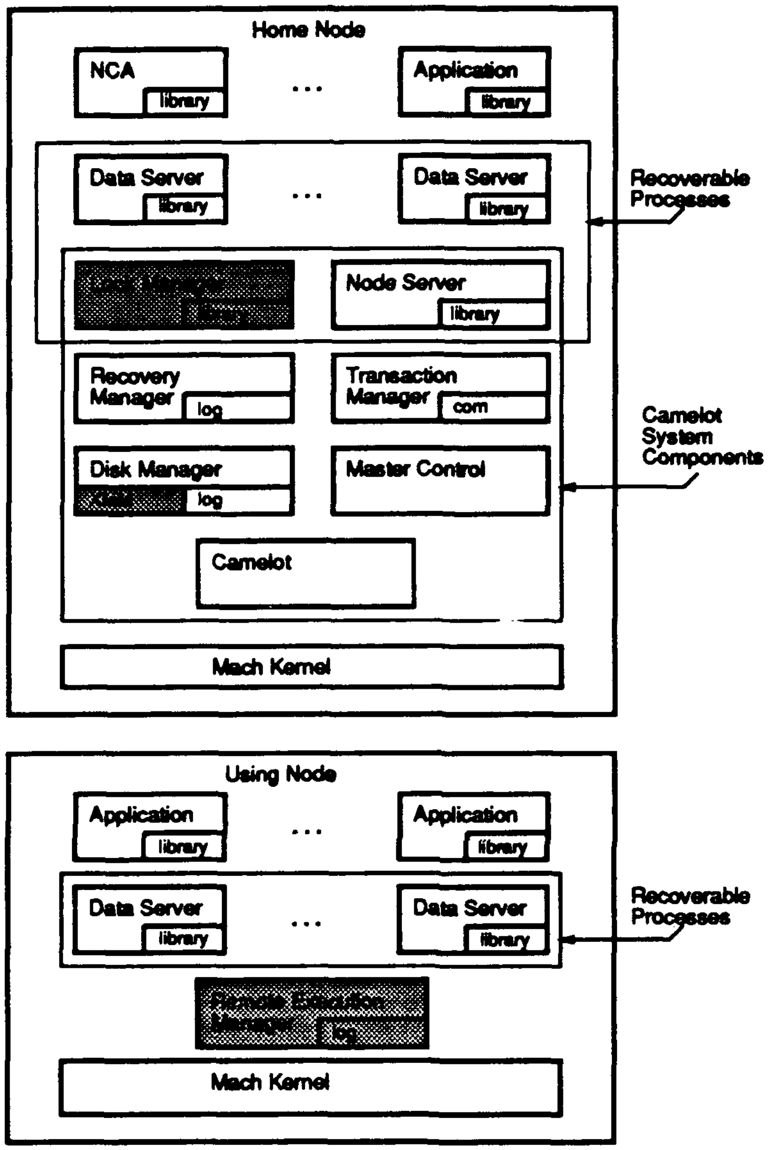
# Chapter 5
# Implementation

The transactional distributed shared memory (TDSM) architecture outlined in Chapter 4 was implemented by extending the existing Camelot 1.0 (release 83) distributed transaction facility, which runs on the Mach operating system. Both Camelot and Mach were introduced in Chapter 3. This chapter parallels Chapter 4 by presenting the TDSM implementation in terms of seven management functions: log management, transaction management, recovery management, buffer management, concurrency management, communication management, and configuration management. Figure 5-1 highlights the additions to the Camelot architecture for TDSM. (The interfaces that were added or changed in Camelot to support TDSM are listed in Appendix B.) Some of the material in this chapter is derived from a comparison of the original Camelot source code to the Camelot/TDSM source code using the UNIX `diff` utility.

## 5.1. Concurrency Management

In the original Camelot system, concurrency control is provided by the Camelot library which is linked with every Camelot server. To obtain a lock via the Camelot library, the server programmer might use this statement: `LOCK(LOCK_NAME(REC(data)),lockSpace, LOCK_MODE_WRITE)`. This statement requests a write lock on an item in recoverable storage named `data`. (The name of the lock is the address of `data` in recoverable storage.) Usually, `lockSpace` is `LOCK_SPACE_PRIMARY`. However, a package supporting an abstract data type may use other lock name spaces to avoid conflicting with other uses of the same lock names.

For Camelot/TDSM, responsibility for concurrency control is divided between the Camelot library in each server, and the home node Lock Manager. The Lock Manager coordinates the use of those lock spaces pre-assigned to the home node. On startup, the Lock Manager identifies itself to the Disk Manager via a `DH_Initialize` remote procedure call (RPC). Later, servers may obtain a port for the Lock Manager from the Disk Manager via a `DS_GetHPort` RPC. In response to a server's lock request on behalf of a given transaction, the Lock Manager may decide to grant the lock to the server for the duration of the requesting transaction. (The server will contact the Manager when the transaction completes.) Or, to eliminate future requests, the Lock Manager may allow the server to *cache* the lock, giving it permission for an unspecified period of time to process lock requests by subsequent transactions. In this latter case, the Lock Manager uses a *call-back* to the server to request the return of the cached lock when another server needs it.



The Lock Manager, Remote Execution Manager, and External Memory Manager (XMM) are additions to Camelot to support TDSM. Compare to Figure 3-1.

**Figure 5-1:** Camelot/TDSM architecture

The Camelot library in each server is responsible for coordinating the use of locks by transactions running in the server. In response to a transaction's lock request, the library must decide whether it may grant the request by itself (when the lock is in a lock space local to the server, or when the lock is cached), or if it must contact the Lock Manager. The library must also respond to a Lock Manager request to return a cached lock.

### 5.1.1. Programmer interface changes

In the original Camelot system, all lock name spaces are managed solely by the Camelot library. For Camelot/TDSM, some lock name spaces are still managed solely by the library, while others are managed collectively by the library and the Lock Manager. If `lockSpace` is `LOCK_SPACE_PRIMARY`, or any value less than `LOCK_SPACE_SHARED`, the lock is local to the server, and the library will not contact the Lock Manager. If `lockSpace` is `LOCK_SPACE_SHARED`, the library will use a lock space that is shared with exactly those servers that share the same recoverable segment. If `lockSpace` is a value greater than `LOCK_SPACE_SHARED`, the lock space will be shared with servers that specify the same value for `lockSpace`. Shared lock spaces are mediated by the Lock Manager.

The division of lock name spaces into local and shared spaces is unfortunate, because it forces programmers to change their code if they wish transactions using a shared RVM segment to be serializable. (Suppose two transactions in different servers wish to update the same object in a RVM segment that they share. If each transaction uses its own local lock space, both transactions can simultaneously hold a write lock on the object. For correctness, the transactions must use a shared lock space, so that only one of them at a time can hold the write lock.) However, because a lock in a shared lock space uses more resources than a lock in a local lock space, the division of lock name spaces allows programmers to avoid the more expensive shared locks when accessing objects that are not shared. (A shared lock is more expensive because an RPC is needed to cache the lock initially, and the cached lock uses a small amount of virtual memory even when it is not held by a transaction.)

### 5.1.2. Cache control

Several servers may cache a lock in read mode, but only one may cache a lock in write mode. However, a transaction may involve multiple servers concurrently, making it possible for a given transaction to hold a write-mode lock in more than one server. (However, at most one server will be caching the lock in this case.)[4] When a lock is held by multiple servers, the Lock Manager consults an application-specified *lock policy* (set via the `SetLockPolicy` procedure call to the lock library) to decide which one of the servers (if any) is permitted to cache the lock.

The Lock Manager consults a lock policy to determine which server is permitted to cache a lock. Since a given transaction may include several servers, only one of which may cache the lock in write mode, the Lock Manager may transfer caching privileges from one server to another while the lock is held in both servers. Four policies are provided:

[4]A lock is granted to a transaction, which may involve multiple threads in one or several servers. Since these threads are part of the same transaction, they may hold a write-mode lock simultaneously, which may violate serializability unless each thread runs as a nested subtransaction. The write-mode lock is held by only one of these subtransactions at a time. But in the Camelot system, when a subtransaction commits, its locks are not released, but assigned to its parent ("anti-inheritance"). Thus, the parent may hold the write-mode lock at the same time as each one of its children.

- **never**: the lock is never cached by the server. All requests are forwarded to the Lock Manager.
- **release**: if a server requests a lock cached by another server, neither server is allowed to cache the lock. But if a server requests a lock when the lock is not held by any other server, it is permitted to cache the lock.
- **first**: if a server requests a lock cached by another server, the other server is permitted to retain its cached lock. That is, the first server to cache a lock will retain the cached lock (until it is requested by a different transaction family in another server, or until another server requests the lock and specifies a policy of **last**).
- **last**: if a server requests a lock cached by another server, the new server will cache the lock, and the server which cached the lock originally will no longer have it cached. That is, the last server to request a lock will retain the cached lock.

The latter three policies (**release**, **first**, and **last**) are equivalent for read-mode locks, and for write-mode locks when the requesting transaction is not in the same family as the transaction holding the lock. (In the latter case, there is a conflict, and the requesting transaction must wait until the holder releases the lock. When the holding transaction releases the lock, it will be uncached by the server, and the requesting transaction will be granted the lock.)

Each lock request that the server library makes to the Lock Manager includes a lock policy, so that each lock may have a different policy, and the policy may vary over time. If two servers specify conflicting lock policies, the Lock Manager should resolve the conflict in favor of the server specifying the policy with the highest priority (**never** has the lowest priority, and **last** has the highest priority). Presently, this scheme is not fully implemented.

The server programmer may also specify how many locks the library may cache per lock space. This limit is evaluated lazily; that is, the server library consults the limit only when a transaction unlocks a lock. If the limit has been exceeded, the library will uncache the lock as it is unlocked. This scheme is incorrect because it fails to uncache locks that have been unused for long time periods. However, it was simpler to implement than a full LRU for the lock cache, and has no adverse effect on the performance results reported in Chapter 6.

### 5.1.3. Failures

To gain speed, most of the state transitions made by the distributed lock manager are recorded only in volatile storage. Parts of this storage are lost when a server or Lock Manager crashes. The Lock Manager can restore its volatile state by contacting each of the data servers known to have obtained locks. Therefore, when the Lock Manager first encounters a server, it records the server's identifier in recoverable storage.

As mentioned previously, the Lock Manager uses a call-back to request a server to return a cached lock. This call will fail when the server or the processor where it resides has crashed, or if there is a communications failure (perhaps a network partition). In the case of a crash, all transactions in the crashed server that have not yet reached the prepared state will abort, so it is safe for the Lock Manager to grant a cached lock to a different server as long as the lock is not held by a prepared transaction. If the lock is held by a prepared transaction, the Lock Manager must wait for the transaction to commit or abort before granting the lock to another server. Thus, if the Lock Manager crashes, it should scan the log, where prepared transactions record the locks they hold, to determine which locks are held by prepared transactions. (This log scan is not presently implemented.)

In the case of a network partition, it would be unsafe for the Lock Manager to grant a cached lock to a different server, because two transactions could then be holding the lock in conflicting modes, and serializability would be violated. In the present implementation, the Lock Manager treats communication failures as crashes, and thus may allow serializability to be violated if failure is due to a network partition. This violation of serializability could be prevented by requiring the Lock Manager to retry failed requests until it succeeds in contacting the server caching the lock, or by allowing it to abort transactions in the uncommunicative server.

### 5.1.4. Lock Manager/library interface

Camelot runs with the support of the Mach operating system, and uses Mach messages to perform RPCs. Most communication between application and server, or between server and server, uses a Camelot variant of the Mach RPC called a SERVER_CALL. The SERVER_CALL embeds additional *history* information in the RPC message in order to track the spread of transactions from node to node, and causes the message recipient to join the sender's transaction and to participate in the two-phase commit protocol at the end of the transaction. To avoid the overhead of distributed two-phase commit, and the embedded history information, the distributed lock manager communicates via the simpler Mach RPC rather than the Camelot SERVER_CALL.

The communication interface between the Lock Manager and the server library is listed in Appendix B. For example, the library may attempt to obtain a write-mode lock on behalf of transaction `tid` via the call: `HS_Lock(hsPort,lockName,LOCK_MODE_WRITE, LOCK_CACHE_FIRST,tid,sPort,&cached)`. The name of the lock (including the lock name space) is contained in variable `lockName`. The library is selecting a lock policy of **first**. The call is directed to the Lock Manager identified by the Mach port `hsPort`; the library is providing the server's port for call-backs in the variable `sPort`. The Lock Manager will return a value of `true` in `cached` if the server library is permitted to cache the lock.

### 5.1.5. Data structures

The *cache status* indicates the mode (read, write, or none) in which a server has cached a lock. When a server has a write-mode lock cached, it does not communicate at all with the Lock Manager to grant a lock. When a server has a read-mode lock cached, it may grant read requests on its own, but must forward write requests to the Lock Manager. When a server does not have the lock cached, it must forward all requests to the Lock Manager. It is possible for a server to have a lock cached in read mode, while several transactions running in the server hold the lock in write mode.

Both the Lock Manager process and the server library use hash tables of lock records, indexed by lock name, to keep track of active locks. (A lock is active if it is cached by a server or held by a transaction.) In the server library, each lock record indicates the cache status, and includes a list of transactions holding the lock, and a list of transactions waiting for the lock. In the Lock Manager, each lock record includes a list of servers caching the lock (and the cache status for each server), a list of server/transaction pairs holding the lock, and a list of server/transaction pairs waiting for the lock.

The Lock Manager also maintains a recoverable storage hash table of server identifiers. When the server library makes a request to the Lock Manager, it includes a special port known as the *server port*, which uniquely identifies the server. When the Lock Manager receives a request that contains a server port it has not previously seen, the Lock Manager make a `DH_PortToServerId` RPC to the Disk Manager to translate the port into a unique identifier known as the *server id*. The Lock Manager records this server id in recoverable storage. If the Lock Manager crashes and recovers, it scans the recoverable server id table. It uses a `DH_PortToServerId` RPC to the Disk Manager to translate each server id into the current corresponding server port, and contacts each server to obtain the data the Lock Manager needs to reconstruct the volatile lock hash table.

## 5.2. Buffer Management

In the original Camelot system, buffer management is provided by the Disk Manager in cooperation with the Mach kernel. The Disk Manager acts as a Mach external pager, processing page-in and page-out requests made by the kernel. It coordinates paging I/O with log I/O to enforce the write-ahead log invariant. The Disk Manager also tracks active servers and active recoverable virtual memory segments.

For Camelot/TDSM, several new components enter the picture. The kernels on several using nodes may make paging requests to the home node. On the home node, the External Memory Manager acts as an intermediary between the Disk Manager and these kernels to maintain the coherency of each RVM segment across nodes. The Remote Execution Manager on each using node acts on behalf of the home node Disk Manager to startup and terminate servers, and to forward buffered log records to the home node.

This section first presents an overview of each component, describing the external interfaces and internal data structures. Next, the algorithms used to implement buffer management functions are outlined.

### 5.2.1. External Memory Manager

Camelot was designed to use the Mach external pager interface to provide recoverable virtual memory. When a server reads or writes a page of a recoverable segment, the Mach kernel on the server's node may make page-in or page-out requests to the Disk Manager on the segment's home node. For Camelot/TDSM, a new component called the External Memory Manager is interposed between the Disk Manager and the Mach kernel. The task of the External Memory Manager to take requests from multiple kernels and make them appear to the Disk Manager as if there were but a single kernel making the requests.

The External Memory Manager, then, has three types of interfaces. To each Mach kernel, it acts as an external pager. To the external pager built-in to the Disk Manager, it acts as a kernel. And to allow the Disk Manager to connect the appropriate kernels to the appropriate RVM segment, it has an interface that allows *paging objects* to be created and hooked together.

The External Memory Manager allows multiple servers to have a read-only copy of a page. But when a server attempts to write a page, all other copies of the page are invalidated. If a dirty page is requested by another kernel, the External Memory Manager tells the writer to send back the dirty page, and turn off write permission. The net effect of this algorithm is to provide the illusion of several threads sharing an address space, when the threads exist on different machines that do not physically share memory.

In its first incarnation, the External Memory Manager was a separate task which communicated with the Disk Manager via Mach messages. For performance reasons, the External Memory Manager was changed to be a library within the Disk Manager.

### 5.2.2. Disk Manager

The Disk Manager has interfaces to practically every other component in the Camelot system; an understanding of the Disk Manager's operation is tantamount to understanding Camelot. The interfaces to the Disk Manager are presented as data structures and algorithms are introduced.

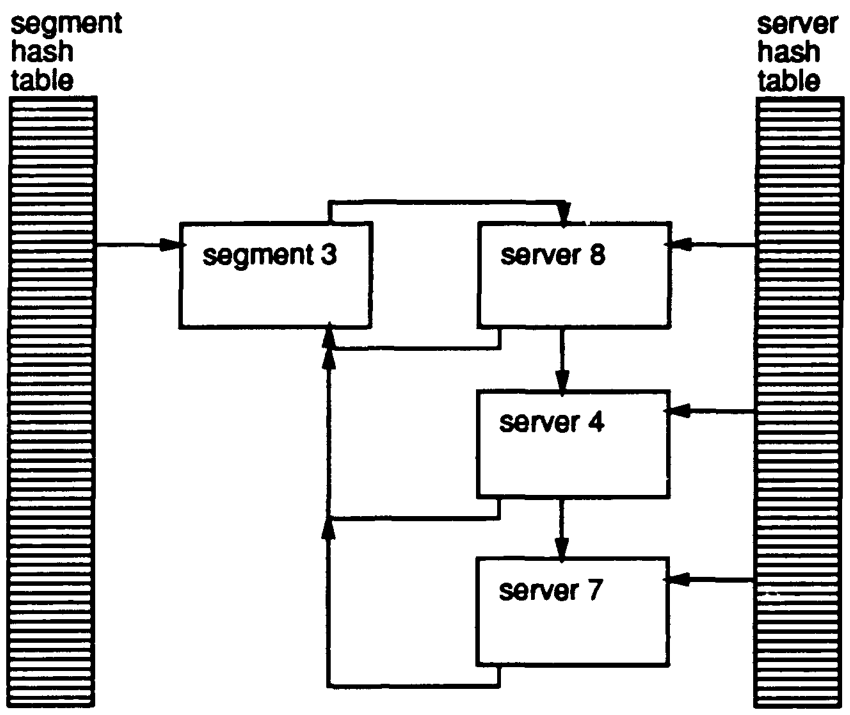
### 5.2.2.1. Server record

The server record `s_record_t` contains everything the Disk Manager needs to know about a particular Camelot server. (Interestingly, the server record does not hold the server's name or executable command line; the Disk Manager makes a `ND_GetRestartAdvice` RPC to the Node Server to obtain this information when needed.) In the original Camelot system, a RVM segment could not be shared between servers, so the server record also contains everything the Disk Manager needs to know about the server's RVM segment. For Camelot/TDSM, a given RVM segment may be shared by more than one server, so a separate segment record (described in the next subsubsection) contains information for a RVM segment. Thus, the information in the server record about the server's RVM segment is replaced with a pointer to the appropriate segment record, and a link field that is used to chain together all servers using a given RVM segment (see Figure 5-2).

Camelot uses the UNIX `fork` operation to start data servers. For a server running on the home node, the server is a child of the Disk Manager, and the Disk Manager stores the child's UNIX process id in the server record. When a child exits, the Disk Manager uses a hash table on UNIX process id to locate the correct server record. With Camelot/TDSM, a server may run on a remote node; in this case, the server is a child of the Remote Execution Manager on that node. For remote servers, the Disk Manager allocates a Mach port `dxPort` which is stored in the server record, and communicated to the Remote Execution Manager. When a server on a remote node exits, the Remote Execution Manager makes a `DX_ServerDied` RPC to the `dxPort` to notify the home node Disk Manager. The Disk Manager uses a hash table on `dxPort` to locate the correct server record. Thus, the `dxPort` plays the same role for remote servers that the UNIX process id plays for local servers. Also in the server record, the Disk Manager stores `nodeId`, the Internet address of the remote node, and `xPort`, the Mach port on which the Remote Execution Manager accepts requests. (The Disk Manager obtains the `nodeId` from the Node Server, and presents it to the Mach netmsgserver in order to obtain `xPort`.)

### 5.2.2.2. Segment record

The segment record `seg_record_t` was added for Camelot/TDSM to track the state of a RVM segment. It contains several fields (`recoveryLock`, `preparedTrans`, `dirty`, `pagingPort`, `requestPort`, and `segDesc`) that were originally in the server record. The `segDesc` is a segment descriptor `cam_segment_desc` which contains the size of the segment and the segment id. (Since a segment may be shared by several servers, the server id that was originally in the segment descriptor was removed.) The Disk Manager uses a hash table on the segment id within `segDesc` to locate segment records. The `pagingPort` is the Mach port on which the Disk Manager receives paging requests from the External Memory Manager; when the Disk Manager receives a request, it uses a hash table on `pagingPort` to locate the proper segment record. The `requestPort` is the Mach port that the Disk Manager uses to make paging requests to the External Memory Manager. The `recoveryLock` is used to ensure that



The Disk Manager finds segment 3 by using a hash table. A linked list starting from the segment record shows that servers 8, 4, and 7 are using segment 3. On the other side, the Disk Manager finds server 4 by using a hash table. A pointer from the server record shows that server 4 is using segment 3.

**Figure 5-2:** Segment and server records

recovery of a server is complete before another attempt is made to restart it. The Disk Manager uses the `preparedTrans` and `dirty` fields to notify the Node Server of the RVM's state with respect to recovery.

The segment record contains one field that was not present in the original Camelot server record. The `serverPtr` field points to a list of servers using the RVM segment (see Figure 5-2).

### 5.2.2.3. Grid

The most complicated data structure in the Disk Manager is the grid which keeps track of log records. The Disk Manager may wish to locate all of the log records for a particular transaction, or all of the log records which reference a particular page. Grid records form a two-dimensional data structure, with a hash table on page ids as one axis, and a hash table on transaction ids as the other axis. In addition to a log sequence number that can be given to the Log Manager to quickly find a log record, each grid record contains a link for each of the two axes (page and transaction), a pointer to the page record, and a pointer to the transaction record.

Given a page id, the Disk Manager uses the page hash table to find the page record for the page. The Disk Manager may then follow the appropriate grid record links to locate all the log records which reference the page. Similarly, given a transaction id, the Disk Manager uses the transaction hash table to find the transaction record, and may follow the appropriate links to locate log records.

In the original Camelot system, the page record includes a pointer to a server record, since the original Camelot stores information about a RVM segment in the server record. For Camelot/TDSM, the page record points instead to the segment record for the RVM segment to which the page belongs.

#### 5.2.2.4. Internal concurrency control

For reasons of programming ease and performance, the Disk Manager was originally implemented as a multi-threaded program. To synchronize access to data structures, the Disk Manager uses latches. The latches are arranged in a hierarchy. If a thread which acquires multiple latches always acquires the latches in the order defined by the hierarchy, deadlocks will not occur.

In the original hierarchy, the `recoveryLock` is the highest priority, with the `serverLatch` (guarding access to a server record) immediately below. In Camelot/TDSM, information about RVM segments (including `recoveryLock`) which was formerly in the server record is moved to a separate segment record data structure which is guarded by a new latch, `segLatch`. Since a given RVM segment may be used by several servers, the `segLatch` has a priority lower than `recoveryLock` but higher than `serverLatch`. Thus, any routine which wishes to access the segment record for the segment in use by a given server must latch the segment record before latching the server record.

### 5.2.3. Remote Execution Manager

The Remote Execution Manager was added for Camelot/TDSM to perform three functions on each using node at the request of the home node Disk Manager. (The Remote Execution Manager also has interfaces to the Communication Manager discussed later.) On request, the Remote Execution Manager will start a server, kill a server, or pick up the shared memory queue of a server. On startup, the Remote Execution Manager registers itself with the netmsgserver so that Disk Managers may use `netname_look_up` to find the Remote Execution Manager.

The shared memory queue (not to be confused with distributed shared RVM) is a virtual memory buffer for log records. On a given using node, each server has a distinct shared memory queue, physically shared with the Remote Execution Manager. To the Remote Execution Manager, a shared memory queue is an uninterpreted array of bytes, with a **head** index and a **tail** index. As the server produces data describing requests, it stores it in the queue and advances the **tail** index. If the **tail** index reaches the **head** index, the queue is full, and the server must make an

RPC to the home node Disk Manager for subsequent requests. The Remote Execution Manager consumes the data on behalf of the home node Disk Manager. It copies the data between the **head** and **tail** indices into a message that it sends to the Disk Manager, and updates the **head** index.

The only other data structure used by the Remote Execution Manager is its own version of the server record that it uses to keep track of its children. The server record includes the child's UNIX process id, the `dxPort` given by the Disk Manager when it ask for the server to be started, and a pointer to the server's shared memory queue. The Remote Execution Manager uses a hash table on `dxPort` to locate the correct server record when it receives a request from the Disk Manager. It uses a hash table on UNIX process id to locate the correct server record when a child exits.

## 5.2.4. Algorithms

### 5.2.4.1. Forward Processing

Camelot offers two forms of transaction logging. With old-value/new-value logging, Camelot records both a before-image and an after-image of the region of RVM being modified. The before-image is used to restore the old value of the region if the transaction aborts. The after-image is used to restore the new value of the region if needed during recovery. With new-value logging, only the after-image is recorded in the log. Camelot must search the log to find a previous modification to the region in order to restore the old value after an abort.

When a old-value/new-value transaction wishes to modify a page, the page is first pinned (by making a request to the Disk Manager), the server makes the change to the page, and the Disk Manager unpins the page when it receives the log record. For each page, the Disk Manager maintains a pin count, and a list of log records describing the modifications made to the page since it was last written to disk. Before writing a page to disk, the Disk Manager forces the log records to stable storage (*write-ahead logging*) and waits for the pin count to become zero.

When a new-value-only transaction wishes to modify a page, the page is pinned and modified, and the log record is sent to the Disk Manager. Unlike an old-value/new-value transaction, however, the page remains pinned until the end of the transaction.

For performance, the pin and log requests from the server to the Disk Manager are stored in a shared memory queue. The Disk Manager checks the shared memory queue at appropriate times to enforce write-ahead logging. The server uses RPCs to make pin and log requests when the queue becomes full.

When a server is on the home node, the shared memory queue is in memory physically shared with the Disk Manager, and the Disk Manager can check the queue directly. Both the original Camelot system and Camelot/TDSM do this. For Camelot/TDSM, when a server is on a different node, the Disk Manager must make a `XD_GetShMemQueue` RPC to the Remote Execution Manager on that node to get a copy of the queue. Once the Disk Manager receives the copy, it uses the same techniques to process the queue as it does for local servers.

Each server has a separate shared memory queue buffer for pin and log requests. For Camelot/TDSM, when several servers share a given RVM segment, the buffering may cause the home node Disk Manager to process the pin and log requests from multiple servers in a order different from that in which they were originally produced. However, the order in which the Disk Manager processes the requests is still guaranteed to be correct. If two requests refer to regions on different pages, the order of processing does not matter. If two requests refer to regions on the same page, the Disk Manager will pick up the first request from the first server's shared memory queue before it allows the page to migrate to the second server. Thus, the two requests will be processed in the proper order.

For Camelot/TDSM, when the Disk Manager stores a record in the grid, it remembers only the RVM segment to which the record belongs, and it forgets which server is responsible for the record. This means that, if a server malfunctions and does not follow the pin-update-log protocol, the Disk Manager does not know which server is malfunctioning, and it will kill **all** of the servers using the RVM segment. This may seem like an unfair penalty for the "innocent" servers to pay, but it simplifies server debugging in that the problems caused by a malfunctioning server are immediately visible, instead of being delayed by the possibility of data corruption.

#### 5.2.4.2. Coherency control

For Camelot/TDSM, the External Memory Manager is able to manage many RVM segments simultaneously. Each RVM segment has a unique `pagingPort` that the External Memory Manager receives paging requests on. When the External Memory Manager receives requests such as `memory_object_data_write`, it passes the requests on to the Disk Manager's external pager routines. When the Disk Manager's external pager wishes to make a `memory_object` request to the kernel, it actually makes the request to the External Memory Manager, which will forward the request to the appropriate kernels.

To ensure the coherency of a given RVM segment, the External Memory Manager may migrate a page from one node to another. To do this, the External Memory Manager makes a `memory_object_lock_request` to the first kernel. This kernel responds with a `memory_object_data_write` containing the page, which the External Memory Manager passes on to the Disk Manager's external pager. The External Memory Manager does not keep a copy of the page it just received, so it makes a `memory_object_data_request` to the Disk Manager's external pager.[5] The Disk Manager responds with a `memory_object_`

[5]The External Memory Manager is a library linked to the Disk Manager, so this request for the page that the External Memory Manager just handed to the Disk Manager is not very expensive. The External Memory Manager does not retain pages because it wishes to avoid the overhead of managing a cache of pages.

data_provided containing the page, and the External Memory Manager can then do the same memory_object_data_provided to give the page to the second kernel.

### 5.2.4.3. Segment activation

When a RVM segment is first activated, the Disk Manager allocates a Mach port pagingPort, and creates a thread to await external pager requests on that port.

For Camelot/TDSM, the Disk Manager asks the External Memory Manager to process the external pager requests, and to call the Disk Manager's external pager routines as needed. As each server using the RVM segment is started, the Disk Manager adds the server to its list of servers using the segment, and delivers the pagingPort to the server via DS_Initialize. The server vm_maps its recoverable segment, specifying via the pagingPort that the External Memory Manager is acting as an external pager.

### 5.2.4.4. Paging

The External Memory Manager presents the image of a single client kernel to the Disk Manager's external pager routines; thus, the original Camelot external pager algorithm needed few alterations for use with Camelot/TDSM.

One set of alterations result from moving the Disk Manager's data for a RVM segment from the server record data structure to a separate segment record data structure. Because of this change, the Disk Manager's external pager routines must latch the segment record, rather than the server record, when it looks at the data. If the external pager detects an error in a request, it no longer can kill a single offending server, but it must kill all of the servers using the offending segment. And on receiving certain requests, such as memory_object_data_write and memory_object_data_unlock, where the external pager must check the shared memory queue in order to enforce the write-ahead log protocol, the external pager must look at the shared memory queues of all servers using the given RVM segment.

Some actions of the External Memory Manager differ from the Mach kernel, and these differences result in additional alterations to the Camelot external pager. Occasionally, because multiple threads proceed in the Disk Manager at different rates, the Disk Manager could ask for a page to be flushed from main memory to the external pager after the corresponding RVM segment has become inactive (i.e., the pagingPort has been destroyed). The Mach kernel simply ignores the bogus flush request; however, the External Memory Manager complains. To avoid this error, some additional checks were added to the Disk Manager to prevent it from making the bogus flush request.

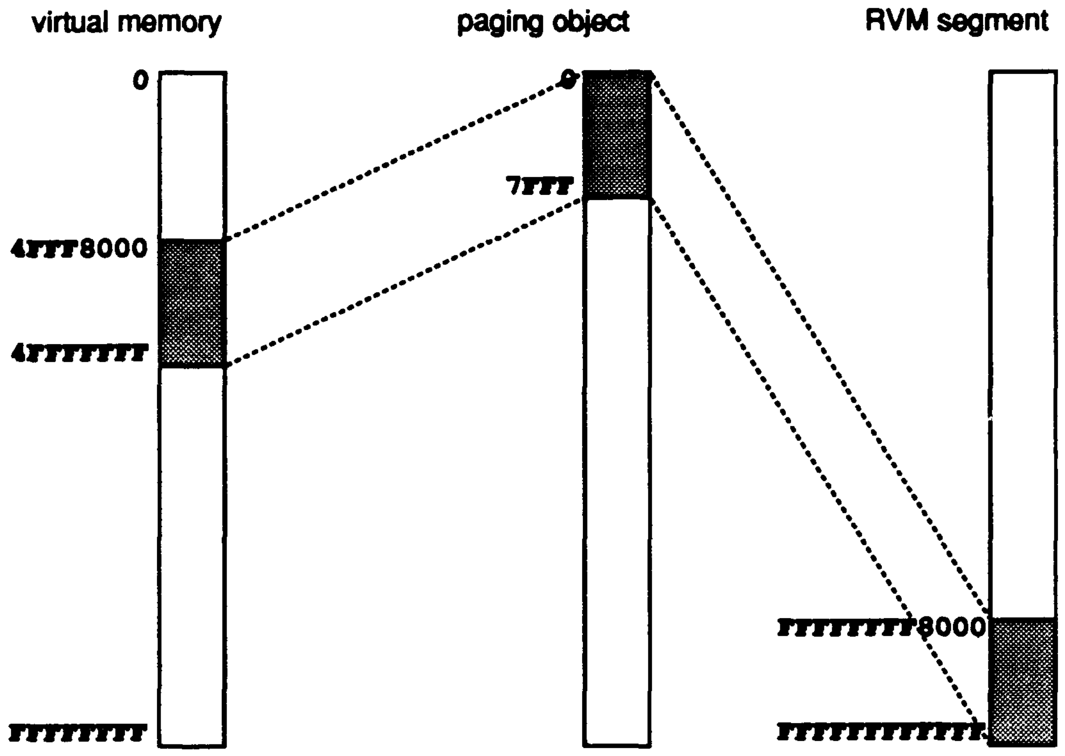
Another difference between the External Memory Manager and the Mach kernel as a client of the external pager is the handling of offsets into the RVM segment. Each RVM segment has a size defined via the Node Server; each page in a segment is identified by its offset within the segment. (The offset does not have to match the virtual memory address of the page. The Camelot library tries to map the RVM segment at a fixed address for each machine architecture; thus, RVM objects may contain pointers to other RVM objects.) Suppose `max` is the maximum possible segment size, and a particular segment is of size `size`. Then, internally, Camelot uses offsets ranging from `max - size` to `max - 1`; i.e., segment offsets are decreasing values, starting from the maximum segment size. When the Camelot library within a server uses `vm_map` to identify a region of the server's virtual memory as a Mach paging object backed by Camelot, it gives the Mach kernel `max - size` as the initial offset within the paging object. The kernel then uses the same offsets within the paging object as Camelot uses with the RVM segment in the messages it sends to and receives from the Camelot external pager. Unfortunately, the External Memory Manager was not designed to deal with large offsets, and requires paging object offsets to range from 0 to `size - 1`. So the Camelot library and the Camelot external pager were changed to use small paging object offsets for `vm_map` and the External Memory Manager, mapping these small paging object offsets to large RVM segment offsets for use internally. (See Figure 5-3.)

The External Memory Manager uses Mach messages to send pages from node to node. Mach, in turn, directs each message to a remote node to the netmsgserver, which transmits the message over a TCP connection to its counterpart on the destination node. The destination netmsgserver extracts the message from the TCP connection, and re-sends it to the correct port on its local node. The netmsgserver in Mach 2.5 exhibited some problems in handling a large number of page-size messages, and, until the bug was fixed, occasionally stopped forwarding messages over the TCP connection.

### 5.2.4.5. Hot Pages

If the kernel has not issued a `memory_object_data_write` of a page for a long time, the Disk Manager determines that the page is *hot*, and asks the kernel to flush the page. However, the Disk Manager cannot write the flushed page to disk if the page is pinned. When the kernel flushes the page, the Disk Manager holds on to the copy, and maintains a separate pin count for the copy. Subsequent pin and log requests are not processed, but queued up in *patch records* by the Disk Manager. When a transaction commits, the Disk Manager goes through the queue of patch records and applies (to the Disk Manager's copy of the page) those patch records that belong to the committed transaction. Thus, the copy's pin count will never be incremented without a corresponding decrement, since pages are always unpinned by the end of transaction. When the pin count reaches zero, the Disk Manager writes its copy out to disk.

Because the External Memory Manager presents the image of a single client kernel to the Disk Manager, the hot page algorithm developed for the original Camelot also works under Camelot/TDSM, with one complication. Suppose the kernel flushes the page and the Disk Manager holds on to the copy as described above. Now, if the page becomes LRU, the kernel may do `memory_object_data_write` again. Unfortunately, the Disk Manager cannot



A RVM segment of size `0x7fff` bytes may be mapped into a server's address space at an arbitrary VM address. The kernel translates the VM address of `0x4fff8000` into an offset within the paging object of `0`. Camelot allocates a RVM segment starting at the maximum offset of `0xffffffffffff`, so the Camelot external pager maps the paging object offset of `0` into a RVM segment offset of `0xffffffff8000`.

**Figure 5-3:** Mapping of offsets

write the page to disk: since the Disk Manager is treating the page as hot, it must be pinned. So the Disk Manager keeps this alternate copy of the page, too. Presumably, since the page is hot, the kernel will soon ask for the page again, and the Disk Manager will simply return the alternate copy to the kernel. The complication is that the original Disk Manager assumes that the kernel discards the page after the second `memory_object_data_write` and panics if the kernel does another `memory_object_data_write` while the Disk Manager was holding the alternate copy. With Camelot/TDSM, the Disk Manager may receive another `memory_object_data_write` while holding the alternate copy if the page migrates from one kernel to another. In this case, the Disk Manager simply replaces its alternate copy with a new alternate copy instead of panicking.

#### 5.2.4.6. Preflush

A server may ask the Disk Manager to preflush a page. Unfortunately with Camelot/TDSM, the Disk Manager's interface to the External Memory Manager does not allow a page to be flushed on a single node; instead, the page will be flushed on all nodes. Fortunately, the preflush request is seldom used. Immediately following segment recovery, a separate request is used to flush all pages. However, for segment recovery, only one server is running, so the flush request acts as expected.

#### 5.2.4.7. Down servers

The original Camelot system tracks the "dirty" or "clean" state of servers that are not presently running. (A server is "dirty" if it goes down with page records in the grid or prepared transactions.) When the Recovery Manager asks the Disk Manager to perform a checkpoint, it passes two lists of servers based on the log records it encountered: a list of recovered servers, and a list of non-recovered servers. When the Node Server asks the Disk Manager for a server's current state with `DN_GetServerState`, it passes the server id.

With Camelot/TDSM, the "dirty" or "clean" state applies to a segment, not to a server. A segment is "dirty" if no running server is using the segment, and when the last server using the segment went down, there were page records in the grid or prepared transactions for the segment. The Recovery Manager passes lists of segments, not servers. And the Node Server includes both server id and segment id in `DN_GetServerState` requests.

One unintended consequence of this change was a deadlock condition between the Node Server and the Disk Manager. In the original Camelot system, the Node Server passes only the server id in `DN_GetServerState`; after receiving the reply, it obtains a lock on its internal `server_entry` data structure. For Camelot/TDSM, the Node Server must look in its `server_entry` data structure to find the segment id; thus, it locks the `server_entry` before calling `DN_GetServerState`. In both systems, the Disk Manager obtains a latch on the appropriate `s_record_t` to process `DN_GetServerState`. The problem is that the latches and locks may be obtained in the opposite order as a server exits: the Disk Manager latches the appropriate `s_record_t`, and then makes a `ND_GetRestartAdvice` RPC to the Node Server to see what to do next. To process `ND_GetRestartAdvice`, the Node Server locks the corresponding `server_entry`. If the Node Server is making a `DN_GetServerState` at the same time the server exits, a deadlock results. To avoid the deadlock, the Node Server was changed to acquire and then drop its `server_entry` lock before calling `DN_GetServerState`.

If a segment is "dirty" when the last server using the segment goes down, Camelot attempts to clean the segment by restarting a server using the segment, and running the segment recovery algorithm. The server is passed a flag that tells it to exit as soon as recovery is complete. In the original Camelot system, the Disk Manager starts the actual server. For Camelot/TDSM, the actual server may be configured to run on a remote node which may be down. Instead of running the actual server, Camelot/TDSM starts a special *surrogate* server on the home node which exits as soon as recovery complete.

### 5.2.4.8. Server startup

The Disk Manager starts servers as requested by the Node Server. In the original Camelot system, the Node Server passes a list of server ids in the `DN_StartDataServers` RPC. The Disk Manager sorts the list by server id, and activates each server in turn. To activate a server, the Disk Manager obtains the server's `recoveryLock`, starts a paging thread for the server's segment, and calls `ND_GetRestartAdvice` to obtain the server's command line. The Disk Manager `forks` and executes the appropriate binary after setting up the shared memory queue and storing a Mach port `dsPort` in the environment.

For Camelot/TDSM, the Node Server passes a list of server ids and corresponding segment ids in the `DN_StartDataServers` RPC. The Disk Manager sorts the (server id, segment id) pairs by segment id, and activates each segment in order, obtaining the segment's `recoveryLock` as the segment is activated. Next, after all segments have been activated, the Disk Manager starts each server in turn. To start a server, the Disk Manager makes a `ND_GetRestartAdvice` RPC back to the Node Server to obtain the command line and Internet address. If the Internet address is non-zero, the Disk Manager uses `netname_look_up` to obtain a Mach port `xPort` for the Remote Execution Manager, and makes a `XD_StartServer` RPC to the Remote Execution Manager. Then, either the Remote Execution Manager (for remote servers) or the Disk Manager (for local servers) will `fork` and execute the appropriate server binary after (1) setting up a section of virtual memory as the shared memory queue and (2) storing a Mach port `dsPort` in the environment so that the server can contact the Disk Manager.

Each server makes a `DS_Initialize` RPC back to the Disk Manager to obtain descriptions of its RVM segment, the shared memory queue, and various ports, including a port for the Disk Manager's pager (original Camelot) or the External Memory Manager (Camelot/TDSM). After performing some initialization, the server enters a loop, awaiting a message from the Recovery Manager.

In the original Camelot system, after all servers have been started, the Disk Manager asks the Recovery Manager to recover all servers, passing a list of server ids via `RD_RecoverServers`. After the Recovery Manager recovers each server, it sends a message to the server that causes it to begin normal operation.

With Camelot/TDSM, after all servers have been started, the Disk Manager asks the Recovery Manager to recover all segments, passing a list of (server id, segment id) pairs for the first server in each segment via `RD_RecoverSegments`. As the Recovery Manager recovers each segment, it eventually sends a message to the first server in each segment that causes the server to begin normal operation. After all segments have been started, the Disk Manager asks the Recovery Manager to recover all remaining servers, passing a list of (server id, segment id) pairs for all remaining servers in each segment via `RD_RecoverServers`. The Recovery Manager sends a message to each server that causes the server to begin normal operation.

Finally, after all servers have begun normal operation, the Disk Manager in both original Camelot and Camelot/TDSM releases the `recoveryLocks`.

### 5.2.4.9. Server termination

When the Disk Manager detects an abnormal condition or receives the appropriate request from another Camelot component, it will terminate a server.

For Camelot/TDSM, the Disk Manager looks at the `xPort` in the server record to determine if the server is running locally or on a remote node. If the server is running locally, the Disk Manager uses the UNIX `kill` system call to kill the server. If the server is running on a remote node, the Disk Manager makes a `XD_KillServer` to the Remote Execution Manager corresponding to the `xPort` to kill the server. When a server on a remote host exits (because the Disk Manager killed the server, or for any other reason), the Remote Execution Manager notifies the Disk Manager via a `DX_ServerDied` RPC.

### 5.2.4.10. System shutdown

A user of the Node Configuration Application (NCA) can make a request to the Node Server to shut down the system. In the original Camelot system, the Disk Manager acts on a `DN_Shutdown` request from the Node Server by making a `MD_ShutdownCamelot` RPC to the Camelot Master Control Program (MCP). The MCP then kills all the Camelot components it knows about.

There are two problems with the original algorithm. First, the MCP may act so quickly that the Node Server may not have time to reply to NCA, and the user will be left hanging. Second, servers started on remote nodes are not children of the Disk Manager, so when the MCP kills the Disk Manager, remote servers will be left hanging. To solve these problems, two changes were made. First, the Disk Manager replies immediately to the `ND_Shutdown` request, and waits for five seconds before calling `MD_ShutdownCamelot`. Second, after calling `MD_ShutdownCamelot`, the Disk Manager starts killing each server via the UNIX `kill` system call or a `XD_KillServer` RPC to the Remote Execution Manager as appropriate.

## 5.3. Recovery Management

The Recovery Manager has two major tasks with respect to returning a RVM segment to a transaction-consistent state. When a transaction aborts, the Recovery Manager scans the log to undo the effects of the transaction. When a server crashes, the the Recovery Manager scans the log to redo the effects of committed transactions.

In the original Camelot system, a RVM segment may be used by only one server, and the Recovery Manager treats recovery of a segment as though it were recovery of a server. For

Camelot/TDSM, many servers can use a given RVM segment, and the Recovery Manager distinguishes between server recovery and segment recovery. Many of the internal and external Recovery Manager interfaces and log record formats which contain a server id in the original design are changed to use a segment id for Camelot/TDSM.

When a server is started, the Recovery Manager receives a Mach port that the Recovery Manager may use to send messages to the Camelot library in the server. The Recovery Manager stores this port with its associated server id in a `s_port_rec`. For Camelot/TDSM, all of the `s_port_recs` for a particular RVM segment are linked together, and a `segment_rec` contains the segment id and a pointer to the list. When the Recovery Manager wishes to perform some action on a RVM segment, it uses a hash table to find the appropriate `segment_rec`. If the Recovery Manager wishes to send a message to a particular server, it searches the list of `s_port_recs` to extract the correct port; otherwise, it uses the port in the first `s_port_rec`. If the message fails (because the server is down, for example), the Recovery Manager can pick another `s_port_rec` and repeat the message.

Transaction abort takes two forms, depending on whether the transaction is initiated using old-value/new-value logging, or new-value-only logging. To limit recovery times, the Disk Manager may initiate a special form of new-value abort, called backstopping, which does all the work needed to determine the proper old value, but does not actually abort the transaction.

### 5.3.1. Old-value/new-value abort

In the original Camelot system, the Transaction Manager calls `RT_Abort` to tell the Recovery Manager to abort a transaction. Once the Recovery Manager identifies the transaction as old-value/new-value, it asks the Disk Manager via `DR_GetTranLSNs` for a list of all log records created by the transaction. The Recovery Manager looks in each log record to identify the server which created the record, and buffers up a request to undo the modification. The requests are sorted by server id. When all requests have been buffered, or the buffer is full, the Recovery Manager calls the server via `SR_RestoreBatch` to undo the modifications. Next, the Recovery Manager notifies the Disk Manager with a `DR_UndoOvnv` call so that the Disk Manager may restore its internal data structures.

For Camelot/TDSM, the algorithm is similar, except that the Recovery Manager extracts segment ids (not server ids) from log records, sorts requests by segment id, and selects an arbitrary server in each segment to undo the modifications via `SR_RestoreBatch`. Because this algorithm may result in one server undoing a modification made by another, the Disk Manager must look at the shared memory queues of all servers using the RVM segment before processing the `DR_UndoOvnv`.

### 5.3.2. New-value-only abort

In the original Camelot system, new-value-only abort is initiated in the same manner as old-value/new-value abort. Once the Recovery Manager identifies the transaction as new-value-only, it asks the Disk Manager via `DR_GetTranLSNs` for a list of all log records created by the transaction. The Recovery Manager scans these log records to build a list of pages that were modified by the transaction. Next, the Recovery Manager asks the Disk Manager via `DR_GetObjectLSNs` for a list of all log records that have modified any of these pages since each page was last written to disk. From this new list of log records, the Recovery Manager determines the values to restore on each page, and buffers up requests to restore the values. The requests are sorted by server id. When all requests have been buffered, or the buffer is full, the Recovery Manager calls the server via `SR_RestoreBatch` to restore the values. Again, the Disk Manager is notified via `DR_Restore`.

For Camelot/TDSM, the algorithm is the same, with the substitution of segment id for server id. Again, the Recovery Manager selects an arbitrary server in each segment to restore the values via `SR_RestoreBatch`.

In both original Camelot and Camelot/TDSM, the backstopping algorithm proceeds as new-value-only abort, except that the Recovery Manager sends the values to the Disk Manager via `DR_Backstop` instead of calling the server.

### 5.3.3. Segment and server recovery

The segment recovery algorithm of Camelot/TDSM evolved from the original Camelot server recovery algorithm, primarily substituting segment ids for server ids. The Disk Manager sends a list of segments needing recovery via the `DR_RecoverSegments` RPC to the Recovery Manager. If the Recovery Manager encounters a fatal error while trying to recover a given segment, it makes a `DR_KillSegment` RPC to the Disk Manager, which will kill all of the servers using the segment.

Segment recovery is performed when there are no servers using a segment. The Disk Manager starts one server for each segment it wishes to recover, and sends a request to the Recovery Manager.[6] To recover a segment, the Recovery Manager reads the log in reverse order and identifies changes made by committed transactions that are not reflected in the disk copy of the page. The Recovery Manager buffers up requests to redo these changes. The Recovery Manager also identifies changes made by aborted transactions that were already written to the

[6]Only one server may be using a given segment during segment recovery. The reason for this is twofold. First, the segment will not reach a transaction-consistent state until recovery completes; thus, even if another server were running, it would have to wait before it could perform useful work. Second, some of the algorithms in the Camelot library (specifically, the algorithm for recovering the recoverable heap) assume that only one thread is running during recovery.

disk copy of the page. The Recovery Manager buffers up requests to undo these changes. When the buffer is full, or all requests have been buffered, the Recovery Manager sends the buffer to the first server using the segment via `SR_RestoreBatch`. After all recovery actions has been performed, the Recovery Manager sends a `SR_RecoveryComplete` message to the first server in each segment it has recovered.

The segment recovery algorithm of Camelot/TDSM has a few differences from the original Camelot server recovery algorithm. The most obvious difference is that Camelot/TDSM keeps lists of segments for recovery, not lists of servers. Server ids no longer appear in **prepare** or **checkpoint** log records, and a segment id was added to the **undo_modify** log record. When processing a **backstop** record, which includes a list of modifications, the segment recovery algorithm must check every item in the list to see if it refers to a RVM segment that was deleted; the original Camelot server recovery algorithm needs only to check the header of the **backstop** record to see if the server had been deleted.

Server recovery in Camelot/TDSM is performed when a segment is already active; that is, there are already servers using the segment. When the Disk Manager starts a server in an active segment, it sends a `RD_RecoverServers` request to the Recovery Manager. The server recovery algorithm is not fully implemented in Camelot TDSM; the Recovery Manager merely sends a `SR_RecoveryComplete` message to the server. This algorithm is correct in that RVM remains in a transaction-consistent state, but there is a problem. If a node crashes, any pages it had exclusive access to become unavailable until the next segment recovery. The server recovery algorithm should utilize information in the log and paging store to reconstruct transaction-consistent pages.

## 5.4. Configuration Management

The Node Server is a distinguished Camelot server that maintains two databases in its recoverable virtual memory segment. One database contains configuration information for Camelot servers. The other database controls the allocation of paging storage to RVM segments. (The Node Server's RVM segment is a fixed size, set at the time the Disk Manager is compiled.) The Node Configuration Application (NCA) is an application, included with the Camelot system, that provides a user interface to the RPC interface of the Node Server. The original Camelot Node Server and NCA were modified to support remote execution, multiple servers per segment, and segment deletion for Camelot/TDSM.

To support execution of a server on a remote node for Camelot/TDSM, the Node Server RPCs `NA_AddServer` and `NA_ShowServer`, and corresponding NCA commands `addserver` and `showserver` have a host name parameter to specify the node on which a server is to execute. A new Node Server RPC `NA_SetSite`, with corresponding NCA command `setsite`, allows a user to change the host name of an existing server. The Node

Server stores the host name in a server_entry in the Node Server's RVM segment. (Because this increases the size of the server_entry, the Disk Manager is compiled with a larger fixed size for the Node Server's RVM segment.) When the Disk Manager makes a ND_GetRestartAdvice RPC to the Node Server, the Node Server translates the host name and returns an Internet address to the Disk Manager.

When configuring a server, a user of NCA must select a small integer as the server id, and a small integer as the segment id. In the original Camelot system, the segment id can be any unused segment id. For Camelot/TDSM, a user of NCA who is configuring a server may select a segment id that is already assigned to another server, as long as that server is owned by the same user. When two servers that share a segment id run concurrently, they share a RVM segment.

In the original Camelot system, the Node Server allows a single server to have an arbitrary number of segments, although this is not supported by the NCA or the Camelot library. To locate the segments assigned to a given server, the original Camelot Node Server scans the entire list of segment descriptors in its RVM segment, and extracts those segment descriptors containing the desired server id. For Camelot/TDSM, the server id is no longer in the segment descriptor, because a given segment may be assigned to an arbitrary number of servers. Thus, the Camelot/TDSM Node Server uses a different algorithm and data structure to locate the segments assigned to a given server (although multiple segments per server are still not supported by the rest of Camelot/TDSM). A single server may have up to NODE_MAX_SEGS_PER_SERVER (currently 4) segments assigned. The segment ids for a given server are stored directly in the server's server_entry, so the Node Server does not need to scan the list of segment descriptors to locate the descriptors referring to the given server.

The Node Server does not supply an interface for deleting a segment. Instead, a user may delete individual servers. When no more servers reference a given segment, the segment is deleted. In the original Camelot system, when a server is deleted, the Node Server asks the Disk Manager to write a log_server_delete record to the log. When the Recovery Manager encounters this record during server recovery, it ignores any previous log records referring to the server. In Camelot/TDSM, the Recovery Manager is responsible for recovering segments, not servers. Thus, when a server is deleted, no record is written to the log. Instead, when the last server referencing a segment is deleted, the Node Server asks the Disk Manager to write a log_segment_delete record to the log, so the Recovery Manager will ignore any previous records referring to the segment.

## 5.5. Communication Management

The Camelot Communication Manager provides a name registration and lookup service for Camelot servers, and tracks the spread of transactions from node to node as Camelot applications and servers make transactional RPCs to other nodes. With TDSM, Camelot servers access data via a shared RVM segment, and are not expected to make transactional RPCs. Thus, the primary function of the Communication Manager under Camelot/TDSM is the name service.

When a server is started by the Disk Manager (either as a child of the Disk Manager or, with Camelot/TDSM, as a child of the Remote Execution Manager), it receives in its environment a port for contacting the Disk Manager. It uses this port to make a `DS_Initialize` RPC to the Disk Manager. In the reply to this request, the Disk Manager includes ports that the server may use to call other Camelot components, such as the Transaction Manager and the Master Control Program. In the original Camelot system, the `DS_Initialize` reply does **not** include a port for the Communication Manager. Instead, the server must use `netname_look_up` to ask the Mach netmsgserver to return a port for the local Communication Manager.

For Camelot/TDSM, a server on a remote node cannot use `netname_look_up` to obtain a port for the Communication Manager because there might not be a Communication Manager on the remote node. Even if a Communication Manager is running on the remote node, it does not know that the server is using a RVM segment on a different node. Thus, for Camelot/TDSM, the `DS_Initialize` reply includes a port for the Communication Manager on the home node. As a result, a server running on a remote node is registered with the Camelot name service on the home node, and appears to the rest of the world as if it were running on the home node. This is not inappropriate, since the paging storage, log, and transaction services for the server all reside on the home node.

Unlike servers, Camelot applications are not started by the Disk Manager, so an application must use `netname_look_up` to locate the Communication Manager. As implied above, Camelot/TDSM allows a server to run on a using node that does not have a Communication Manager. To reduce RPC costs, a user may wish to run a Camelot application on the same node as this server, but without a Communication Manager, the application cannot run. To remedy this problem, the Remote Execution Manager can act as a surrogate for the Communication Manager on another node. A command-line option tells the Remote Execution Manager the name of the node to which it is to forward all Communication Manager requests. To accommodate forwarded requests, the Communication Manager on the home node checks each incoming request to see if it includes a port that the Communication Manager sent out in a previous message; if so, the Communication Manager breaks the forwarding loop, and handles the request itself.

## 5.6. Log Management

The Camelot Log Manager is a library that is linked with the Recovery Manager and the Disk Manager. For Camelot/TDSM, the original Camelot Log Manager is used. Because of the tight coupling between buffer management and log management, and between recovery and log management, the functions of the Log Manager are discussed above in Sections 5.2 and 5.3.

The primary difference between logging in the original Camelot system and in Camelot/TDSM is the fact that log records for a given segment may be generated by several servers, and thus buffered in several shared memory queues. In the original Camelot system, the Disk Manager need inspect the shared memory queue of only a single server for most operations. For Camelot/TDSM, the Disk Manager must inspect the shared memory queue of **all** servers using a given segment when it receives any external pager requests for the segment, or when it receives `DR_UndoOvnv`, `DR_Restore`, or `DR_Checkpoint` RPCs for the segment from the Recovery Manager.

## 5.7. Transaction Management

The Camelot Transaction Manager is a protocol engine that handles distributed agreement. The original Camelot Transaction Manager needs no modifications for Camelot/TDSM. To the Transaction Manager, remote servers appear to execute on the home node, since the log, paging store, and other Camelot services are on the home node.

The only change made to the Transaction Manager for Camelot/TDSM was to fix a previously existing bug, a typo where a conditional test was inverted. The condition appears more frequently with Camelot/TDSM because the RPCs that the Transaction Manager makes to a server take more time when the server is not on the home node.

## 5.8. Summary

This chapter has described the changes made to the original Camelot system to support TDSM. The algorithms and data structures used in two new components, the Lock Manager and the Remote Execution Manager, as well as algorithms and data structures used in existing components were presented.

Table 5-1 illustrates the number of lines of code in the Camelot system. The Camelot library had the most changes, primarily to support the distributed Lock Manager, and the Disk Manager also had many changes (as described in Section 5.2). Omitting the External Memory Manager, which was provided by Joe Barrera, I added 7,626 lines of code to Camelot, and changed 4,144 lines.

| Component | Original | Added | Changed |
|---|---|---|---|
| Disk manager | 14,382 | 1,671 | 1,271 |
| Recovery manager | 14,796 | 126 | 736 |
| Transaction manager | 29,573 | 5 | 21 |
| Communication manager | 3,324 | 14 | 29 |
| Log manager | 48,942 | 98 | 142 |
| Node server | 3,969 | 260 | 254 |
| Node configuration application | 1,835 | 49 | 20 |
| Library | 36,681 | 1,809 | 1,671 |
| Control programs | 6,532 | 10 | 3 |
| Lock manager | | 2,490 | |
| Remote execution manager | | 1,104 | |
| External memory manager | | 9,535 | |
| Total | 153,502 | 17,161 | 4,144 |

This table shows the number of lines of code (including comments) in the original Camelot system, the number of lines added to support TDSM, and the number of lines of the original system that were changed to support TDSM. "Control programs" includes the initialization program Camelot, and the Master Control Program. "Library" includes the Camelot low-level library `-lcam` and the high-level library `-lcamlib`.

**Table 5-1:** Lines of code

# Chapter 6
# Performance

This chapter measures the performance of the transactional distributed shared memory (TDSM) implementation described in Chapter 5. Since the implementation is a prototype to prove the feasibility of TDSM, I do not expect to show that this implementation of TDSM performs better than any alternatives. Rather, the purpose of the measurements is to understand the system and the relative costs of operations in the system. An analysis of the measurements shows the implications of choices made in the design of the system, aids in directing the design of future systems, and allows the performance of these future systems to be predicted. The predicted good performance of these future systems supports the thesis that TDSM is useful.

Although the numbers reported in this chapter are measured accurately, the analysis of individual operations is somewhat imprecise due to concurrency between system components. Nevertheless, the analysis is qualitatively correct, and I believe that the information gleaned from the analysis is accurate. More precision could be obtained through stochastic modelling.

The performance measurements are done in three sets. In the first set, a set of carefully chosen transactions is measured under various conditions. These measurements are analyzed to obtain the cost of the relevant operations. In the second set, the ET1 benchmark is used to measure the throughput of the system. The measured throughput is then compared to the throughput that can be predicted based on the analysis of the individual operations. The final set of performance measurements compares an application using function shipping to an application using data sharing.

## 6.1. Experimental Environment

All experiments were performed on one to three IBM RT/PC APC workstations, running Camelot 1.0 (release 83) on top of Mach 2.5 (versions CS7k through CS7r)[7]. Each workstation has 12 megabytes of main memory and is rated at approximately 2.5 VAX MIPS. The workstations are connected by a 4 megabit/second token ring that also connects several other unrelated workstations. All workstations were running in multi-user mode with the usual set of

[7]The kernel versions varied only in bug fixes to authentication and distributed file system mechanisms that were not involved in the experiments.

system daemons during the experiments. In addition to the processes required to perform the experiments, some display processes were running to passively monitor the tests. However, during the tests, no one was actively using the workstations running the experiments.

The choice of the IBM RT/PC as the basis for the experimental environment may appear unreasonable. By today's standards, the IBM RT/PC is slow and obsolete. However, at the inception of this work, the standard set of workstations available at CMU included the Sun 3, the Microvax II, and the IBM RT/PC. Camelot and Mach were developed on these three architectures. Shortly after I joined the Camelot project, the principal investigator, all of the research programmers, and most of the senior graduate students left the project for greener pastures. As a result, Camelot was never ported to new architectures as they became available, even though another research project attempted to do so, but failed (in part due to bugs in the Mach kernel). Nevertheless, Camelot and Mach on the IBM RT/PC proved to be a stable platform for developing a TDSM system, and the the IBM RT/PC is a reasonable choice for the purpose of proving feasibility. For the purpose of proving utility, the measurements in this chapter allow the performance of the system to be characterized in terms of "primitive operations." In Chapter 7, the cost of these primitive operations is extrapolated to modern hardware to predict the performance of TDSM.

## 6.2. Primitives

To provide a basis for understanding the performance of various operations, the characteristics of the underlying platform are measured:

- Interprocess communication (IPC): including CPU time and network transfer time for remote messages, and CPU time only for local messages.
- Disk I/O: rotational latency, seek time, and transfer rate.
- Page fault handling: operating system overhead of the external pager interface.

These primitive operations are independent of Camelot and TDSM, and are a function of the workstation, network, and kernel. The cost of these primitives is measured by executing the primitive 1000 times, recording the elapsed time, and dividing by the number of repetitions. (The primitive is repeated 1000 times because the clock resolution is larger than the duration of a single primitive operation.)

Using this technique, the latency for sequential 4096-byte reads or writes (on a UNIX raw partition) is determined to be 21.1 milliseconds; sequential 512-byte reads or writes take 17.2 milliseconds. If we assume that the time for a read or write is linear in the size of the block, then the rotational latency of the disk is 16.6 milliseconds, and data is transferred at a rate of 1.09 microseconds per byte. The average rotational latency for random accesses should be half that of sequential accesses, or 8.3 milliseconds. A seek followed by a read or write of 4096 bytes takes 35 milliseconds on average. Subtracting the data transfer time (4.5 milliseconds) and the average rotational latency (8.3 milliseconds) produces an average seek time of 22 milliseconds.

| Message | Local times (ms.) | Remote times (ms.) |
|---|---|---|
| XD_GetShMemQueue (RPC; 80 bytes) | 4.09 | 28.4 |
| XD_GetShMemQueue (RPC; 0 bytes) | 2.50 | 17.2 |
| HS_Lock (RPC) | 2.66 | 17.7 |
| ST_Vote (RPC) | 2.35 | 16.2 |
| ST_Commit | .89 | 13.4 |
| TS_Join (RPC) | 2.29 | 16.1 |
| TA_GetTids (RPC; 1 tid) | 2.82 | 22.3 |
| TA_End (RPC) | 2.43 | 16.1 |
| memory_object_lock_request+ memory_object_lock_completed | 3.96 | 22.6 |
| memory_object_data_request+ memory_object_data_provided (4096 bytes) | 6.07 | 186.2 |
| memory_object_data_write (4096 bytes) | 3.70 | 185.6 |
| cpa_read_s (RPC; 32 bytes) | 2.30 | 17.1 |
| cpa_read_l (RPC; 1024 bytes) | 2.88 | 24.4 |

The "Local" column reports elapsed times when both processes are on the same workstation. The "Remote" column reports elapsed times when sender and receiver are on different workstations. The notation "RPC" indicates that the time includes both the request and reply messages.

**Table 6-1:** Representative IPC times

To measure raw IPC times, two processes are used. One process uses the MIG-generated server code to receive a message, call an empty service routine, and send a message in reply if one is required. The other process uses MIG-generated stubs to generate and send a message, and await a reply if the request is a remote procedure call (RPC). This latter process starts a timer, makes a number of requests in sequence (200 requests if the processes are on different workstations, 1000 requests if they are on the same workstation), and records the time elapsed on the timer. This time is then divided by the number of requests to determine the cost of the primitive; some IPC times that are used later in the chapter are given in Table 6-1.

The cost of a message round depends on the size, type, and number of parameters in both the request and reply. Thus, for example, Table 6-1 includes two lines for XD_GetShMemQueue for two different parameter sizes, and the times for HS_Lock and ST_Vote differ because of the different parameters each RPC uses. When two processes are on the same workstation, and the message includes a pointer to a large region of memory, Mach uses the *copy-on-write* optimization to avoid copying the large region. Using the virtual memory hardware, Mach maps the region read-only into both the sender's and receiver's address space. The region is copied only if (and when) one of the processes attempts to modify the region. (Thus, the `memory_object_data_write` call is very fast locally since the data need not be copied.) Of course, when the two processes are on different workstations, Mach cannot avoid copying the data. (Unfortunately, Mach 2.5 copies the data several times in this case. This accounts for the huge values for `memory_object_data_provided` and `memory_object_data_request`.)

In the Mach external pager interface, the primitives of interest are page in and page out. An external pager is not required to access the disk while satisfying requests, so the time measured should include page fault and IPC times, but exclude any disk activity. To measure these times, two processes are used. The external pager process responds to each `memory_object_data_request` message with the same memory-resident page. In response to all other messages, it does nothing. The client process cycles through all pages in its address space, reading or writing one byte on each page. For write tests, the client is run through enough cycles to ensure that main memory is filled with dirty pages, so that each time it accesses a new page, the kernel must send an old page out to the external pager. On a workstation with 12 megabytes of main memory, the experiment is guaranteed to reach steady state after accessing 3000 pages (4096 bytes per page). After reaching steady state, the client process records the elapsed time to read or write 4000 pages. Using this technique, the time for the client to read fault, and the kernel to request and receive a page from the external pager is determined to be 11.2 milliseconds. The time for the client to write fault, and the kernel to page out an old page, to request and receive a new page from the external pager is determined to be 14.1 milliseconds. These activities use the `memory_object` messages given in Table 6-1. Thus, processing a page fault takes about 5 milliseconds plus the IPC time.

## 6.3. Operation Costs

Rather than characterizing the performance of the system by reporting the latency of a "typical" transaction, I use the *primitive analysis* methodology of Spector and Daniels [Spector and Daniels 85]. The goals of this methodology are to predict the performance of a transaction by identifying the costs of its component operations, to model the cost of these operations in terms of the underlying primitives, and to predict the system's performance when primitive costs are altered by changes to the system's configuration and algorithms. The performance of a transaction can be predicted by summing the costs of its component operations. To construct the model of operation costs, the following steps are followed:

1. Measure the cost of the underlying primitives (as reported in Section 6.2.
2. Measure the cost of the component operations.
3. Based on an understanding of the system and its algorithms, analyze each

component operation to determine the number of each type of primitive used by the operation. The remaining time not accounted for by the primitives is CPU processing time. Express the operation time as a linear sum of the primitive times and CPU time.

When an operation cost cannot be measured directly, the methodology instead measures the *incremental latency* added to a transaction when a single operation of the specific type is added to the work performed by the transaction. To determine the incremental latency of a operation, the latency of two test transactions is measured: one transaction executes the operation *m* times, another transaction executes the operation *n* times. The incremental latency is then the difference of the transaction latencies divided by the difference of *m* and *n*. Subtracting the latency of the *m* or *n* operations from the total latency of the transaction then gives the *transaction overhead*, latency that can be attributed to initiating and committing a transaction of a particular type. (Transaction overhead varies with the type of transaction and may not be directly measurable. For example, a read-only transaction can be committed with no log force, while an update transaction requires a log force. It is not possible to directly measure the overhead of an update transaction since a transaction that performs zero writes is considered read-only.) This method of determining incremental latency assumes that the latency of a test transaction is a linear function of the number of operations. Operations must be chosen carefully to ensure the validity of this assumption.

The latency of an operation may include several components that can occur concurrently. For example, a network page fault may include processor time to id^ntify the fault and construct a request message, network transfer time to send the message to another node, time on another processor to interpret the received message and determine the page's location on disk, waiting time to allow a previous disk operation to complete, and seek and transfer time to read the page from disk. The choice of operations should separate these components whenever possible.

This section reports operation costs in many different configurations so as to be able to describe the latency of many types of transactions. However, only two of these configurations, configurations 3 and 5 (Figures 6-3 and 6-5) are significant in the remainder of the dissertation. The key operations to keep in mind are 32-byte non-paging reads, 32-byte non-paging writes, paging reads, paging writes, read-only transaction overhead, and update transaction overhead.

## 6.3.1. Experimental parameters

Many parameters affect the performance of the system and must be considered when choosing operations. For example, accessing a region of virtual memory is much cheaper when the region is already present in main memory than when a page fault must be taken. In the original evaluation of Camelot's virtual memory system, Eppinger varied his experimental parameters along four dimensions [Eppinger 89]:

- Region size. Accessing a larger region takes longer than accessing a smaller region.

However, the system may impose a smaller cost on accesses to subsequent bytes after accessing the first byte of a region. Measuring accesses to different region sizes allows the incremental cost of each additional byte to be determined.

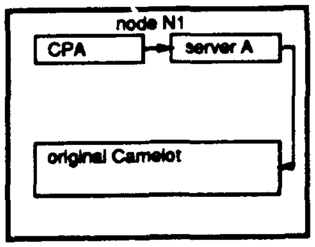
- Non-paging vs. paging. A non-paging test measures the cost of reading or writing a region that is currently cached in physical memory. A paging test incurs the additional cost of reading the page from disk and possibly writing back a dirty page.
- Reads vs. writes. A write is much more expensive than a read because modifications must be logged to stable storage by the end of the transaction, and modified pages must eventually be written back to non-volatile storage.
- Virtual memory vs. buffer pool. Eppinger compared the virtual memory implementation to an equivalent buffer pool implementation to demonstrate the practicality of the virtual memory approach.

Camelot with TDSM does not include a buffer pool implementation of TDSM, so this parameter is not considered in this dissertation. However, some additional dimensions are of interest:

- Local servers vs. remote servers. A local server is one that is running on the same node as Camelot; a remote server runs on a different node. Many latencies are higher for remote servers because of longer communication times. Comparing the latencies for local and remote servers identifies the communication component of the performance model.
- Local locks vs. distributed locks. When two servers share a recoverable segment, they must share locks via RPCs to the Lock Manager for correct operation. When only one server uses a recoverable segment, the lock library within the server can independently grant all locks.
- External Memory Manager vs. no External Memory Manager. If all servers run on the same node, the Camelot external pager can bypass the External Memory Manager and communicate directly with the Mach kernel. Varying this dimension allows the overhead of the External Memory Manager to be measured.
- Camelot with TDSM vs. original Camelot. Camelot with TDSM provides a superset of the original Camelot functionality. Comparing the performance of the two implementations identifies the overhead incurred by applications that do not use the additional functionality.
- Multiple servers vs. one server. To demonstrate the viability of data sharing, the cost of multiple servers sharing a TDSM segment must be measured.

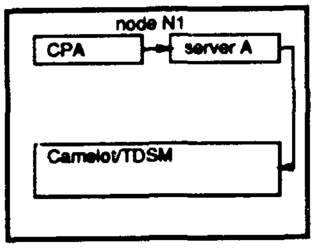
The parameters are not completely independent. The original Camelot does not support remote servers, multiple servers, distributed locking, nor the External Memory Manager. Camelot with TDSM must use the External Memory Manager to support multiple servers.

The operations of interest in characterizing TDSM are reads and writes (paging and non-paging), and transaction overhead (initiate and commit). To measure these operation times using the primitive analysis methodology, I ran a number of tests written in the Camelot Performance Analyzer (CPA) language [Eppinger 87]. The CPA tests are executed by an interpreter, which initiates a transaction and then makes RPCs to one or more Camelot servers to perform operations on recoverable virtual memory. For most tests, the servers perform multiple operations for each RPC, since the cost of a single operation is small relative to the cost of an RPC. Eight different configurations are used during the course of the tests:



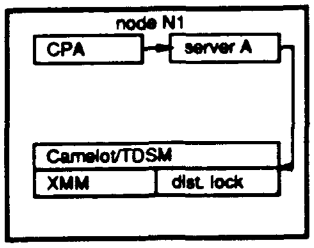
Local server, original Camelot.

**Figure 6-1:** Test configuration 1



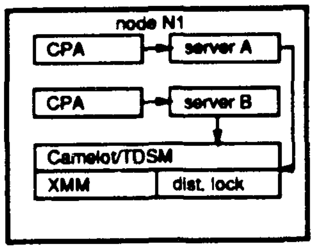
Local server, Camelot/TDSM.

**Figure 6-2:** Test configuration 2



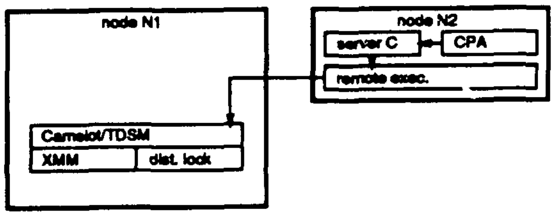
Local server, distributed locks, Camelot/TDSM and External Memory Manager.

**Figure 6-3:** Test configuration 3



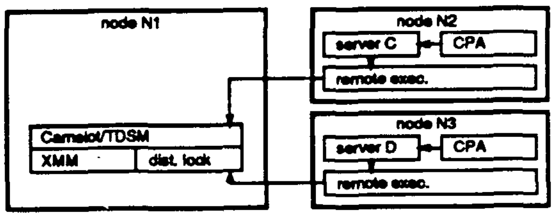
Two local servers, distributed locks, Camelot/TDSM and External Memory Manager.

**Figure 6-4:** Test configuration 4



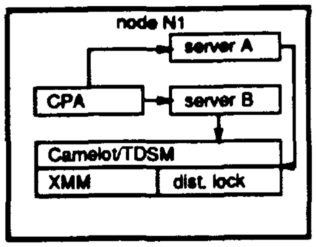
Remote server, distributed locks, Camelot/TDSM and External Memory Manager.

**Figure 6-5:** Test configuration 5



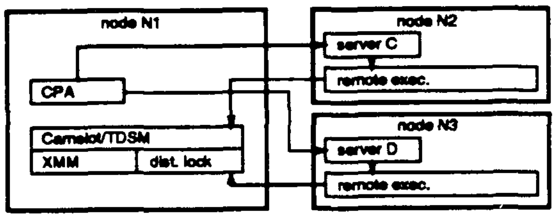
Two remote servers, distributed locks, Camelot/TDSM and External Memory Manager.

**Figure 6-6:** Test configuration 6



Two local servers, interleaved calls, distributed locks, Camelot/TDSM and External Memory Manager.

**Figure 6-7:** Test configuration 7



Two remote servers, interleaved calls, distributed locks, Camelot/TDSM and External Memory Manager.

**Figure 6-8:** Test configuration 8

1. The CPA interpreter makes an RPC to a server which accesses a recoverable segment using the original Camelot. All of the processes (interpreter, server, and Camelot) are on the same node. Locks are granted locally by the library in the server. This configuration provides a baseline for comparison with the other configurations. (Figure 6-1.)

2. The CPA interpreter makes an RPC to a server which accesses a recoverable segment using Camelot with TDSM, but without the External Memory Manager. All processes are on the same node and locks are granted locally. Except for a different version of Camelot, this configuration is identical to configuration 1, and a comparison of the results shows the performance impact of the changes to Camelot. (Figure 6-2.)

3. The CPA interpreter makes an RPC to a server which accesses a recoverable segment using Camelot with TDSM and the External Memory Manager. All processes are on the same node. Locks are granted via the Lock Manager. This configuration adds distributed locks and the External Memory Manager to configuration 2; comparing results in the two configurations identifies the overhead imposed by distributed locks and the External Memory Manager. (Figure 6-3.)

   Two test variants are used in this configuration. In one variant, the server performs several operations on each RPC, for comparison with configuration 2. In the other variant, only one operation is performed on each RPC. This variant provides a baseline for comparison with configuration 7.

4. Two servers share a recoverable segment using Camelot with TDSM and the External Memory Manager. Two CPA interpreters, one for each server, make RPCs to their respective servers concurrently. All processes are on the same node. The servers physically share memory, and locks are granted via the Lock Manager. This configuration adds a second CPA interpreter and server to configuration 3. Comparing the results of the two configurations shows the cost of sharing a segment within a node. (Figure 6-4.)

   This configuration is used only for read operations. Because two servers can cache a read lock simultaneously, the order in which transactions obtain locks does not matter, and the two servers may execute concurrently with no coordination between the two. This configuration cannot be used to obtain meaningful results for write operations, because the servers must contend with each other to cache a write lock. The order in which servers obtain locks does matter, because it is much less expensive for a transaction to obtain a lock when the lock is already cached by the server where the transaction is running. Thus, to measure write operations, the two servers must be coordinated by a single CPA interpreter.

5. A CPA interpreter on node *N2* makes an RPC to a server, also on node *N2*. Via the Remote Execution Manager, the server accesses a recoverable segment on node *N1* using Camelot with TDSM and the External Memory Manager. Locks are granted by the Lock Manager. This configuration modifies configuration 3 by moving the CPA interpreter and server to a different node. A comparison of the results shows the cost of accessing a recoverable segment remotely. (Figure 6-5.) As in configuration 3, two test variants are used, one for comparison with configuration 8.

6. A server on node *N2* and a server on node *N3* share a recoverable segment on node *N1* using Camelot with TDSM and the External Memory Manager. A CPA interpreter on node *N2* and a CPA interpreter on node *N3* make RPCs to the server on their respective nodes concurrently. The External Memory Manager maintains the coherency of the shared segment, and locks are granted via the Lock Manager.

This configuration adds a second CPA interpreter and server on another node to configuration 5, and modifies configuration 4 by moving the CPA interpreters and servers to two separate nodes. A comparison of the results of these configurations shows the cost of sharing a segment between nodes. (Figure 6-6.) As with configuration 4, this configuration is used only for read operations.

7. Two servers share a recoverable segment using Camelot with TDSM and the External Memory Manager. A single CPA interpreter interleaves RPCs to each server, so that an operation on one server is always followed by an operation on the other server. All processes are on the same node. The servers physically share memory, and locks are granted via the Lock Manager. This configuration is similar to configuration 4, but it uses one, instead of two, CPA interpreters, so that write operations may be measured. Each server performs only one operation per RPC. A comparison of the results of this configuration with configuration 3 shows the cost of sharing a segment within a node. (Figure 6-7.)

8. A server on node *N2* and a server on node *N3* share a recoverable segment on node *N1* using Camelot with TDSM and the External Memory Manager. A single CPA interpreter on node *N1* interleaves RPCs to each server, so that an operation on one server is always followed by an operation on the other server. The External Memory Manager maintains the coherency of the shared segment, and locks are granted via the Lock Manager. This configuration is similar to configuration 6, but it uses one, instead of two, CPA interpreters, so that write operations may be measured. A comparison of the results of this configuration with configurations 3, 5, 6, and 7 shows the cost of sharing a segment between nodes, including inter-node page invalidations. (Figure 6-8.)

### 6.3.2. Non-paging tests

These tests measure the incremental cost to a transaction of reading or writing a region of a virtual memory page that is already present in physical memory. Once a page is cached in physical memory, there should be little or no difference between various configurations in the cost of reading a region; however, write operations and the overhead per transaction should be higher for remote servers than for local servers.

For non-paging operations, the following test is used:

```
repeat $t$ times
  begin transaction
    make one RPC to server to do
      repeat $k$ times
        lock & read or write $x$ bytes on the same page
    return from RPC
  end transaction
```

#### 6.3.2.1. Non-paging reads

For these tests, the number of transactions $t$ is either 100 or 500 depending on the configuration. Within each configuration, 9 sets of parameters are used, independently varying both the size $x$ of the virtual memory region (32, 128, and 1024 bytes) and the number of times $k$ each transaction accesses the region (50, 100, and 200 times). For a given configuration and set of parameters $x$ and $k$, the following methodology is used. Run the test and measure the elapsed time. Divide the elapsed time by $t$, the number of transactions, to determine the average cost per transaction. Repeat the test many times to ensure the system is in a steady state (locks and pages cached, disk storage allocated). Repeat the test several times to obtain several measurements of the average time per transaction, and record the median value of these measurements. (The median measurement is used in a attempt to discard statistical outliers; for example, tests where the disk was particularly well synchronized with the CPU, or when the network was especially congested. But note that the median measurement is always close to the minimum measurement in non-paging tests.)

| Configuration | trans. overhead (ms.) | 32 byte read (ms.) | 128 byte read (ms.) | 1024 byte read (ms.) | fixed cost per read (ms.) | variable cost per byte (μs.) |
|---|---|---|---|---|---|---|
| 1. Local server original Camelot ($t$=500) | 9.5 | .294 | .313 | .461 | .290 | .167 |
| 2. Local server Camelot/TDSM ($t$=500) | 9.5 | .313 | .332 | .473 | .310 | .160 |
| 3. Local server Camelot/TDSM/XMM ($t$=100) | 10.0 | .307 | .326 | .480 | .303 | .173 |
| 4. Two local servers Camelot/TDSM/XMM ($t$=100) | 10.0 | .304 | .345 | .490 | .311 | .175 |
| 5. Remote server Camelot/TDSM/XMM ($t$=100) | 57.0 | .279 | .320 | .460 | .285 | .172 |
| 6. Two remote servers Camelot/TDSM/XMM ($t$=100) | 62.5 | .280 | .307 | .460 | .279 | .177 |

The "trans. overhead" column reports the computed time to initiate and terminate a read-only transaction in each configuration. The "32 byte read," "128 byte read," and "1024 byte read" columns record the incremental cost to read a region of the given size. The "fixed cost per read" and "variable cost per byte" columns are computed from a linear regression on the 32, 128, and 1024 byte columns.

**Table 6-2:** Non-paging read times

Next, for a given configuration and given region size $x$, the measurements of average transaction times vs. different values of $k$ (the number of operations per transaction) are entered into a linear regression computation to determine the overhead per transaction and incremental cost per $x$-byte read operation. (In configurations 4 and 6, the linear regression uses $2k$ instead of $k$, since the two servers collectively execute twice as many transactions in a given time period.)

These incremental costs and overheads are reported in Table 6-2 in the "trans. overhead," "32 byte read," "128 byte read," and "1024 byte read" columns.

Finally, the incremental times per read operation of 32, 128, and 1024 byte regions are entered into a linear regression computation to determine the fixed cost per read and variable cost per byte. These fixed and variable costs are reported in Table 6-2 in the "fixed cost per read" and "variable cost per byte" columns. The correlation coefficient for all linear regressions is greater than 0.99. (Results for configurations 7 and 8 are reported in subsection 6.3.4.) The measured times per transaction include the time to obtain a lock on the object.

Eppinger reported a fixed cost of .016 milliseconds per read with a variable cost of .163 microseconds per byte for non-paging virtual memory reads. The results in Table 6-2 for configuration 1 using substantially the same version of Camelot are .290 milliseconds per read and .167 microseconds per byte. The difference between the times per byte is inconsequential; the difference between the times per read can be attributed to the cost of obtaining a read lock, which was not included in Eppinger's measurements. The per transaction overhead of 9.5 milliseconds includes the IPC time of `TA_GetTids` amortized over several transactions and the sum (7.07 milliseconds) of the local IPC times for `TS_Join`, `TA_End`, and `ST_Vote`, plus CPU time for commit processing.

The per transaction overhead remains substantially the same in configurations 2, 3, and 4; IPC times are the dominant cost, and the times are the same in all local configuration. The per transaction overhead increases to 57.0 milliseconds in configuration 5 because of increased IPC times: 48.4 milliseconds for `TS_Join`, `TA_End`, and `ST_Vote`, plus CPU time for commit processing and the IPC time of `TA_GetTids` amortized over several transactions. In configuration 6, the per transaction overhead is still higher (62.5 milliseconds) because messages of nodes *N2* and *N3* must contend with each other for the CPU on node *N1*, increasing IPC times.

The lower fixed cost per read in configurations 5 and 6 relative to configuration 1 can be attributed to fewer context switches (since Camelot is on a different node than the servers) resulting in a higher CPU cache hit ratio. I cannot account for the higher variable cost per byte in configurations 3 through 6, the higher fixed cost per read in configuration 2, or the lower variable cost per byte in configuration 2, although the number of transactions per test may be a contributing factor in some cases.

#### 6.3.2.2. Non-paging writes

The same methodology as used for non-paging read tests is used to determine the overhead per transaction and incremental cost per *x*-byte write operation. These incremental costs and overheads are reported in Table 6-3 in the "trans. overhead," "32 byte write," "128 byte write," and "1024 byte write" columns. Similarly, the fixed cost per write and variable cost per byte are computed via linear regressions, and are reported in the "fixed cost per write" and "variable cost per byte" columns. This table reports write times only for single-server configurations. Subsection 6.3.4 describes a different test used to order the sequence of operations to obtain meaningful results for writes in multi-server configurations. The measured times per transaction include the time to obtain a lock on the object and to write log records to disk. (The cost of obtaining a cached distributed lock is within 0.1 milliseconds of the cost of obtaining a local lock.)

| Configuration | trans. overhead (ms.) | 32 byte write (ms.) | 128 byte write (ms.) | 1024 byte write (ms.) | fixed cost per write (ms.) | variable cost per byte (μs.) |
|---|---|---|---|---|---|---|
| 1. Local server original Camelot (*t*=500) | 17.5 | 1.43 | 1.93 | 5.34 | 1.37 | 3.88 |
| 2. Local server Camelot/TDSM (*t*=500) | 18.0 | 1.47 | 1.96 | 5.35 | 1.40 | 3.86 |
| 3. Local server Camelot/TDSM/XMM (*t*=100) | 17.5 | 1.44 | 2.02 | 5.57 | 1.40 | 4.08 |
| 5. Remote server Camelot/TDSM/XMM (*t*=100) | 103 | 2.93 | 4.79 | 20.9 | 2.42 | 18.0 |

The "trans. overhead" column reports the computed time to initiate and terminate an update transaction in each configuration. The "32 byte write," "128 byte write," and "1024 byte write" columns record the incremental cost to write a region of the given size. The "fixed cost per write" and "variable cost per byte" columns are computed from a linear regression on the 32, 128, and 1024 byte columns.

**Table 6-3:** Non-paging write times

Eppinger reported a fixed cost of .760 milliseconds per write with a variable cost of .631 microseconds per byte for non-paging virtual memory writes. The results in Table 6-3 for configuration 1 using the same version of Camelot are 1.37 milliseconds per write and 3.88 microseconds per byte. The difference between the times per byte can be attributed to the incremental cost of writing the log record to disk, which Eppinger excluded from his measurements. (Eppinger measured the log I/O cost separately, and reported a value of .930 microseconds per byte.) The difference between the times per write can be attributed to the cost of obtaining a write lock (not measured by Eppinger) and the incremental cost of writing the log record to disk. The per transaction overhead of 17.5 milliseconds includes the sum of the local IPC times for `TS_Join`, `TA_End`, `ST_Vote`, and `ST_Commit` (7.96 milliseconds), the IPC time of `TA_GetTids` amortized over several transactions, time to initiate the log write to disk, plus CPU time to process the shared memory queue (pin and log requests) and for commit processing. The log write apparently overlaps some of the CPU processing.

The per transaction overhead remains substantially unchanged in configurations 2 and 3; IPC times are the dominant cost, and the times are the same in all local configuration. The per transaction overhead increases to 103 milliseconds in configuration 5 because of increased IPC times (a higher amortized cost for `TA_GetTids`, plus 61.8 milliseconds for `TS_Join`, `TA_End`, `ST_Vote`, and `ST_Commit`) and an additional message to obtain the shared memory queue (17.2 milliseconds for `XD_GetShMemQueue`). Also, the page being modified must be flushed every checkpoint interval in order to limit recovery times. This cost of this flush is amortized over many transactions, but the IPC time for `memory_object_data_write` increases from 3.70 milliseconds in configurations 1, 2, and 3 to 185.6 milliseconds in configuration 5.

The fixed cost per write and variable cost per byte are approximately the same in configurations 1, 2, and 3. The higher fixed cost per write in configuration 5 can be attributed to copying the pin and log requests in the shared memory queue back to the home node. The higher variable cost per byte in configuration 5 can be attributed to the incremental cost of copying a byte in a shared memory queue log request back to the home node.

## 6.3.3. Paging tests

These tests measure the cost to a transaction of reading or writing a region of a virtual memory page that is not present in physical memory. For read faults, the kernel must discard an old clean page, and obtain a new page from the Camelot external pager. For write faults, the kernel must write back an old, dirty page, and obtain a new page from the Camelot external pager. Thus, read operations, write operations, and the overhead per transaction should all be higher for remote servers than for local servers.

For paging operations, the following test is used:

```
repeat t times
  begin transaction
    make one RPC to server to do
      repeat k times
        lock & read or write x bytes on a new page
    return from RPC
  end transaction
```

The number of transactions *t* is either 10 or 20 depending on the configuration and whether a read or write test is being performed. Nine sets of parameters are used within each configuration: the size *x* of the virtual memory region varies from 32 to 128 to 1024 bytes, and the number of pages *k* accessed by each transaction varies from 25 to 50 to 100 pages. As with non-paging tests, median measurements are recorded. (The median measurement is usually close to the average measurement in paging tests, but the range of actual measured values is relatively large, in most cases varying up to 5% from the mean. Disk reads and writes may occur in parallel with CPU computations, but because the CPU times are much smaller than disk times, the CPU sometimes must wait for a disk operation to complete. This causes the measured transaction times to vary.) The same methodology is used to compute the incremental cost per $x$-byte read or write operation. These incremental costs are reported in Table 6-4. This table reports paging operation times only for single-server configurations. Paging operation times for multi-server configurations are addressed in subsection 6.3.4.

| Configuration | 32 byte read (ms.) | 128 byte read (ms.) | 1024 byte read (ms.) | 32 byte write (ms.) | 128 byte write (ms.) | 1024 byte write (ms.) |
|---|---|---|---|---|---|---|
| 1. Local server original Camelot ($t$=10) | 27.4 | 27.2 | 27.6 | 100 | 105 | 109 |
| 2. Local server Camelot/TDSM ($t$=20 for reads; $t$=10 for writes) | 27.0 | 26.8 | 26.4 | 111 | 111 | 106 |
| 3. Local server Camelot/TDSM/XMM ($t$=20 for reads; $t$=10 for writes) | 34.0 | 34.3 | 34.0 | 115 | 119 | 120 |
| 5. Remote server Camelot/TDSM/XMM ($t$=20 for reads; $t$=10 for writes) | 207 | 207 | 210 | 385 | 372 | 382 |

The "32 byte read," "128 byte read," and "1024 byte read" columns record the incremental cost to discard a clean page, page in a new page, and read a region of a given size on the new page. The "32 byte write," "128 byte write," and "1024 byte write" columns record the incremental cost to page out a dirty page, page in a new page, and modify a region of a given size on the new page. The values are valid to within approximately 5%.

**Table 6-4:** Paging times

The results in Table 6-4 show only the incremental costs of paging reads and writes. The fixed cost per operation and the variable cost per byte are not shown in the table. The additional cost added to a paging operation by accesses to additional bytes on a page is extremely small relative to the cost of servicing the page fault, and cannot be reliably measured. This variable cost per byte should be the same for paging and non-paging operations. The fixed cost per paging operation, and the incremental cost of a 128- or 1024-byte operation is, to two significant digits, identical to the incremental cost of the corresponding 32-byte operation.

The overhead per transaction is not shown in Table 6-4 because it, too, cannot be reliably measured from the tests. The overhead per transaction should be the same for both paging and non-paging operations, so the expected overhead per transaction for a paging write test in configuration 1 is 17.5 milliseconds. The total time per transaction for 25 32-byte paging write operations is expected to be 2522 milliseconds. An error of only 1% in the measurement of the time for this transaction would eliminate or double the per transaction overhead. Since the measured times varied by more than 1%, the overhead per transaction cannot be reliably measured from these paging tests.

The measured times per transaction include the time to obtain a lock on the object and to write log records to disk.

Eppinger reported an incremental cost of 5.23 milliseconds for a 32-byte paging read, and 12.1 milliseconds for a 32-byte paging write. The results in Table 6-4 for configuration 1 using the same version of Camelot are 27.4 milliseconds for a 32-byte paging read, and 100.2 milliseconds for a paging write. The difference between the read times can be attributed to the cost of reading a page from the disk (21.1 milliseconds with no seek, since the tests access the pages sequentially and there are no intervening log writes), which Eppinger excluded from his measurements. (Eppinger measured the paging I/O cost separately, and reported a latency of 21 milliseconds for sequential 4096-byte accesses.) The difference between the write times can be attributed to the cost of seeking and writing a dirty page to the disk (35 milliseconds), seeking and reading a clean page from the disk (35 milliseconds), and the amortized cost of seeking and writing the log at the end of the transaction, which Eppinger again excluded from his measurements. (Eppinger reported an average rotational latency of 8.3 milliseconds, an average seek time of 28 milliseconds, a latency of 21 milliseconds for sequential 4096-byte accesses, and a DMA transfer rate of .930 milliseconds per byte.)

The incremental cost of a 32-byte paging read remains the same in configuration 2. The incremental cost of a 32-byte paging write appears to increase in configuration 2, but results for 128- and 1024-byte writes indicate that the time is substantially unchanged within the error of the measurements. The incremental cost of paging reads and paging writes increases by 5 to 10 milliseconds from configuration 2 to configuration 3. This can be attributed to the additional processing performed by the External Memory Manager in configuration 3. Some of this extra time could be eliminated by merging the External Memory Manager into Camelot, eliminating some duplicated data structures.

The incremental cost of a paging read increases to 207 milliseconds in configuration 5 because of increased IPC times for `memory_object_data_request` and `memory_object_data_provided`. (The 180 millisecond increase in IPC time is equivalent within the error of the measurements to the 173 millisecond increase in incremental cost.) The incremental cost of a paging write increases to 385 milliseconds in configuration 5 because of increased IPC times for `memory_object_data_write`, `memory_object_data_request`, and `memory_object_data_provided`. Also, on each pageout Camelot requests the shared memory queue log buffer (even though it is empty) from the Remote Execution Manager, which adds another 17.2 milliseconds to the cost of a paging write. The Mach kernel attempts to minimize the number of times it must exercise its pageout algorithm, so writebacks of dirty pages occur in batches that are not synchronized with page faults by the paging test program. Because of the increased IPC times for remote servers, many writebacks of dirty pages are delayed long enough to be outside the scope of the test measurement. Thus, the increase in measured incremental cost (270 milliseconds) is less than the expected increase in IPC time (379 milliseconds).

### 6.3.4. Multi-server tests

These tests measure the incremental cost to a transaction of reading or writing a region of a virtual memory page that is in use by another server. For read operations, both servers cache the page and corresponding lock, and no invalidations take place. For write operations, the actions taken by Camelot depend on the location of the servers. When the servers are on different nodes, the page and corresponding lock migrates from node to node. When the servers are on the same node, the lock migrates from server to server, but both servers can physically share memory, and no page faults should occur.[8]

When two servers on the same node share a recoverable segment, they physically share memory, and the order in which their accesses are interleaved should not affect the test results. However, when two servers on different nodes share a recoverable segment, a page must be shipped from node to node each time a different node modifies the page, so the order of accesses does matter. To ensure that the accesses alternate, a single CPA interpreter controls both servers and interleaves calls to each server. Thus, for configurations with two servers, the following test is used:

```
repeat t times
  begin transaction
    repeat k times
      make RPC to first server to do
        lock & read or write x bytes on same page
      return from RPC
      make RPC to second server to do
        lock & read or write x bytes on same page
      return from RPC
  end transaction
```

In the above test, the CPA interpreter makes an RPC for each server operation, and the results are not comparable to those in subsections 6.3.2 and 6.3.3, where one RPC results in many server operations. To provide a new baseline for comparison, the following test is used in single-server configurations:

[8]Unfortunately, the IBM RT/PC has inverted page tables, which means that a physical memory page may not have two virtual memory addresses. Thus, when a server attempts to access a page that is currently mapped to a virtual memory address in another server, it takes a page fault. The Mach kernel has a "fast path" for this type of page fault which immediately remaps the physical page to the alternate virtual memory address.

```
repeat t times
  begin transaction
    repeat k times
      make RPC to server to do
        lock & read or write x bytes on same page
      return from RPC
  end transaction
```

### 6.3.4.1. Multi-server reads

For multi-server read tests, the number of transactions *t* is 20 in all configurations. Within each configuration, 9 sets of parameters are used, varying both the size *x* of the virtual memory region (32, 128, and 1024 bytes) and the number of times *k* each transaction calls the servers. In the local configurations (3 and 7), *k* is varied from 50 to 100 to 200. Because of longer test times in the remote configurations (5 and 8), *k* is varied from 25 to 50 to 100.

| Configuration | trans. overhead (ms.) | 32 byte read (ms.) | 128 byte read (ms.) | 1024 byte read (ms.) | fixed cost per read (ms.) | variable cost per byte (μs.) |
|---|---|---|---|---|---|---|
| 3. Local server Camelot/TDSM/XMM (*t*=20) | 7.0 | 2.25 | 2.33 | 2.93 | 2.24 | .679 |
| 5. Remote server Camelot/TDSM/XMM (*t*=20) | 33.5 | 17.9 | 18.0 | 25.5 | 17.3 | 7.94 |
| 7. Two local servers Camelot/TDSM/XMM (*t*=20) | 9.5 | 2.36 | 2.45 | 3.05 | 2.35 | .685 |
| 8. Two remote servers Camelot/TDSM/XMM (*t*=20) | 71.0 | 17.9 | 18.2 | 25.5 | 17.4 | 7.86 |

The "trans. overhead" column reports the computed time to initiate and terminate a read-only transaction in each configuration. The "32 byte read," "128 byte read," and "1024 byte read" columns record the incremental cost to read a region of the given size. The "fixed cost per read" and "variable cost per byte" columns are computed from a linear regression on the 32, 128, and 1024 byte columns.

**Table 6-5:** Multi-server read times

The same methodology as used for non-paging read tests is used to determine the overhead per transaction and incremental cost per *x*-byte read operation, although minimum, rather than median, times are recorded for multi-server read tests. (The minimum time is used to discount the effects of other traffic on the network.) Also, in configurations 7 and 8, the linear regression uses 2*k* instead of *k*, since the two servers collectively execute twice as many transactions in a given time period. These incremental costs and overheads are reported in Table 6-5 in the "trans. overhead," "32 byte read," "128 byte read," and "1024 byte read" columns. Similarly, the fixed cost per read and variable cost per byte are computed via linear regressions, and are reported in the "fixed cost per read" and "variable cost per byte" columns. The correlation coefficient for all linear regressions is greater than 0.99. (The results reported are intended to character multi-server configurations. See subsections 6.3.2 and 6.3.3 for other configurations.) The measured times per transaction include the time to obtain a lock on the object.

Table 6-2 reports a fixed cost of .303 milliseconds per read with a variable cost of .167 microseconds per byte for non-paging virtual memory reads in configuration 3. The results in Table 6-5, with an RPC from the CPA interpreter to the server for each read operation, are 2.24 milliseconds per read and .679 microseconds per byte. The higher fixed cost per read is the result of the extra RPC. The higher variable cost per byte results because the RPC copies the data back to the CPA interpreter. (The average cost of a 32-byte read `cpa_read_s` local RPC is 2.30 milliseconds, and the cost of a 1024-byte read `cpa_read_l` local RPC is 2.88 milliseconds. This gives a fixed cost per local read RPC of 2.28 milliseconds, and a variable cost per byte of .58 microseconds.) The transaction overhead appears lower than in Table 6-2 because it is a minimum, rather than a median, measurement.

In configuration 7, the fixed and variable costs per read are slightly higher than configuration 3 because the CPU cache hit ratio is lower due to contention by the two servers. The overhead per transaction increases because two more RPCs are performed for the second server, `TS_Join` and `ST_Vote`.

In configuration 5, the fixed and variable costs per read are higher than configuration 3 due to increased IPC times for the RPC from the CPA interpreter to the server. (This RPC has a fixed cost of 16.9 milliseconds per call with a variable cost of 7.4 microseconds per byte.) The transaction overhead increases because of increased IPC times for `TS_Join` and `ST_Vote` (32.3 milliseconds in configuration 5 as compared to 4.64 milliseconds in configuration 3). The transaction overhead for configuration 5 is smaller here than in table 6-2 because the CPA interpreter is on the same node as Camelot here, and cost of the `TA_GetTids` and `TA_End` RPCs made by the CPA interpreter is lower.

Finally, the fixed and variable costs per read in configuration 8 are the same as configuration 5 as expected. The overhead per transaction increases because two more RPCs are performed to the second server, `TS_Join` and `ST_Vote`.

### 6.3.4.2. Multi-server writes

The same methodology as used for multi-server read tests is used to determine the overhead per transaction and incremental cost per *x*-byte write operation. The number of transactions for write tests is either 10 or 20 depending on the configuration. These incremental costs and overheads are reported in Table 6-6 in the "trans. overhead," "32 byte write," "128 byte write," and "1024 byte write" columns. Similarly, the fixed cost per write and variable cost per byte are computed via linear regressions, and are reported in the "fixed cost per write" and "variable cost per byte" columns. The results reported are intended to characterize multi-server configurations. See subsections 6.3.2 and 6.3.3 for other configurations. The measured times per transaction include the time to obtain a lock on the object and to write log records to disk. (The cost of obtaining a cached distributed lock is within 0.1 milliseconds of the cost of obtaining a local lock.)

| Configuration | trans. overhead (ms.) | 32 byte write (ms.) | 128 byte write (ms.) | 1024 byte write (ms.) | fixed cost per write (ms.) | variable cost per byte (μs.) |
|---|---|---|---|---|---|---|
| 3. Local server Camelot/TDSM/XMM (*t*=20) | 10.0 | 2.90 | 3.97 | 8.07 | 3.02 | 4.96 |
| 5. Remote server Camelot/TDSM/XMM (*t*=20) | 49.0 | 20.2 | 23.1 | 50.1 | 19.2 | 30.1 |
| 7. 2 local servers Camelot/TDSM/XMM (*t*=20) | 15.5 | 3.56 | 4.09 | 8.05 | 3.46 | 4.48 |
| 8. 2 remote servers Camelot/TDSM/XMM (*t*=10) | n.a. | 430 | 503 | 523 | 460 | 65.2 |

The "trans. overhead" column reports the computed time to initiate and terminate an update transaction in each configuration. The "32 byte write," "128 byte write," and "1024 byte write" columns record the incremental cost to write a region of the given size. The "fixed cost per write" and "variable cost per byte" columns are computed from a linear regression on the 32, 128, and 1024 byte columns.

**Table 6-6:** Multi-server write times

Table 6-3 reports a fixed cost of 1.40 milliseconds per write with a variable cost of 4.08 microseconds per byte for non-paging virtual memory writes in configuration 3. The results in Table 6-6, with an RPC from the CPA interpreter to the server for each write operation, are 3.02 milliseconds per write and 4.96 microseconds per byte. The higher fixed cost per write is the result of the extra RPC. The higher variable cost per byte results because the RPC copies the data from the CPA interpreter to the server. (The fixed and variable costs for write RPCs should be the same as for read RPCs, 2.28 milliseconds and .58 microseconds, respectively.) The transaction overhead appears lower than in Table 6-3 because it is a minimum, rather than a median, measurement.

As with read operations, the fixed and variable costs per write in configuration 7 are roughly equal to those of configuration 3. The overhead per transaction increases because three more messages are needed for the second server, `TS_Join`, `ST_Vote`, and `ST_Commit`.

In configuration 5, the fixed cost per write is higher than configuration 3 due to increased IPC time for the RPC from the CPA interpreter to the server. The variable cost per byte is higher because of increased IPC times to copy a byte from the CPA interpreter to the server, and to copy a byte in a shared memory queue log request back to the home node. The transaction overhead increases because of increased IPC times for `TS_Join`, `ST_Vote`, and `ST_Commit` (45.7 milliseconds in configuration 5 as compared to 5.43 milliseconds in configuration 3). The transaction overhead for configuration 5 is smaller here than in table 6-3 because the CPA interpreter, which makes the `TA_GetTids` and `TA_End` RPCs, is on the same node as Camelot here.

The write test in configuration 8 is the only multi-server test in which pages actually migrate across the network. The overhead per transaction is not shown for this configuration because it is smaller than the error of the measurements, and cannot be reliably measured from the tests. The overhead per transaction is expected to be on the order of 100 milliseconds (with extra `TS_Join`, `ST_Vote`, and `ST_Commit` messages for the second server as compared to configuration 5) which is less than 1% of the total time per transaction. (Each transaction is at least 23,000 milliseconds.) A write operation by the server on node *N2* causes the kernel to send a `memory_object_data_request` to Camelot on node *N1*. Camelot does a `memory_object_lock_request` to the kernel on node *N3*, which results in a `memory_object_data_write` reply, followed by `memory_object_lock_completed`. After flushing log records to stable storage (using `XD_GetShMemQueue` to pick up the log records from node *N3*) and writing the updated page to disk, Camelot on node *N1* can reply to the request of the kernel on node *N2* with a `memory_object_data_provided`. A write operation by the server on node *N3* causes a similar sequence of events. The IPC time for the `XD_GetShMemQueue` and `memory_object` calls on each write operation is 411.6 milliseconds. The increase in the fixed cost per write in configuration 8 as compared to configuration 5 can be attributed to this extra IPC time and the cost of flushing the log to stable storage and writing the page to disk. The variable cost per byte increases because of the extra log writes performed on each page migration.

## 6.3.5. Summary

Table 6-7 summarizes the latency of key operations in the two configurations (configurations 3 and 5) that are significant in the remainder of the dissertation. A non-paging read is slightly less expensive in the remote configuration (configuration 5) because of reduced context switch overhead. A non-paging write is slightly more expensive in the remote configuration because log records must be transferred over the network at the end of transaction. Paging reads and writes are much more expensive because of the high cost of sending a page over the network. And finally, transaction overheads are higher in the remote configuration because transaction management messages must be sent over the network.

| Operation | Local | Remote |
|---|---|---|
| Non-paging 32-byte read (ms.) | .307 | .279 |
| Non-paging 32-byte write (ms.) | 1.44 | 2.93 |
| Paging read (ms.) | 34.0 | 207 |
| Paging write (ms.) | 115 | 385 |
| Read-only Transaction (ms.) | 10.0 | 57.0 |
| Update Transaction (ms.) | 17.5 | 103 |

The "Local" column reports the latency of the given operations in configuration 3. The "Remote" column reports the latency of the given operations in configuration 5.

**Table 6-7:** Summary of performance measurements

## 6.4. Throughput

The debit-credit, or ET1, benchmark is commonly used to measure the throughput of transaction processing systems [Anonymous et al 85]. The ET1 benchmark is studied here both to measure throughput, and as an example for the performance model reported in the next chapter.

The throughput of a system is given in TPS, the number of ET1 transactions per second it can achieve with at least 95% of the transactions completing in 1 second or less. The standard ET1 database for a system capable of 100 TPS holds balances for a bank with 1,000 branches, 10,000 tellers, and 10,000,000 accounts (a total of 1,001,100,000 bytes). The database also includes a 90-day history file (1 gigabyte). Each ET1 transaction updates a branch balance, a teller balance, and an account balance, and appends a record to the history file.

The ET1 standard specifies an environment with block-mode terminals and X.25 connections to a large mainframe. The benchmark was adapted by Pausch for the Camelot environment, where an "ATM" process communicating with a presentation server replaces block-mode terminals, TCP/IP replaces X.25, and workstations replace the mainframe [Pausch 88]. The presentation server forwards the request from the ATM process to a data server, which updates the ET1 database in recoverable virtual memory. All processes are multi-threaded, so that several transactions can be in progress simultaneously. An ET1 transaction from the perspective of the ATM process is given in Figure 6-9.

For the ET1 tests described in this section, the ATM process, the presentation server, and data server all run on the same node, and the data for all branches is stored in a single recoverable

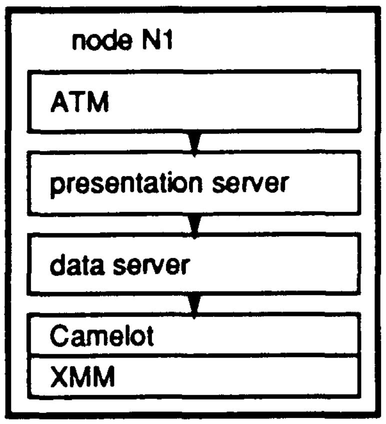
```
pick random branch number, teller number,
account number
begin transaction
  make RPC to presentation server to do:
    make RPC to data server to do:
      lock branch balance
      update branch balance
      lock teller balance
      update teller balance
      lock account balance
      update account balance
      begin lazy server-based transaction
        lock history pointer
        increment history pointer
      end lazy server-based transaction
      update history entry
    return from RPC to data server
  return from RPC to presentation server
end transaction
```

**Figure 6-9:** ET1 transaction

segment. The system used for the tests can achieve on the order of 10 TPS, so 10/100 of the standard database is used: 100 branches, 1,000 tellers, and 1,000,000 accounts (a total of 100,110,000 bytes). Because of limited disk space and relatively short test durations, a smaller history file of only 5 megabytes is used (approximately 4.5 days).

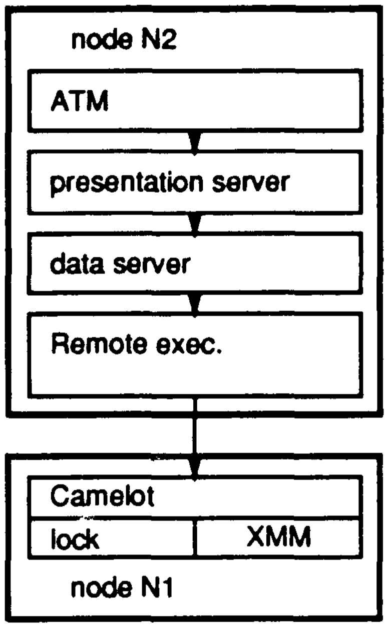
The ET1 test is run in two *configurations*. In the "local" configuration, all processes run on the same node, and locking is done entirely within the server library (Figure 6-10). The local configuration is the traditional configuration for running the Camelot ET1 benchmark. In the "remote" configuration, Camelot including the Lock Manager runs on node *N1*, while the ATM process, the presentation server, the data server, and the Remote Execution Manager run on node *N2*. The data server must make RPCs to the Lock Manager to cache locks (Figure 6-11). The remote configuration is meant to illustrate the performance of data sharing, since the data server must obtain pages and transaction services across the network, although the server's RVM segment is not actually shared with another server. Both configurations use Camelot with TDSM and the External Memory Manager.

To determine the TPS rating of Camelot, the following methodology is used. Start the system, and execute 3000 ET1 transactions. The database contains 24,415 pages of account records, while main memory holds less than 2500 pages. Each ET1 transaction accesses an account record at random, so main memory should be filled with account records after 3000 transactions, and the system will be in steady state. Next, start a global timer, and execute 1000 or 4000 transactions depending on the configuration. For each transaction, record the elapsed time, noting whether it is less or greater than 1 second. After completing all transactions, record the elapsed time on the global timer. Divide the total number of transactions by the total elapsed time to obtain the TPS rating. The TPS rating is valid if 95% or more of the transactions took 1 second or less.



All processes on the same node, local locks.

**Figure 6-10:** ET1 "local" configuration



Two nodes, distributed locks.

**Figure 6-11:** ET1 "remote" configuration

Because some processing (disk, network, and CPU) can occur in parallel, increasing the number of ATM threads that initiate transactions may increase the overall TPS rating, by allowing several transactions to execute simultaneously. However, if too many transactions attempt to execute simultaneously, the TPS rating may decrease as the system becomes CPU-, disk-, or network-bound. Also, as transactions wait for busy resources, the time per transaction increases, and the TPS rating may become invalid if too many transactions take longer than 1 second. For the local configuration, the highest rating of 8.06 TPS occurs with 2 threads, for an average transaction latency of 124 milliseconds. (Eppinger reported 11.3 TPS with 3 threads. I can match this rating using a smaller database of 15 megabytes. It may be that Eppinger used the smaller database.) For the remote configuration, the highest valid rating of 1.73 TPS occurs with 1 thread, for an average transaction latency of 578 milliseconds. (With 1 thread, the local configuration achieves a rating of 7.40 TPS, for an average transaction latency of 135 milliseconds.)

To understand an ET1 transaction (as shown in Figure 6-9), a line-by-line analysis may be helpful:

1. To begin a transaction, the ATM process obtains several TIDs with `TA_GetTids`. This cost is amortized over several transactions.
2. When the presentation server receives the RPC from the ATM process, it makes a `TS_Join` RPC to Camelot.
3. When the data server receives the RPC from the presentation server, it makes a `TS_Join` RPC to Camelot.
4. In steady state, the branch balance lock is cached by the data server in the remote configuration, so no RPC to the Lock Manager is needed.

5. The branch balance update causes pin and log requests to be appended to the shared memory queue. The page is cached in memory, so no requests to the External Memory Manager are needed.

6. In steady state, the teller balance lock is also cached by the data server in the remote configuration.

7. The teller balance update causes pin and log requests to be appended to the shared memory queue. The page is cached in memory, so no requests to the External Memory Manager are needed.

8. The account balance lock is not cached by the data server in the remote configuration, so it makes a `HS_Lock` RPC to the Lock Manager.

9. The account balance update causes pin and log requests to be appended to the shared memory queue. The account balance page is not cached in memory, so the kernel makes a `memory_object_data_write` request to send a dirty page to the External Memory Manager. In the remote configuration, Camelot responds with a `XD_GetShMemQueue` request to obtain log records. In both configurations, the kernel requests the account balance page with `memory_object_data_request`, and Camelot replies with `memory_object_data_provided`.

10. A lazy server-based transaction can be initiated without communicating with Camelot.

11. The history pointer lock is cached by the data server in the remote configuration, so no RPC to the Lock Manager is needed.

12. The history pointer update causes pin and log requests to be appended to the shared memory queue. The page is cached in memory, so no requests to the External Memory Manager are needed.

13. A lazy server-based transaction can be committed without communicating with Camelot.

14. The history record update causes pin and log requests to be appended to the shared memory queue. Just under 79 history records fit in a page, so the page is usually cached in memory. After every 78 or 79 transactions, a page fault occurs, and is satisfied as described for line 9 above.

15. The RPC reply from the data server to the presentation server is a single message.

16. The RPC reply from the presentation server to the ATM process is a single message.

17. To commit the transaction, the ATM process makes a `TA_End` RPC to Camelot. Camelot makes `ST_Vote` RPCs to both the presentation and data servers. Next, Camelot in the remote configuration obtains the shared memory queue from both servers via `XD_GetShMemQueue` RPCs; the presentation server's queue is empty. Finally, Camelot notifies the data server that the transaction committed via a `ST_Commit` message. (Camelot does not notify the presentation server since the presentation server did not make any updates.)

In the local configuration, the time for an ET1 transaction can be broken down as follows. The paging write in line 9 takes 111 milliseconds. The occasional paging write in line 14 adds 1.4 milliseconds. The non-paging writes in lines 5, 7, 12, and 14 add 5.6 milliseconds. The

RPCs in lines 2/15 and 3/16 add 5 milliseconds. The transaction overhead for an update transaction with two servers, including the RPCs mentioned in lines 1, 2, 3, and 17, is 15.5 milliseconds. This gives a total of 138.5 milliseconds, for an expected rating of only 7.2 TPS. Because there are multiple transactions running concurrently, some of the log and paging I/O overlap with CPU processing, allowing the higher throughput.

In the remote configuration, the analysis is as follows. The paging write in line 9 takes 385 milliseconds. The occasional paging write in line 14 adds 4.9 milliseconds. The `HS_Lock` request in line 8 adds 17.7 milliseconds. The non-paging writes in lines 5, 7, 12, and 14 add 9.6 milliseconds. The RPCs in lines 2/15 and 3/16 add 5 milliseconds. The transaction overhead for a single server update transaction is 103 milliseconds. The second server requires additional RPCs `TS_Join`, `ST_Vote`, and `XD_GetShMemQueue` which add 49.5 milliseconds to the transaction overhead. This gives a total of 574.7 milliseconds, for an expected rating of 1.74 TPS. This is extremely close to the measured rating.

In the remote configuration, higher TPS ratings can be achieved by increasing the number of threads initiating transactions. Unfortunately, the higher TPS ratings are not valid, because fewer than 95% of the transaction complete in 1 second or less. For example, with 2 threads, the remote configuration achieves 1.89 TPS, but only 81.6% of the transactions complete in less than 1 second. The majority of the elapsed time of an ET1 transaction is spent waiting for pages to be transferred over the network. When two or more transactions execute simultaneously, some of this waiting time can be spent doing useful work at the home node or the remote node. However, if a transaction that is already 574 milliseconds long has to wait an additional 385 milliseconds for the network while another transaction is transferring a page, it comes dangerously close to exceeding 1 second in total elapsed time. The percentage of transactions completing in less than 1 second drops precipitously as the number of threads increases. Only 4.5% complete in less than 1 second with 5 threads, although this configuration achieves 2.36 TPS.

The ET1 benchmark is not an ideal choice of application to demonstrate the benefits of data sharing. The ET1 benchmark was intentionally designed with a hot spot (the history pointer) and limited locality of reference: every transaction must access a page that is not cached in main memory. The limited locality of reference implies that the ET1 benchmark is primarily a measure of paging throughput. As will be shown in Chapter 7, paging throughput need not be significantly reduced by the data sharing model.

However, the hot spot can cause performance problems for the data sharing model. The ET1 benchmark uses remote access to a recoverable segment, but the segment is not actually shared between two servers. If the segment were shared between servers on two nodes, ET1 performance would be degraded by the cost of migrating the page containing the history pointer from node to node. To eliminate this degradation, the history file could be divided into sections, one for each node, with a separate history pointer for each section. Since the history sections are all part of a single recoverable segment, the history file is shared by all nodes, but updates occur in independent partitions, and there is no concurrent write sharing. The account database could be similarly partitioned to eliminate migration of account records from node to node, but the benefits of such a partitioning are negligible, given that a random account record is much more likely to be on paging disk than stored in main memory on another node.

## 6.5. Function Shipping vs. Data Sharing

In this section, the performance of function shipping is compared to the performance of data sharing through the use of simple tests written in the CPA language. Ideally, a data sharing test should not require any RPCs; however, the architecture of Camelot does not allow a process that uses recoverable storage to have any user interface but RPC. Thus, each iteration of the data sharing test contains an initial RPC to a server with recoverable storage, which then does the remainder of the work in the test:

```
repeat 20 times
  make one RPC to local server to do
    begin transaction
      repeat k times
        lock & read or write 32 bytes
    end transaction
  return from RPC
```

The function shipping test is structured in a similar manner, including the initial RPC. That is, the CPA interpreter makes an initial RPC to a local server which does the remainder of the work in the test:

```
repeat 20 times
  make one RPC to local server to do
    begin transaction
      repeat k times
        make one RPC to remote server to do
          lock & read or write 32 bytes
        return from RPC to remote server
    end transaction
  return from RPC to local server
```

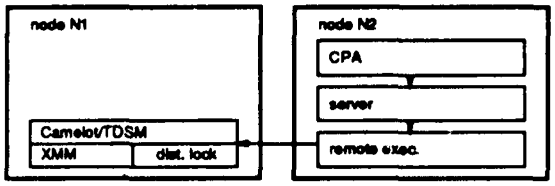


Figure 6-12: Data sharing configuration

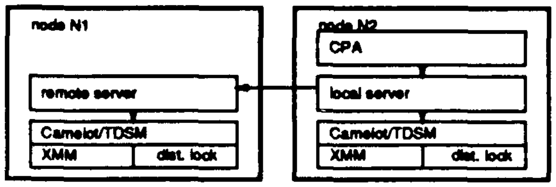


Figure 6-13: Function shipping configuration

The configurations for these two tests is shown in Figures 6-12 and 6-13. The test methodology is similar to that in Section 6.3: repeat the test many times to determine the median value of the average time per transaction. (Note that $t$, the number of transactions per test, is fixed at 20, and $x$, the size of the virtual memory region, is fixed at 32 bytes.) Both paging and non-paging versions of the test are run, and the number of iterations of the innermost loop $k$

varies from 1 to 2 to 10. (In the paging version of the tests, each iteration of the innermost loop accesses a 32-byte region on a new page. In the non-paging version, each iteration accesses the same region repeatedly.) The median measurements for reads and writes are reported in Table 6-8. Although each non-paging test accesses the same region *k* times, the results may also be interpreted as if the tests accessed *k* different regions that are all cached.

| Test Operation | *k*=1 non-pag. (ms.) | *k*=2 non-pag. (ms.) | *k*=10 non-pag. (ms.) | *k*=1 paging (ms.) | *k*=2 paging (ms.) | *k*=10 paging (ms.) |
|---|---|---|---|---|---|---|
| Data sharing reads | 75 | 75 | 77 | 186 | 384 | 1865 |
| Function shipping reads | 51 | 76 | 280 | 76 | 128 | 511 |
| Data sharing writes | 217 | 218 | 218 | 431 | 555 | 3365 |
| Function shipping writes | 106 | 131 | 346 | 135 | 144 | 771 |

The "non-pag." (non-paging) columns show the median times for a transaction that accesses the same region *k* times. The "paging" columns show the median times for a transaction that accesses *k* regions, each on a different page, that are not cached.

**Table 6-8:** Function shipping vs. data sharing

Inspection of Table 6-8 shows that, in this experimental environment, data sharing suffers in comparison to function shipping when only one region is accessed. The reason for the poorer performance of data sharing is that the overhead to commit the transaction is higher: Camelot on node N1 needs 4 RPCs to the server on node N2 (and must also transfer log data from node N2 to node N1) when committing a data sharing transaction, while Camelot on node N2 needs only 1 RPC to Camelot on node N1 to commit a function shipping transaction. A more optimized implementation of data sharing could eliminate the extra RPCs, bringing the overhead of a data sharing transaction closer to the overhead of a function shipping transaction.

Data sharing also suffers in comparison to function shipping when paging occurs. This is to be expected, since data sharing not only shares the cost of disk access incurred by function shipping, but also pays the additional penalty of transferring pages over the network. If the applications ran multiple transactions concurrently, and the network transfer time were closer to disk latencies, the effect on throughput would be less significant, since network transfer could overlap disk I/O.

Fortunately, Table 6-8 does have some good news for data sharing. When a transaction reads two or more regions that are cached, data sharing outperforms function shipping, because the additional RPCs needed by function shipping quickly outweigh the higher transaction overhead of data sharing. Update transactions have a higher overhead than read-only transactions, so a data sharing transaction needs to write more than two cached regions before it outperforms a function shipping transaction that writes the same number of regions, but Table 6-8 shows that the breakeven point is between 2 and 10 different regions in this environment.

## 6.6. Summary

The performance results in this chapter show that the system behaves as expected, given the poor effective bandwidth provided by Mach 2.5 on this hardware. Mach utilizes only 5% of the bandwidth of the 4 megabit/second token ring: a `memory_object_data_write` message which transfers 4096 bytes takes 185.6 milliseconds. This translates to a bandwidth of 22 kilobytes/second, or 176 kilobits/second. Chapter 7 develops a model of the system which allows its performance to be predicted in an environment with an effective bandwidth that is more reasonable.

Even without drawing on the performance model in Chapter 7 to understand the effects of more modern hardware and communication systems, TDSM apparently performs well with frequently-accessed, read-only data. A non-paging read operation is just as fast in remote configurations as local configurations, although a read-only transaction using remote access to a TDSM segment incurs an additional per-transaction overhead compared to local access. The additional overhead for remote access is directly attributable to network IPC time.

It is less clear from the results in this chapter that TDSM performs well with data that is updated. A 32-byte non-paging write operation takes more than twice as long in remote configurations as local configurations. TDSM also appears to suffer significantly when infrequently-accessed data must be paged over the network, or when frequently-updated data is paged from node to node. However, this disparity appears only because the effective network throughput is so small relative to disk and memory speeds. Chapter 7 shows how this disparity would be less significant if TDSM is used on modern hardware.

# Chapter 7
# Analysis

This chapter reports on a model for the performance results reported in Chapter 6. This model is used to project the performance of the system to different environments, and to project the effects of changes made in the algorithms used by the system.

First, parameters of the performance model are introduced, and results are projected for two alternative environments. Next, the model is used to demonstrate the effects of the cache hit ratio on throughput. The next section proposes changes to algorithms to improve performance and discusses effects of those changes. Finally, the weaknesses of the implemented system are exposed and discussed.

## 7.1. Performance Model

Several factors make it difficult to model the performance of the system precisely. CPUs, disks, and networks operate concurrently. Messages may contain "inline" data, or pointers to "out-of-line" data, which have different effects on network message transmission. Messages are split into packets for transmission over the network. Log data is formatted into blocks for disk writes. Thus, the performance model developed below only approximates the behavior of the system.

Extrapolating the performance of the system to different hardware and software environments is also problematic. Disks and networks may offer higher throughput that is precisely defined, but the latency of disk and network operations depends in part on the hardware controller, the interface it presents to the remainder of the hardware system, and the operating system's proficiency in accessing the controller. Similarly, while newer CPUs may promise 4- to 20-fold performance improvements for typical application code, operating systems generally do not realize the same benefit. System calls, interrupt handling, context switching, data copying, and many memory-intensive operations do not fully benefit from modern architectural innovations [Anderson et al 91]. However, as a developing system, many parts of Mach 2.5 were not optimized. The assumption underlying the extrapolated results reported in this chapter is that, even though operating system operations may not benefit as much as application code from improvements in system architecture, optimizing the operations can mitigate this effect. In any case, the results are meant to be interpreted qualitatively, not quantitatively, as a demonstration that TDSM is a useful technique for structuring applications, and that access to a page on disk over the network need not be significantly more expensive than access to a page on a local disk. Indeed, the fact that memory-intensive operations do not fully benefit from modern architectural innovations is a disadvantage for both forms of page access.

### 7.1.1. Parameters of the model

The performance model is based on the characteristics of the hardware (CPU, disk, and network) and operating system on which the system executes. Eight parameters describe the hardware and operating system:

- $c$, the time to execute one CPU instruction.
- $n_{f,l}$, the time for a null RPC (sending a message containing no data and awaiting a reply that contains no data) between two processes on the same node. This parameter is a function of the number of instructions that the operating system executes to transmit a message.
- $n_{v,l}$, the incremental time to include another byte in a message sent between processes on the same node.
- $n_{f,r}$, the time for a null RPC between two processes on different nodes. The value of this parameter depends on the operating system overhead (on both sending and receiving nodes), latency of the network interface (sender and receiver), latency of the network (the time it takes a bit to propagate from sender to receiver) and the network bandwidth (even a null RPC must transmit some control information).
- $n_{v,r}$, the incremental time to include another byte in a message sent between processes on different nodes.
- $d_f$, the time to initiate a disk operation (including operating system overhead and average rotational latency).
- $d_s$, the average disk seek time.
- $d_v$, the incremental time to read or write one byte from disk.

All times in the model are expressed in terms of these eight parameters. These parameters may be changed to extrapolate the performance of the system to a different platform (faster hardware, or a more efficient operating system).

The model is built on the assumption that only one transaction executes concurrently. Later, the analysis relaxes this assumption to demonstrate how higher throughput may be obtained.

### 7.1.2. Actual parameters

The actual performance measurements reported in Chapter 6 are used to determine the parameter values for the performance model. Table 7-1 summarizes the parameter values for the experimental environment, and for two hypothetical scenarios that are introduced subsequently.

The 2.5 MIPS speed of the IBM RT/PC CPU defines $c = .4$ microseconds. Analysis of the times for the `cpa_read_s` and `cpa_read_l` RPCs reveals that $n_{f,l}$ is 2.28 milliseconds, and

| Parameter | | Actual | Scen-ario 1 | Scen-ario 2 |
|---|---|---|---|---|
| time per CPU instruction | $c$ | .4 µs. | .1 µs. | .04 µs. |
| null RPC time (local) | $n_{f,l}$ | 2.28 ms. | .30 ms. | .12 ms. |
| RPC time per byte (local) | $n_{v,l}$ | .58 µs. | .05 µs. | .02 µs. |
| null RPC time (remote) | $n_{f,r}$ | 16 ms. | 2.0 ms. | .24 ms. |
| RPC time per byte (remote) | $n_{v,r}$ | 14 µs. | 2 µs. | .07 µs. |
| average rotational latency | $d_f$ | 8.3 ms. | 8.3 ms. | 5 ms. |
| average seek time | $d_s$ | 22 ms. | 22 ms. | 10 ms. |
| disk transfer time per byte | $d_v$ | 1 µs. | 1 µs. | .2 µs. |

**Table 7-1:** Performance model parameters

$n_{v,l}$ is .58 microseconds. Since local RPCs are performed entirely by the CPU, these values can be expressed as 5700*$c$ and 1.45*$c$, respectively. Again, analysis of `cpa_read_s` and `cpa_read_l` RPCs reveals that $n_{f,r}$ is 16 milliseconds, and $n_{v,r}$ is 14 microseconds. If these times were entirely CPU-bound, they could be expressed as 40000*$c$ and 35*$c$, respectively. Although experiments show that node-to-node RPCs use large amounts of CPU time, and CPU time makes the largest contribution to RPC latency in the actual experimental environment, it is not the sole component of RPC latency, and the model uses other terms to express these parameters. The measurements reported in Chapter 6 lead to disk parameters of $d_f$ = 8.3 milliseconds, $d_s$ = 22 milliseconds, and $d_v$ = 1 microsecond.

The operation costs measured in Chapter 6 can be modeled in terms of the eight parameters. This model is developed in Appendix A. Because the performance model does not explicitly address concurrency, several operation times in the model are larger than the actual measured times reported in Chapter 6. The operation times that are expressed in terms of the eight parameters are:

- Non-paging reads. The fixed cost per read is $R_{np,f}$ and the variable cost per byte is $R_{np,v}$.
- Non-paging writes. The fixed cost per write is $W_{np,f,l}$ when the writer is on the home node, and $W_{np,f,r}$ when the writer is on a different node. The variable cost per byte is $W_{np,v,l}$ when the writer is on the home node, and $W_{np,v,r}$ when the writer is on a different node.
- Paging reads. The time for a paging read is $R_{p,l}$ when the page is on the local disk, and $R_{p,r}$ when the page must be fetched over the network.
- Paging writes. The time for a paging write is $W_{p,l}$ when the writer is on the home node, and $W_{p,r}$ when the writer is on a different node.
- Read-only transaction overhead. The overhead for a read-only transaction is $T_{ro,l}$ for a server on the home node, and $T_{ro,r}$ for a server on a node other than the home node.

- Update transaction overhead. The overhead for an update transaction is $T_{u,l}$ for a server on the home node, and $T_{u,r}$ for a server on a different node.

Given the above variables for operation times, the performance of the ET1 benchmark can be modeled in two configurations: in the "local" configuration, all processes run on the same node (Figure 6-10), and in the "remote" configuration, Camelot runs on a different node than the data server and its client (Figure 6-11). The remote configuration is meant to illustrate the performance of data sharing, since the data server must obtain pages and transaction services over the network, although the RVM segment is not actually shared with another server. The local configuration serves as a baseline for comparison as a degenerate form of function shipping.

The latency of a single ET1 transaction is modeled by summing the appropriate operation times (note that the subscript $_l$ or $_r$ must be appended as appropriate):

- Transaction overhead. To the single-server update transaction overhead add 3 messages: $T_u + 3*n_f$
- Paging write operations. One paging write occurs every transaction. Another paging write occurs every 78 transactions. The sum is $1.02*W_p$.
- Non-paging write operations There are 4 non-paging writes each transaction: $4*(W_{np,f} + 32*W_{np,v})$.

Substituting the given parameter values and adding the terms together produces a time of 595 milliseconds for an ET1 transaction in the remote configuration, and 149 milliseconds for an ET1 transaction in the local configuration. These times are somewhat higher than the actual measured times from Chapter 6 of 578 and 135 milliseconds, respectively, for a single ET1 thread. Table 7-2 summarizes the component costs of an ET1 transaction using the actual parameters.

| | Local configuration | Remote configuration |
|---|---|---|
| ET1 transaction (ms.) | 149 | 595 |
| CPU time (ms.) | 21.0 | 21.0 |
| Paging disk (ms.) | 70.2 | 70.2 |
| Log disk (ms.) | 30.4 | 30.4 |
| Message transfer (ms.) | 5.21 | 126 |
| Message setup (ms.) | 21.8 | 348 |

"ET1 transaction" is the latency of a transaction in the model, assuming no concurrent execution of transactions. "CPU time" does not include message setup and transfer time that is reported separately.

**Table 7-2:** Modeled ET1 performance (actual parameters)

Inspection of Table 7-2 reveals that, in the local configuration, paging disk access makes the largest contribution to ET1 transaction latency. If CPU, paging disk, and log disk could all run at full speed concurrently, the local configuration could achieve at most 14.2 TPS. In practice, however, the CPU must synchronize with the disks, so the actual throughput is lower. If two paging disks were used and two ET1 transactions were run concurrently, the paging throughput would increase by a factor of two. In this case, CPU time (including message setup and transfer time) would limit ET1 transaction throughput in the local configuration, and the maximum throughput would be 20.8 TPS.

In the remote configuration, message setup is the largest component of ET1 transaction latency. If both CPUs, network, and disks could run at full speed concurrently, and half of the message setup time is allocated to each CPU, then the remote configuration could achieve at most 5.1 TPS. In practice, the CPUs must synchronize with each other, the network, and the disks, so the actual throughput is lower. If message setup and transfer operations were 7 times faster, the latency of an ET1 transaction in the remote configuration would become bound by the paging disk throughput.[9] In this case, the maximum throughput would be the same as the local configuration, 14.2 TPS.

The poor performance of the remote configuration on the ET1 transaction benchmark is clearly due to the high cost of network RPCs. The operating system used in this environment is not well-tuned for network RPCs. In the following subsections, we explore the qualitative effects of tuning the operating system by extrapolating the model and its parameters to different scenarios.

### 7.1.3. Scenario 1

In Scenario 1, the performance of an ET1 transaction is extrapolated to an environment with modern hardware and faster message passing. The CPU is 4 times faster and the operating system is streamlined for faster local and network RPCs.

The IBM RT/PC APC could easily be replaced with 10 MIPS CPU; in Scenario 1, $c = .1$ microseconds. The Mach 2.5 operating system used for the actual performance measurements apparently takes 5700 instructions to send a null message between two processes on the same machine, with 5.8 instructions per word (1.45 instructions per byte) to copy the data. Optimizing the operating system implementation for fast local messages (as has been done with Mach 3.0) could yield 3000 instructions per message and 2 instructions per word, $n_{f,l} = 3000*c$ and $n_{v,l} = .5*c$ for local messages. For network messages, Mach 2.5 takes 16 milliseconds for a null RPC, with 14 microseconds for each additional byte transferred. These times correspond to the time for the IBM RT/PC APC to execute 40,000 instructions and 35 instructions, respectively. Optimizing the implementation for fast network messages could yield a network null RPC time, $n_{f,r}$, of 2.0 milliseconds (the time to execute 20000 instructions on a 10 MIPS CPU). Transferring

[9]With Mach on the IBM RT/PC, network RPCs are effectively CPU-bound. If we assume that network setup and transfer times scale directly with CPU speed, then the CPU would have to be 7 times faster before network RPCs would be bound by the network bandwidth of 4 megabits/second.

data at the full 4 megabit/second bandwidth of the token ring means the time per byte $n_{v,r}$ is 2 microseconds (the time to execute 20 instructions). These parameters are summarized in Table 7-1. Substituting the Scenario 1 values for $n_{f,l}$, $n_{v,l}$, $n_{f,r}$, $n_{v,r}$, and $c$ produces a remote ET1 transaction time of 167 milliseconds, and a local ET1 transaction time of 109 milliseconds. (See Table 7-3.)

| | Local configuration | Remote configuration |
|---|---|---|
| ET1 transaction (ms.) | 109 | 167 |
| CPU time (ms.) | 5.26 | 5.26 |
| Paging disk (ms.) | 70.2 | 70.2 |
| Log disk (ms.) | 30.4 | 30.4 |
| Message transfer (ms.) | .45 | 18.0 |
| Message setup (ms.) | 2.86 | 43.5 |

"ET1 transaction" is the latency of a transaction in the model, assuming no concurrent execution of transactions. "CPU time" does not include message setup and transfer time that is reported separately.

**Table 7-3:** Modeled ET1 performance (Scenario 1)

Inspecting Table 7-3 shows that, for Scenario 1, paging disk access still makes the largest contribution to ET1 transaction latency in the local configuration. This means that the maximum achievable throughput is still 14.2 TPS. If the paging disk throughput were 3 times faster (e.g., by using 3 disks and running 3 transactions concurrently), ET1 throughput would be bound by log disk throughput; the maximum achievable throughput would be 32.9 TPS. Log disk throughput can be increased by using *group commit*; that is, delaying the commitment of some transactions so that one log force can commit several transactions simultaneously. Paging disk throughput would have to be 9 times faster, and log disk throughput would have to be 4 times faster, before ET1 throughput would be CPU-bound in the local configuration. If ET1 throughput were entirely CPU-bound, the maximum achievable throughput would be 117 TPS.

Paging disk access also is the largest component of ET1 transaction latency in the remote configuration. If the paging disk throughput were 2 times faster, ET1 throughput would be bound by message setup time; allocating half of this time to each CPU, the maximum achievable throughput would be 37.0 TPS. If the CPU, disks, and network interfaces were fast enough that ET1 throughput were bound by network transfer time, the maximum achievable throughput would be 55.6 TPS.

The disparity between local and remote configurations with respect to the time for a non-paging write operation decreases in Scenario 1 relative to the actual measurements reported in Chapter 6. In Chapter 6, a 32-byte non-paging write takes 2.0 times as long in the remote configuration as in the local configuration. In Scenario 1, a 32-byte non-paging write takes 1.5

times as long in the remote configuration as in the local configuration. The decreased disparity results from the relative improvements to local and remote message times. From Table 7-2 to Table 7-3, the message setup time $n_{f,r}$ for the remote configuration improves by a factor of 8.0 while the message setup time $n_{f,l}$ for the local configuration improves by a factor of 7.6.

Scenario 1 exhibits a much smaller penalty for running an ET1 transaction in the remote configuration instead of the local configuration, and is achievable using readily available hardware with a finely-tuned operating system.

### 7.1.4. Scenario 2

In Scenario 2, the performance of the system is extrapolated to an environment with leading-edge technology. The CPU, disk, and network are all faster.

Replacing the 10 MIPS processor of Scenario 1 with a 25 MIPS processor reduces $c$ to .04 microseconds. With this processor, I assume a network where page-size messages may be sent in a single packet, and the network RPC setup time is very close to the local RPC setup time. This is the type of network that might be used internally in a loosely-coupled multiprocessor. The fixed cost per message $n_{f,r}$ is .35 milliseconds. With an available bandwidth of 14 megabytes/second, the variable cost per byte $n_{v,r}$ is .07 microseconds. Since pages do not need to be fragmented when sent in an RPC, the Scenario 2 formula for paging read message time in the remote configuration, $R_{p,r}$ matches the formula for the local configuration. The Scenario 2 formula for paging write message time in the remote configuration, $W_{p,r}$, includes only one more RPC message (for obtaining a distributed lock) than in the local configuration. Using aggressive disk technology, rotational latency is reduced to $d_r$ = 10 milliseconds, seek time is reduced to $d_s$ = 10 milliseconds, and the transfer rate is reduced to $d_v$ = .2 microseconds per byte. These parameters are summarized in Table 7-1. With these parameters and changes to the model, an ET1 transaction takes 54 milliseconds in the remote configuration, and 51 milliseconds in the local configuration. (See Table 7-4.)

For Scenario 2, Table 7-4 reveals that ET1 throughput in both local and remote configurations is bound first by the paging disk throughput (with a maximum achievable throughput of 31.0 TPS), and then by the log disk throughput. If the paging disk throughput was 3 times faster, then the maximum achievable ET1 throughput would be 66.7 TPS. The difference in message time between the two configurations is only 7% of the total transaction time. Message time can be further reduced by combining messages sent between nodes. As suggested previously, paging disk throughput can be increased by using multiple disks, and log disk throughput can be increased by using group commit. These changes will decrease the contribution of paging writes $W_p$ and update transaction overhead $T_u$ (respectively) to total ET1 transaction latency. Paging disk throughput would have to improve by a factor of 3 before log disk throughput would become the bottleneck. If log disk throughput improved by a factor of 5, paging disk throughput would have to improve by a factor of 6 before ET1 throughput becomes

| | Local configuration | Remote configuration |
|---|---|---|
| ET1 transaction (ms.) | 50.7 | 54.4 |
| CPU time (ms.) | 2.10 | 2.10 |
| Paging disk (ms.) | 32.3 | 32.3 |
| Log disk (ms.) | 15.0 | 15.0 |
| Message transfer (ms.) | .18 | .53 |
| Message setup (ms.) | 1.14 | 4.40 |

"ET1 transaction" is the latency of a transaction in the model, assuming no concurrent execution of transactions. "CPU time" does not include message setup and transfer time that is reported separately.

**Table 7-4:** Modeled ET1 performance (Scenario 2)

bound by RPC time in the remote configuration, and a factor of 10 before ET1 throughput becomes limited by CPU time in the local configuration.

Scenario 2 incurs a small (less than 10 percent) penalty for running an ET1 transaction in the remote configuration over the local configuration. In this environment, locality of reference becomes less important in choosing the data sharing approach over function shipping.

## 7.1.5. Conclusion

The qualitative predictions of the model are significant. The model demonstrates that TDSM is a reasonable alternative to function shipping on an appropriate hardware/software platform, even for an application such as ET1 which has little or no locality of reference. Indeed, when message passing becomes reasonably efficient, remote access becomes feasible. However, the numbers predicted by the performance model should not be taken literally, since the model is a rough approximation to the behavior of distributed systems.

The remote configuration is an unusual environment in which to run the ET1 benchmark, but it is useful for making the point above. In a more realistic environment, the data for the benchmark would be partitioned across several home nodes. A transaction router would make its best attempt to route incoming transactions to the home node of the data referenced by the transaction. Thus, most transactions would run as in the experimental "local configuration," with the "remote configuration" reserved for those transactions which the router is unable to direct to the correct home node.

## 7.2. Throughput vs. Cache Hit Ratio

As in any data sharing system, the performance of TDSM improves as the cache hit ratio increases. This section provides a back-of-the-envelope illustration of this maxim.

The performance model discussed previously is used to determine the latency of a test transaction. (Here, exactly five write operations are grouped to form a test transaction.) The latency is broken down into components of CPU (including message setup), network transfer, paging disk, and log disk. When multiple transactions execute concurrently, these components are allowed to overlap (i.e., assume no scheduling delays). If we also assume that paging disk and log disk throughput can be arbitrarily increased by running multiple disks in parallel, then test transaction throughput is limited by either the available CPU time, or the network bandwidth.

The test transactions execute in an environment with $N$ using nodes that share a single network. When $N > 1$, each node acts as both a home node and a using node. A transaction on node $i$ accesses data from node $i+1$ (modulo $N$); thus, the load is evenly distributed among the nodes. When $N = 1$, a second node acts as the home node for the sole using node. Each transaction writes five 32-byte regions of RVM. Since each using node has a unique home node, there is no contention between nodes for RVM regions. The results reported here also assume there is no contention between transactions within a given node for RVM regions.

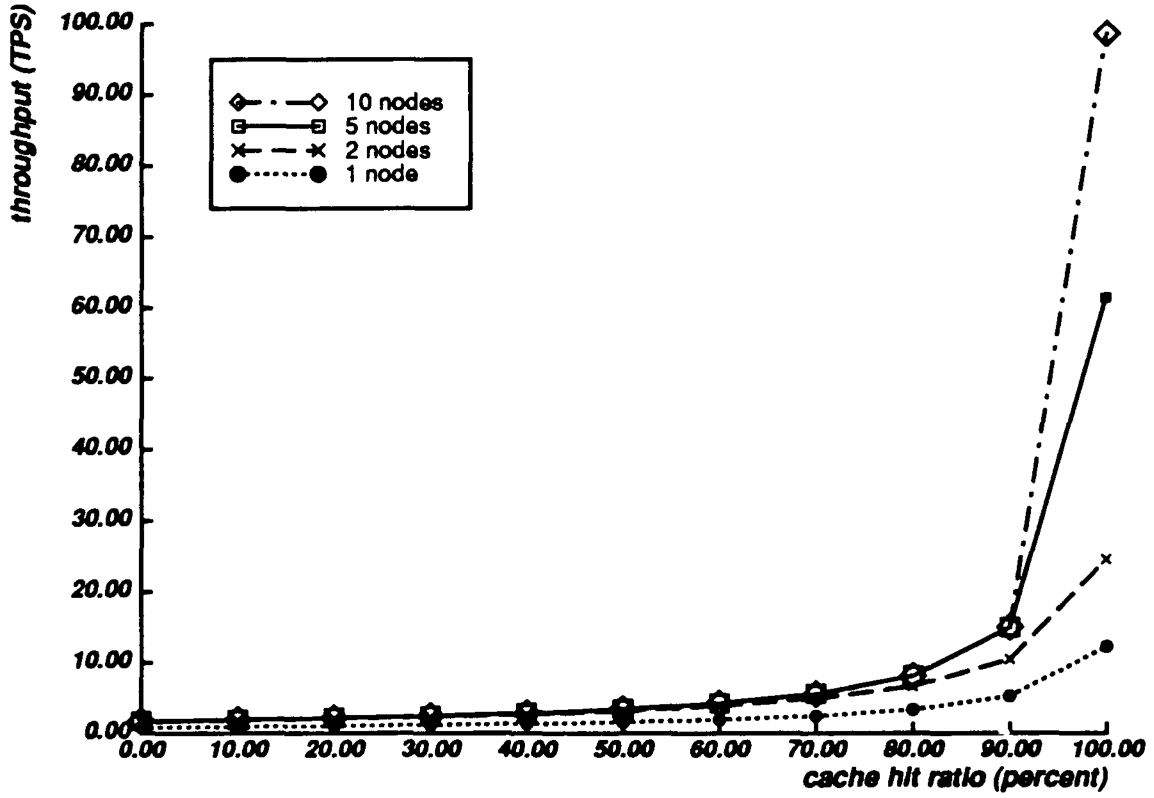


**Figure 7-1:** Throughput vs. cache hit ratio, actual parameters

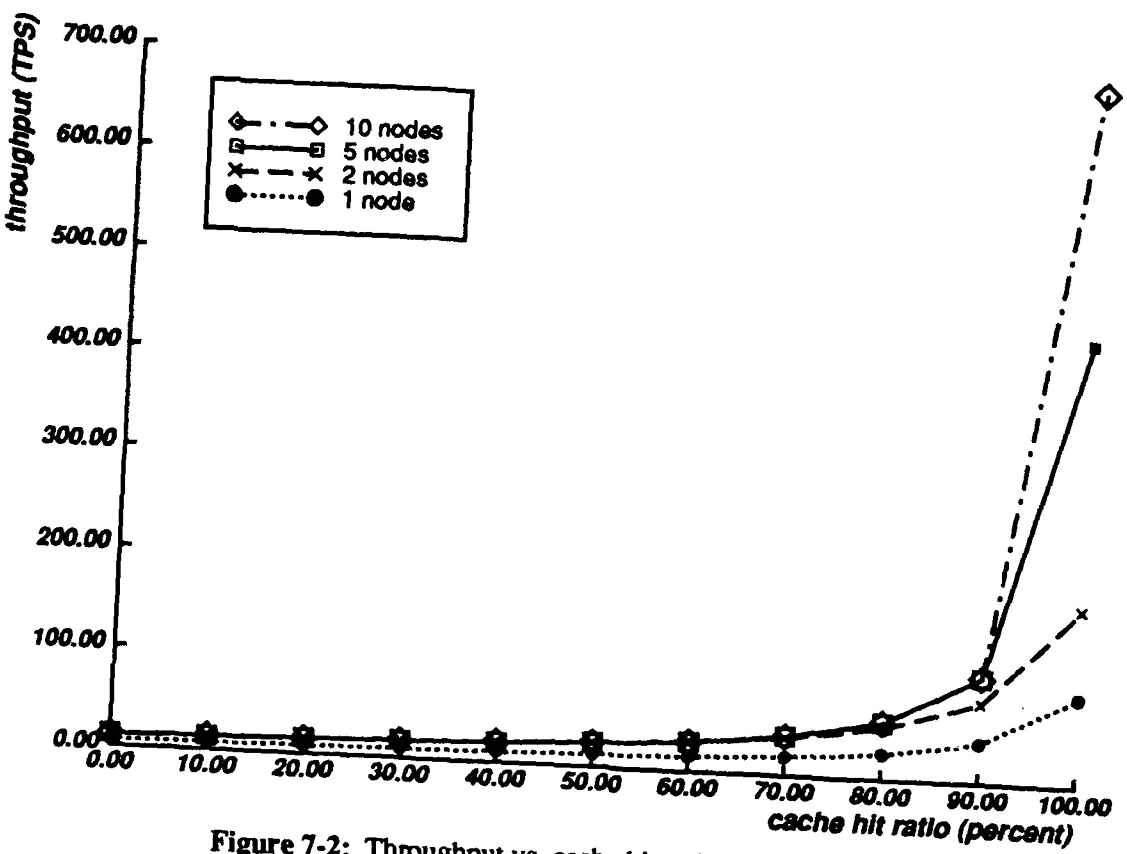


Figure 7-2: Throughput vs. cache hit ratio, Scenario 1

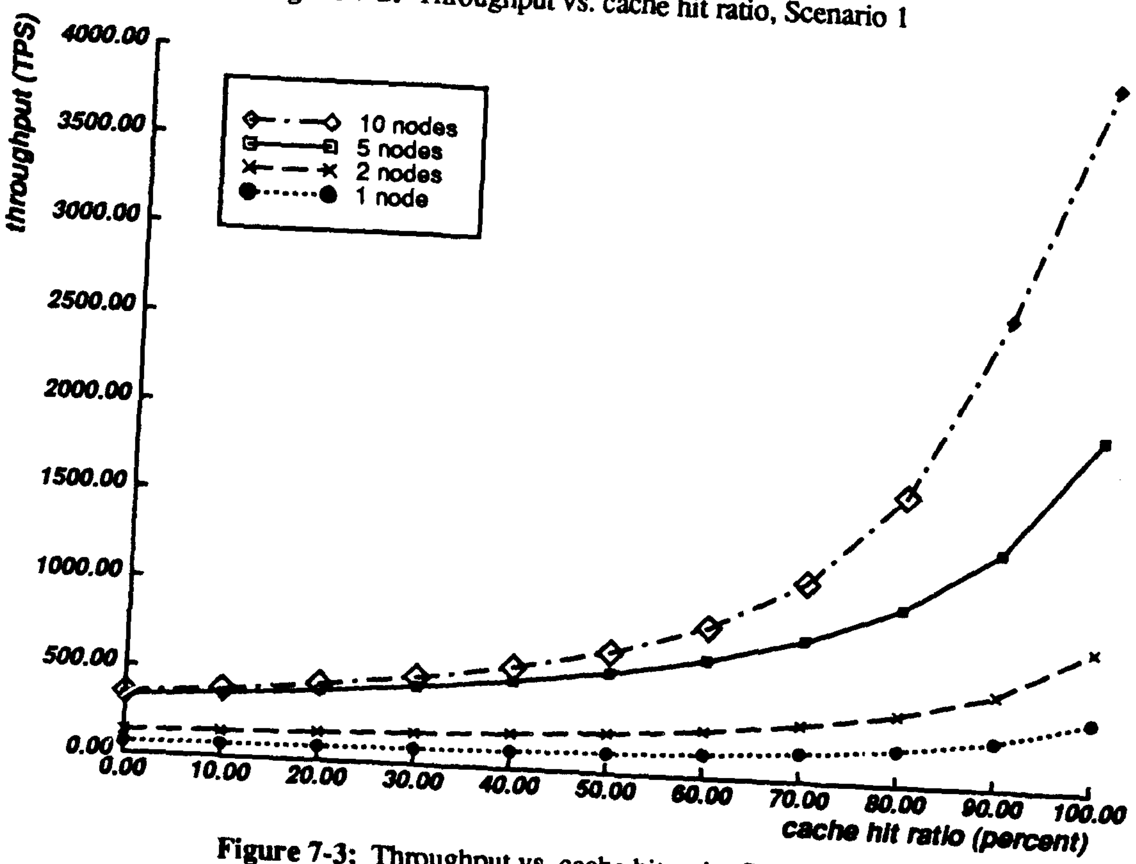


Figure 7-3: Throughput vs. cache hit ratio, Scenario 2

Each write access may find a region cached in physical memory, or not cached (in which case a dirty page is written back to the home node, and the requested page fetched from the home node). The cache hit ratio is the ratio of the number of accesses that find a region cached to the total number of accesses. When the cache hit ratio is small and there are multiple nodes, throughput may be limited by the network bandwidth, since all nodes use a common network. Otherwise, throughput depends on the total CPU time available.

Modeled throughput figures are reported in Figures 7-1, 7-2, and 7-3 for the actual parameters and two scenarios. The throughput value is the total number of test transactions that can be executed by all nodes in one second. Individual data points are marked with large symbols when throughput is limited by network bandwidth, and with small symbols when throughput is limited by CPU capacity.

All three figures show that throughput is limited by CPU capacity for one or two nodes, since one or two nodes do not generate enough traffic to overwhelm the network. The figures also show that throughput is roughly proportional to the number of nodes when the cache hit ratio is 100 percent.

For the actual parameters and Scenario 1, Figures 7-1 and 7-2 indicate that, until the cache hit ratio reaches 100 percent, throughput for 5 nodes is limited by network bandwidth, and adding additional nodes does not increase throughput. For 10 nodes, throughput is always limited by network bandwidth: even with a 100 percent cache hit rate and no network paging, the network messages needed to commit transactions use all the available bandwidth.

Figure 7-3 draws a different picture for Scenario 2. Throughput is limited by network bandwidth only for 10 nodes with a cache hit ratio less than 90 percent. CPU capacity limits throughput for 1, 2, or 5 nodes independent of the cache hit ratio, and for 10 nodes with a cache hit ratio of at least 90 percent.

When the cache hit ratio is low, paging activity of only a few nodes can saturate the network. A higher cache hit rate and higher network bandwidth is more suitable for TDSM. Even with lower network bandwidth, TDSM provides reasonable throughput given a high cache hit ratio. (The overhead of transaction commit for TDSM could be reduced to a single RPC per node in a more sophisticated implementation.)

## 7.3. Performance Improvements

Two tracks are available for improving the performance of TDSM. This section models improvements to the performance of TDSM through optimizing the architecture presented in Chapter 4. Improving the performance by substituting a different architecture is also discussed, but not modeled.

### 7.3.1. Optimizations

Since message time is a prominent component of several TDSM operations, an obvious technique to improve TDSM performance is to reduce the number of messages sent between nodes.

Tam has suggested combining concurrency control and coherency control messages [Tam 91]. In his 2PL* algorithm, when a node requests a read (write) lock on an object, a read-only (writable) copy of the object is delivered with the lock. The 2PL* algorithm reduces the cost of paging operations in the model. It can be implemented in the TDSM system described in this thesis as follows. Instead of allowing a server on a using node to contact the Lock Manager directly, install the Remote Execution Manager as a surrogate for the Lock Manager. When the Remote Execution Manager receives a lock request from a server, it sends a special request to the External Memory Manager on the home node. The External Memory Manager makes a request to the real Lock Manager on behalf of the server. When the Lock Manager replies, the External Memory Manager merges the reply with a copy of the page corresponding to the lock, and sends the merged message back to the Remote Execution Manager. The Remote Execution Manager extracts the lock reply (which it forwards to the server) and the page (which it forwards to the kernel).

In Tam's analysis, the combined coherency/concurrency control algorithm improves throughput by a factor of two over independent coherency and concurrency control. However, Tam assumes that the latency of a coherency control message is only twice the latency of a concurrency control message, and Tam's analysis does not include the cost of accessing the paging disk. For the scenarios presented in this chapter, the ratio of coherency control message latency to concurrency control message latency is larger (10.5 for the actual measurements, 11.0 in Scenario 1, and 2.8 in Scenario 2). Ignoring the extra intra-node communication, combining coherency and concurrency control can be modeled by subtracting the cost of a separate message to obtain a distributed lock, $n_{f_r}$, from the formula for a paging write operation $W_{p,r}$ in the remote configuration. The effects of this change on ET1 throughput in the remote configuration are given in Table 7-5.

Another possibility f‥ reducing the number of messages is to eliminate explicit fetches of the shared memory queue log buffer by the home node. In the TDSM system design, when the home node receives a pageout message for a given recoverable segment, it makes an RPC to

| | Actual | Scenario 1 | Scenario 2 |
|---|---|---|---|
| Original ET1 transaction latency (ms.) | 595 | 167 | 54.4 |
| 2PL* ET1 transaction latency (ms.) | 579 | 165 | 54.1 |

The "original" row is repeated from Tables 7-2, 7-3, and 7-4 for the remote configuration. The "2PL*" row reflects reduced latencies from combining concurrency and coherency control messages.

**Table 7-5:** Effects of combining concurrency and coherency control

obtain the log buffer to every node using a the segment. (A using node may send a pageout message because it is trying to free up physical memory, or because it was requested to do so by the External Memory Manager as part of the coherency control algorithm.) This RPC can be eliminated by sending the log buffer along with every pageout message. To do this, on each using node install the Remote Execution Manager in the path between the kernel and the home node External Memory Manager. When the Remote Execution Manager receives a pageout message from its local kernel, it appends the log buffer and forwards the message to the home node External Memory Manager. The External Memory Manager separates the log buffer from the memory page and forwards both to the Disk Manager.

The home node also requests the shared memory queue log buffer for a given server when any transaction that has executed in the server attempts to commit. This RPC can be eliminated by sending the log buffer along with the server's vote. Finally, the home node scans the log buffer for all servers whenever any transaction aborts. With some extra communication within Camelot, inter-node RPCs to obtain the log buffer can be restricted to only those servers that were actually involved in the transaction.

These changes to eliminate explicit shared memory queue fetches can be modeled by subtracting the cost of an RPC, $n_{f,r}$, from the remote configuration formulas for a paging write operation $W_{p,r}$ and update transaction overhead $T_{u,r}$. The effects of this "log optimization" in conjunction with the 2PL* speedup are shown in Table 7-6.

With the "log optimization," both the modified page and log records describing modifications to that page are sent together in a message to the home node. The size of this message can be reduced at the cost of some additional processing by eliminating either the modified page or the log records. If the modified page is eliminated, the home node can reconstruct it by scanning the log records and applying the modifications described by the log records to an older version of the page. If the log records are eliminated, the home node can reconstruct them by comparing the modified page to an older version of the page. The drawback of either scheme is that the home node must maintain a correct "older version" of the page. To avoid "double-paging," the home node must keep this older version on its paging disk. Thus,

| | Actual | Scenario 1 | Scenario 2 |
|---|---|---|---|
| Original ET1 transaction latency (ms.) | 595 | 167 | 54.4 |
| 2PL* + log opt. ET1 transaction latency (ms.) | 547 | 161 | 53.4 |

The "original" row is repeated from Tables 7-2, 7-3, and 7-4 for the remote configuration. The "2PL* + log opt." row reflects reduced latencies from eliminating explicit shared memory queue fetches, and combining concurrency and coherency control messages.

**Table 7-6:** Effects of "log optimization"

the savings from not transferring the log records or modified page over the network are offset by the expense of an additional disk read and additional CPU processing at the home node.

If the "log optimization" technique is further optimized by eliminating the network transfer of the modified page, the home node can reconstruct the modified page by reading an older version from the paging disk, and applying the modifications in the log records. This can be modeled in the remote configuration as follows, assuming that only a single 32-byte region is modified on the page. Subtract the page transfer time, $4096{*}n_{v,r}$, from the cost of a paging write operation $W_{p,r}$. Add the cost of transferring a 32-byte log record, $32{*}n_{v,r}$, the CPU time for repeating a 32-byte modification, $W_{np,f,l} + 32{*}W_{np,v,l}$, and the time to read an older version of the page from disk, $d_s + d_f + 4096{*}d_v$. Table 7-7 shows that this optimization is beneficial when network messages are expensive (as with the actual parameters), but in Scenarios 1 and 2, the cost of reading the old page from disk and extra CPU processing outweigh any savings in network transfer time.

| | Actual | Scenario 1 | Scenario 2 |
|---|---|---|---|
| 2PL* + log opt. ET1 transaction latency (ms.) | 547 | 161 | 53.4 |
| Omit page ET1 transaction latency (ms.) | 525 | 188 | 69.4 |

The "2PL* + log opt." row is repeated from Table 7-6. The "Omit page" row includes the same optimizations with a further modification to eliminate the network writeback of a modified page via reconstruction at the home node from log records and an older version of the page.

**Table 7-7:** Effects of omitting writeback of modified page

In the present implementation, whenever a page migrates from node *N1* to node *N2*, log records from node *N1* are forced to stable storage, and the page is written to the paging disk. The write to paging disk is not strictly necessary. It occurs, in part, because only the External Memory Manager knows that a migration is in progress. To the rest of Camelot, the pageout message from node *N1* looks like a pageout due to housecleaning of inactive pages by the kernel. If the page is inactive, writing it to disk is a good idea, since the page will not be modified again soon. But writing the page to disk during a node-to-node migration is not as beneficial (although it may help limit recovery times) since the page is still active and will be modified again soon. With a closer coupling of the External Memory Manager to the rest of Camelot, this paging disk write during migration can be eliminated. (The effects of this change cannot be expressed in the ET1 model because an ET1 transaction does not include any node-to-node page migration.)

Another potential opportunity for improving performance within the architecture arises in the coherency control algorithm. A server written for Camelot is structured so that Camelot knows the exact boundaries of a region being modified in recoverable virtual memory. With some changes to the locking interface, Camelot could also know the exact boundaries of regions being read in recoverable virtual memory. Given this information, Camelot could use a granularity other than the system page size for coherency control. That is, instead of transferring whole pages from node to node, and ensuring that whole pages are coherent, Camelot could transfer objects of a different size, and ensure coherency for objects of a different size.

The performance effects of such a change are heavily dependent on the recoverable virtual memory reference pattern of each server. If Camelot uses a granularity smaller than a page, performance may improve because Camelot sends less data over the network when it fetches an object. Unfortunately, if the server references several objects residing on the same page, performance may decrease, because Camelot will have to send more messages to obtain the same amount of data. However, the smaller granularity may improve performance when two servers on different nodes update two distinct objects on the same page: since Camelot is maintaining the coherency of smaller objects, neither server will have to wait for the other server to finish updating the page. If Camelot uses a granularity larger than a page because a server references an object larger than a page, performance may improve because Camelot may be able to transfer the object from node to node with fewer messages. However, if Camelot uses a larger granularity, performance may decrease because Camelot may transfer more data than is needed by the server.

Because the performance effects of using different granularities depends so strongly on the server characteristics, and the actual algorithm for maintaining coherency with varying granularities is complicated, these effects are not expressed in the model. Physical locking (as in the IBM 801 architecture [Chang and Mergen 88]) could make smaller granularities more tractable.

### 7.3.2. Architectural changes

Instead of attempting to improve the performance of TDSM by optimizing the architecture of Chapter 4, let us consider the more radical approach of designing a new architecture. The home node concept causes several common operations to include a network component: transactions are committed over the network, and pages are written to non-volatile storage over the network. An alternative architecture could eliminate the network component of these operations if each node uses its local disk as non-volatile storage for data pages, and each node uses a local log service to commit transactions locally. As long as pages are not migrated from node to node, all transactions in this hypothetical architecture run as "local" transactions.

There are two difficult problems to solve in this hypothetical architecture. The first is how to locate a given page. With a home node, there is no question of how to locate a given page, since the home node always has current information. With no home node, page location information becomes distributed, and a page location algorithm is necessary.

The second, more difficult, problem is how to perform recovery after a transaction aborts or after a crash. With a home node, all relevant log records are easily located, and recovery is straightforward. With no home node, log records become distributed, and an algorithm is needed to locate them.

To better understand these problems, consider an example. Suppose transaction *T1* on node *N1* modifies a region on a given page. Before *T1* commits, the page migrates to node *N2*, where transaction *T2* modifies a second, distinct region. Now, if transaction *T1* aborts, either log records must migrate from node *N1* to node *N2*, or the page must migrate from node *N2* to *N1*, so that transaction *T1*'s modification can be undone. If node *N1* crashes before the log records migrate to node *N2*, node *N2* is stuck with a copy of the page that is inconsistent. If node *N2* crashes after the page is migrated back to node *N1*, node *N1* is stuck with a copy of the page that is inconsistent.

Now suppose both nodes *N1* and *N2* crash. At same time, a third node *N3* must determine the outcomes of transactions *T1* and *T2* so that it may locate (or generate) a consistent copy of the page, yet the information node *N3* needs may not be available. (The architecture described in Chapter 4 does not have this problem exactly: it can survive the crashes of nodes *N1* and *N2*, as long as neither node is the home node.)

One option for solving the page location and recovery problems is to use broadcast. A page can be located by broadcasting a request message to all nodes. When a page migrates from node to node, relevant log records must migrate along with it. When a transaction commits or aborts, a broadcast message notifies all nodes which may need to force log records to disk, or perform abort actions as appropriate. To recover after a crash, a node broadcasts a request for all relevant log records. The problem with this broadcast approach is that it suffers performance problems of its own. To ensure correctness, many broadcast messages will have to be reliable broadcasts. Reliable broadcast essentially forces the transaction commit protocol down into the communications layer, making reliable broadcast relatively expensive.

A second option for solving the page location and recovery problems is to use write-through techniques. When a node modifies a page, the modification is "written-through" the node's non-volatile cache by sending a notification to all interested nodes. Transaction outcomes are also written-through to all interested nodes. A page is easily located by contacting one of the interested nodes. Recovery is straightforward, because each of the interested nodes has all of the information needed. The problem with this write-through approach is the tradeoff between performance and availability. To improve performance, the frequency of write-throughs and the number of interested nodes must be restricted. For best availability, the set of interested nodes should be the entire distributed system, and write-throughs must be unrestricted.

Architectural changes are also discussed in the next section where certain deficiencies of the system are addressed.

## 7.4. Deficiencies of the System

Several deficiencies of the TDSM design were brought up in Chapter 4. Some additional deficiencies are omissions in the implementation. These deficiencies and possible solutions are addressed below.

### 7.4.1. Recovery

Server recovery is not fully implemented. If a server on node *N1* fails but node *N1* is still accessible, recovery is performed correctly as each transaction that was executing in the server aborts. When a transaction aborts, the Recovery Manager directs undo requests to a server on another using node *N2*. As the server processes the undo requests, pages from the failed server will migrate from node *N1* to node *N2*. No redo operations are necessary, since modifications made by committed transactions will be present on the migrated page.

But if a using node fails, the home node does not recover pages that were in use by the failed node, and these pages are unavailable for further access. The problem is that Camelot can execute its segment recovery algorithm only as the first server using a given segment is started, or after the last server using a given segment terminates. The segment recovery algorithm scans the log backwards. As it encounters modifications made by committed transactions on a given page, it generates redo requests, until it finds a record of the page being written to paging disk. Then, as it encounters modifications made by aborted transactions, it generates undo requests until it reaches the last checkpoint. These undo and redo requests are then applied to the page on paging disk.

To solve the problem of pages from a failed node being unavailable, a separate server recovery algorithm is needed. The server recovery algorithm is similar to the segment recovery algorithm, except that it only processes modifications from servers on the failed node. The server recovery algorithm must work in conjunction with the Lock Manager to abort any transactions in progress on the failed server, and release locks cached by the server.

Implementing this server recovery algorithm requires some additional interfaces between the Disk Manager, Lock Manager, and Recovery Manager to identify transactions in failed servers and abort them, releasing their cached locks. Since the server recovery algorithm can be adapted from the existing segment recovery algorithm, it could be developed with as little as one man-month of effort.

This deficiency affects only the recovery algorithm, and rectifying the problem will not affect the performance measurements reported in Chapter 6, since these measurements include only forward processing, not recovery.

## 7.4.2. Configurability

In the Camelot/TDSM architecture, using nodes have relatively little autonomy. The configuration of a recoverable virtual memory segment is defined on its home node; every server using the segment is defined on the home node, even if the server runs on a different node. A server may be started or killed only from the home node. To allow using nodes more autonomy, the server configuration could be distributed, with the configuration for each server residing on the node where the server runs. The segment configuration remains on the home node.

The only real difficulty with this approach arises from the way Camelot stores configuration information. Since Camelot stores configuration information about servers and recoverable virtual memory in recoverable virtual memory, storing server configuration on each using node will require a full-fledged Camelot to run on each using node; each using node must have its own non-volatile and stable storage for Camelot to use. When a using node starts a server configured to use a remote recoverable virtual memory segment, the using node contacts Camelot on the home node, and everything continues as in the original Camelot/TDSM design.

## 7.4.3. Security

Closely related to the node autonomy deficiency addressed in the previous subsection is a security deficiency. The Remote Execution Manager on each using node processes an RPC from the home node in which the home node may request an arbitrary program to be executed. To guard against malicious programs, each using node should be able to control the set of programs that may be executed. If the server configuration is stored on each using node (as described above) then this control can be achieved by changing the interface between the Remote Execution Manager and the home node. Instead of providing the name of the program to be executed, the home node should present a server ordinal to the Remote Execution Manager. The Remote Execution Manager can consult the server configuration information on its own node to determine the name of the program to execute.

Another security problem is endemic to the data sharing paradigm. While function shipping allows arbitrary operations to be restricted to particular callers, data sharing cannot control operations other than **read** and **write**. The current implementation of Camelot/TDSM does not provide even this restriction: once a server meets the access controls for a given recoverable segment, it has read and write access to the entire segment. As a step in the right direction, Camelot could define access lists for individual pages, so that each server has read, write, or no access to each page of a recoverable segment.

Other security features that bear consideration include encrypting messages sent over the network (including data pages), and forcing Camelot and the Remote Execution Manager to authenticate each other. These features are beyond the scope of the security guarantees envisioned by the original designers of Camelot.

### 7.4.4. Availability

The primary drawback to the Camelot/TDSM design is that failure of the home node effectively causes all servers on using nodes to be unavailable. When the home node fails, servers cannot commit or abort transactions, or access any pages except those pages already cached in main memory.

An obvious approach to solving this availability problem is to harden the single point of failure. The physical components of the home node can be duplexed (CPU, memory, controllers, disks, even the network) to reduce the probability of failure. Alternatively, the home node could be replicated over the network, perhaps forwarding all log records, page stores, and other necessary information to a "hot standby."

A variant of the hardening approach is to harden only the log service. Daniels demonstrated the viability of distributed, replicated log service [Daniels 88]. Using this log service as a common log for all nodes enables each using node to commit or abort a transaction without contacting the home node. Given a checkpoint and the log, any node can reconstruct a consistent state for paging store, so replication of checkpoints ensures availability after crashes. The trick is to achieve consistent, replicated checkpoints without halting forward processing.

A second approach is that outlined in Section 7.3.2: replicate pages and log records by distributing them among using nodes, and rely on the redundancy of using nodes to improve availability. When a region of shared recoverable virtual memory is modified, a log record describing the modification is written to the local log, and also forwarded to another node. Rahm has studied this technique in an environment where disks are shared by all nodes [Rahm 89], but much of the study is still relevant to a general distributed system. Each node can commit or abort a transaction on its own, relying on the replication of log records to notify other nodes of transaction outcomes.

Non-volatile storage for pages is also replicated. Tam has investigated a scheme which allows the ownership of pages to migrate, while still allowing easy location of a page for recovery [Tam 91]. Thus, this second approach improves availability by providing a backup site (or sites) for pages and log records, at the cost of additional communication. Replication of log records can be done after a transaction commits locally, so as to avoid increasing transaction latency. Replication of pages may add to transaction latency.

### 7.4.5. Incremental Growth

Closely related to the availability problem of Camelot/TDSM is scalability: the home node is both a single point of failure, and a potential performance bottleneck. The load on the home node and the required communications bandwidth increases with the number of transactions; the number of transactions is expected to increase as the number of using nodes increases.

The hardening approach outlined above doesn't address the incremental growth problem, since the performance of the hardened service still must grow with the number of transactions. However, if Camelot/TDSM were extended to allow each server access to multiple segments, and the database can be divided among multiple segments, then incremental growth in both storage capacity and processing power could be achieved by adding additional home nodes.

The second approach of replicating pages and log records in a distributed manner does offer a path to incremental growth of processing power and storage capacity. Each node uses its local disk to store a subset of the database, and the transaction load is distributed among the nodes. The number of messages needed to replicate a page or log record depends on the number of replicas, not on the total number of nodes in the system. As long as access to data is evenly distributed among the nodes, increasing the number of nodes does not increase the load on any one node.

## 7.5. Summary

The performance model used in this chapter demonstrates the practicality of TDSM given an appropriate (and reasonable) hardware and software environment. Indeed, the model even predicts that low locality of reference is not as significant a factor favoring function shipping when a high-speed network is available. Of course, data sharing achieves higher throughput as the cache hit ratio increases. The model shows that, while the implementation is lacking some possible optimizations to the TDSM architecture, these optimizations do not have a significant effect on the system's performance.

The TDSM architecture described in this dissertation has shortcomings in the areas of recovery, configurability, and security that could be rectified easily. More serious deficiencies are poor availability and incremental growth. Alternative architectures that replicate and distribute the functionality of the home node can correct these deficiencies, but may have a negative impact on performance.

# Chapter 8
# Conclusions

## 8.1. Motivation

Transactional distributed shared memory (TDSM) aids in building distributed programs that meet many application requirements, including:

- Ease of programming. Distributed shared memory extends the familiar notion of shared memory to a distributed system. The benefits of transactions can be obtained by adding a few statements to a well-designed program.
- Incremental growth. A sophisticated TDSM system allows easy expansion of processing and storage capacity by adding more nodes. (Note that the prototype implementation described in this dissertation has limits to the growth of these capacities.)
- Multiple access to data. Distributed shared memory enables shared data to be accessed in the same manner as local data. Transactions ensure that concurrent accesses do not result in inconsistent states.
- Data integrity. Transactions guarantee permanence and prevent inconsistencies arising from failures or concurrent updates.
- Performance. Distributed shared memory offers automatic partitioning for high locality of reference, and automatic replication for read-only performance.

TDSM offers an interesting alternative to the traditional technique of extending transactions to a distributed system via *transactional RPC*. Remote procedure call (RPC) is attractive because it uses the familiar paradigm of procedure call to hide the details of constructing a message, sending it, awaiting a reply, and unpacking the reply. Unfortunately, RPC cannot hide the time it takes to send and receive a message. As a result, programmers who wish to improve the performance of their applications may try to reduce the number of RPCs they make through the use of a cache of requests and replies. In a general-purpose RPC system where each operation may have complicated semantics, it is difficult to keep caches consistent. The beauty of TDSM is that the semantics of ''memory'' are simple and memory is used by almost every application. Memory caching is a popular research topic, and should benefit from extensive research in the field.

The benefits of TDSM over transactional RPC include:

- Ease of use. Data flows automatically to the site of use with no explicit communication required.

- Automatic replication of read-only data. Read-only data is cached at each node where it is used.
- Automatic adaptation to changing locality. Data that is written at only one node remains cached at that node.
- Elimination of distributed commit. With function shipping, a transaction may update data stored at several nodes. Each node in the transaction must agree on the transaction's success, and must force log data to disk. With TDSM, data is shipped to the transaction's node, where it is updated locally. Only one node must force log data to disk.
- Efficient software implementation. Since only a few operations are involved, the remote procedure calls needed to implement TDSM can be hand-coded and optimized. Such optimization is difficult for transactional RPC given the wide variety of options available in a general-purpose RPC system.
- Efficient hardware implementation. Again, since only a few operations are involved, the remote procedure calls need to implement TDSM could be implemented (at least partially) in hardware. Virtual memory hardware can improve performance.

The performance advantages claimed for TDSM should not be considered in isolation. Knowing the semantics of a given application, a programmer could design an application-specific cache that transfers just the data that is needed, and thus outperforms the more general caching of TDSM. But TDSM offers performance gains in conjunction with ease of use: an application benefits from TDSM caching with no additional programming.

TDSM may not be suitable for all applications. TDSM suffers in several areas:

- Security. With function shipping, each operation is invoked by a separate RPC, and thus each operation may be individually restricted. With distributed shared memory, operations are invoked via appropriate read and write operations. The distributed shared memory system can only restrict reads and writes of particular regions.
- Availability. To ensure data integrity, a transaction system may prevent access to data that is not known to be consistent. This difficulty may be exaggerated in TDSM systems unless the system takes special care to ensure that data and the log records describing modifications to the data are available after a crash.
- Performance. TDSM may transfer more data than is needed by the application. If the cost of data transfer is not amortized over repeated accesses, it may be less expensive to ship function requests to the original location of the data.

The major challenge in designing a TDSM system is transaction recovery, the process of undoing or redoing the effects of a transaction to ensure failure atomicity or permanence. Traditional approaches to transaction recovery do not apply to TDSM because recovery of a TDSM object may involve many nodes instead of just one: nodes that are updating the object, nodes holding records of previous updates to the object, nodes that are already recovering the object, and nodes holding pieces of different versions of the object.

## 8.2. Contributions

This dissertation argues that it is feasible and useful to provide TDSM as a tool for constructing distributed applications. The specific contributions of the dissertation are summarized below:

- An architecture for implementing TDSM in a general purpose transaction processing facility. The key design decision underlying the architecture is the concept of the home node, which provides a simple, direct method for achieving transaction guarantees. The challenge of transaction recovery is addressed by storing log records and data blocks on the home node, and coordinating recovery from the home node. Caching of data and log records is used to improve performance and reduce the load on the home node.

- An overview of the implementation of this architecture in the Camelot distributed transaction facility. Camelot's concept of a recoverable virtual memory (RVM) segment is extended in two ways. First, each RVM segment can be shared by multiple Camelot servers. Second, Camelot services and pages of any RVM segment can be accessed over the network.

  The major additions to Camelot for TDSM are the Lock Manager (to ensure serializability of accesses to shared data), the Remote Execution Manager (to assist in remote access to Camelot services and RVM segments), and the External Memory Manager (to maintain the consistency of RVM pages across nodes). To support TDSM, data structures in the Camelot Disk Manager, Recovery Manager, and Node Server are modified to separate RVM segments from the servers that access them. Forward processing and recovery algorithms are similarly affected.

- A description of the implementation's performance, reporting the latency of individual operations and ET1 throughput figures in various configurations. The performance results show that reading a RVM page that is cached is no more expensive for remote access to a RVM segment than for local access. Writing a cached page and committing a transaction is somewhat more expensive because of the cost to send log records and transaction commit messages across the network. Access to a page that is <u>not</u> cached is significantly more expensive for remote access because the underlying RPC system in the experimental environment performs poorly on large (page-size) messages.

- A characterization of the applications for which TDSM is suitable. An illustration in Section 7.2 shows how aggregate throughput for one to ten nodes is limited by CPU capacity and not network bandwidth once the cache hit ratio reaches 90 percent. The illustration uses a test transaction that performs five write operations. TDSM read operations are much cheaper than write operations: a read operation that hits the cache is entirely CPU-bound, and a cache miss on a read operation is less than half the cost of a cache miss on a write operation. Thus, an application need not necessarily have high locality of reference as long as the ratio of communication costs imposed by network paging is low in comparison to application processing costs.

- An analysis of the TDSM system's performance in terms of characteristics of the underlying hardware and software. Using a model, the performance of the system is extrapolated to two different platforms. ET1 throughput is limited primarily by paging disk throughput, and thus the ET1 benchmark (as used in the performance measurements) is not really an appropriate application for TDSM. Even so, when the underlying platform is optimized for TDSM, an ET1 transaction that pages over the network can achieve a latency within 10% of the latency of an ET1 transaction that uses a local paging disk. Adapting the ET1 benchmark to TDSM to improve locality would further reduce the latency.

- A performance analysis of possible optimizations to the TDSM implementation. The failure atomicity property of transactions may be provided in part by writing a log containing all of the changes made to a data structure. This log write may also serve to update the conten*s of the caches in a distributed shared memory system, eliminating the need for additional messages to keep caches coherent. Another possible optimization is to combine locking with cache coherency: locks can be obtained automatically as an item is cached, and released automatically when an item leaves the cache. The model shows that these optimizations offer modest improvements in performance.

- A discussion of the deficiencies of the architecture and its implementation. Relatively simple changes to a few Camelot components can improve configurability and security. Limitations to incremental growth are slightly more difficult to remedy. Incremental growth in processing capacity is limited by the load of paging requests and transaction commit requests that the home node can process. Incremental growth in storage capacity is limited by the capacity of the home node, although it is conceptually simple to allow a given server to use segments from multiple home nodes.

  A more serious deficiency is limited availability. Failure of the home node effectively causes all data to be unavailable. Some of these deficiencies can be addressed by hardening the home node through replicating hardware or software services. A more promising alternative is to eliminate the home node completely, distributing its functions among all nodes in the environment.

- A discussion of some of the issues in designing future TDSM systems. During recovery, log records must be located in order to restore data blocks to a transaction-consistent state. Pages must be located during both recovery and forward processing. To improve availability, pages and log records should be replicated, complicating the algorithms that locate them, and the transaction commit protocols. Allowing a modified block to migrate before transaction commit can improve throughput, but complicates recovery.

- Some practical evidence that TDSM is feasible and useful. In the course of the thesis work, I implemented a system that provides TDSM, thus showing feasibility. As part of testing and analyzing the system, I took several multi-threaded transactional data servers that had been implemented on the original Camelot system, and ran copies of these servers on multiple nodes, with each set of servers of a particular type sharing a TDSM segment. This demonstrates the utility of TDSM in that, without recoding the servers, I was able to provide distributed, consistent, cached access to the data in the TDSM segment.

## 8.3. Future work

Although the TDSM implementation described in this dissertation is based on recoverable virtual memory, the conclusions of the thesis are not restricted to virtual memory. Many transaction processing systems provide recoverable storage through read and write operations on a buffer pool. Since the user-level pager interface of Mach is uncommon in other operating systems, it is instructive to consider how non-virtual memory TDSM could be implemented. When an application requests a block from a TDSM buffer pool manager, the manager may locate the block on disk as before, or it may ask the buffer pool manager on another node for the block. When an application releases a buffer to a TDSM buffer pool manager, the manager may write the block to disk as before, or it may send the block to another node that is requesting the block. Just as pages of virtual memory are managed to provide distributed shared virtual memory, buffers in the buffer pool can be managed to provide a shared memory.

The need for a distributed lock manager could be eliminated in future designs of TDSM. Suppose the shared memory coherency algorithm enforces a single-writer, multiple-reader protocol on data blocks. To obtain a lock, ask the coherency algorithm for a copy of the block in the same mode as the lock. Use a local lock manager to serialize local transactions, and prevent the coherency algorithm from invalidating a block until all local locks are dropped. This method should be compared and contrasted to the distributed lock manager. Another alternative that eliminates locks altogether is optimistic concurrency control.

The current design of TDSM behaves poorly when concurrent write-sharing is frequent. Future work should address this problem. One alternative is to eliminate false sharing through compiler techniques (forcing independent objects to separate pages) or modifications to the granularity of coherency control (applying the coherence algorithm to objects smaller than a page). An alternative of interest is to relax the single-writer constraint that forces an entire page to be shipped from writer to writer. Instead, multiple nodes could have write access to a given page, and could use write-through techniques to keep the copies coherent.

Chapter 7 presents a brief overview of alternative architectures for TDSM. A promising area for future work is to explore these alternatives in more detail. A difficult problem in architecting a TDSM system is finding the proper tradeoff between performance and availability when designing the transaction recovery algorithm. The architecture described in this dissertation limits availability in favor of performance and ease of implementation. Other architectures may choose to improve availability at the price of reduced performance and/or a more complicated implementation.

## 8.4. Conclusion

As applications become more sophisticated in their use of RPC, caching of requests and replies will become more common. This implies that a smart application won't use RPC directly. The question for system designers is whether to concentrate on providing the tools needed to build RPC-based systems, or to look instead at providing shared access to data with a common caching mechanism for all applications. In this dissertation, I investigated the second option in the context of transactions, presenting evidence that transactional distributed shared memory is a feasible and useful tool for constructing distributed applications.

Although this dissertation has studied TDSM in the context of a general purpose distributed system of engineering workstations, the analysis in Chapter 7 shows that the results have broader application. Given low-cost messages through the use of optimized software and hardware implementations, TDSM is a natural application for distributed or loosely coupled multiprocessors. With some additional work, TDSM could also provide a caching mechanism for database systems that use SQL to communicate between client and server.

# Appendix A
# Performance Model

This appendix develops the performance model used in Chapter 7, based on the performance results reported in Chapter 6. This model is used to project the performance of the system to different environments, and to project the effects of changes to the algorithms used by the system.

## A.1. Parameters

Eight parameters characterize the hardware and operating system:

- $c$, the time to execute one CPU instruction.
- $n_{f,l}$, the time for a null RPC (sending a message containing no data and awaiting a reply that contains no data) between two processes on the same node. This parameter is a function of the number of instructions that the operating system executes to transmit a message.
- $n_{v,l}$, the incremental time to include another byte in a message sent between processes on the same node.
- $n_{f,r}$, the time for a null RPC between two processes on different nodes. The value of this parameter depends on the operating system overhead (on both sending and receiving nodes), latency of the network interface (sender and receiver), latency of the network (the time it takes a bit to propagate from sender to receiver) and the network bandwidth (even a null RPC must transmit some control information).
- $n_{v,r}$, the incremental time to include another byte in a message send between processes on different nodes.
- $d_f$, the time to initiate a disk operation (including operating system overhead and average rotational latency).
- $d_s$, the average disk seek time.
- $d_v$, the incremental time to read or write one byte from disk.

These parameters and their values in the three scenarios described in Chapter 7 are given in Table A-1. The model is built on the assumption that only one transaction executes concurrently. All times in the model are expressed in terms of the eight parameters.

| Parameter | Actual | Scen-ario 1 | Scen-ario 2 |
|---|---|---|---|
| time per CPU instruction $c$ | .4 µs. | .1 µs. | .04 µs. |
| null RPC time (local) $n_{f,l}$ | 2.28 ms. | .30 ms. | .12 ms. |
| RPC time per byte (local) $n_{v,l}$ | .58 µs. | .05 µs. | .02 µs. |
| null RPC time (remote) $n_{f,r}$ | 16 ms. | 2.0 ms. | .24 ms. |
| RPC time per byte (remote) $n_{v,r}$ | 14 µs. | 2 µs. | .07 µs. |
| average rotational latency $d_f$ | 8.3 ms. | 8.3 ms. | 5 ms. |
| average seek time $d_s$ | 22 ms. | 22 ms. | 10 ms. |
| disk transfer time per byte $d_v$ | 1 µs. | 1 µs. | .2 µs. |

**Table A-1:** Performance model parameters

## A.2. Operation Model

Operation times in the model can be expressed in terms of the eight parameters. Formulas for the modeled times are summarized in Table A-2.

| | Local configuration | Remote configuration |
|---|---|---|
| Non-paging read fixed cost $R_{np,f}$ | $725*c$ | $725*c$ |
| Non-paging read cost/byte $R_{np,v}$ | $.425*c$ | $.425*c$ |
| Non-paging write fixed cost $W_{np,f}$ | $3500*c + 73*n_{v,l}$ | $3500*c + 73*n_{v,r}$ |
| Non-paging write cost/byte $W_{np,v}$ | $7.5*c + n_{v,l} + d_v$ | $7.5*c + n_{v,r} + d_v$ |
| Paging read operation $R_p$ | $15000*c + n_{f,l}*2 + 4096*n_{v,l} + d_s + d_f + 4096*d_v$ | $15000*c + n_{f,r}*8 + 4096*n_{v,r} + d_s + d_f + 4096*d_v$ |
| Paging write operation $W_p$ | $32000*c + 2*(n_{f,l} + 4096*n_{v,l}) + 2*(d_s + d_f + 4096*d_v)$ | $32000*c + 13*n_{f,r} + 2*4096*n_{v,r} + 2*(d_s + d_f + 4096*d_v)$ |
| Read-only transaction overhead $T_{ro}$ | $7500*c + 3*n_{f,l} + 200*n_{v,l}$ | $7500*c + 3*n_{f,r} + 200*n_{v,r}$ |
| Update transaction overhead $T_u$ | $5000*c + 3.5*n_{f,l} + 200*n_{v,l} + d_s + d_f$ | $5000*c + 4.5*n_{f,r} + 200*n_{v,r} + d_s + d_f$ |

**Table A-2:** Performance model summary

The modeled operation time formulas are derived from the operation costs measured in Chapter 6:

- Non-paging reads. A non-paging read operation involves only CPU time. From Table 6-2, the fixed cost per read is $R_{np,f} = 725*c$, and the variable cost per byte is $R_{np,v} = .425*c$.
- Non-paging writes. The difference in fixed cost per write from Table 6-3 between local and remote configurations is 1.02 milliseconds, due to increased message costs. The remaining 1.40 milliseconds is CPU time, so the formula for fixed cost per write is $W_{np,f,l} = 3500*c + 73*n_{v,l}$ in the local configuration, and $W_{np,f,r} = 3500*c + 73*n_{v,r}$ in the remote configuration. The variable cost includes writing one byte to the log disk, and the data is copied once in a message, so the formula for variable cost per byte is $W_{np,v,l} = 7.5*c + n_{v,l} + d_v$ in the local configuration, and $W_{np,v,r} = 7.5*c + n_{v,r} + d_v$ in the remote configuration.
- Paging reads. The page must be read from the paging disk, so $R_{p,d} = d_s + d_f + 4096*d_v$. There are two messages, to transfer a total of one page of data, $R_{p,n,l} = n_{f,l}*2 + 4096*n_{v,l}$, for the local configuration. In the remote configuration, the message must be split into 1500 byte packets, so the empirically determined formula is $R_{p,n,r} = n_{f,r}*8 + 4096*n_{v,r}$. Eppinger measured 5.2 milliseconds of CPU time [Eppinger 89], to which is added .7 milliseconds for locking, so the total CPU time is $R_{p,c} = 15000*c$. The time for a paging read is then the sum $R_p = R_{p,c} + R_{p,n} + R_{p,d}$.
- Paging writes. A dirty page (4096 bytes) is written to the paging disk, and a clean page is read from the paging disk. Since both operations require a seek, the formula is $W_{p,d} = 2*(d_s + d_f + 4096*d_v)$. There are two messages, each transferring one page of data, $W_{p,n,l} = 2*(n_{f,l} + 4096*n_{v,l})$, for the local configuration. Again, the data must be split into packets for the remote configuration. Since the remote configuration uses distributed locks, an extra RPC is needed to obtain a write lock. Another RPC transfers the (possibly empty) shared memory queue log buffer, so the formula is $W_{p,n,r} = 13*n_{f,r} + 2*4096*n_{v,r})$. Eppinger measured 12.1 milliseconds of CPU time [Eppinger 89], to which is added .7 milliseconds for locking, so the total CPU time is $W_{p,c} = 32000*c$. The time for a paging write is then the sum $W_p = W_{p,c} + W_{p,n} + W_{p,d}$.
- Read-only transaction overhead. Three messages transfer a total of approximately 200 bytes of data, giving the formula $T_{ro,n,l} = 3*n_{f,l} + 200*n_{v,l}$ in the local configuration, and $T_{ro,n,r} = 3*n_{f,r} + 200*n_{v,r}$ in the remote configuration. Subtracting from the values in Table 6-2 and rounding gives a CPU time of $T_{ro,c} = 7500*c$. The time for a read-only transaction is then the sum $T_{ro} = T_{ro,c} + T_{ro,n}$.
- Update transaction overhead. Initiating the write to the log disk takes $T_{u,d} = d_s + d_f$. In the local configuration, there are 3.5 messages to transfer roughly 200 bytes of data (three messages are round-trip RPCs, one message is a one-way notification), $T_{u,n,l} = 3.5*n_{f,l} + 200*n_{v,l}$. In the remote configuration, there are 4.5 messages transferring roughly 200 bytes of data, $T_{u,n,r} = 4.5*n_{f,r} + 200*n_{v,r}$. (The additional message in the remote configuration transfers the shared memory queue log buffer. The amount of data transferred depends on the operations performed by the transaction, and is assigned in the model to the appropriate operation.) This leaves CPU time of roughly $T_{u,c} = 5000*c$. This time $T_{u,c}$ is lower than the CPU time for read-only transactions $T_{ro,c}$ because some of the CPU processing for an update transaction overlaps the disk I/O. The total time is then $T_u = T_{u,c} + T_{u,n} + T_{u,d}$.

Given these formulas for operation times, the latency of a single ET1 transaction can be modeled by summing the appropriate operation times:

- Transaction overhead. To the single-server update transaction overhead add 3 messages: $T_{u,l} + 3*n_{f,l}$ or $T_{u,r} + 3*n_{f,r}$
- Paging write operations. One paging write occurs every transaction. Another paging write occurs every 78 transactions. The sum is $1.02*W_p$.
- Non-paging write operations There are 4 non-paging writes each transaction: $4*(W_{np,f} + 32*W_{np,v})$.

## A.3. Modeled results

Modeled times for each operation can be computed by substituting each set of parameter values from Section A.1 into the formulas from Section A.2. These results are given in Tables A-3, A-4, and A-5. The results show the total time for operation. The total is also broken down into CPU time (excluding RPC time), message time (combining setup and transfer time), and disk time (combining log disk and paging disk time).

| | Total time (ms. ; µs.) | CPU time (ms. ; µs.) | Message time (ms. ; µs.) | Disk time (ms. ; µs.) |
|---|---|---|---|---|
| **Local configuration** | | | | |
| Non-paging read $R_{np,f}$; $R_{np,v}$ | .29 ; .17 | .29 ; .17 | 0 ; 0 | 0 ; 0 |
| Non-paging write $W_{np,f}$; $W_{np,v}$ | 1.44 ; 4.58 | 1.40 ; 3 | .04 ; .58 | 0 ; 1 |
| Paging read operation $R_p$ | 47.3 | 6.00 | 6.94 | 34.4 |
| Paging write operation $W_p$ | 90.9 | 12.8 | 9.31 | 68.8 |
| Read-only transaction overhead $T_{ro}$ | 9.96 | 3.00 | 6.96 | 0 |
| Update transaction overhead $T_u$ | 40.4 | 2.00 | 8.10 | 30.3 |
| ET1 transaction | 149 | 21.0 | 27.0 | 101 |
| **Remote configuration** | | | | |
| Non-paging read $R_{np,f}$; $R_{np,v}$ | .29 ; .17 | .29 ; .17 | 0 ; 0 | 0 ; 0 |
| Non-paging write $W_{np,f}$; $W_{np,v}$ | 2.42 ; 18 | 1.40 ; 3 | 1.02 ; 14 | 0 ; 1 |
| Paging read operation $R_p$ | 226 | 6.00 | 185 | 34.4 |
| Paging write operation $W_p$ | 404 | 12.8 | 323 | 68.8 |
| Read-only transaction overhead $T_{ro}$ | 53.8 | 3.00 | 50.8 | 0 |
| Update transaction overhead $T_u$ | 107 | 2.00 | 74.8 | 30.3 |
| ET1 transaction | 595 | 21.0 | 474 | 101 |

Parameters reflect the hardware and operating system used in Chapter 6. Disk and message parameters and non-paging operations have a fixed cost per operation (milliseconds), plus a variable (incremental) cost per byte (microseconds); other times (milliseconds) are per operation.

**Table A-3:** Modeled performance (actual parameters)

| | Total time (ms. ; µs.) | CPU time (ms. ; µs.) | Message time (ms. ; µs.) | Disk time (ms. ; µs.) |
|---|---|---|---|---|
| **Local configuration** | | | | |
| Non-paging read $R_{np,f}$; $R_{np,v}$ | .07 ; .04 | .07 ; .04 | 0 ; 0 | 0 ; 0 |
| Non-paging write $W_{np,f}$; $W_{np,v}$ | .35 ; 1.80 | .35 ; .75 | .00 ; .05 | 0 ; 1 |
| Paging read operation $R_p$ | 36.7 | 1.50 | .805 | 34.4 |
| Paging write operation $W_p$ | 73.0 | 3.2 | 1.01 | 68.8 |
| Read-only transaction overhead $T_{ro}$ | 1.66 | .75 | .91 | 0 |
| Update transaction overhead $T_u$ | 31.9 | .50 | 1.06 | 30.3 |
| ET1 transaction | 109 | 5.26 | 3.31 | 101 |
| **Remote configuration** | | | | |
| Non-paging read $R_{np,f}$; $R_{np,v}$ | .07 ; .04 | .07 ; .04 | 0 ; 0 | 0 ; 0 |
| Non-paging write $W_{np,f}$; $W_{np,v}$ | .50 ; 3.75 | .35 ; .75 | .15 ; 2 | 0 ; 1 |
| Paging read operation $R_p$ | 60.1 | 1.50 | 24.2 | 34.4 |
| Paging write operation $W_p$ | 114 | 3.2 | 42.4 | 68.8 |
| Read-only transaction overhead $T_{ro}$ | 7.15 | .75 | 6.4 | 0 |
| Update transaction overhead $T_u$ | 40.2 | .50 | 9.40 | 30.3 |
| ET1 transaction | 167 | 5.26 | 61.5 | 101 |

Parameters reflect the hardware and operating system improvements discussed for Scenario 1. Disk and message parameters and non-paging operations have a fixed cost per operation (milliseconds), plus a variable (incremental) cost per byte (microseconds); other times (milliseconds) are per operation.

**Table A-4:** Modeled performance (Scenario 1)

| | Total time (ms. ; µs.) | CPU time (ms. ; µs.) | Message time (ms. ; µs.) | Disk time (ms. ; µs.) |
|---|---|---|---|---|
| Local configuration | | | | |
| Non-paging read $R_{np,f}$; $R_{np,v}$ | .03 ; .02 | .03 ; .02 | 0 ; 0 | 0 ; 0 |
| Non-paging write $W_{np,f}$; $W_{np,v}$ | .14 ; .52 | .14 ; .30 | .00 ; .02 | 0 ; .2 |
| Paging read operation $R_p$ | 16.7 | .60 | .32 | 15.8 |
| Paging write operation $W_p$ | 33.3 | 1.28 | .40 | 31.6 |
| Read-only transaction overhead $T_{ro}$ | ..66 | .30 | .36 | 0 |
| Update transaction overhead $T_u$ | 15.6 | .20 | .42 | 15.0 |
| ET1 transaction | 50.7 | 2.10 | 1.32 | 47.3 |
| Remote configuration | | | | |
| Non-paging read $R_{np,f}$; $R_{np,v}$ | .03 ; .02 | .03 ; .02 | 0 ; 0 | 0 ; 0 |
| Non-paging write $W_{np,f}$; $W_{np,v}$ | .15 ; .57 | .14 ; .30 | .00 ; .07 | 0 ; .2 |
| Paging read operation $R_p$ | 17.4 | .60 | .99 | 15.8 |
| Paging write operation $W_p$ | 34.9 | 1.28 | 1.97 | 31.6 |
| Read-only transaction overhead $T_{ro}$ | 1.36 | .30 | 1.06 | 0 |
| Update transaction overhead $T_u$ | 16.8 | .20 | 1.59 | 15.0 |
| ET1 transaction | 54.4 | 2.10 | 5.03 | 47.3 |

Parameters reflect the hardware and operating system improvements discussed for Scenario 2. Disk and message parameters and non-paging operations have a fixed cost per operation (milliseconds), plus a variable (incremental) cost per byte (microseconds); other times (milliseconds) are per operation.

**Table A-5:** Modeled performance (Scenario 2)

# Appendix B
# Interfaces

This appendix presents the RPC interfaces that Camelot system components use to communicate with each other and with Camelot servers and applications. Only those RPCs that were added or changed for Camelot/TDSM are included. (Thus, this appendix may be considered a supplement to the corresponding appendix in *Camelot and Avalon: A Distributed Transaction Facility* [Eppinger et al 91].)

Each interface has a two letter name. The first letter represents the component that receives the call, and the second represents the component that sends the call. The component letters are as follows:

- A - application
- D - Disk Manager
- H - Lock Manager
- N - Node Server
- R - Recovery Manager
- S - data server
- X - Remote Execution Manager

## B.1. DH Interface

```
routine DH_Initialize(
                dsPort          : port_t;
                hPort           : port_t;
            OUT dhPort          : port_t);
```

`DH_Initialize` is used by the Lock Manager to identify itself to the Disk Manager.

- `dsPort` - The port on which the Disk Manager receives requests from servers.
- `hPort` - The port on which the Lock Manager will receive requests from servers.
- `dhPort` - The port on which the Disk Manager will receive subsequent requests from the Lock Manager.

```
routine DH_PortToServerId(
                    dhPort              : port_t;
                    sPort               : port_t;
                OUT serverId            : cam_server_id_t);
```

`DH_PortToServerId` is used by the Lock Manager to store the identity of its clients in non-volatile storage.

- `dhPort` - The port on which the Disk Manager receives requests from the Lock Manager
- `sPort` - The port used to receive requests from Camelot system components.
- `serverId` - The id of the server corresponding to this port.

```
routine DH_ServerIdToPort(
                    dhPort              : port_t;
                    serverId            : cam_server_id_t;
                OUT sPort               : port_t);
```

`DH_ServerIdToPort` is used by the Lock Manager to store the identity of its clients in non-volatile storage.

- `dhPort` - The port on which the Disk Manager receives requests from the Lock Manager
- `serverId` - The id of the server.
- `sPort` - The port used by the given server to receive requests from Camelot system components.

## B.2. DN Interface

```
routine DN_GetServerState(
                    dnPort              : port_t;
                    serverId            : cam_server_id_t;
                    segmentId           : cam_segment_id_t;
                OUT state               : cam_server_state_t);
```

`DN_GetServerState` is used by the Node Server to find out the state of a server. If the Disk Manager has no record of a server, it will say the server is CAM_SS_DOWN_CLEAN.

- `dnPort` - The port on which the Disk Manager receives requests from the Node Server.
- `serverId` - The ID of the server whose state is requested.
- `state` - The state of the server.

```
simpleroutine DN_Initialize(
                        dnPort          : port_t;
                        ndPort          : port_t;
                        serverIds       : cam_server_id_list_t;
                        segmentIds      : cam_segment_id_list_t);
```

The Node Server uses `DN_Initialize` at startup time to give the Disk Manager a port for `ND_` calls, and to give the Disk Manager a list of all servers in the Node Server's database. The Disk Manager does an `ND_GetRestartAdvice()` for each server in the list, showing a transition from `CAM_SS_UNDEFINED` to `CAM_SS_DOWN_CLEAN` or `CAM_SS_DOWN_DIRTY` as appropriate.

- `dnPort` - The port on which the Disk Manager receives requests from the Node Server.
- `ndPort` - The port on which the Node Server receives requests from the Disk Manager.
- `serverIds` - The ids of all the servers in the Node Server's database.
- `segmentIds` - The segment ids corresponding to the server ids.

```
routine DN_DeleteSegment(
                        dnPort          : port_t;
                        segmentId       : cam_segment_id_t;
                        tid             : cam_tid_t);
```

`DN_DeleteSegment` is an intermediate step in deleting a server. The Disk Manager will spool a segment-deletion record that references the specified tid, so that if that transaction's family commits the segment will be permanently deleted.

- `dnPort` - The port on which the Disk Manager receives requests from the Node Server.
- `segmentId` - The ID of the segment to be deleted.
- `tid` - The identifier of the transaction on behalf of which the segment is being deleted.

## B.3. DR Interface

```
routine DR_Checkpoint(
                        drPort          : port_t;
                        recovered       : cam_segment_id_list_t;
                        notRecovered    : cam_segment_id_list_t);
```

`DR_Checkpoint` is a request from the Recovery Manager to take a checkpoint. This request is given after each node or segment recovery pass.

- `drPort` - The port on which the Disk Manager receives requests from the Recovery Manager.
- `recovered` - A list of ids of the segments that have been recovered.

- `notRecovered` - A list of ids of the segments that have not been recovered.

```
routine DR_UndoOvnv(
                        drPort              : port_t;
                        tid                 : cam_tid_t;
                        regptr              : cam_regptr_t;
                        oldValue            : pointer_t;
                        lsnUndone           : cam_lsn_t);
```

`DR_UndoOvnv` allows the Disk Manager to re-build its structures when the Recovery Manager un-does changes on behalf of an aborted or incomplete OVNV transaction. It also writes the undo record on behalf of the Recovery Manager.

- `drPort` - The port on which the Disk Manager receives requests from the Recovery Manager.
- `tid` - The id of the transaction that is being undone.
- `regptr` - The point to the region being manipulated.
- `oldValue` - The old value of the region, sent out of line.
- `lsnUndone` - The lsn of the log record that is being undone.

```
routine DR_Backstop(
                        drPort              : port_t;
                        tid                 : cam_tid_t;
                        segment             : cam_segment_id_t;
                        lastOne             : boolean_t;
                        backstopDataPtr     : pointer_t);
```

`DR_Backstop` allows the Recovery Manager to write a backstop record.

- `drPort` - The port on which the Disk Manager receives requests from the Recovery Manager.
- `tid` - The id of the transaction that is being backstopped.
- `segment` - The id of the segment whose records are copied in this backstop record.
- `lastOne` - A flag indicating if this is the last backstop record for this transaction.
- `backstopDataPtr` - That backstop data, sent out of line.

```
routine DR_KillSegment(
                        drPort              : port_t;
                        segmentId           : cam_segment_id_t);
```

If, while recovering a segment, the Recovery Manager does not receive a response to messages it sends to servers, the Recovery Manager uses `DR_KillSegment` to kill all servers using the segment.

- `drPort` - The port on which the Disk Manager receives requests from the Recovery Manager.

• segmentId - The id of the segment.

## B.4. DS Interface

```
routine DS_Initialize(
                dsPort              : port_t;
            OUT serverId            : cam_server_id_t;
            OUT recoveryOnly        : boolean_t;
            OUT tsPort              : port_t;
            OUT mPort               : port_t;
            OUT sPort               : port_t
                        = (MSG_TYPE_PORT_ALL, 32);
            OUT cPort               : port_t;
            OUT sharedMemAddr       : vm_address_t;
            OUT segDescList         :
                        cam_segment_desc_list_t;
            OUT segPortList         : port_array_t);
```

DS_Initialize gives the server all the information and capabilities it needs to reference recoverable regions.

- dsPort - The port on which the Disk Manager receives requests from this server.
- serverId - The id of this server. This value is informational. For security reasons, data servers never provide serverIds in requests to Camelot. The data server's serverId is derived from the port in which the request is made. *It is* conceivable that a data server might use its serverId in the NA interface to ask the Node Server to give it information about itself, or to change its resource allocations.
- recoveryOnly - A flag is set to TRUE if the server is expected to just recover and exit. Otherwise the server will recover and make itself available for use.
- tsPort - The port used in calls to the Transaction Manager for beginning, joining, and killing transactions.
- mPort - The port used to send debugging information to the MCP.
- sPort - The port used to receive requests from Camelot system components. These requests indicate that transactions have prepared, committed, or aborted. During transaction abort and recovery, these messages describe changes which must be made to recoverable storage. Upon receiving the reply to the DS_Initialize message, the data server will get receive and ownership rights on the sPort.
- cPort - The port used to contact the Camelot Communication Manager.
- sharedMemAddr - The address of the shared memory queue. This queue is used to efficiently pin regions and spool log records.
- segDescList - A list of this server's segment descriptors.
- segPortList - A list of ports which are used in the vm_map call to map recoverable storage into a data server's address space.

```
routine DS_GetHPort(
                    dsPort          : port_t;
                OUT hPort           : port_t);
```

DS_GetHPort obtains a port for the Lock Manager.

- dsPort - The port on which the Disk Manager receives requests from this server.
- hPort - The port on which the Lock Manager receives requests.

## B.5. DX Interface

```
routine DX_ServerDied(
                    dxPort          : port_t);
```

The Remote Execution Manager uses DX_ServerDied to notify the Disk Manager that a server exited.

- dxPort - The port on which the Disk Manager receives requests from Remote Execution Manager about a particular server.

## B.6. HS Interface

```
routine HS_Lock(
                    hsPort          : port_t;
                    lockName        : cam_lock_name_t;
                    lockMode        : cam_lock_mode_t;
                    lockPolicy      : cam_lock_policy_t;
                    tid             : cam_tid_t;
                    sPort           : port_t;
                OUT cacheable       : boolean_t);
```

HS_Lock obtains a lock on behalf of a transaction. The call blocks until the lock is available. If cacheable is true, the server is permitted to cache the lock in the mode it requested.

- hsPort - The port on which the Lock Manager receives requests from servers.
- lockName - The name of the lock being requested.
- lockMode - The mode in which the server is requesting the lock.
- lockPolicy - The caching policy to use.
- tid - The transaction on behalf of which the lock is being requested.
- sPort - The port on which the server receives requests from Camelot system components. The Lock Manager may use this port to request the return of a cached lock.
- cacheable - Indicates whether the server may cache the lock.

```
routine HS_TryLock(
                    hsPort          : port_t;
                    lockName        : cam_lock_name_t;
                    lockMode        : cam_lock_mode_t;
                    lockPolicy      : cam_lock_policy_t;
                OUT success         : boolean_t;
                    tid             : cam_tid_t;
                    sPort           : port_t;
                OUT cacheable       : boolean_t);
```

HS_TryLock is a non-blocking form of HS_Lock.

- success is TRUE if the lock request succeeded.

```
routine HS_Unlock(
                    hsPort          : port_t;
                    lockName        : cam_lock_name_t;
                    sPort           : port_t;
                    tid             : cam_tid_t);
```

HS_Unlock indicates that the given transaction has released the lock, or, if tid is CAM_TID_NULL, that the server is uncaching the lock.

- hsPort - The port on which the Lock Manager receives requests from servers.
- lockName - The name of the lock being released.
- sPort - The port on which the server receives requests from Camelot system components.
- tid - The transaction that is releasing the lock.

```
routine HS_DemoteLock(
                    hsPort          : port_t;
                    lockName        : cam_lock_name_t;
                    sPort           : port_t;
                    tid             : cam_tid_t);
```

HS_DemoteLock indicates that the given transaction is demoting a write lock to a read lock, or, if tid is CAM_TID_NULL, that the server is changing a cached write lock into a cached read lock.

- hsPort - The port on which the Lock Manager receives requests from servers.
- lockName - The name of the lock being released.
- sPort - The port on which the server receives requests from Camelot system components.
- tid - The transaction that is demoting the lock.

## B.7. NA Interface

```
camelotroutine NA_AddServer(
                     key               : cam_tid_t;
              INOUT  serverID          : cam_server_id_t;
                     owner             : user_name_t;
                     autoRestart       : boolean_t;
                     commandLine       : camelot_string_t;
                     site              : camelot_string_t;
              INOUT  segmentID         : cam_segment_id_t;
                     quotaChunks       : u_int);
```

NA_AddServer allows the addition of new servers. If the serverID parameter is nullServerId, the Node Server will pick and return an unused server ID. If the segmentID parameter is nullSegmentId, the Node Server will pick and return an unused segment ID.

- key - The key identifies the caller.
- serverID - This is the server identifier that will be assigned to the new server.
- owner - This is the name of the user that owns the server.
- autoRestart - This is true if the server should be restarted whenever Camelot is restarted.
- commandLine - This is the command line that will be used to start the server.
- site - This is the site at which the server will run.
- segmentID - This is the segment identifier for the recoverable segment that the server will use.
- quotaChunks - This is the size of the server's recoverable segment in chunks.

```
camelotroutine NA_ShowServer(
                     key               : cam_tid_t;
                     serverID          : cam_server_id_t;
                OUT  owner             : user_name_t;
                OUT  autoRestart       : boolean_t;
                OUT  commandLine       : camelot_string_t;
                OUT  site              : camelot_string_t;
                OUT  segDescList       :
                             cam_segment_desc_list_t;
                OUT  chunksUsedList    : cam_size_list_t;
                OUT  state             : cam_server_state_t);
```

NA_ShowServer shows information about a server. This includes a list of all segments tied to that server ID, number of chunks used for each segment, the server's command line, whether or not it is scheduled to be automatically restarted at node startup, its owner, and its state.

- key - The key identifies the caller.
- serverId - This is the identifier of the server that the caller wants information about.

- `owner` - The name of the user that owns the server.
- `autoRestart` - This is true, if the server should automatically be restarted when Camelot is restarted.
- `commandLine` - This is the command used to start the server.
- `site` - This is the site at which the server will run.
- `segDescList` - This is an array of descriptors describing the recoverable segments associated with the server.
- `chucksUsedList` - This array lists the number of chunks used by the server in each of its recoverable segments.
- `state` - This is the current state of the server.

```
camelotroutine NA_SetSite(
                    key             : cam_tid_t;
                    serverID        : cam_server_id_t;
                    site            : camelot_string_t);
```

`NA_SetSite` changes the site at which a server is to run.

- `key` - The key identifies the caller.
- `serverID` - The identifier of the server to be changed.
- `site` - This is the site at which the server will run.

## B.8. ND Interface

```
camelotroutine ND_GetRestartAdvice(
                    serverId        : cam_server_id_t;
                    atYourRequest   : boolean_t;
                    oldState        : cam_server_state_t;
                    newState        : cam_server_state_t;
                OUT shouldRestart   : boolean_t;
                OUT recoveryOnly    : boolean_t;
                OUT commandLine     : pointer_t;
                OUT nodeId          : cam_node_id_t;
                OUT segDescList     :
                         cam_segment_desc_list_t;
                OUT chunkDescList   : cam_chunk_desc_list_t);
```

`ND_GetRestartAdvice` is used to ask the Node Server for advice when a server transitions to a new down state, or when the Node Server has requested that the server be started. The Node Server is consulted on this because its recoverable database contains all relevant information about the server and about any variable parameters needed to implement the restart policy.

- `serverId` - The id of the server about which we inquire.
- `atYourRequest` - A flag that if `TRUE` means the Node Server has already requested the server be started and should supply the necessary information. If the `atYourRequest` flag is `FALSE`, it means the server has transitioned to a new

down state, and the Node Server has the option of restarting it. The Disk Manager describes the transition, and is told whether to restart the server.

- oldState - The previous state of the data server.
- newState - The new state of the data server.
- shouldRestart - A boolean specifying whether the server should be restarted at all.
- recoveryOnly - A boolean specifying whether the server should just run recovery and exit or should actually stay up. Not meaningful if shouldRestart is FALSE.
- commandLine - The command line that should be executed to start the server. Not meaningful if shouldRestart is FALSE.
- nodeId - The node on which the server should run.
- segDescList - A list of segment descriptors for the server. Not meaningful if shouldRestart is FALSE.
- chunkDescList - A list of disk mappings for the server. Not meaningful if shouldRestart is FALSE.

## B.9. RD Interface

```
routine RD_RecoverServers(
                    rdPort          : port_t;
                    segmentIDs      : cam_segment_id_list_t;
                    serverIDs       : cam_server_id_list_t;
                    srPorts         : port_array_t);
```

The Disk Manager may request recovery of individual servers with RD_RecoverServers.

- rdPort - The port on which the Recovery Manager receives messages from the Disk Manager.
- serverIds - The list of servers to be recovered.
- segmentIds - The list of segments to be recovered.
- srPorts - A list of ports that the Recovery Manager uses to send messages to the servers.

```
routine RD_RecoverSegments(
                    rdPort          : port_t;
                    segmentIds      : cam_segment_id_list_t;
                    serverIds       : cam_server_id_list_t;
                    srPorts         : port_array_t;
                    highRecNbr      : u_int;
                    cachePtr        : pointer_t =
                    ^array [] of (MSG_TYPE_CHAR, 8, dealloc);
                OUT nbrRecordsProcessed : u_int;
                OUT milliseconds    : u_int);
```

Segment recovery is first requested by the Node Server, since that is where the information about servers and recoverable segments is kept. But the Node Server sends its request to the Disk Manager, since that is where the list of srPorts is kept. The Disk Manager then uses RD_RecoverSegments to tell the Recovery Manager to do the work.

- rdPort - The port on which the Recovery Manager receives messages from the Disk Manager.
- segmentIds - The list of segments to be recovered.
- serverIds corresponding to the segments to be recovered.
- srPorts - A list of ports that the Recovery Manager uses to send messages to the servers.
- highRecNbr is the record number assigned (internally by the Disk Manager) to the last record currently in the log. As the Recovery Manager reads the log, it decrements this record number so that when it backlinks old records into the grid, it can indicate to the Disk Manager exactly how far back they are in the log. The Disk Manager uses this information to control page flush (and potentially other) strategies.
- cachePtr refers to the data that must be transferred from the version of logger being used by the Disk Manager to the (read-only) version used by the Recovery Manager to make them consistent.
- nbrRecordsProcessed and milliseconds are given to the Node Server by the Disk Manager for use in setting parameters that will affect recovery time.

## B.10. SH Interface

```
routine SH_Lock(
                    sPort           : port_t;
                    lockName        : cam_lock_name_t;
                    lockMode        : cam_lock_mode_t;
                    tid             : cam_tid_t;
                OUT cacheStatus     : cam_lock_status_t;
                OUT readers         : cam_tid_list_t;
                OUT writers         : cam_tid_list_t);
```

SH_Lock is used by the Lock Manager to request a server to release a cached lock on behalf of the given transaction. The server may elect to grant the lock to the transaction without releasing the cached lock. Or, it may demote a cached write lock to a cached read lock, or release the cached lock entirely. The return parameter cacheStatus indicates which selection it made. If the server is releasing a cached lock while it still has transactions holding the lock, it will return a list of these transactions in readers and writers.

- sPort - The port on which the server receives requests from Camelot system components.
- lockName - The name of the lock being requested.
- lockMode - The mode in which the lock is requested.
- tid - The transaction on behalf of which the lock is being requested.
- cacheStatus - The status of the lock in the server's cache.

- readers - A list of transactions still holding the lock in read mode.
- writers - A list of transactions still holding the lock in write mode.

```
routine SH_TryLock(
                sPort            : port_t;
                lockName         : cam_lock_name_t;
                lockMode         : cam_lock_mode_t;
            OUT success          : boolean_t;
                tid              : cam_tid_t;
            OUT cacheStatus      : cam_lock_status_t;
            OUT readers          : cam_tid_list_t;
            OUT writers          : cam_tid_list_t);
```

SH_TryLock is a non-blocking form of SH_Lock.

- success is TRUE if the request succeeded.

```
routine SH_WaitForLockToBreak(
                sPort            : port_t;
                lockName         : cam_lock_name_t;
                lockMode         : cam_lock_mode_t;
                tid              : cam_tid_t;
            OUT formerHolders    : cam_tid_list_t);
```

SH_WaitForLockToBreak is used when the Lock Manager wishes to obtain a lock from a server when that lock is not cached by the server. The return parameter formerHolders is a list of all the transactions that were holding the lock. (This optimization eliminates extra HS_Unlock calls by the server.)

- sPort - The port on which the server receives requests from Camelot system components.
- lockName - The name of the lock being requested.
- lockMode - The mode in which the lock is being requested.
- tid - The transaction that is requesting the lock.
- formerHolders - A list of transactions that were holding the lock.

```
routine SH_GetLockInfo(
                sPort            : port_t;
            OUT lockNames        : cam_lock_name_list_t;
            OUT lockModes        : cam_lock_mode_list_t;
            OUT cached           : cam_boolean_list_t;
            OUT tids             : cam_tid_list_t);
```

SH_GetLockInfo is used by the Lock Manager when it recovers after a crash to obtain the information it needs for its volatile hash tables.

- sPort - The port on which the server receives requests from Camelot system components.

- lockNames - A list of names of locks held by the server.
- lockModes - A list of lock modes, corresponding to lockNames.
- cached - A list of booleans, each one true if the corresponding lock in lockNames is cached by the server.
- tids - A list of transactions, corresponding to lockNames, holding the lock.

## B.11. XD Interface

```
routine XD_StartServer(
                   xPort           : port_t;
                   dsPort          : port_t;
                   dxPort          : port_t;
                   commandLine     : pointer_t;
               OUT pid             : int;
               OUT shMemQueueAddr  : vm_address_t);
```

XD_StartServer is used by the Disk Manager to start a server on a remote host.

- xPort - The port on which the Remote Execution Manager receives requests.
- dsPort - The port on which the Disk Manager receives requests from this server.
- dxPort - The port that the Disk Manager uses to receive requests from the Remote Execution Manager about this server.
- commandLine - This is the command line that will be used to start the server.
- pid - The UNIX process id of the server.
- shMemQueueAddr - The address of the shared memory queue. This queue is used to efficiently pin regions and spool log records.

```
routine XD_KillServer(
                   xPort           : port_t;
                   dxPort          : port_t);
```

XD_KillServer is used by the Disk Manager to kill a running server on a remote host.

- xPort - The port the Remote Execution Manager uses to receive requests.
- dxPort - The port that the Disk Manager uses to receive requests from the Remote Execution Manager about this server.

```
routine XD_GetShMemQueue(
                   xPort           : port_t;
                   dxPort          : port_t;
               OUT shMemQueue      : pointer_t);
```

XD_GetShMemQueue is used by the Disk Manager to pick up the contents of the shared memory queue.

- xPort - The port the Remote Execution Manager uses to receive requests.

- `dxPort` - The port that the Disk Manager uses to receive requests from the Remote Execution Manager about this server.
- `shMemQueue` - The contents of the shared memory queue. The Remote Execution Manager returns only the active portion of the queue.

# References

[Agarwal et al 88] Anant Agarwal, Richard Simoni, John Hennessy, Mark Horowitz.
An Evaluation of Directory Schemes for Cache Coherency.
In *Proc. Fifteenth Annual Int'l Symposium on Computer Architecture*, pages 280-289. IEEE, 1988.
Computer Architecture News Volume 16 Number 2.

[Anderson et al 91]
Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, Edward D. Lazowska.
The Interaction of Architecture and Operating System Design.
In *Proc. Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 108-120. ACM/IEEE, April, 1991.

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

[Archibald and Baer 86]
James Archibald, Jean-Loup Baer.
Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model.
*ACM Trans. Computer Systems* 4(4):273-298, November, 1986.

[Baron et al. 90] Robert V. Baron, David L. Black, William Bolosky, Jonathan Chew, David B. Golub, Richard F. Rashid and Avadis Tevanian Jr., Michael Wayne Young.
Mach Kernel Interface Manual.
August, 1990.
Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.

[Barrera 92] Joseph S. Barrera.
*Kernel Support for Distributed Memory Multiprocessors.*
PhD thesis, Carnegie Mellon University, 1992.
Forthcoming.

[Bellew et al 90] Matthew Bellew, Meuchun Hsu, Va-On Tam.
Update Propagation in Distributed Memory Hierarchy.
In *Proc. Sixth Int'l Conf. on Data Engineering.* IEEE, February, 1990.

[Bernstein and Goodman 82]
Philip A. Bernstein, Nathan Goodman.
A Sophisticate's Introduction to Distributed Database Concurrency Control.
In *Proc. Eighth Int'l Conf. on Very Large Data Bases*, pages 62-76. VLDB, 1982.

[Bershad and Zekauskas 91]
Brian Bershad, Matthew Zekauskas.
*Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors.*
Technical Report CMU-CS-91-170, Carnegie Mellon University, September, 1991.

[Bisiani et al 89] Roberto Bisiani, Andreas Nowatzyk, Mosur Ravishankar.
Coherent Shared Memory on a Distributed Memory Machine.
In *Proc. 1989 Int'l Conf. on Parallel Processing*, pages 133-141. 1989.

[Black et al 89] David L. Black, Anoop Gupta, Wolf-Dietrich Weber.
Competitive Management of Distributed Shared Memory.
In *Compcon: IEEE Computer Society Int'l Conf.*, pages 184-190. March, 1989.

[Bloch 91a] Joshua J. Bloch.
The Camelot Library.
In Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector (editor), *Camelot and Avalon: a Distributed Transaction Facility*, pages 21-56. Morgan Kaufmann, San Mateo, California, 1991.

[Bloch 91b] Joshua J. Bloch.
The Design of the Camelot Library.
In Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector (editor), *Camelot and Avalon: a Distributed Transaction Facility*, pages 163-188. Morgan Kaufmann, San Mateo, California, 1991.

[Bolosky et al 89] William J. Bolosky, Robert P. Fitzgerald, Michael L. Scott.
Simple But Effective Techniques for NUMA Memory Management.
In *Proc. Twelfth Annual Symposium on Operating Systems Principles*. ACM, December, 1989.

[Bottos 92] Beth Bottos.
July, 1992
Personal Communication.

[Carey and Livny 88]
Michael J. Carey, Miron Livny.
Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication.
In *Proc. Fourteenth Int'l Conf. on Very Large Data Bases*, pages 13-25. VLDB, 1988.

[Chang and Mergen 88]
Albert Chang, Mark F. Mergen.
801 Storage: Architecture and Programming.
*ACM Trans. Computer Systems* 6(1):28-50, February, 1988.

[Cheriton 86] David R. Cheriton.
Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design.
In *Proc. Sixth Int'l Conf. on Distributed Computing System*, pages 190-197. IEEE, 1986.

[Cheriton 87] David R. Cheriton.
UIO: A Uniform I/O System Interface for Distributed Systems.
*ACM Trans. Computer Systems* 5(1):12-46, February, 1987.

[Cheriton 88] David R. Cheriton.
*The Unified Management of Memory in the V Distributed System.*
Technical Report STAN-CS-88-1192, Stanford University, 1988.

[Cheriton et al 89] David R. Cheriton, Hendrik A. Goosen, Patrick D. Boyle.
Multi-Level Shared Caching Techniques for Scalability in VMP-MC.
In *Proc. Sixteenth Annual Int'l Symposium on Computer Architecture*, pages 16-24. IEEE, June, 1989.

[Cox and Fowler 89]
Alan L. Cox, Robert J. Fowler.
The Implementation of a Coherent Memory Abstraction on a NUMA Processor: Experiences with PLATINUM.
In *Proc. Twelfth Annual Symposium on Operating Systems Principles.* ACM, December, 1989.

[Daniels 88] Dean S. Daniels.
*Distributed Logging for Transaction Processing.*
PhD thesis, Carnegie Mellon University, December, 1988.
Also available as Tech. Report CMU-CS-89-114, Carnegie Mellon University, August, 1988.

[Davidson 89] Susan B. Davidson.
Replicated Data and Partition Failures.
In Sape Mullender (editor), *Distributed Systems*, pages 265-292. ACM Press, 1989.

[Dias et al 87] Daniel M. Dias, Balakrishna R. Iyer, John T. Robinson, Philip S. Yu.
Design and Analysis of Integrated Concurrency-Coherency Controls.
In *Proc. Thirteenth Int'l Conf. on Very Large Data Bases*, pages 463-471. VLDB, 1987.

[Dias et al 89] Daniel M. Dias, Balakrishna R. Iyer, John T. Robinson, Philip S. Yu.
Integrated Concurrency-Coherency Controls for Multisystem Data Sharing.
*IEEE Transactions on Software Engineering* 15(4):437-448, April, 1989.

[Dubois et al 86] M. Dubois, C. Scheurich, F. Briggs.
Memory Access Buffering in Multiprocessors.
In *Proc. Thirteenth Annual Int'l Symposium on Computer Architecture*, pages 434-442. IEEE, 1986.
Computer Architecture News Volume 14 Number 2.

[Duchamp 89] Dan Duchamp.
*Transaction Management.*
PhD thesis, Carnegie Mellon University, June, 1989.
Also available as Tech. Report CMU-CS-88-192 Carnegie Mellon University, June, 1989.

[Eppinger 87] Jeffrey L. Eppinger.
CPA: The Camelot Performance Analyzer.
August, 1987.
Camelot Working Memo 12.

[Eppinger 89] Jeffrey L. Eppinger.
*Virtual Memory Management for Transaction Processing Systems.*
PhD thesis, Carnegie Mellon University, February, 1989.
Also available as Tech. Report CMU-CS-89-115 Carnegie Mellon University, February, 1989.

[Eppinger 91] Jeffrey L. Eppinger.
The Design of the Camelot Disk Manager.
In Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector (editor), *Camelot and Avalon: a Distributed Transaction Facility*, pages 197-235. Morgan Kaufmann, San Mateo, California, 1991.

[Eppinger and Dietzen 92]
Jeffrey L. Eppinger, Scott Dietzen.
Encina: Modular Transaction Processing.
In *Compcon: IEEE Computer Society Int'l Conf.*. February, 1992.

[Eppinger and Michaels 91]
Jeffrey L. Eppinger, George Michaels.
Camelot Node Configuration.
In Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector (editor), *Camelot and Avalon: a Distributed Transaction Facility*, pages 57-62. Morgan Kaufmann, San Mateo, California, 1991.

[Eppinger and Nichols 91]
Jeffrey L. Eppinger, Sherri Menees Nichols.
Recoverable Storage Management in Camelot.
In Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector (editor), *Camelot and Avalon: a Distributed Transaction Facility*, pages 111-120. Morgan Kaufmann, San Mateo, California, 1991.

[Eppinger et al 91]
Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector.
*Camelot and Avalon: a Distributed Transaction Facility.*
Morgan Kaufmann, San Mateo, California, 1991.

[Fleisch 88] Brett D. Fleisch.
Distributed Shared Memory in a Loosely Coupled Distributed System.
In *Frontiers in Computer Communications Technology: Proc. ACM SIGCOMM '87 Workshop*, pages 317-327. 1988.

[Fleisch and Popek 89a]
Brett D. Fleisch, Gerald J. Popek.
Mirage: A Coherent Distributed Shared Memory Design.
In *Proc. Twelfth Annual Symposium on Operating Systems Principles.* ACM, December, 1989.

[Fleisch and Popek 89b]
Brett D. Fleisch, Gerald J. Popek.
*Mirage: A Coherent Distributed Shared Memory Design.*
Technical Report CSD-890020, UCLA, April, 1989.

[Forin et al 88] Alessandro Forin, Joseph Barrera, Michael Young, Richard Rashid.
*Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach.*
Technical Report CMU-CS-88-165, Carnegie Mellon University, August, 1988.

[Gharachorloo et al 90]
Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, John Hennessy.
Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessor.
In *Proc. 17th Annual International Symposium on Computer Architecture,* pages 15-26. IEEE, May, 1990.

[Gifford and Spector 84]
David K. Gifford, Alfred Z. Spector.
A Case Study: The TWA Reservation System.
*Communications of the ACM* 27(7):650-665, July, 1984.

[Goodman 89] James R. Goodman.
*Cache Consistency and Sequential Consistency.*
Technical Report 61, SCI Committee, March, 1989.

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

[Gray and Cheriton 89]
Cary G. Gray, David R. Cheriton.
Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency.
In *Proc. Twelfth Annual Symposium on Operating Systems Principles.* ACM, December, 1989.

[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.

[Hastings 90] Andrew B. Hastings.
Distributed Lock Management in a Transaction Processing Environment.
In *Proc. 9th Symposium on Reliable Distributed Systems,* pages 22-31. IEEE, October, 1990.

[Hsu and Tam 88] Meichun Hsu, Va-On Tam.
*Managing Databases in Distributed Virtual Memory.*
Technical Report TR-07-88, Center for Research in Computing Technology, Harvard University, March, 1988.

[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.

[Kageyama 89] Yukihisa Kageyama.
*CICS Handbook.*
McGraw-Hill, New York, 1989.

[Kong et al 90] Mike Kong, Terence H. Dineen, Paul J. Leach, Elizabeth A. Martin, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Wyant.
*Network Computing System Reference Manual.*
Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

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

[Lamport 79] Leslie Lamport.
How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.
*IEEE Transactions on Computers* C-28(9):241-248, September, 1979.

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

[LeLann 81] G. Le Lann.
A Distributed System for Real-Time Transaction Processing.
*Computer* 14(2):43-48, February, 1981.

[Li 89] Kai Li.
IVY: A Shared Virtual Memory System for Parallel Computing.
In *Proc. 1988 Int'l Conf. on Parallel Processing*, pages 94-101. 1989.

[Li and Hudak 86] Kai Li, Paul Hudak.
Memory Coherence in Shared Virtual Memory Systems.
In *Proc. Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 229-239. August, 1986.

[Li and Schaefer 89]
Kai Li, Richard Schaefer.
A Hypercube Shared Virtual Memory System.
In *Proc. 1989 Int'l Conf. on Parallel Processing*, pages 125-132. 1989.

[Mockapetris 83a] P. Mockapetris.
*Domain Names - Concepts and Facilities.*
Technical Report RFC 882, Network Working Group, November, 1983.

[Mockapetris 83b] P. Mockapetris.
*Domain Names - Implementation and Specification.*
Technical Report RFC 883, Network Working Group, November, 1983.

[Mockapetris 86] Paul Mockapetris.
*Domain System Changes and Observations.*
Technical Report RFC 973, Network Working Group, January, 1986.

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

[Mummert et al 91]
Lily B. Mummert, Dan Duchamp, Peter D. Stout.
The Design of the Camelot Transaction Manager.
In Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector (editor), *Camelot and Avalon: a Distributed Transaction Facility.* Morgan Kaufmann, San Mateo, California, 1991.

[OSF 92] Open Software Foundation.
*Distributed Computing Environment, An Overview.*
Open Software Foundation, Cambridge, MA, 1992.

[Pausch 88] Randy Pausch.
*Adding Input and Output to the Transactional Model.*
PhD thesis, Carnegie Mellon University, August, 1988.
Also available as Tech. Report CMU-CS-88-171, Carnegie Mellon University, August, 1988.

[Pausch et al 91] Randy Pausch, Dean S. Thompson, Jeffrey L. Eppinger.
An Introduction to Mach for Camelot Users.
In Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector (editor), *Camelot and Avalon: a Distributed Transaction Facility,* pages 13-20. Morgan Kaufmann, San Mateo, California, 1991.

[Peterson and Strickland 83]
R. J. Peterson, J. P. Strickland.
LOG Write-Ahead Protocols and IMS/VS Logging.
In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,* pages 216-243. ACM, March, 1983.

[Rahm 89] Erhard Rahm.
*Recovery Concepts for Data Sharing Systems.*
Technical Report 14/89, University Kaiserslautern, Department of Computer Science, October, 1989.

[Ramachandran and Khalidi 88]
Umakishore Ramachandran, M. Yousef A. Khalidi.
*An Implementation of Distributed Shared Memory.*
Technical Report GIT-ICS-88/50, Georgia Institute of Technology, December, 1988.

[Ramachandran and Mohindra 88]
Umakishore Ramachandran, Ajay Mohindra.
*A Suite of Hierarchical Cache Coherence Protocols.*
Technical Report GIT-ICS-88/51, Georgia Institute of Technology, December, 1988.

[Ramachandran et al 89]
Umakishore Ramachandran, Mustaque Ahamad, M. Yousef A. Khalidi.
Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer.
In *Proc. 1989 Int'l Conf. on Parallel Processing,* pages 160-169. 1989.

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

[Satyanarayanan et al 85]
M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, Michael J. West.
The ITC Distributed File System: Principles and Design.
In *Proc. 10th Symposium on Operating System Principles*, pages 35-50. ACM, December, 1985.

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

[Schwarz and Spector 84]
Peter M. Schwarz, Alfred Z. Spector.
Synchronizing Shared Abstract Types.
*ACM Trans. Computer Systems* 2(3):223-250, August, 1984.
Also available in Stanley Zdonik and David Maier (editors), Readings in Object-Oriented Databases. Morgan Kaufmann, 1988. Also available as Tech. Report CMU-CS-83-163, Carnegie Mellon University, November 1983.

[Snaman and Thiel 87]
William E. Snaman Jr., David W. Thiel.
The VAX/VMS Distributed Lock Manager.
*Digital Technical Journal* (5):29-43, September, 1987.

[Spector 89a] Alfred Z. Spector.
Achieving Application Requirements on Distributed Systems Architectures.
In Sape Mullender (editor), *Distributed Systems*, pages 19-33. ACM Press, 1989.

[Spector 89b] Alfred Z. Spector.
Distributed Transaction Processing Facilities.
In Sape Mullender (editor), *Distributed Systems*, pages 191-214. ACM Press, 1989.

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

[Stout 91] Peter D. Stout.
The Design of the Camelot Communication Manager.
In Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector (editor), *Camelot and Avalon: a Distributed Transaction Facility*, pages 287-291. Morgan Kaufmann, San Mateo, California, 1991.

[Stout et al 91] Peter D. Stout, Eric C. Cooper, Richard P. Draves, Dean S. Thompson.
Mach for Camelot Implementors.
In Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector (editor), *Camelot and Avalon: a Distributed Transaction Facility*, pages 93-110. Morgan Kaufmann, San Mateo, California, 1991.

[Stumm and Zhou 90]
Michael Stumm, Songnian Zhou.
Algorithms Implementing Distributed Shared Memory.
*IEEE Computer* 23(5):54-64, May, 1990.

[Sun 86] Sun Microsystems, Inc.
*Networking on the SUN Workstation*
Mountain View, California, 1986.

[Sun 88] Sun Microsystems.
*RPC: Remote Procedure Call Protocol Specification Version 2.*
Technical Report RFC 1057, Network Working Group, June, 1988.

[Tam 91] Va-On Tam.
*Transaction Management in Data Migration Systems.*
PhD thesis, Harvard University, January, 1991.

[Thompson 91] Dean Thompson.
The Design of the Camelot Recovery Manager.
In Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector (editor), *Camelot and Avalon: a Distributed Transaction Facility*, pages 237-250. Morgan Kaufmann, San Mateo, California, 1991.

[Thompson and Jaffe 91]
Dean Thompson, Elliot Jaffe.
The Design of the Camelot Local Log Manager.
In Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector (editor), *Camelot and Avalon: a Distributed Transaction Facility*, pages 189-196. Morgan Kaufmann, San Mateo, California, 1991.

[Thompson and Michaels 91]
Dean Thompson, George Michaels.
Camelot Node Management.
In Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector (editor), *Camelot and Avalon: a Distributed Transaction Facility*, pages 139-148. Morgan Kaufmann, San Mateo, California, 1991.

[Unix System Laboratories 92]
Unix System Laboratories.
*Tuxedo System 4.1 Product Overview and Master Index.*
Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[Welch 90] Brent Welch.
February, 1990
Personal Communication.

[Yu et al 85] Philip S. Yu, Daniel M. Dias, John T. Robinson, Balakrishna R. Iyer, Douglas Cornell.
Modelling of Centralized Concurrency Control in a Multi-System Environment.
In *Proc. 1985 ACM SIGMETRICS Conf.*, pages 183-191. ACM, 1985. Performance Evaluation Review, Volume 13, Number 2.

[Yu et al 86] Philip S. Yu, Douglas W. Cornell, Daniel M. Dias, Balakrishna R. Iyer.
On Affinity Based Routing in Multi-System Data Sharing.
In *Proc. Twelfth Int'l Conf. on Very Large Data Bases*, pages 249-256. VLDB, August, 1986.

[Yu et al 87] Philip S. Yu, Daniel M. Dias, John T. Robinson, Balakrishna R. Iyer, and Douglas W. Cornell.
On Coupling Multi-Systems Through Data Sharing.
*Proceedings of the IEEE* 75(5):573-587, May, 1987.