Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report

June 17, 1987

Alfred Z. Spector, Dean Thompson, Randy F. Pausch, Jeffrey L. Eppinger,
Dan Duchamp, Richard Draves, Dean S. Daniels, Joshua J. Bloch

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Camelot is a distributed transaction facility that runs on top of the Berkeley Unix 4.3 compatible Mach operating system. Camelot runs on a variety of different hardware and supports the execution of distributed transactions on shared user-defined objects, and hence the operation of distributed network services. The Camelot library, akin to the Unix Man 3 library, provides about 30 macros and procedure calls to simplify the development of applications and distributed services. To achieve good performance, Camelot is implemented using a combination of multi-thread tasks, shared memory, messages, and datagrams. This paper reports on a number of latency experiments to show the overhead of Camelot Release 0.4(22).

Index Terms - Availability, distributed databases, distributed systems, operating systems organization, reliability, transaction-based systems, persistent objects.

Technical Report CMU-CS-87-129

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

Table of Contents

1. Introduction

Distributed transaction facilities are increasingly recognized as providing important systems support for many types of reliable, distributed applications. A *distributed transaction facility* supports the execution of transactions and the implementation of permanent, abstract data objects that are shared by multiple, distributed applications. Examples of such data objects include general or special purpose databases, files, and mail queues. Examples of distributed applications that share access to such objects include control programs for automated teller terminals, computer aided design programs, and mail spoolers.

A transaction is a collection of operations bracketed by two markers: *Begin_Transaction* and *End_Transaction*. Transactions provide failure atomicity, permanence, and serializability, all of which reduce the amount of attention an application programmer must pay to concurrency and failures [19, 54].

- **Failure atomicity** ensures that if a transaction's work is interrupted by a failure, any partially completed results will be undone. A programmer can then attempt the work again in entirety by reissuing the same or a similar transaction. This property simplifies the implementation of most replication algorithms, and makes it easier to achieve high availability.

- **Permanence** guarantees that if a transaction completes successfully, the results of its operations will never be lost, except in the event of a catastrophe. Systems can be designed to reduce the risk of catastrophes to any desired probability.

- **Serializability** assures that while transactions are allowed to execute concurrently, the results will be the same as if the transactions executed serially; other concurrently executing transactions cannot observe inconsistencies. Programmers are therefore free to cause temporary inconsistencies during the execution of a transaction knowing that their partial modifications will never be visible.

When transactions contain operations on objects scattered across a distributed system, they are called *distributed transactions*. The distribution of objects on multiple processing nodes permits increased performance and system availability.

Application programmers also benefit from other functions of a distributed transaction facility. Such facilities simplify the development, use, and management of objects (or protected subsystems) that persist across program invocations. Having a single transaction facility underlying a number of abstractions permits their combined use. For example, a computer aided design system can be built using a relational database management system and a transactional file system.

Although the functions provided by a distributed transaction facility may vary somewhat, they must include the following:

- Primitives for beginning, committing, and aborting transactions;

- Communication primitives and control structures that simplify the invocation of operations on objects from within transactions;

- Synchronization, recovery, and storage manipulation primitives that permit the implementation of permanent abstract objects;
- Server management primitives that support grouping collections of abstract objects into coherent sets of related functions;
- Node management primitives that permit the configuration of processing nodes and their recovery after media failures.

Interest in transaction processing has recently spread beyond the developers of traditional on-line commercial transaction processing and database systems such as CICS, IMS, TMF, R*, and Ingres/Star [27, 28, 23, 34, 38]. These systems have relatively restrictive transaction models and are intended for mainframes or specialized transaction processing installations. Many of these systems are part of database systems and require the programmer to use a particular data model to get the benefits of transactions.

Researchers in the areas of distributed operating systems (e.g., Locus, Quicksilver, TABS), file systems (e.g., Alpine, 801) and object-oriented languages (e.g., Argus, Avalon, Garden), have been attempting to make transactions more generally useful [66, 22, 56, 7, 8, 35, 26, 50].

A major challenge driving these projects has been to embed transaction concepts within easy to use, general purpose programming environments, be they programming languages or operating systems. At Carnegie Mellon, we began to investigate transaction processing in the early 1980's [54, 49]. We developed TABS, a distributed transaction facility, as a layer on top of the Accent Kernel [57, 56]. The knowledge we gained from TABS, combined with advances made by our colleagues at other institutions, convinced us that we should undertake a methodical effort to add transaction facilities to a Unix™-like programming environment.

Camelot (**C**arnegie **M**ellon **L**ow **O**verhead **T**ransaction facility), which runs on top of the Unix BSD 4.3 compatible Mach operating system, is the result of this effort [44, 52, 58, 55]. We intend Camelot to be machine independent, easy to use in a variety of problem domains, and efficient, even in comparison with highly tuned commercial on-line transaction processing systems. Through this substantial implementation effort, we are trying to demonstrate that support for distributed transactions can and should be embedded within general purpose operating systems environments.

Furthermore, we believe that Camelot can serve as a basis for research in a variety of areas, including highly available systems. In our view, past research in distributed replication algorithms has lacked an efficient and easy to use foundation on which to experiment. For example, we intend to implement and use some replication algorithms that we have developed [6, 5, 25]. Camelot should also simplify the development of special purpose databases that support programmers and designers.

Broadly, Camelot supports the execution of distributed transactions, and the definition, management, and use of *data servers*, which encapsulate shared, recoverable objects. Camelot is based on the client-server model and uses remote procedure calls (RPCs) both locally and remotely to provide communication among applications and servers.

Camelot Release 0.4(22) is operating experimentally within CMU on about one dozen machines including IBM RT PCs and DEC Vaxes, including a dual processor Vax 8800. (Most features are implemented.) We anticipate that Camelot will also run on the SUN 3, Encore Multimax, and Sequent 21000 computers in the near future, since these three machines support Mach. We plan to make a fully functional Release 1.0 available to outsiders in the early Fall of 1987.

The following section describes the hardware architecture that Camelot assumes and the software architecture that Camelot implements. Section 3 describes Camelot's programming library and its interfaces for managing servers and processing nodes. Section 4 describes the most important implementation techniques that Camelot uses. Section 5 briefly summarizes Camelot's internal structure. Section 6 describes Camelot's current performance. Section 7 concludes with a brief discussion of Camelot and its architecture.

2. Hardware and Software Architecture

Camelot assumes an underlying hardware model that consists of both processing nodes and a communication network, as illustrated by Figure 2-1. Processing nodes may have different hardware architectures and may be either uniprocessors or shared memory multiprocessors. Processor failures are assumed to be detectable, not byzantine.

Processing nodes have three types of storage: volatile, non-volatile, and stable. Portions of objects reside in volatile storage when being accessed; objects reside in non-volatile storage when they have not been recently accessed; and stable storage is composed of memory that is assumed to retain information, despite failures. The contents of volatile storage are lost after a system crash, and the contents of non-volatile storage are lost with lower frequency, but always in a detectable way. Stable storage can be implemented using two non-volatile storage units on a node or using a network service [12, 10].

Camelot assumes a communication network that provides datagram-oriented, inter-networked ISO Level 3 functions [68] such as the Arpanet IP protocol [43]. In some cases, applications using Camelot may need high availability, so communication networks should have sufficient redundancy to render network partitions unlikely. However, Camelot assumes network partitions may occur, and guarantees that all behavior will be serializable and failure atomic.

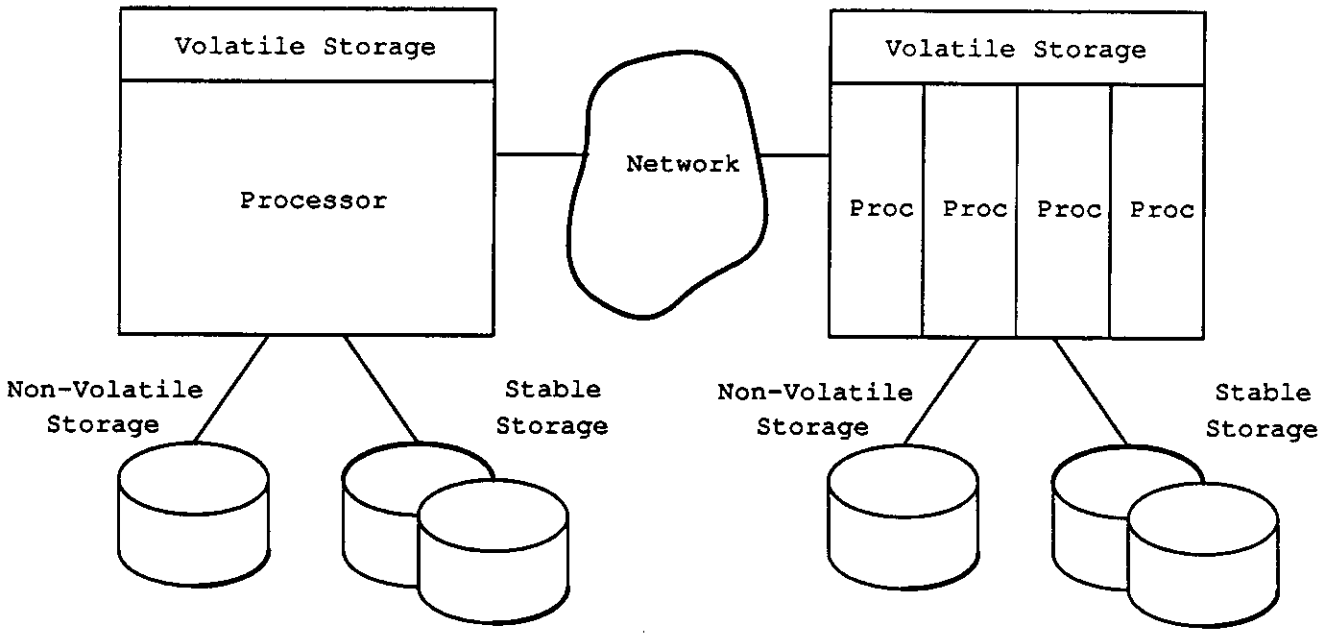

Figure 2-1: Assumed Hardware

This figure shows the components of the hardware on which Camelot runs. Stable storage and non-volatile storage, though pictured here as disks, may be implemented with other media, such as battery backed-up memory. Stable storage may not be local to a node, but rather implemented as a distributed service.

Camelot supports a software architecture based on the *client-server* model [64]. *Applications* execute operations on shared local or remote data objects that are encapsulated within server processes, called *data servers*. Camelot uses RPCs to simplify operation invocations [4, 30, 51]. A transaction either *commits*, in which case all its updates appear to be atomically made, or it *aborts*, in which case no changes appear at all. Transactions may be *nested* within one another [46, 41]. Nested transactions permit a transaction to spawn children that may run in parallel, but are synchronized so that the entire transaction still exhibits serializability. Nested transactions also are permitted to abort without causing the loss of the entire parent transaction's work.

Servers encapsulate one or more data objects and export an RPC interface to clients. To implement operations, servers read or modify data they directly control and invoke operations on other servers. Servers may also begin and commit transactions. After an operation is performed, servers return a result to the client. (Servers storing long term data are called *Resource Managers* in R* and *Guardians* in

Argus [34, 35].)

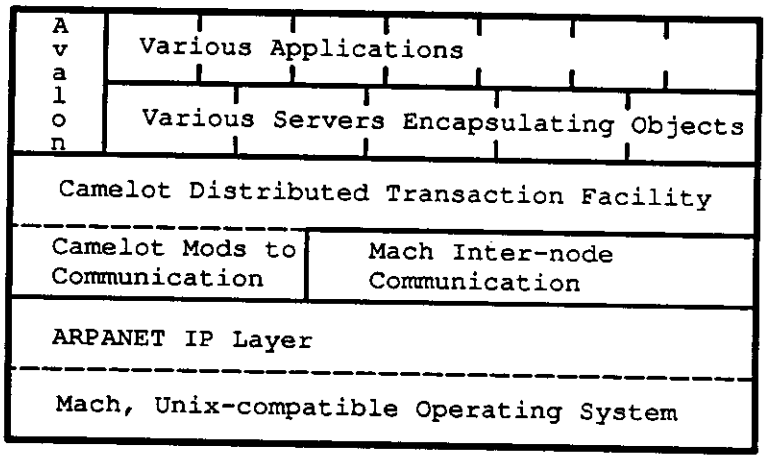

Figure 2-2: Logical Organization of a Camelot Processing Node

This figure illustrates the software that exists on a uniprocessor or multiprocessor Camelot node. Mach is at the lowest level. The Arpanet IP communication layer is logically, although not physically, layered on top of Mach. Inter-node communication, provided jointly by both Mach and Camelot facilities, is layered next. Camelot transactional processing support includes transaction coordination and recovery. The Avalon language support, though not described in this paper, will provide additional linguistic support for reliable systems and generates calls on Camelot and Mach. Users define servers encapsulating shared, recoverable objects and applications that use those objects. Applications do not internally contain long-lived data but rather use data contained in servers.

The logical organization of Camelot is shown in Figure 2-2. At the base is Mach: the operating system kernel that implements processes, virtual memory, local synchronization, and local communication. Camelot uses Mach, but conceptually, it could be supported by other kernels, such as the V Kernel [9]. Above the kernel are the internet datagram and inter-node message communication facilities. Camelot directly uses internet datagrams to coordinate transaction commitment and support distributed logging. The message communication facility is used for implementing RPCs. The Camelot distributed transaction facility uses the kernel and communication facilities to implement transactions and to support data servers and applications.

3. Camelot Release 1.0 Interfaces

There are two types of interfaces to Camelot: *programming interfaces* for writing applications and data servers, and *operational interfaces* for configuring processing nodes, installing data servers, and recovering from miscellaneous failures. The programming interfaces comes in two varieties: The Camelot library contains easy to use routines and macros, akin to those found in Section 3 of the *UNIX Programmer's Reference Manual* [62]. The library hides most of the complex details of building new Camelot applications and data servers. Underlying the library is the Camelot primitive interface, which uses shared-memory or message-based communication with central Mach and Camelot components. These components jointly manage processes, recoverable storage, transactions, and recovery. We anticipate that most programmers will use the Camelot library, and the following section describes it. Sections 3.2 describes Camelot's operational interfaces, and Section 3.3 contains fragmentary examples to demonstrate the use of Camelot.

3.1. The Camelot Library

The Camelot library permits programmers to rapidly implement data servers and applications. For servers, it provides a common message handling framework and standard processing functions for system messages (e.g., for performing recovery and participating in commit protocols). Thus, the task of writing a server is essentially reduced to writing procedures for the operations supported by the server.

The library also provides several categories of support primitives to help a programmer write these procedures. Transaction management primitives provide the ability to initiate and abort nested and top level transactions. Data declaration primitives create static recoverable objects. Data access routines permit reading and writing recoverable objects, and automatically perform the logging of modifications. Locking routines maintain the serializability of transactions. Lock inheritance among families of nested transactions is handled automatically. The library also provides primitives to make calling remote procedures almost as easy as local ones.

The Camelot library balances ease of use, performance, and flexibility concerns while remaining compatible with normal C programming practices. In total, the Camelot library contains about thirty routines and macros.

Applications that execute transactions and perform operations on data servers use most of the Camelot library except for the primitives dealing with the declaration, modification, and synchronization of recoverable objects.

Sections 3.1.1 to 3.1.5 detail the macros and routines in the Release 1.0 Camelot Library.

3.1.1. Initializing and Terminating Applications and Servers

The Camelot library provides two routines to initialize the operation of a data server. The first, `Camlib_InitializeServer`, establishes communication with various lower-level Camelot system components, cooperates with them to recover the state of a server after a crash, initializes threads of control to handle system messages (e.g., during two-phase commit), and potentially registers the name of the server in a network-wide name service. The three arguments to `Camlib_InitializeServer` are a procedure that will be executed as a transaction the first time the server is ever started (to initialize its recoverable storage); the name of the server; and a boolean that specifies whether that name should be registered with the name service. A server does not register its name if it wants to limit the clients that can access it.

The second data server initialization routine is called `Camlib_Serve`. The first of its five arguments specifies the name of a procedure that dispatches incoming RPCs to particular internal procedures. (This procedure is automatically generated by the Mach Interface Generator (MIG) RPC compiler [29].) The next three arguments, which are typically null, specify procedures to be called when a transaction has committed (`CommitProc`), after a transaction has aborted (`AbortProc`), or before a transaction votes whether to prepare to commit (`PrepareProc`). The library passes each of these procedures a transaction identifier and expects `PrepareProc` to return a boolean value. This value permits programmer-supplied logic to contribute to the prepare decision. The fifth and final argument specifies the maximum number of concurrently executing operations (multiprogramming level) that the Camelot library will allow.

The `Camlib_TerminateServer(message)` routine causes a server to exit and print a message to the Camelot error log explaining the reason for the exit.

Applications initialize themselves with a procedure called `Camlib_InitializeAppl`. This procedure takes a single argument, which is the name of the application, and returns a boolean value indicating whether or not Camelot is running. If Camelot is running, the application can use all the Camelot facilities except those in Section 3.1.4. If Camelot is not running, the application will be limited to invoking operations on servers. Each operation is then executed within the server as a new top-level transaction and committed locally by the server before the operation returns.

3.1.2. Declaring and Using Recoverable Objects

Recoverable objects may be declared within a server by enclosing any legal C object declaration between the `DECLARE_RECOVERABLE_OBJECTS` and `END_RECOVERABLE_DECLARATIONS` macros. The `REC` macro is used to access recoverable objects. The `MODIFY` macro is used to update objects and is the analogue of the C assignment operator for recoverable objects. For example, the following assigns

10 to the recoverable integer `count`.

```
MODIFY(REC(count), 10)
```

The `MODIFY` macro has the side effect of spooling log data to stable storage, so recoverable objects can be recovered automatically after transaction aborts, node failures, or media failures. (See Section 4.3.) However, the programmer has the responsibility of ensuring that there be at most one concurrent writer to any logged object, because Camelot Release 1.0 uses value logging for recovery.

Camelot Release 1.0 provides four locking calls as a simple basis for doing 2-phase locking [13, 3]. `Camlib_Lock` obtains a shared or exclusive lock on a lock specified by a lock name, which is a 32-bit quantity. `Camlib_Lock` does not return control to the caller until the lock is obtained. `Camlib_TryLock` obtains the lock if it is available, but always immediately returns a boolean indicating whether or not the specified lock was obtained. `Camlib_LockStatus` returns the status of a lock without waiting, but does not attempt to acquire the lock. `Camlib_Unlock` allows a transaction to prematurely return a lock. This procedure must be used with extreme caution, as misuse can cause behavior that is non-serializable and non-failure atomic. The `LOCK_NAME` macro coerces any 32-bit quantity into a lock name.

3.1.3. Calling Servers

The `SERVER_CALL` macro invokes operations on remote or local servers. The macro's first argument is the name of the server to call. The second argument returns a code indicating whether the remote procedure call succeeded. The third argument specifies the remote procedure and its parameters. The parameters to every remote procedure are preceded by the keywords `NOARGS` or `ARGS`, which are syntactic sugar that hides a communication port and a transaction identifier from the programmer. (NOARGS is used in the unlikely event the remote procedure has no arguments.) For example,

```
SERVER_CALL("testserver", retCode, record_write(ARGS index, value))
```

invokes the procedure `record_write` at the server "testserver", with arguments `index` and `value` and return code `retCode`.

The server name in the `Server_Call` macro has small amount of syntax associated with it:

```
Server_Name :=  <identifier> | <identifier>@<domain_name> |
                <identifier>@*
```

In all instances, the identifier must refer to a string that a server has previously passed to `Camlib_InitializeServer`. In the first alternative, the server is assumed to be on the same node as the caller. In the second, the server is assumed to located on the node with the specified Arpanet domain name. In the third alternative, the asterisk specifies that a broadcast should be done on the local subnetwork to locate the server.

A default timeout of 60 seconds is associated with RPCs. If the SERVER-CALL is executed within a transaction and a timeout occurs, the innermost transaction aborts and execution continues after the end of the aborted transaction. If a timeout occurs when the SERVER_CALL has been called by an application outside the scope of a transaction, it returns a timeout return code to the user. The library also provides a SERVER_CALL_2 macro, which is an alternate form that takes an additional, user-specified timeout value.

3.1.4. Using Transactions

Camelot provides a number of primitives for beginning, committing, and aborting transactions. The BEGIN_TRANSACTION and END_TRANSACTION(status) macros enclose arbitrary C code (including local and remote procedure calls), which is then executed as a single transaction. The status variable indicates if a transaction committed or aborted. (Mach, Camelot, the RPC compiler, and the server supply different classes of error codes in the abort case.) Nested transactions result if BEGIN_TRANSACTION ... END_TRANSACTION blocks are dynamically nested. To initiate a nested top-level transaction, Camelot provides BEGIN_TOP_LEVEL_TRANSACTION ... END_TOP_LEVEL_TRANSACTION macros.

Camelot's lower level components provide flexibility in specifying the type of recovery and commit protocol to be used. In particular, Camelot allows programmers to specify whether a transaction will modify few or many pages of data (See Section 4.3.), and whether the commit protocol should be fast or non-blocking. (See Section 4.5.) The BEGIN_TRANSACTION_2(transType) ... END_TRANSACTION_2(commitProtocol, status) macros permit these more specialized options to be used.

Applications or servers can voluntarily abort transactions by invoking Camlib_AbortTransaction(Status) or Camlib_AbortTopLevelTransaction(Status). After a Camlib_AbortTransaction call occurs, the transaction initiator continues execution following the innermost dynamically enclosing END_TRANSACTION or END_TOP_LEVEL_TRANSACTION. After a Camlib_AbortTopLevelTransaction call occurs, the initiator of the innermost *top level* transaction will continue execution after the end of the transaction. Note that a sudden change in the flow of control can happen as the result of a failure of another (possibly remote) participant aborting the transaction or a network or node failure; therefore, programmers can not assume that code between BEGIN_TRANSACTION and END_TRANSACTION will always execute in its entirety. However, the failure atomicity properties of Camelot will automatically undo any partially completed updates to recoverable storage and release the appropriate locks.

THIS_TID returns an transaction identifier specifying the currently executing transaction. The identifier may be used, for example, to implement lock managers having more functions than the built-in Camelot

lock manager.

3.1.5. Parallelism

Camelot permits parallel top-level transactions to be invoked with the `Camlib_ConcurrentTopLevelTrans` routine. This call takes a single procedural argument, and executes that procedure within a concurrent top-level transaction. A variant, `Camlib_ConcurrentTopLevelTrans2`, takes two additional arguments that specify the recovery technique and commit protocols.

There are two constructs for beginning nested transactions. The first is a `COBEGIN ... COEND` form:

```
COBEGIN
        TRANS(proc1, arg1, status1);
        TRANS(proc2, arg2, status2);
                .
                .
        TRANS(procn, argn, statusn);
COEND
```

Each procedure, $proc_i$, is invoked as a nested transaction and is passed an integer argument arg_i. The parent is joined when all the children have committed or aborted, and $status_i$ then contains the status of the i^{th} transaction. Note that this form creates a collection of top-level transactions if it is called outside the scope of a transaction.

The `COFOR` form of this provides a looping structure to establish concurrently executing nested transactions:

```
COFOR(loopVar, initVal, finalVal)
        TRANS(proc, arg, status);
COEND
```

Some servers and applications may need multiple threads of control within a single transaction, though such programs must guard against unwanted concurrency anomalies. The following two forms are analogous to the `COBEGIN` and `COFOR` forms, but do not begin nested transactions:

```
THREAD_COBEGIN
        THREAD(proc1, arg1);
        THREAD(proc2, arg2);
                .
                .
        THREAD(procn, argn);
THREAD_COEND

THREAD_COFOR(loopVar, initVal, finalVal)
        TRANS(proc, arg);
COEND
```

3.1.6. Observations

The Camelot library does not hide transaction processing notions from programmers, but it does make them easy to use. Recoverable objects blend well into the standard C object space, but server programmers must use the special primitives `REC`, `MODIFY`, and `Camlib_Lock` to access them. While we could have built locking into the `REC` or `MODIFY` calls, we did not believe we should restrict the locking options available to the programmer.

Programmers are not required to use the system-supplied unique transaction identifier, which is common to all threads running within a single transaction. Should they wish to use it, the `THIS_TID` macro makes it available. The same applies to the procedural arguments to the `Camlib_Serve` call. For example, the `CommitProc` is useful to implement hybrid atomicity [24], but its use is not required with locking.

This flexibility is an important characteristic of the Camelot library. Simple servers are easy to write and they should have high performance. However, the library permits more complex implementation techniques to be used for more complex servers. In Weihl's terminology, the Camelot library supports both explicit and implicit approaches to server development [65]. The explicit techniques are the ones that provide the flexibility.

3.2. Operational Interfaces

Before Camelot can be used, it is necessary to configure a node with appropriate raw disks for Camelot's recoverable storage and a log, if local logging is being used. The `newfs` program on Mach, which also comes with BSD 4.3 Unix [62], is used for this purpose.

When starting Camelot for the first time, Camelot will prompt for this information: the type and location of the log (local simplex, local duplex, or distributed); the size of the log if it is local; the location of node's archive dump server; and the location and size of the recoverable storage. Recoverable storage is permitted to span multiple raw disk partitions. When restarting Camelot, the previous configuration values are used, unless the `-config` flag is used to request changes to Camelot's configuration. Careful procedures must be followed for changing sensitive parameters, such as log location. To change a log, Camelot must have been previously shut down cleanly, so that it can be recovered from the new (empty) log.

Once Camelot is started, Camelot uses a distinguished Camelot data server, called the *node server*, to determine which data servers should be restarted, and what recoverable storage they are using. A special application, called the *node configuration application* (NCA), permits authorized users create, delete, start, restart, and crash one or more servers. The NCA also permits these users to specify disk

quotas for servers, the maximum recovery time that Camelot should take to recover from a node failure, and the preferable times of the day for taking archival dumps. Finally, the NCA provides commands for authorizing and deauthorizing users, and for showing a node's current configuration. All commands in the NCA are implemented in the node server as transactions.

3.3. Examples

The first step in developing a data server is to specify its RPC interfaces. As an example, MIG accepts the following interface definition for a procedure that takes an index and a 32-byte record as input parameters:

```
#define          SIZE_OF_SMALL    32
type small_record_t = array[SIZE_OF_SMALL] of
                          (MSG_TYPE_UNSTRUCTURED, 8);

camelotroutine record_write(
                port             : port_t;
                tid              : tid_t;
                index            : int;
                value            : small_record_t);
```

The port and tid arguments are required by Mach. They must be omitted in the `SERVER_CALL` macro, and are ignored in the body of the data server's `record_write` procedure, below.

The following is an example of a server's declaration of a recoverable array of 32 byte records:

```
typedef
struct {
    char data[32];
} small_record_t;

DECLARE_RECOVERABLE_OBJECTS

    small_record_t small_record_array[ARRAY_SIZE];

END_RECOVERABLE_DECLARATIONS
```

A typical main program for a simple server contains calls on `Camlib_InitializeServer` and `Camlib_Serve`. The former procedure instructs the Camelot library to call the procedure `array_init` the first time the server is ever started within a transaction. The FALSE and "testserver" parameters cause the server to be registered with the name server under the name "testserver". The latter procedure specifies that the MIG generated routine, `small_record_array_server` be used to process requests to execute remote procedures. The maximum multiprogramming level for client requests is specified at 10.

```
main()
{
    extern boolean_t small_record_array_server();

    Camlib_InitializeServer(array_init, FALSE, "testserver");
    Camlib_Serve(small_record_array_server, NULL, NULL, NULL, 10);
}
```

The `record_write` procedure executes in the server and is automatically called in response to the appropriate RPC. It takes an index and 32-character record and writes the record in the specified position in a recoverable array.

```
int record_write(port, tid, index, value)
port_t port;    tid_t tid;  /* System Parameters (ignored) */
int index;
small_record_t value;
{
    /* If index is out of bounds, return an error code */
    if (index < 0 || index >= ARRAY_SIZE)
        return(INDEX_OUT_OF_BOUNDS);

    Camlib_Lock(LOCK_NAME(small_record_array[index]),
                                LOCK_MODE_WRITE);
    MODIFY(REC(small_record_array[index]), value);

    return(0);
}
```

To install the server testserver, the NCA is used. After logging in, a user issues the `addserver` command to add the server to the database and specifies that the server be limited to using twenty 8192-byte chunks of disk space for recoverable storage. Then, the user uses the `startserver` to start it executing. (The text in bold is computer output.)

```
% nca
login: azs
Password: camelot
 connected users: azs

nca> addserver
 server id (0 for next available:) 0
 owner: admin
 auto restart [y,n]: n
 command line: /usr/azs/bin/testserver
 segment id (0 for next available:) 0
 quota (in chunks, each chunk is 8192 bytes:) 20
server: 5   owner: admin   auto-restart: no
segment: 5     quota: 163840 bytes
cmdline: /usr/azs/bin/testserver

nca> startserver
 server id: 5
Server 5 being started.
nca> q
%
```

To use the server, one may write a procedure `do_transaction`, which begins a transaction and

executes an RPC specifying the `record_write` procedure. The syntax "testserver@wales" indicates the remote procedure invocation is done to "testserver" at the Arpanet site "wales".

```
boolean do_transaction(index, value)
int index;
small_record_t value;
{
        int rc;
        status_t status;

        BEGIN_TRANSACTION
               SERVER_CALL("testserver@wales", rc,
                           record_write(ARGS index, value));
        END_TRANSACTION(status);

        if (status != COMMITTED)
        {
            printf("Transaction failed.\n");
            return(FALSE);
        }
    }
    return(TRUE);
}
```

4. Essential Camelot Implementation Techniques

Camelot uses a number of techniques to support high performance distributed transaction processing. These are used to implement data server and application tasks, communication, inter-transaction synchronization, recoverable storage, stable storage, and transactions.

4.1. Task and Communication Structures

Under Unix, a process is both a basic unit of resource allocation and protection and a single thread of control. When multiple threads of control are needed within one process, they must be run as coroutines. The Mach kernel supports Unix-like processes, which are called *tasks*. However, each task provides an execution environment for any number of threads. The threads within a task are scheduled independently by the kernel, and share all of that task's resources. In addition, each task can share certain areas of memory with other tasks. Camelot makes use of both tasks, threads, and shared memory between tasks:

- Camelot applications, data servers, and internal system components are multi-threaded using the facilities of Mach and the C thread management library [11]. Multi-threading permits real parallelism and the effective use of multiprocessors, at the programming expense of requiring careful synchronization. (The Camelot library locking calls automatically do this synchronization for data servers.) Importantly, multi-threading permits the overlap of processing and disk I/O. The latter is a traditional problem in many Unix systems due to their synchronous I/O interfaces.

- Each data server shares a separate region of memory with Camelot. The region is treated as a very efficient message queue, where data servers can place requests for asynchronous

processing by Camelot. Shared memory substantially reduces local message passing overheads. For example, all log records written by data servers are written to this shared memory area, and processed by Camelot only when necessary.

Mach supports messages that are arbitrarily long vectors of typed information, addressed to ports [45]. Many tasks may have send rights to a port, but only one has receive rights. Mach messages are the basis for RPCs issued to either local or remotely located tasks. As mentioned above, MIG is used to generate the client and server stubs.

Finally, Mach provides a UDP datagram interface. This interface is used for internode communication for transaction commitment, as described below.

4.2. Inter-Transaction Synchronization

Camelot supports two compatible types of synchronization: locking and hybrid atomicity [13, 24]. The Camelot library directly supports locking. Hybrid atomicity has features of both timestamps and locking, and requires direct use of the primitive Camelot interface.

Since both types of synchronization are implemented above the level of Camelot system components, implementations may be tailored to provide the highest concurrency. For example, with *type-specific* locking, server implementors can obtain increased concurrency by defining type-specific lock modes and lock protocols [31, 49].

Both locking and hybrid atomicity may delay transaction execution, even if that delay leads to a deadlock. Some systems implement local and distributed deadlock detectors that identify and break cycles of waiting transactions [42, 34]. Although future releases of Camelot will include a deadlock detector, Camelot Release 1.0, like many other systems, relies on time-outs. In Camelot, time-outs occur when RPC responses are delayed.

4.3. Recoverable Storage

Recovery in Camelot is based upon *write-ahead logging* [37, 18, 33, 20, 21, 48]. This technique uses an append-only sequence of records in stable storage. These records may contain a redo component that permits the effects of committed transactions to be redone, and possibly an undo component that permits the effects of aborted transactions to be undone. Updates to data objects are made by modifying a representation of the object residing in volatile storage and by spooling one or more records to the log. The Camelot library's `Modify_Object` call automatically formats log records, and spools them to stable storage by writing them in a shared memory queue.

Logging is called "write-ahead" because log records must be safely written (forced) to stable storage

before transactions commit, and before the volatile representation of an object is copied to non-volatile storage. Because of this strategy, there are log records in stable storage for all the changes that have been made to non-volatile storage, and for all committed transactions. Thus, the log can be used to recover from aborted transactions, system crashes, and non-volatile storage failures. However, because log records are rarely forced, the cost of logging during forward processing is low. For example, many transactions have to do only one log force on each site on which they execute.

Other advantages of write-ahead logging over other schemes have been discussed elsewhere, and include the potential for increased concurrency, reduced I/O activity at commit time, and contiguous allocation of objects on secondary storage [20, 61, 47]. All objects in Camelot Release 1.0 share a common log and use one of two write-ahead logging techniques: old value/new value logging or new value logging.

The simpler technique is called *old value/new value* logging, in which the undo and redo portions of a log record contain the old and new values of an object's representation. During recovery after node or server crashes, objects are reset to their most recently committed values during a one pass scan that begins at the last log record written and proceeds backward. If value logging is used, only one transaction at a time may modify any individually logged component of an object, assuming failure atomicity and permanence is desired.

The other technique is called *new value* logging, in which log records contain only a redo component. New value logging requires less log space than old value/new value logging, but it requires that pages not be written back to their home location on disk until the modifying transaction completes. This can result in double paging should the kernel need to temporarily write these pages to paging store. Camelot assumes that the transaction invoker knows if the transaction will modify few pages and, in such cases, will specify that new value logging should be used.

The Camelot recovery algorithms are similar to other previously published write-ahead log-based algorithms [18, 33], particularly those of Schwarz [48]. However, they have been extended to support aborts of nested transactions, new value recovery, and the logging of arbitrary regions of memory.

Periodically, Camelot initiates node-wide *checkpoints*. Checkpoints reduce the amount of log data that must be available for crash recovery and shorten the time to recover after a crash [21]. At checkpoint time, a list of the pages currently in volatile storage and the status of currently active transactions are written to the log. Camelot also periodically forces certain (hot) pages in volatile storage to non-volatile storage and may abort long running transactions to lessen the amount of log that must be rapidly accessible. To reduce the cost of recovering from disk failures, Camelot infrequently *dumps* the contents of non-volatile storage into an off-line archive. Camelot takes fuzzy dumps of recoverable storage, so it

does not have to stop all processing while this is going on.

One unusual aspect of Camelot and its predecessor system, TABS, is the integration of virtual memory management with recoverable storage. References to this may be found in [61, 60, 14, 16, 59]. Camelot permits a data server to map large (up to 2^{48} bytes) regions of recoverable storage into a data server[1]. Camelot then automatically ensures that the write-ahead log protocol is followed. Furthermore, Camelot automatically recovers the state of recoverable storage following transaction aborts, node failures, or media failures. Using the Camelot library, the storage for recoverable objects is allocated automatically in response to the `Declare_Recoverable_Objects` declaration.

For efficiency, Camelot allocates recoverable storage in large contiguous chunks on Mach raw disk partitions, and demand pages to those partitions. Should a server have knowledge of page access patterns, there are primitives in the Camelot primitive interface to permit data servers to prefetch or disgard pages. These primitives are useful for sequential scans in a relational database, for example.

To implement recoverable storage in virtual memory, Camelot uses the Mach external pager interface [67]. Using this interface, Mach permits a task external to the kernel to manage the backing storage of certain regions of virtual memory. In the case of Camelot, this task backs recoverable storage and writes log records to stable storage.

4.4. Stable Storage

Camelot writes log records to simplexed or duplexed disks locally, or to a remotely located log service. The distributed log service replicates log records on a collection of dedicated network log servers [12].

Each remote log server implements replicas of multiple logs and therefore supports multiple Camelot nodes. Because the log servers use uninterruptable power and every log is replicated on multiple servers, records can be forced to stable storage by writing them to two or more log servers. It is never necessary to wait for the servers to write the records to disks.

The distributed logging service uses special purpose UDP protocols to read and write log records. To reduce communication delays, log servers are expected to run on the same local area network as their clients. A form of weighted voting [17] is used as the replication technique. However, all voting can be done at node start-up time because a log is logically used by only a single client at a time. Hence, a log read can be directed to a single log server.

---

[1]The Camelot library currently restricts these to a few gigabytes, but the larger limit can be achieved using the primitive Camelot interface.

While local logging is simpler to implement, it requires relatively large amounts of disk space— a potentially high cost in certain environments. In addition to reducing the need for local disk space, the use of a shared network logging facilities may also have survivability, operational, and performance advantages. Survivability is likely to be better for a replicated logging facility because it can tolerate the destruction of one or more entire processing nodes. Operational advantages accrue because it is easier to manage high volumes of log data at a small number of logging nodes, rather than at all transaction processing nodes. Performance could be better in some instances because the shared facilities can have faster hardware than could be afforded for each processing node.

4.5. Transaction Management

Camelot system facilities support the library commands that initiate, commit, and abort top-level and nested transactions. Two options exist for commit: *blocking* commit and *non-blocking* commit. Blocking commit may result in data that remains locked until the coordinator is restarted or a network is repaired. Non-blocking commit, though more expensive in the normal case, reduces the likelihood that a node's data will remain locked until another node or network partition is repaired. In addition to these standard transaction management functions, Camelot provides an inquiry facility for determining a transaction's status. This is used to support locking in nested transactions.

Supporting distributed commitment requires both communication and stable storage transitions. For performance, the transaction manager communicates exclusively with UDP datagrams. In the case of blocking commit, Camelot forces only one log record per transaction per node used in the transaction, and sends only three datagrams per transaction per node. The non-blocking commit algorithms are roughly twice as expensive. All the usual read-only optimizations are implemented [40], and nested transaction commit does not require any network communication. Additionally, Camelot supports *group-commit*, which slightly delays the commitment of a transaction so that groups of log records can be forced at once. This reduces contention on high latency logging devices, like disks.

4.6. Other Techniques

Camelot Release 1.0 provides a logical clock [32] and Release 2.0 will provide a synchronized real-time clock. These clocks are useful not only for supporting hybrid atomicity, but other distributed algorithms, for example replication using optimistic timestamps [5]. Camelot also extends the Mach naming service to support multiple servers with the same name. This is useful to support replicated objects.

5. Camelot Internal Structure

Some functions, such as locking, are performed primarily within the Camelot library. However, many functions require support from central Mach or Camelot components. Figure 5-1 shows the seven tasks in Release 1.0 of Camelot. The Camelot tasks are the master control task, disk manager, communication manager, recovery manager, transaction manager, node server, and NCA.

- The **master control** task restarts Camelot after a node failure and interprets all (re)configuration commands. It also funnels error and debugging output from Camelot processes, data servers, and applications to a single error log.

- The **disk manager** is a Mach external pager that allocates and deallocates recoverable storage writes log records, and enforces the write-ahead log invariant. Additionally, the disk manager writes pages to/from the disk when Mach needs to service page faults on recoverable storage or to clean primary memory. It performs checkpoints to limit the amount of work during recovery and works closely with the recovery manager when failures are being processed. It periodically initiates the writing of hot pages, also to ensure faster recovery.

- The **communication manager** forwards inter-node Mach messages and provides Camelot's logical clock services. In addition, it keeps a list of all the nodes that are involved in a particular transaction. This information is provided to the transaction manager for use during commit or abort processing. It maintains information to aid in detecting orphans. Finally, the communication manager provides a name service. (The transaction manager and distributed logging service use UDP datagrams for efficiency; they bypass the communication manager.)

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

- The **transaction manager** coordinates the initiation, commit, and abort of local and distributed transactions.

- The **node server** is the repository of configuration data necessary for restarting the node. (See Section 3.2.) It stores its data in recoverable storage and is recovered before other servers. A primitive interface is defined for it, so that it may be accessed by user written application programs.

- The **NCA** permits Camelot users to update configuration data in the node server and to crash and restart servers.

The interfaces of the low-level Camelot components together constitute the primitive interface to Camelot, briefly referred to in Section 3. For example, the disk manager implements a procedure called `DS_LogOldValueNewValue`, which accepts a transaction identifier, an object location, and two log values, and spools a corresponding record to the log. As another example, Camelot assumes that all data servers implement a procedure, `SR_RestoreObject`, that sets an object to a specified value. The recovery component calls this procedure to restore the value of an object to a committed or aborted state. (The Camelot library automatically performs the `SR_RestoreObject` functions for users of the library.)

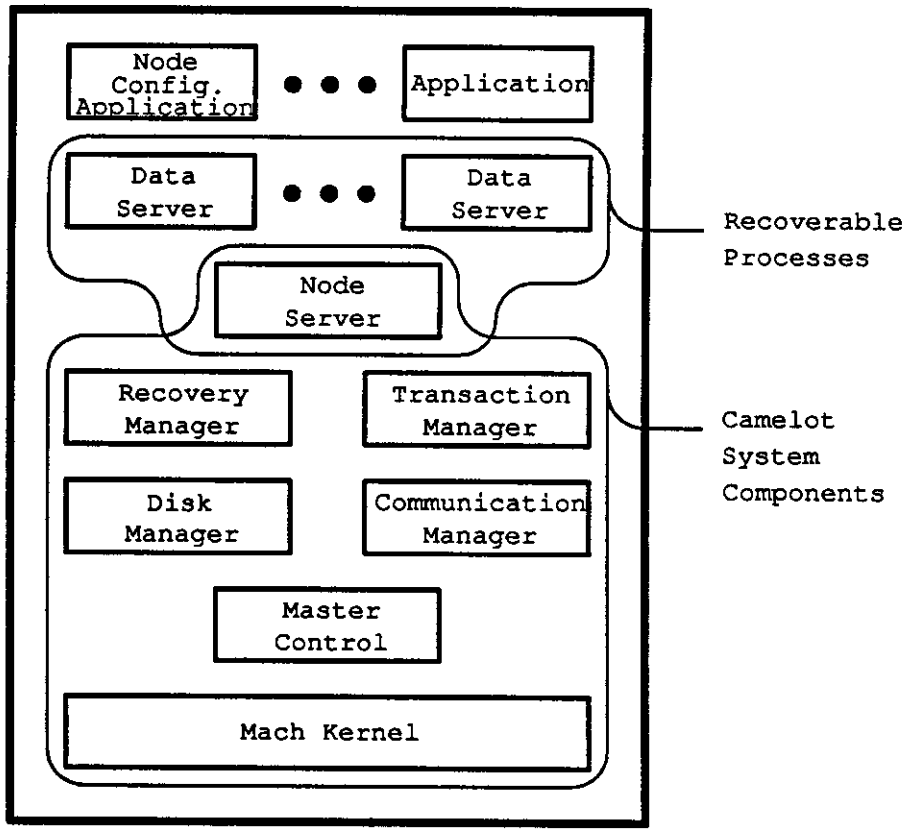

Figure 5-1: Tasks in Camelot Release 1

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

The remaining component of Camelot not yet described is the distributed logging service, which comprises multiple dedicated log servers running with uninterruptable power supplies [12]. The Camelot disk manager directly communicates with the log servers when it needs to read or write log records, or when it needs to truncate the log. This relationship between Camelot components is graphically described in Figure 5-2.

Camelot Release 0.4(22) contains about 40,000 non-comment, non-machine generated lines of C code. (There will probably be approximately 60,000 lines of code in Release 1.0)

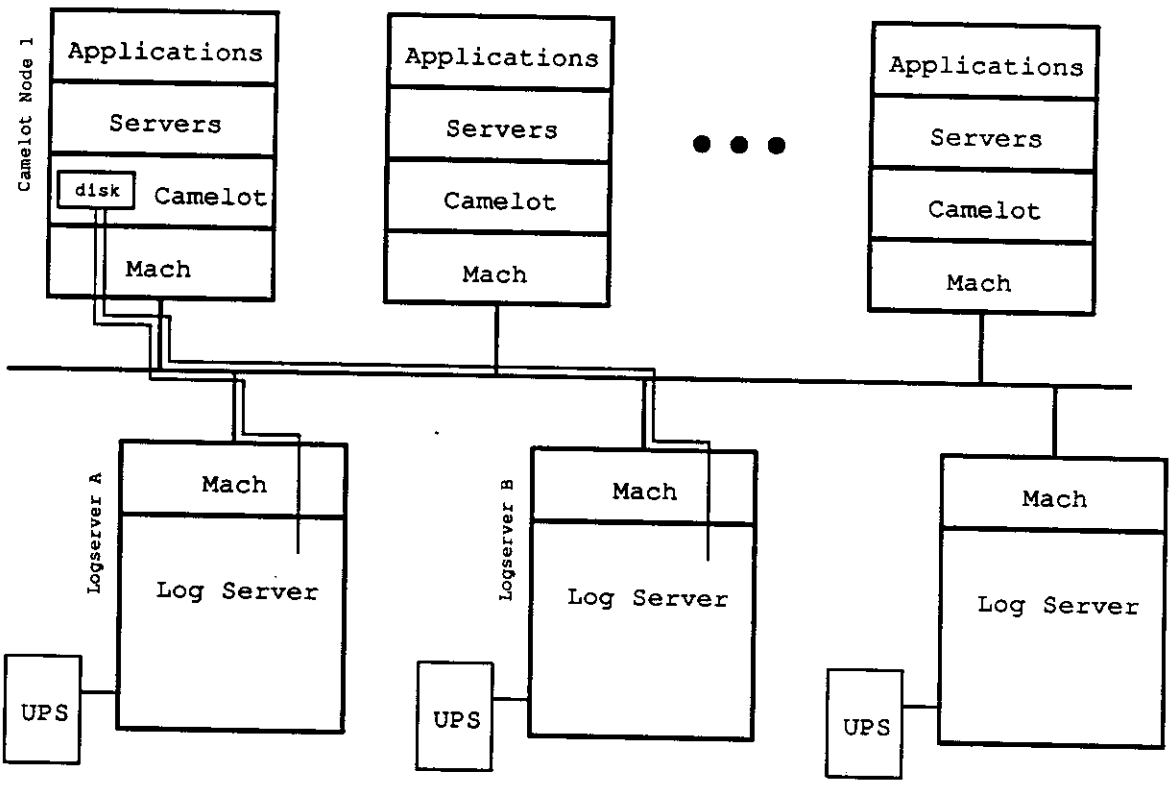

Figure 5-2: A Camelot Network with Log Servers

This shows a collection of Camelot nodes on a local area network, which could be bridged to a long haul network and other Camelot nodes. On the local area network are three log server nodes, ready to accept or provide log data. Camelot node 1 is shown as using log servers A and B actively. Node 1 would begin to use the remaining log server should either log server A or B become unavailable.

6. Performance Evaluation

A complete evaluation of the performance of a system like Camelot is difficult: Camelot provides a number of diverse functions, many of which involve distributed processing. There is substantial room for parallelism in Camelot and its client tasks on multiprocessors and on distributed systems. There is also instruction execution overlap with disk I/O and inter-node communication. Counterbalancing this parallelism is potential contention in Camelot, Mach, and data servers. Camelot runs on a wide variety of different hardware, and can use both long-haul and local area networks. Further complicating performance evaluation is the diversity of applications that Camelot is intended to support: activities ranging from commercial on-line transaction processing systems, to military command and control

systems, to CAD/CAM databases. Finally, a thorough performance analysis would be lengthy since it would require studies of both transaction latency and throughput, and a sensitivity analysis to show the effects of Mach's performance, CPU speed, disk performance, and multiprocessor architecture on performance.

Thus, it would be impractical to embed a complete evaluation of Camelot in this paper. Instead, this section describes the Release 0.4(22) performance of a number of important Camelot functions on three different machines.[2] Its goal is to convince the reader that Camelot's performance is fine for many applications and that transaction processing, per se, adds little overhead to distributed processing. The methodology used here is a subset of that of that we used to evaluate TABS, which we described in [53].

In the following section, we describe the hardware and testing methods we used. Section 6.2 describes the measured performance of the operating system and hardware functions on which Camelot depends. These measurements provide a basis for both calibrating the performance of the three individual machines and operating systems, and also for evaluating the overhead added by Camelot. Section 6.3 describes the measured latency of transaction execution on a number of simple benchmarks. Section 6.4 details the elapsed time of transaction execution as a function of the number of local 1 kilobyte write operations. Section 6.5 summarizes Camelot's performance, and provides a few performance estimates not explicitly measured for this paper.

6.1. Hardware Used

The performance measurements were done on three types of hardware:

1. IBM RT PCs [63] with 6 megabytes of memory, running at about 2 RT MIPS/second. Both Camelot and Mach were compiled using the Metaware [39] optimizing C compiler. All distributed experiments done between RT PCs used a 4 megabit/second (IEEE 802.5) IBM token ring. All disks on RT's were 70 megabyte 5 1/4" hard disks without hardware DMA. Mach uses a sector interleaving factor of 4 on these disks.

2. A DEC MicroVax II [15] with 6 megabytes of memory, running at about 1 VAX MIPS/second. Both Camelot and Mach were compiled using the Portable C compiler. No distributed experiments with Camelot are reported between Vaxes. The logging disk on the MicroVax II is a 140 megabyte 5 1/4" hard disk, with hardware DMA. Mach uses a sector interleaving factor of 1 on these disks.

3. A DEC VAX 8800, 2-way multiprocessor with 32 megabytes of memory, running at approximately 5 to 7 VAX MIPS per CPU. Both Camelot and Mach were compiled using the portable C compiler. No distributed experiments are reported with the Vax 8800. All

[2]We expect the performance numbers to be approximately 20% to 100% better in Release 1.0 in the Fall of 1987. This is primarily due to a reduction in some per-transaction overhead and forthcoming improvements in Mach message passing and datagram times.

disks on the Vax 8800 were DEC RA81s. Mach uses a sector interleaving factor of 1 on these disks. Mach was in a preliminary stage of development on this machine so performance is less well tuned than for the other machines.

All performance evaluation was done on machines that were relatively unloaded. The RT PCs were connected to the CMU internet, but we ran the experiments on a Sunday when there was relatively little other network traffic.

6.2. Operating System and Hardware Primitives

Camelot and its clients depend heavily on the following low-level operations during processing:

- **RPCs where client and server are located on the same node:** These are used by the `SERVER_CALL` macro when the client and server are collocated, and for some local communication with and between Camelot components.
- **RPCs across nodes:** These are used by `SERVER_CALL` for RPCs across the network.
- **Inter-node datagrams:** These are used during the transaction commit protocols, for lock resolution within nested transactions, and for supporting the distributed logging service.
- **Log writes to a raw disk:** These are used to implement local logging.
- **Paging reads and writes:** These are the times to do 4K byte page reads and writes of recoverable storage. For this paper, we assume all pages are accessed randomly, not sequentially. The performance of sequential paging would be substantially better.

The times for these operations, and the familiar "puzzle" benchmark are described in Table 6-1.

6.3. Camelot Benchmarks

To make the performance evaluation tractable, we ran benchmarks transactions serially from a single application, and recorded only elapsed times. In particular, we did not measure throughput when multiple clients were attempting to perform operations the same server. (Throughput would be greater than the inverse of the latency for all transactions except read-only local transactions, since Camelot exploits most sources of parallelism.) Also, we did not measure Camelot's paging performance. The reason is that Camelot's disk manager does not interfere with normal paging, except for occasionally writing hot pages to disk.

We ran the following benchmarks:

- Transactions on 1, 2, and 3 local data servers that read (write) one 32 byte record. These benchmarks demonstrate the time to execute simple read (write) transactions. Additionally, they demonstrate the extra cost of involving more than one data server.

Primitive Operation	IBM RT PC	DEC MicroVax	DEC VAX 8800
Local 32 byte RPC	4.1	4.4	0.89
Local 1024 byte RPC	5.0	5.4	1.1
Remote 32 byte RPC	40	NA	NA
Remote 1024 byte RPC	57	NA	NA
Single 64 byte inter-node datagram	7	NA	NA
Average 512 byte disk write, streamed raw I/O, includes track steps	3.3	2.7	1.5
Random access 4096 byte paged I/O, read or write	55	43	37
Puzzle Performance	4.6	8.9	1.8

Table 6-1: Primitive Operation Times in Msecs

This chart presents the elapsed time in milliseconds to do 8 benchmarks on the IBM RT PC, DEC Microvax, and DEC VAX 8800. These numbers are average elapsed times produced during long running tests. They reflect the performance of the Mach Alpha release kernel of late May 1987. The elapsed time for local communication and puzzle benchmarks is comprised entirely of CPU time. The NAs indicate performance numbers that we did not obtain; they are not needed to interpret the benchmarks presented in Section 6.3, since we ran distributed experiements on only the RT PC. Mach is probably most highly optimized on the MicroVax II, with the RT PC implementation second best. The -O (optimize) compiler switch was used on the VAX compiler for the puzzle benchmark.

- Transactions on 1, 2, and 3 local data servers that read (write) ten 32 byte records, each as a separate RPC. These benchmarks demonstrate the incremental costs of doing additional operations on servers. When coupled with the numbers from the one read transactions, these benchmarks enable one to deduce the local per-transaction overhead. In the case of write transactions, these transactions show the effects of log forces on transaction performance.

- All of the above with 1 kilobyte records. These benchmarks demonstrate the additional cost of transmitting the larger records.

- All of the above with each data server on a separate remote node. These benchmarks demonstrate the cost of doing non-local operations committing distributed transactions. We ran these benchmarks only on the RT PCs on the token ring, though they could have been run on the Vaxes, or with mixed RT PCs and Vaxes.

Note that 10 operations per server per transaction execute up to 30 operations. For the benchmarks using 1 kilobyte records, these are relatively heavy-duty, 30 kilobyte update transactions.

To get the numbers in the tables, we used a simplex log, and executed new-value only transactions.

Transaction Benchmarks	1 Server	2 Servers	3 Servers
Local Read Transactions			
1 32 byte read/server	27	44	61
10 32 byte reads/server	76	137	202
1 1K byte read/server	28	46	63
10 1K byte reads/server	86	160	240
Local Write Transactions			
1 32 byte write/server	52	80	110
10 32 byte writes/server	118	234	372
1 1K byte write/server	68	97	137
10 1K byte writes/server	216	403	688
Remote Read Transactions			
1 32 byte read/server	110	190	240
10 32 byte reads/server	590	1090	1680
1 1K byte read/server	130	210	300
10 1K byte reads/server	770	1450	2150
Remote Write Transactions			
1 32 byte write/server	170	260	320
10 32 byte writes/server	660	1170	1740
1 1K byte write/server	190	290	400
10 1K byte writes/server	900	1590	2270

Table 6-2: Benchmarks on 1-4 RT PCs, elapsed msecs/transaction

These benchmarks were run on 6 megabyte RT PCs interconnected with a 4 megabit/second token ring. These benchmarks indicate, for example, that the RT PC is capable of executing at least 19 simple, main memory write transactions per second on a single local server without group commit. With paging transactions, write transaction latency would increase substantially due to the need to move data to/from disks. Mach remote RPC times of 40 - 57 msecs currently account for much of the elapsed times of multi-operation, distributed transactions. Note that elapsed times would barely increase as the number of remote servers increase if the distributed benchmarks had executed the `SERVER_CALL`s in parallel, using the `COBEGIN ... COEND` library constructs. (See Section 6.5.)

We did not use group commit, since we were looking to achieve low latency. (Of course, system throughput would increase for short write transactions.)

The tables report the average elapsed time per transaction for the median run that we executed. (For most tests, we ran 30 runs of 20 transactions per run, for 600 transactions total. For the very long distributed transactions, we ran 10 runs of 10 transactions per run.) In general, this value was close to the average elapsed time per transaction across all runs. We chose to report the median value in an attempt to factor out the effects of paging and network traffic not reflecting Camelot's underlying performance. Approximately, 50,000 transactions were executed to produce these numbers. The numbers reflect Camelot Version 0.4(22) of June 10, 1987.

Transaction Benchmarks	1 Server	2 Servers	3 Servers
Local Read Transactions			
1 32 byte read/server	37	58	80
10 32 byte reads/server	97	181	263
1 1K byte read/server	38	61	83
10 1K byte reads/server	112	207	298
Local Write Transactions			
1 32 byte write/server	66	97	126
10 32 byte writes/server	258	318	456
1 1K byte write/server	72	115	147
10 1K byte writes/server	257	475	688

Table 6-3: MicroVax II Benchmarks, elapsed msecs/transaction

These benchmarks were run on a 6 megabyte MicroVax II. The numbers ar relatively close to the RT PC numbers: The MicroVax executes C code more slowly, but has a faster disk.

Table 6-2 shows the results for all these benchmarks on the RT PC. Tables 6-3 and 6-4 show the local benchmarks for the MicroVax II and VAX 8800.

Transaction Benchmarks	1 Server	2 Servers	3 Servers
Local Read Transactions			
1 32 byte read/server	8	13	18
10 32 byte reads/server	16	32	49
1 1K byte read/server	8	13	19
10 1K byte reads/server	19	38	58
Local Write Transactions			
1 32 byte write/server	33	33	33
10 32 byte writes/server	33	67	85
1 1K byte write/server	34	34	52
10 1K byte writes/server	76	133	198

Table 6-4: VAX 8800 Benchmarks, elapsed msecs/transaction

These are elapsed times per transaction for running the benchmarks on a VAX 8800. There is a small amount of parallelism in the benchmarks, but they primarily run on one CPU. The large jumps in some of the write transactions (e.g., 1 Kbyte write operations) is apparently due to the delay caused by missing a disk rotation.

6.4. Performance Graphs

We ran one more detailed measurement on an RT PC, a MicroVax, and VAX 8800 to learn the performance of long update transactions. Long update transactions strain write-ahead logging techniques, but are useful in applications such as CAD/CAM. Graph 6-1 shows the elapsed time as a function of the number of 1 kilobyte updates, for up to 100 separate updates. Approximately 8, 12, and 2 msecs of CPU time is spent executing the update operation in the server on the RT PC, MicroVax, and VAX 8800, respectively. Most of the rest of the time is due to logging I/O. This time would decline substantially on the RT PC with a DMA disk controller. We conclude that the logging of very large updates will not be an intolerable performance penalty for many CAD/CAM activities.

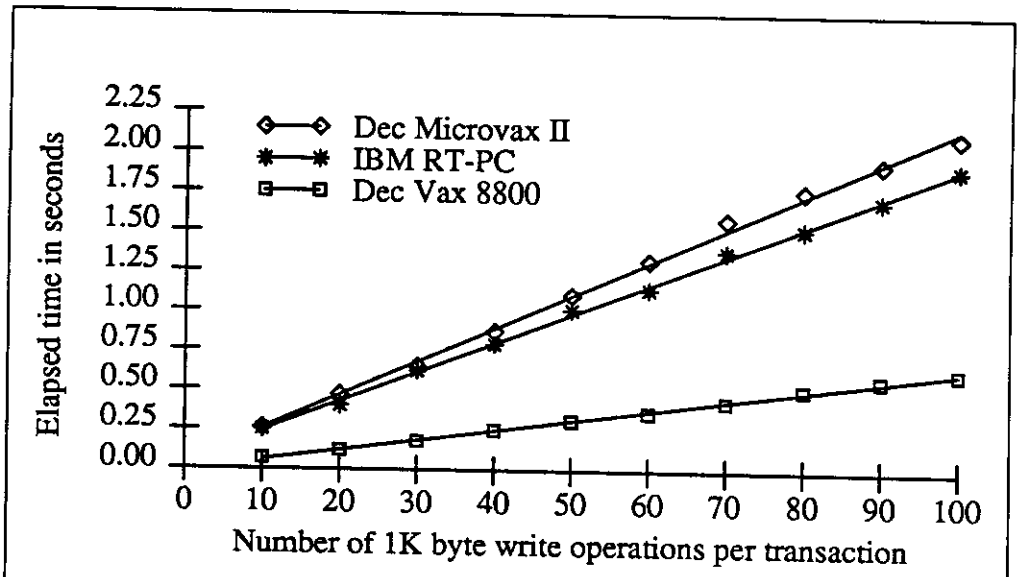

Figure 6-1: Times for Large Update Transactions, in Seconds

This graph shows the elapsed time in seconds of local transactions as a function of the number of 1 kilobyte updates, from 10 to 100 separate updates, in gradations of 10.

6.5. Discussion

The approximate elapsed times of certain Camelot functions are reported in Table 6-5. These numbers are derived from the observed performance, ad reported in the previous two sections. Comparing these derived performance numbers with the primitive times reported in Table 6-1 helps to understand the overhead of transaction execution in Camelot.

For example, Table 6-5 shows that the cost on RT PCs of beginning and committing a read-only transaction on *n* servers is approximately 10+12n msecs for the local servers and 50+10n msecs for remote servers. The higher base in the distributed case is a result of extra processing and

communication delays in the distributed commit protocol, and the CPU overhead of tracking the spread of distributed transactions. The lower slope is due to parallelism in the distributed protocol.

Camelot Function	IBM RT PC	DEC MicroVax	DEC VAX 8800
Cost/Transaction			
Local read-only	10+12n	14+15n	2.8+4.3n
Local write	21+22n	31+22n	6.8+6n
Remote read-only	50+10n		
Remote write-only	120+10n		
Cost/Read Operation			
Local 32 byte	5.5	7	1.1
Local 1024 byte	6.5	8	1.6
Remote 32 byte	52		
Remote 1024 byte	70		
Cost/Write Operation			
Local 32 byte	8.5	12	2.1
Local 1024 byte	17.5	20	6.2
Remote 32 byte	52		
Remote 1024 byte	73		

Table 6-5: Approximate Elapsed Time of Various Camelot Functions

In the cost per transaction formulas, *n* refers to the number of servers involved in a transaction. For the distributed case, the formulas apply to a single coordinator node and *n* servers running on *n* separate nodes. The formulas are usually accurate within 5%, and are off by more than 10% only in cases where log forces consistently miss disk revolutions.

In comparing the cost per operation, a read operation adds about 1 msecs to the cost of a local RPC on Mach on the RT PC. A write operation adds about 4+8m msecs to the cost of an RPC, where *m* is the number of kilobytes written. Camelot adds roughly 11 msecs to the cost of a remote read operation, and roughly 11+3m msecs to the cost of a remote write operation. These formulas generally match the numbers measured and are consistent with the expected behavior of the underlying recovery algorithms.

This paper has not mentioned paging, except to say that Camelot only minimally interferes with the normal paging of recoverable objects. It also has not mentioned the overhead of nested transactions. Nested transactions have a small fixed overhead, for example, of about 8 milliseconds on the RT PC. Note that the commit of a nested transaction never requires a 2-phase protocol or any non-local message communication. Nested transactions become expensive in Camelot only when there are intra-family lock conflicts. Resolving these could require multiple network datagrams to be sent between transaction managers.

Importantly, the paper has not mentioned recovery time after node failure. Recovery time is linear in the amount of log between the tail of the log (at the time of the crash) and the most recent checkpoint record, the first log record applying to the oldest active recoverable page, or the first log record of the longest running transaction, whichever is further back. Checkpoints are very inexpensive in Camelot, and can be taken frequently. Very long running transactions are automatically aborted. The Camelot disk manager tracks the oldest log record that applies to every recoverable page to ensure that hot pages are written with sufficient frequency. We do not have specific performance numbers, as we have not yet done empirical studies of recovery times.

While we have not reported any throughput measurements in the preceding section, we have done a small amount of experimentation. Here are two small throughput results:

- Five transaction processing nodes were able to drive two RT PCs running as a distributed log service at the the rate of sixty-nine 800 byte log forces per second. Due to the fact that each log force was sent to two RTs, (for replication), this used about 25% of the 4 Mbits/second available on the token ring.

- Three clients competing for local read-only service on one server achieved 92% utilization of each processor on the VAX 8800. There was no throughput degradation when 3 clients competed for read-only service on one Camelot server on an RT PC with read-only transactions.

These numbers suggest the system will perform well, but definitive proof would require more evidence. We are willing to project that Camelot Release 1.0 will execute roughly 10 local ET-1 [2] debit/credit transactions per second on an RT PC with 8 megabytes of memory, provided there is sufficient disk capacity to do local logging.

Camelot's performance should improve substantially with improvements to local and remote message passing, and Camelot's implementation. For example, we can usually eliminate the call on the transaction manager to allocate a unique transaction identifier by piggybacking the identifier on a previous acknowledgement.

Only recently have *detailed* performance evaluations of the transactional costs of other systems begun to be published, and there is still very little data. However, a recent paper on Argus [36], provides additional evidence that transaction overhead, particularly for nested transactions, can be very low. The Quicksilver system [22] appears to have overhead per transaction that is similar to Camelot's but Quicksilver has substantially faster local and network communication.

There have also been a number of performance studies of the ET-1 benchmark, providing throughput results. In a recent paper, Tandem described a large experiment in which they executed a sustained rate of 6.5 ET-1 transactions/second on a network of 32 3 MIPS processors [1]. (Fifteen percent of these

transactions were distributed.) It seems that Camelot's performance on RT PCs is in the same league, though it must be said that Tandem's system provides functions (e.g., process pairs and support for large networks) that Camelot does not.

7. Conclusions

Camelot integrates and extends the best features of many systems. For example, the commit protocols are optimizations of those developed for R* [40]. The nested transaction model was developed at MIT and used in Argus [35]. RPC protocols and stub compilers became of wide interest in the early 80's. Write-ahead logging was developed for commercial database management systems, such as IBM [28].

However, there are also a number of quite new ideas in Camelot. The hybrid non-blocking commit protocol, the combination of old-value/new-value and new value-only logging; the distributed logging algorithms; the C-oriented interface for transaction processing; Camelot's machine independence and effective use of multiprocessors; and the integration of virtual memory management and recoverable storage [16] are some of the more important ones. These will be the subject of Ph.D. dissertations over the next year.

The goal of this paper has not been to describe the new algorithms in Camelot, but rather to describe the implementation and integration of diverse ideas from the fields of distributed systems, operating systems, and database systems. The Camelot interfaces bundle many diverse techniques into an easy to use and efficient package for a Unix-like environment. We hope this paper will help to make the case that distributed transaction processing can be an easy to use and relatively efficient basis for supporting reliable, distributed systems.

Acknowledgments

Joshua Bloch developed the Camelot library. Dean Daniels, with the assistance of Dean Thompson, developed the distributed logging service. Richard Draves developed the Camelot extensions for MIG and the Mach inter-node message server. Dan Duchamp developed the transaction manager. Jeffrey Eppinger developed the disk manager, and node server. Dean Thompson has maintained our programming environment, and with help from Sherri Menees, developed the recovery manager. The authors wish to thank Kathryn Swedlow for her editorial assistance and Bruce Horn for assistance with the graph.

References

[1] Anon et Al.
A Benchmark of Tandem's Nonstop SQL Demonstrating over 200 Transactions Per Second.
May, 1987.
Tandem Memo.

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

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

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

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

[6] Joshua J. Bloch, Dean S. Daniels, Alfred Z. Spector.
A Weighted Voting Algorithm for Replicated Directories.
JACM 34(4), October, 1987.
To appear. Also available as Technical Report CMU-CS-86-132, Carnegie-Mellon University, July 1986.

[7] Mark R. Brown, ???
The Alpine File System.
ACM Trans. on Computer Systems 3(4):261-293, November, 1985.

[8] A. Chang, M. Mergen.
801 Storage: Architecture and Programming.
In *Proceedings of the 11th Symposium on Operating System Principles*, pages ???-???. ACM, November, 1987.

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

[10] D.R. Cheriton.
Log Files: An Extended File Service Exploiting Write-once Optical Disk.
In *Proceedings of the 11th Symposium on Operating System Principles*, pages ???-???. ACM, 1987.

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

[12] Dean S. Daniels, Alfred Z. Spector, Dean Thompson.
Distributed Logging for Transaction Processing.
In *Sigmod '87 Proceedings*. ACM, May, 1987.
Also available as Technical Report CMU-CS-86-106, Carnegie-Mellon University, June 1986.

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

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

[15] *VAX Architecture Handbook*
Digital Equipment Corporation, 1981.

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

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

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

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

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

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

[22] R. Haskin, Y. Malachi, W. Sawdon.
Recovery Management in QuickSilver.
In *Proceedings of the 11th Symposium on Operating System Principles*, pages ???-???. ACM, November, 1987.

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

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

[25] Maurice P. Herlihy.
A Quorum-Consensus Replication Method for Abstract Data Types.
ACM Transactions on Computer Systems 4(1), February, 1986.

[26] M. P. Herlihy, J. M. Wing.
Avalon: Language Support for Reliable Distributed Systems.
In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing.* IEEE, July, 1987.

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

[28] *IMS/VS Version 1 General Information Manual*
GH20-1260 edition, IBM Corp., White Plains, NY., 1980.

[29] Mike Jones, Mary Thompson.
MIG - The Mach Interim Matchmaker.
1987.
Mach Group document.

[30] 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.

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

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

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

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

[35] Barbara H. Liskov, Robert W. Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[36] B. Liskov, P. Johnson, R. Scheifler.
Implementation of Argus.
In *Proceedings of the 11th Symposium on Operating System Principles*, pages ???-???. ACM, November, 1987.

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

[38] R. McCord.
INGRES/STAR: A Heterogeneous Distributed Relational DBMS.
In *Sigmod '87 Proceedings.* ACM, May, 1987.

[39] *High C™ Programmer's Guide*
MetaWare Incorporated, Santa Cruz, CA, 1986.

[40] C. Mohan, B. Lindsay.
Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions.
In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 76-88. ACM, August, 1983.

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

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

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

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

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

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

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

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

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

[50] Andrea H. Skarra et. al.
An Object Server for an Object-Oriented Database System.
In *Proceedings International Workshop on Object-Oriented Database Systems*, pages 196-204. September, 1986.

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

[52] Alfred Z. Spector.
Distributed Transaction Processing and the Camelot System.
In Yakup Paker et al. (editors), *Nato Advanced Study Institute Series - Computer and Systems Sciences: Distributed Operating Systems: Theory and Practice*, pages 331-353. Springer-Verlag, 1987.
Also available as Carnegie Mellon Report CMU-CS-87-100, January 1987.

[53] 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.

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

[55] Alfred Z. Spector, Kathryn R. Swedlow, ed.
The Guide to the Camelot Distributed Transaction Facility: Release 1
0.4(14) edition, Carnegie Mellon University, Pittsburgh, PA, 1987.

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

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

[58] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, Dean S. Thompson.
The Camelot Project.
Database Engineering 9(4), December, 1986.
Also available as Technical Report CMU-CS-86-166, Carnegie-Mellon University, November 1986.

[59] Lindsey L. Spratt.
The Transaction Resolution Journal: Extending the Before Journal.
Operating Systems Review 19(3):55-62, July, 1985.

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

[61] Irving L. Traiger.
Virtual Memory Management for Database Systems.
Operating Systems Review 16(4):26-48, October, 1982.
Also available as Technical Report RJ3489 IBM Research Laboratory, San Jose, California, May, 1982.

[62] *UNIX Programmer's Reference Manual (PRM)*
April 1986 edition, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, 1986.

[63] Waters, F. (ed.).
IBM RT Personal Computer Technology.
International Business Machines Corporation, 1986.

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

[65] William E. Weihl.
Specification and Implementation of Atomic Data Types.
PhD thesis, Massachusetts Institute of Technology, March, 1984.

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

[67] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, R. Baron.
The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.
In *Proceedings of the 11th Symposium on Operating System Principles*, pages ???-???. ACM, November, 1987.

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