1992

Design and Implementation of the RAID V2 Distributed Database System

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Karl Friesen

Abdelsalum Helal

Enrique Malfa

John Riedel

Report Number:
90-962
THE DESIGN AND IMPLEMENTATION OF THE RAID-V2 DISTRIBUTED DATABASE SYSTEM

Bharat Bhargava
Kari Friesen
Abdelsalam Helal
Engrique Maia
John Riedl
Jagannathan Srinivasan

CSD-TR-962
March 1990
(Revised August 1992)
The Design and Implementation of the RAID-V2 Distributed Database System*

Bharat Bhargava
Karl Friesen
Abdelsalam Helal
Enrique Mafia
John Riedl
Jagannathan Srinivasan

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
(317) 494-6013

August 20, 1992

Abstract

The RAID distributed database system has proven to be a valuable tool for the experimental investigation of reliability, adaptability, communication, and checkpointing in transaction-based systems. As we studied these issues and others, we realized that some of design decisions that we made had alternatives that could improve the flexibility and the reliability of the system. Our experience with RAID has prompted us to re-examine these alternatives and implement a second version. In this paper, we present a brief synopsis of the original RAID system and our motivation for modifying it. We detail the changes and enhancements that distinguish the new version. We then give examples of the experimental studies that can be conducted using the RAID software. Finally, we list some insights about adaptability and reliability obtained

*This research is supported in part by a grant from AIRMICS, and National Science Foundation grant IRI-8821398.
from participating in the restructuring of a large software system and present possible directions for further research.
Contents

1 Introduction 4

2 Motivation for RAID Version 2 5
   2.1 Control Flow 6
   2.2 Failure Detection 6
   2.3 Concurrency Control 7
   2.4 Functional Layering 7
   2.5 Communication 8

3 RAID Servers 8
   3.1 Transaction Flow of Control 9
   3.2 Server Interfaces 10
      3.2.1 Inter-server Data Structures 11
      3.2.2 RAID Database Structure 12
      3.2.3 Action Driver (AD) 13
      3.2.4 Replication Controller (RC) 13
      3.2.5 Concurrency Controller (CC) 14
      3.2.6 Access Manager (AM) 14
      3.2.7 AM Functions 14
      3.2.8 Evaluation of Implementation Choices 15
      3.2.9 Atomicity Controller (AC) 19
   3.3 Communications 20
   3.4 Off-line Replication Management 20

4 Experimental Infrastructure 21
   4.1 Adaptability Features in RAID 22
   4.2 Benchmark Data 23
   4.3 Action Driver Simulator 26
   4.4 Open versus Closed Experiments 28
   4.5 Experimentation with Restart Policies 28
   4.6 RAID Experimental Procedure 29

5 Conclusion 33
1 Introduction

RAID [BR89a, BR89b] is a distributed database system being developed on Sun workstations running the Unix operating system. RAID is a server-based system, with each server encapsulating a distinct and well-defined subset of the services desirable in a transaction processing system. Servers are independent entities — they share no state and interact only by passing messages along established paths of communication. These messages carry the requests, replies, and data that execute transactions.

RAID is comprised of six servers (Figure 1: the User Interface (UI), the Action Driver (AD), the Replication Controller (RC), the Atomicity Controller (AC), the Concurrency Controller (CC), and the Access Manager (AM). The UI and the AD are known as the User Management servers, and the RC, AC, CC, and AM are known as the Transaction Management servers.

**User Interface** The User Interface is the server that is responsible for the presentation of the database to the user. It may be a simple command line interface, or a complex graphical display. Each user communicates with the database through a dedicated UI. Multiple implementations of the UI server may be active at the same time allowing different users to take advantage of different display environments. The UI sends a query encapsulating the user's request to the Action Driver for further processing.

**Action Driver** The Action Driver processes the input received from the User Interface, translates it into read and write actions on database items, and assigns it a unique transaction identifier. The resulting read and write actions are sent to the Replication Controller for further processing. Each user has a separate Action Driver server.

**Replication Controller** The Replication Controller is responsible for managing replicated copies of database items transparently. The RC presents a single-copy database to the User Level servers and translates their instructions into actions on the replicated copies.

**Atomicity Controller** The Atomicity Controller ensures that all transactions complete uniformly across all sites. This is done by executing a commit protocol with the ACs on other sites.

**Concurrency Controller** The Concurrency Controller checks the consistent processing of concurrent transactions. As read and actions occur the CC checks for violations of serializability. Transactions are delayed or restarted to maintain serializability, depending on the type of concurrency control running.
Figure 1: The RAID System Architecture

**Access Manager** The Access Manager is responsible for managing the physical storage of database items. When a read or write action occurs, the Access Manager issues a timestamp for that action. The timestamps are used by the Concurrency Controller for its serialization checks.

**Communications** Since RAID is a server-based system, communication is of fundamental concern. The design, reliability, and performance of the servers reflects the underlying communication design. Communication support in RAID is implemented in several layers, from UDP to high-level RAID multicast. The RAID oracle translates logical names to physical addresses. Each server maintains a cache of the addresses of its neighboring servers, using the oracle to update cache entries as neighboring servers recover or relocate.

## 2 Motivation for RAID Version 2

RAID was originally developed to facilitate the investigation of reliability issues in distributed transaction systems [BR89a, BR89b]. It has been a valuable tool for experimental studies of replication, communications, and concurrent checkpointing. However, the first version of
Raid was not suitable for some of the topics that we wished to explore because of omissions in the system design. These topics included a dynamically adaptable transaction system, and adaptability to failures. In addition to adding the capabilities necessary to approach these problems, enhancements were made in the new design that increase the flexibility and power of RAID.

2.1 Control Flow

One of the major differences between version 1 and version 2 of RAID is the structure of the communication paths. Inter-server communication is based on the RPC paradigm. All inter-server actions consist of a request and a reply. Once a request is issued by a server for a transaction, further progress on that transaction by that server is blocked until a reply is received. In some instances, the multi-RPC model is more appropriate. In those situations, one server sends the same request to several destinations before blocking. Processing resumes when all recipients of the request have responded. We use the multi-RPC model for accessing read quorums and for performing distributed commitment.

RPC-based communication is useful for error detection and correction. If a server does not receive a timely reply to a request, a time-out event occurs. A server responds to a time-out event by resending the last request sent for the transaction, or by attempting to abort the transaction. If a server receives the same request for a transaction multiple times, all of the requests after the first are handled as time-out events. Messages are otherwise idempotent.

All time-out processing in RAID is initiated by the AD — the same server that originates transactions. When a transaction arrives at the AD, a timer is set based on the number of read and write actions in the transaction. If the transaction does not complete before the timer expires, the AD generates a time-out event and resends its last request for the transaction. This may result in other time-out events at other servers until the time-out reaches the source of the actual problem. In this way, we detect errors at a high level, and can propagate time-out events quickly throughout the system.

The adoption of the RPC model has increased the number of messages required to execute a transaction. Most of the additional communication is between servers on the same site, and is less costly than communication with remote sites. Furthermore, servers on a single site may be merged into a single process resulting in low-cost local communication. This allows us to use a communication paradigm that allows effective error detection with only minor costs in transaction processing performance.

2.2 Failure Detection

RAID-V2 features increased fault-tolerance and high system availability. The on-line replication control server can opt to use quorum methods which are highly tolerant to site failures
and network partitions. A fully-replicated directory is used to implement partial replication. Partial replication contributes to an increased fault-tolerance since full replication as well as no replication are in general fault-intolerant. In addition, partial replication is a key parameter in tuning system availability and performance.

The RAID-V2 replication controller provides view information of the RAID instance through a surveillance-based failure/repair detection protocol. Since this information can be temporarily out-of-date, it is considered to be a view hint. Without a view hint, failures are more likely to be detected late in a transaction’s execution (e.g. at commit time). The availability of a view hint makes it possible to avoid a priori sites or replicas that are not available. View hints have the added advantage of being able to detect the repair of failures. This feature can be used to automatically initiate recovery procedures.

View hints also help the replication controller to learn of periods of prolonged failure. During these periods, the replication controller can prescribe a non-blocking commit protocol (like a three-phase commitment) to be used by the atomicity controller instead of a blocking two-phase commit protocol.

The RAID-V2 replication controller implements an adaptable version of the quorum consensus replication control method. Quorum parameters are structured as a RAID data relation, called the quorum relation. Updating the quorum relation dynamically changes the quorums and therefore the site access pattern. With this adaptability features, quorums can be reconfigured to avoid failures that are highly anticipated to occur (e.g. periodic maintenance, or unreliable connections).

2.3 Concurrency Control

Another major change in RAID V2 is an on-line concurrency control server. In RAID V2, the concurrency controller sees every action that is to be performed by the access manager. This allows the CC to determine transaction dependencies incrementally. More importantly, it allows it to make it possible for the concurrency controller to implement blocking algorithms, notably certain locking schemes. This extends the capabilities of the CC of RAID V1.

2.4 Functional Layering

RAID V2 was designed in a layered fashion that presents different abstractions of the database to different database servers. Because the AM performs all physical reads and writes to physical storage, it does not need to know about the distributed or replicated nature of the database. Similarly, the concurrency controller is only responsible for local data items. The replication controller and the atomicity controller are the subsystems in RAID that manage the distributed aspects of the system. The replication controller receives requests for operations on a single-copy database. It then translates these requests into operations on the replicated copies. When reading, it constructs the current value of each item
from the information supplied by the remote RCs. The atomicity controller ensures that all sites commit or abort a transaction in a uniform manner. The AC receives a list of sites that will participate in commitment from the RC. This way, the AC needs no information about the locations of data items. This layering has simplified the server interfaces and has increased the level of abstraction in the system. This simplified the coding and testing efforts considerably.

2.5 Communication

We made a number of changes for pragmatic reasons. RAID V2 improves upon the original RAID communication package by using a binary data format to encode messages instead of ASCII strings. Strings were originally used to make debugging easier, and to allow RAID to work in a heterogeneous environment of processors with different bit and byte orders. As a debugging tool this worked well, since messages could easily be examined for errors. Unfortunately, performance suffered, with as much as half of the time of a round-trip message being spent formatting and parsing the message. There are three ways to support heterogeneity with a binary data format. First, each message can be converted to a network byte order for transmission, and then be translated into the local host byte order on reception. This method requires two conversion routines: one for in-coming messages and one for outgoing messages. Second, each message can be converted into the destination byte order before transmission. This requires the sender to have conversion routines for each type of destination host. Third, each message can be sent in the host byte order of the sender, and be converted to the destination byte order on reception. This case also requires conversion routines for each type of host. The latter two approaches have the advantage that only machines with different byte orders perform message translation. The third approach has the additional advantage that the byte order of the sender can be encoded in the packet header allowing the receiver to dynamically determine the correct conversion routine needed to unpack the rest of the message. In RAID V2, we chose to use a network byte order because of the smaller number of conversion routines needed. Also, a standard network byte order — XDR — was readily available to us. We examined the performance of the communication routines and discovered that while using XDR was comparable to using ASCII strings for simple messages, XDR only required half as much time as ASCII strings for the more complicated messages used in transaction processing.

3 RAID Servers

RAID servers are independent entities. No RAID server relies on information internal to a server of a different type. This feature contributed greatly in achieving a clean, adaptable implementation and was made possible through the careful design of the transaction flow of
control and a simple inter-server message protocol. In this section, we present the transaction flow of control in RAID. Inter-server interaction is described through inter-server data structures and message protocol. Finally for each server, we describe the functional goals and outline the main features in our design.

3.1 Transaction Flow of Control

Figure 1 shows the communications paths between servers in RAID. The flow of control for a transaction that commits is as follows:

0. Transaction arrives at a AD from its UI.
1. AD processes transaction into read and write operations, reading through the RC. Write operations are saved in the AD until commit time.
2. The RC reads by communicating with its CC and with remote RCs.
3. The CC reads by communicating with its AM.
4. Tuples are returned from the AM to the CC, ...
5. from the CC or remote RCs to the originating RC, ...
6. and from the RC to the AD.
7. When the AD has completed the read phase of transaction processing it passes the list of tuples that should be updated to the RC.
8. The RC passes the update list to the AC. The update list includes the set of sites to participate in distributed commit, with read-only sites specified separately, and the list of updates, including the set of sites at which each update must be completed.
9. The AC requests precommit from all ACs.
10. Each AC requests precommit from its CC.
11. Each CC requests precommit from its AM.
12. The AMs establish write locks if possible and return the guaranteed write timestamp to the CCs.
13. The CCs check for serializability using the write timestamp, and prepare data structures to guarantee that the transaction will be kept serializable until commit or abort. The CCs reply to the ACs.
14. The ACs reply to the originating AC.

CCs that do not need write TSs, such as two-phase locking, may check precommit at step 11.
15. If the originating AC receives all 'yes' votes, it commits the transaction by writing the commit record to the log. Once the commit record is on stable storage the AC replies to the RC that the transaction has committed.

16. The originating AC tells all ACs to write the transaction updates to the database.
17. Each AC tells its CC to write the updates.
18. Each CC tells its AM to write the updates.
19. Each AM writes the updates to the database and frees the commit locks. The AM returns to the CC.
20. Each CC frees its data structures relating to the transaction. The CC returns to the AC.
21. The ACs indicate the completion of the updates to the originating AC.
22. Once the originating AC has received acknowledgements for commitment from all other ACs all information about the transaction — including log records — can be purged.

Transactions can be aborted by the originating AD or AC. Other servers may reply to any message with a negative acknowledgement, following the same path that positive acknowledgements follow. A negative acknowledgement will eventually cause the transaction to abort under many transaction processing algorithms, but cannot be assumed to be final decisions either by the server that issued the negative acknowledgement or by other servers that see the message. Leaving the final decision with the AD and AC supports future algorithms that can choose alternate approaches to commit transactions despite unavailability of some data items at some sites.

If the AD decides to abort a transaction it sends an abort message to the RC, which passes it on to all RCs involved in reads for the transaction, which in turn inform their respective CC. If the transaction has already been sent to the AC for commitment when the RC receives an abort request from the AD, the RC forwards the abort request to the AC instead. If the transaction is in an abortable state, then it is aborted. Otherwise the message acts as a timeout, causing the AC to retry the last round of messages it sent. If the originating AC decides to abort a transaction, it notifies all remote ACs involved in commit, and the local RC, which propagates it to the local AD. ACs forward the abort to their local CCs. CCs pass abort messages on to their AM. Abort messages are acknowledged like other messages so the originating site can eventually release transaction state, including log records.

3.2 Server Interfaces

This section describes the role of each RAID server in the system. Included are a description of the interface between servers, and the role of each server in the system. Specific
transaction processing methods are not mentioned for each server, since these vary between implementations.

3.2.1 Inter-server Data Structures

The control messages transmit the following data structures:

- **timestamp**: unsigned integer. Unique timestamps are assigned by the local AM to be unique across the entire system, and all timestamps are guaranteed to be non-increasing. Uniqueness is guaranteed by appending the site id to a counter.

- **transaction id**: unique timestamp read from the local logical clock when transaction was started.

- **relation id**: (database id, relation id within database).

- **tuple id**: tuple number within relation.

- **attribute tag**: enumerated type from (string, integer, float, char), and a length.

- **attribute**: union type from (string, integer, float, char).

- **tuple tag**: list of attribute tags.

- **tuple value**: list, each item of which is an attribute. The first three attributes are referred to as system attributes. Attribute 0 is the tuple id. Attribute 1 is RC-specific data (often a version number). Attribute 2 is the use flag (for the AM). Its value is zero to indicate a deleted tuple.

- **relation descriptor**: describes a sub-relation desired from the database. Includes a relation id and a list of tuple ids. A relation descriptor has additional fields that can be used to define a sub-relation in terms of key fields of the database (section 3.2.8).

- **relation**: a sub-relation of some database relation. Includes an id, a tuple tag, and a list of tuples.

- **write relation**: a list of relations.

- **relation update**: (list of write sites, relation to be updated).

- **update list**: list of relation updates.

- **RC commit request**: list of read sites, list of write sites, update list.
3.2.2 RAID Database Structure

A RAID database is essentially a collection of relations. In addition, the schema information (metadata) describing the relations is also stored as part of the database. Here we describe the components of the RAID database.

**.metadata file**  This file contains schema information of the RAID database. For each relation it specifies the relation name, number of attributes, followed by attribute information. The attribute information is specified by following three things:

1. *type:* specifies the type (an integer code) of the attribute.
2. *size:* specifies the size of the attribute (in bytes).
3. *index:* specifies whether it is an indexed attribute.

**System Relations**  There are three system relations:

- *RELATION* lists all the relations for this database.
- *ATTRIBUTE* lists the relation and attribute pairs for all the relations. In addition for each attribute, its name, type, size, and index information is maintained.
- *CONTROL_RELATION* specifies the algorithms to be used by different RAID servers.

User can perform a query on RELATION and ATTRIBUTE relation to get schema information of the RAID database.

**User Relations**  There can be any number of user relations. The user relations can have any number of attributes. Currently, the attributes of user relation must be one of the following types: integer, floating point, character, or fixed length strings.

**System Attributes**  The following system attributes are prepended to all relations:

1. *TUPLE.ID:* This unique identifies a tuple in a relation. The read and write requests for tuples are specified by (relation, tuple-id) pair.
2. *VERSION:* This attribute is used by replication controller to manage replicas of tuples.
3. *USED:* This attribute indicates if the tuple id is used or not. When a request to delete a tuple is received the used attribute field is set to false. The deleted tuple’s space can be reclaimed at a later time.

---

2 An index structure on the attribute's value should be maintained to allow quick access.
The data for each relation (including the system relations) is stored in a separate UNIX file. Each tuple is stored as a collection of ASCII strings. White space separates the strings corresponding to different attributes of the tuple. If an attribute is of type string then it is stored either as a quoted string or as an unquoted string with a special (predetermined) character as delimiter.

### 3.2.3 Action Driver (AD)

The AD is responsible for interpreting parsed transactions received from the UI. The AD executes reads as they appear in the transaction, and queues writes to be executed at the end of the transaction. Reads are handled by the RC. The final commit message is also sent to the RC.

As transactions arrive in the system the real-time clock value is recorded, and they are appended to a ready list. When the degree of multiprogramming allows, the head of the ready list is activated. A newly activated transaction requests a transaction id from the timestamp server (currently handled by the AM), and begins processing. Transactions that receive nack messages are automatically aborted and scheduled to be restarted. Statistics, including response time based on the arrival timestamp, are recorded for transactions that commit.

### 3.2.4 Replication Controller (RC)

The RAID RC is responsible for providing replication transparency and for maintaining system availability in presence of site failures and network partitions. One-copy serializability is used as the correctness criterion for maintaining mutual consistency among the database replicas.

The RC maps logical actions from a local AD into physical actions on available replicas. The mapping is done through a quorum-based interface that unifies access to a library of replication control methods. When presented with a logical action on some relation, the interface consults a fully-replicated directory to locate all existing replicas of that relation. It then uses a view hint vector to identify which replicas are currently available. The interface then passes the set of sites where replicas are currently available, along with the action needed on to the replication control method. The latter decides whether available replicas are enough to perform the action, in which case it returns a quorum of sites that is a subset of the available sites. Since it is possible to have more than one quorum, some optimization heuristics are used to select minimum size quorums, while load-balancing the RAID sites. The heuristics almost always favor quorums that include the local site.

The RC interfaces with other RAID servers as follows. It receives Read requests from local ADs or remote RCs, and StartCommit requests from local ADs. When the RC receives a Read request from its local AD, it passes it to the quorum interface. If no quorums are
available, the RC sends a NackRead back to the AD. Otherwise, it maps the AD request into requests to a quorum of physical copies, and transfers the request to the RCs on the sites that contain the physical copies (requests to the local site are passed on to the local CC). Read requests from remote RCs are for physical copies contained on the local site are also passed on to the local CC. After the RC has obtained the necessary replies from remote RCs and its local CC, it checks if all replies are positive, in which case, it returns the most up-to-date tuples to the AD. If one or more of the replies are negative, it sends a NackRead back indicating that the request can not be satisfied due to concurrency conflict.

Similar to the Read requests, the RC handles StartCommit requests by finding a quorum for each Write operation in the StartCommit request. The RC construct a commit request only if a quorum is found for every Write operation. The commit request contains the set of sites from which the transaction read (union of read quorums), the set of sites to which the transaction wants to write (union of write quorums), and an update list that contains relations with their respective write quorums. If a quorum is found for every write operation, the RC constructs and forwards the commit request to its local AC to start a commitment session. Otherwise, the RC sends a NackStartCommit back to the local AD.

3.2.5 Concurrency Controller (CC)

The CC is responsible for guaranteeing serializability of transactions that commit. The CC receives Read requests from the RC, and PreCommit, and Commit requests from the AC. The CC may delay the requests before or after sending them to the AM. The CC may also choose to reply with a NackRead or NackPreCommit, indicating that the request violates serializability or that the AM responded with a Nack.

3.2.6 Access Manager (AM)

The RAID Access Manager (AM) is implemented as a separate server which manages physical access to RAID databases. It receives and responds to read and write requests from the RAID Concurrency Controller (CC). At present a simplified version of AM has been implemented. The goal was to get the collection of the RAID servers functional as soon as possible because we needed all the servers to be operational to do any form of debugging and testing. To reduce implementation effort and time we simplified the AM design considerably, but at the same time we have avoided oversimplifying of AM so that realistic experiments can still be conducted and the data collected can be of practical use. Here we describe the functions performed by AM, and evaluate the design and implementation decisions made.

3.2.7 AM Functions

Loading and Unloading of Databases At the startup time AM reads the entire RAID database into memory which involves reading the relation data files. In addition for each
relation, B-tree\textsuperscript{39} like index structures are built for each index attribute(s)\textsuperscript{3} which index
has to be maintained. The current version of AM supports building indices both on unique
and non-unique attribute values. We use the G++ Bag class to build these index structures.
At the end of the session (that is when AM is terminated) the entire in-memory database is
written back to the secondary storage. The built in-memory index structures are discarded
at the termination of AM.

Responding to Requests from RAID Concurrency Controller (CC) After the
loading of database AM is ready to receive and respond to the requests from the CC. AM
receives a Read request in the form of a relation descriptor consisting of a relation name and a
list of tuple ids. The read request are queued if their is a conflict and subsequently when the
conflicting transaction is terminated (either committed or aborted) the tuples corresponding
to the read request are returned along with a timestamp to the CC. An Index Read request
is also received in the form of a relation descriptor. The AM responds with a list of tuple
ids matching the index read request to CC. In the next pass a read request consisting of
previously returned tuple ids is received and processed. The PreCommit request contains
a list of relations. Each relation contains a list of tuples that have to be written to the
database. For each tuple a write lock is obtained. When all the write locks are obtained
a yes is sent to the CC indicating that it can commit the request. Finally, on receiving a
Commit request, the AM goes through the list of tuples that needs to written (the possible
operations are update, insert and deletes) and updates the in-memory database accordingly.
In case of a delete operation the tuple is marked as deleted. If an Abort request is received,
all the data structures (queues, etc.) associated with that transaction are deleted.

3.2.8 Evaluation of Implementation Choices

Memory resident Database The decision to load the entire database at the start up time
simplified the design considerably. We avoided building a buffering scheme for maintaining
the most frequently accessed items, the implementation of which would have been fairly
complex. Instead, we load the entire database in memory and let the underlying operating
system (in our case UNIX) do swapping based on pages accessed. The access time for a
tuple is considerably reduced because it involves a memory access as opposed to a possible
disk access. One of the questions that needs to be answered is how does the reduced access
time for tuples affect the experimental data.

If we assume that the database loaded represents the portion of a database which is
most frequently accessed, then the experimental data obtained from RAID will not be very
different from a DBMS which uses a sophisticated buffering scheme. More specifically, if
the buffering scheme results in a high cache hit factor (percentage of times a tuple accessed

\textsuperscript{3}The attribute for which index has to be maintained.
will be found in memory) then the data obtained from the current version of RAID would be valid. On the other hand one has to be careful while analyzing experimental data which is very sensitive to parameters such as hot-spot size fraction and hot-spot access fraction. The reason is that in DBMS which support buffering, there will be substantial difference between the access times for tuples corresponding to the hot-spots vs. others, because the most frequently accessed items would be found in memory, where accessing other tuples will involve disk access. In our scheme this difference will not be observed as the entire database is kept in the memory.

The current scheme can result in large initial setup time for AM especially for large databases since the entire database needs to be loaded. This is not much of a problem since RAID is an experimental system, and furthermore the experimentation process is automated. However, we do agree that the current scheme limits the kind of experiments one can perform. For example we cannot experiment with very large databases. Also we cannot study the effect of clustering in our system. Thus in the next version of RAID we plan to implement a buffering scheme in AM.

Ignoring Updates The databases for experimentation are created by performing a series of inserts on different relations in the database. Once a database is created, user can perform experiments. Normally the updated in memory copy of the database is written to the corresponding set of UNIX files when the AM terminates. To facilitate experimentation with the same database we provide a ignore flag which when set ignores the updates, and does not write back the updated in memory database to corresponding set of UNIX files. This flag was used extensively to avoid recreation of the database during debugging and for running pilot experiments. The flag was also used in many experiments because it did not matter whether the updates were written to the disk at the termination of the AM. Moreover, it guaranteed that all the experiments will be performed with the same database.

Indices In the current design in memory index structures are maintained by the AM. The index structures are built when the AM starts up. The read request is implemented as an extension of the read request from the AD to the AM (through the RC and CC). This read request is in the form of a relation descriptor, which is shown in figure 2. In the original form, this read request directly specifies a list of tuple ids to be returned. In the extended form, indicated by a special, negative, value in the number of tuples field, the AD includes at least three additional items: a key column, a key value, and a comparator. The key column specifies which index is to be used by the AM. The key value specifies a value to be searched for in the index, and the comparator specifies a comparison function to be applied to the

4The entire database must fit into the heap area of AM process.
5Related data is clustered together on disk to reduce access time. Usually when a tuple needs to be read the page which contains the tuple is brought to memory to enhance performance.
6The time to write the updated in memory copy of the database to disk was not charged to the experiment.
A relation descriptor identifies a relation and a list of tuples within the relation. This descriptor uniquely identifies a sub-relation within the database. It is used primarily for specifying sets of tuples to be read.

```c
typedef enum {comNULL = 1, comLT = 2, comLE = 3, comGT = 4, comGE = 5, comEQ = 6, comInRange = 7, comExRange = 8} comparator_type;

typedef struct {
    database_id_type database_id;
    relation_id_type relation_id;
    int n_tuples; /* could be ALL_TUPLES or INDEX */
    tuple_id_list_type tuple_id_list; /* n_tuples */
    int column_id; /* in relation, for indexing */
    comparator_type comparator; /* method to select tuples from index */
    attribute_tag_type tag; /* type of index column */
    attribute_type key1, key2; /* keys for indexing (often only one used) */
} relation_descriptor_type;
```

Figure 2: The relation descriptor data structure.

key value and corresponding attributes from the key column of the relation. There are seven comparators supported: less than, less than or equal to, equal to, greater than, greater than or equal to, inclusive range, and exclusive range. The latter two specify range queries, and use the second key field to specify the upper end of the range.

The comparators are applied in the AM by using the index to choose a starting point and then progressing sequentially through the tuples in increasing key order until the comparator is no longer satisfied. The tuple ids of the tuples that satisfy the comparator are packed into a special relation with only one column, and returned to the AD. Index reads are treated like other reads for the purpose of replication and concurrency control. When the AD receives the response it creates a new relation descriptor, copies the tuple ids from the returned relation into the relation descriptor, and issues a second read request to read the desired tuples.

The reason for the two-pass design is that the index must be maintained in the AM so they are subject to concurrency and replication control, but used in the AD. If the matching queries that use a key that is not in the index are implemented by scanning the entire relation. These queries do not occur often in practice.

---

7Index queries that use a key that is not in the index are implemented by scanning the entire relation. These queries do not occur often in practice.
tuples were directly returned from the AM in response to the index request the CC would not have a chance to perform concurrency control on the actual read. A problem with this design is that index reads implicitly read all tuples of the relation. That is, a transaction reading the index must conflict with any transaction performing a write to any tuple in the relation — even a tuple other than those read. The reason is that any write may modify the index, which could potentially change the results of the index read. For example, a transaction that reads all students whose name is “John” must conflict with a transaction that changes all students whose name is “Ian” to “John”. The first transaction does not read the tuples with name “Ian”, but if the second transaction serializes first this is an error. One solution is to implement predicate locking in the concurrency controller to specify exactly the database tuples that are being used by a transaction. This solution has poor performance, though, and is difficult to extend to other data models such as those used by object-oriented systems.

Our solution is for the CC to read-lock the entire relation for any transaction that performs an index read. For correctness, only transactions that update an attribute in an index column need to write-lock the index. Unfortunately, it is not easy to recognize such transactions in the CC, so in our current design all writes must lock the index. Since this significantly affects potential concurrency, and since most of the experiments we performed did not modify index attributes, we include a flag to tell the concurrency controller not to lock the indices. We used this flag in our experiments.

In the next version we plan to implement the index for a relation as a special index relation. The tuples of this index relation all contain key fields from the original relation and tuple ids of other tuples in the index relation. The tuples are organized into a B-tree [Com79], with the leaves of the tree pointing to tuples of the original relation. When the AD wishes to search the index, it proceeds from the root of this b-tree issuing read requests for other nodes in the tree as necessary. When it reaches a leaf node, it records the tuple id and uses it to directly read the desired tuple. The advantage of the scheme is that concurrency control and replication control are performed for the index relations exactly as they are for ordinary relations. Transactions searching the index obtain read-locks on each node (tuple) they visit, and transactions updating the index (usually because they change attributes in the indexed column) must obtain write locks on index tuples that they change.

A Generic Interface to CC AM provides a generic interface to CC allowing CC to run any concurrency control algorithm of its choice. This is acheived by returning a timestamp for both read and precommit requests received from CC. The AM guarantees the CC that the requests for which timestamps are returned will be performed in the increasing value of timestamps. To this end AM maintains one pre-Commit queue per tuple. When a read/write request is received AM returns immediately with ACK along a timestamp if there is no conflict. In case of a conflict the request is queued in the pre-commit queue for the corresponding tuple. Subsequently when a transaction terminates (due to abort or commit) AM scans the precommit queues and processes requests which no longer conflict. At present,
the all tuples read request of a relation, and the index read request are handled slightly differently by assuming that such requests do NOT conflict with write requests. Thus we do not queue such requests in AM.

Maintaining relation data as ASCII strings It is natural to have a separate UNIX file for each relation. The design to store tuples as collection of ASCII strings was done for convenience. It helped us during debugging as we could simply view the files which contained relation data. We will need to change the storage scheme if we decide to support buffering scheme in AM. The AM can be designed to support transfer of pages between main memory and secondary storage, and the relation data can be stored in the binary format.

Recovering from Failures The current version does not support operations to recover from a failure. At present the Atomicity Controller (AC) does log the updates on to stable storage. To support recovery the AM will have to support a restart operation which will read the log written by AC and perform recovery[BHG87].

3.2.9 Atomicity Controller (AC)

The AC supervises the distributed commitment of transactions. A transaction may be committed by employing either a 2-phase or a 3-phase protocol. Selection of a commit protocol is performed on a per-transaction basis by the RC at the time that the transaction is ready to begin the commit process. The RC is not required to select a commit protocol. In such cases, the current default protocol is used.

In addition to commit protocol information, the AC receives from the RC a list of the sites that should participate in the commit process for the current transaction. This information is derived by the RC from the quorum assignments for the transaction. Finally, the RC sends the AC the list of updates that must be written to the database.

The AC also performs logging of transactions in order to ensure consistency of the database upon recovery from a site failure. Log records are forced to stable storage just prior to each transition from one commit phase to another.

The AC responds to StartCommit messages from the RC, and to a variety of messages from other ACs. The AC that received the StartCommit message coordinates the phase transitions in the commit protocol. In the initial phase of the commit protocols each AC requests a PreCommit from the CC. If the PreCommit is granted, then the AC signals that it can proceed with commitment; otherwise it indicates that the transaction should be aborted. When sufficient replies are received, the coordinator sends either a Commit or an Abort message to all participating ACs and to the local RC. After all sites acknowledge completion of the transaction, the transaction is purged.
3.3 Communications

RAID servers communicate by sending typed messages to each other. The RAID communications library enables servers to exchange information in a heterogeneous environment. The communication library was designed in a layered fashion. At the lowest level messages are sent as UDP/IP datagrams. Layered just above UDP/IP is the RAID LDG (for long datagram) layer. This layer is present to handle the fragmentation and reassembly of messages that are larger than UDP socket can handle. The LDG fragment size is tunable to experiment with the difference between user-level fragmentation and the further fragmentation of messages by IP gateways. Above the LDG layer are the procedures that support heterogeneous communication.

The RAID communications package employs a flexible naming scheme that allows RAID communications to be abstracted from the underlying transport mechanism. A RAID address is a 4-tuple consisting of a RAID instance number, a RAID site number, a RAID server, and a service instance number. This abstraction makes it much easier to port RAID to a new network architecture (e.g. from a IP-based network, to an OSI network). All mappings of RAID addresses to transport-level addresses are managed by a nameserver known as the Oracle. When a RAID server sends a message, it checks its cache for the address translation. If the mapping is not found, the oracle is consulted.

3.4 Off-line Replication Management

A RAID database is constructed off-line using the RAID support commands. The database schemas and replication information are first entered into a fill-in-the-space spec file. For each relation, the spec file contains the relation name, a list of attribute descriptors, and a list of Host-Path pairs. Three internal attributes are always part of every relation. These are the tuple_id, version_number, and the used_bit attributes. The tuple_id attribute is used to implement tuple-level granularity for the concurrency controller. The version_number attribute is used by a variety of replication control methods to identify up-to-date copies. The used_bit attribute acts as a marker for deleted tuples. An attribute descriptor consists of the attribute name, type, length, and primary key flag. The list of the Host-Path pairs specifies the locations at which a relation is to be replicated. Host is an internet domain name and Path is an absolute path name of the database directory on the associated host. For example, "/uraid10/raid/newraid/databases" is the absolute path where the "DebitCredit9" database is to be stored at the host "raid10.cs.purdue.edu".

Once the spec file is created, the database can be constructed and its relations can be initialized using the dbcreate command. The dbcreate command takes two arguments. The spec file and the name to be given to the database. Dbcreate reads in the spec file and creates the database directory and configuration file. The directory is a digestification of the replication information found in the spec file. Figure 3 depicts the RAID directory
layout. The configuration file contains a mapping of the Host-Path pairs into logical unique id's. This mapping is mandated by the RAID high-level communication routines, where servers addresses are not specified in terms of host names but rather in terms of virtual sites that are named by unique logical id's. As will be shown, the mapping is also used to automate RAID instantiation. In addition to the directory and the configuration file, dbcreate creates user and meta relations. The meta relations contains user-relation schemas. User relations are optionally initialized by inserting tuples from a specified input file. After creating all these files, remote copies the directory, the configuration file, the meta relations and the spec file itself to every Host-Path pair found in the configuration file. User relations are then remote copied according to the replication information found in the directory.

In addition to dbcreate, RAID provides commands to remove, reset, and extend the database. It also provides a powerful command that automatically starts up an appropriate instance of RAID at all hosts where a database is stored. The dbrm command destroys a database by removing all local and remote files related to that database. The dbreset command combs all the tuples from user relations and leaves the database as if it were just created with empty relations. The dbextend allows more relations to be added to the database. Finally, the raid command takes a database path and uses it to start up an appropriate RAID instance. From the configuration file, which is found in the database path, the raid command learns of all the Host-Path pairs and their associated unique logical id's. For each Host-Path pair \((H, P)\) with logical id \(k\), the command remotely creates a replication controller, an atomicity controller, a concurrency controller, and an access manager on the host \(H\). As a command-line argument to the raid command, the logical id \(k\) is passed as the virtual site number to all the instantiated servers. In addition, the replication controller and the access manager servers are passed the path \(P\), as arguments. This way, the replication controller knows how to locate the RAID directory and the access manager knows where to find the schema information. The raid command can also accept servers' command line arguments, and passes them on to the respective server when it is instantiated.

4 Experimental Infrastructure

In this section we describe the experimental infrastructure of the RAID project at Purdue University. The Raid laboratory has five Sun 3/50s, and four Sun SparcStation-1s, all with local disks connected by a 10Mb/s Ethernet. The SparcStations were acquired recently so some of the reported experiments were done on Sun 3/50s. Measurements are facilitated by microsecond resolution timers that were obtained from Zytec Corporation. Adaptability features in RAID make it possible to test different algorithms and implementation techniques under the same conditions using the same benchmarks. A single independent variable can be chosen, and can be varied over a range of values while the rest of the system remains constant.
For instance, many different replication controllers can be tested with the same atomicity controller, concurrency controller, and access manager, and under the same workload. This provides a fair comparison between the performance of the different implementations. In the following subsections, we discuss the benchmarks for distributed databases that we developed by extending the DebitCredit benchmark [A+S85]. We outline the action driver simulator which parametrizes and applies the benchmark to the RAID system. We also describe the transaction restart policy and its effects on the stability of experiments. Finally, the RAID experimental procedure is detailed.

### 4.1 Adaptability Features in RAID

Three of the RAID servers have built in adaptability features — the concurrency controller (CC), the replication controller (RC), and the atomicity controller (AC). Each of these servers implements a number of algorithms and has the mechanism necessary to convert from one algorithm to another.

The CC implements five algorithms for concurrency control: timestamp ordering (T/O), two-phase locking (2PL), generic timestamp ordering (gen-T/O), generic locking (gen-2PL) and generic optimistic (gen-OPT). The first two algorithms are implemented using specialized data structures, while the last three use the same general data structures. In the case of T/O and gen-T/O, the implementations enforce the same concurrency control policy, but one uses a generic data structure specifically designed for adaptability, while the other uses an ad hoc data structure designed specifically for T/O.

The RC implements an adaptable version of the quorum consensus (QC) algorithm [Gi79], where the quorum assignments are determined by a quorum-parameters relation. Quorum assignments may be changed by updating this relation. Many of the standard replication control methods can be expressed using this mechanism. The quorum parameters may be set to enforce the read-one-write-all policy (QC-ROWA), the read-same-as-write policy...

<table>
<thead>
<tr>
<th>Relation Name</th>
<th>Host-Path List</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch</td>
<td>raid8.cs.purdue.edu/uraid8/databases/DC</td>
</tr>
<tr>
<td>teller</td>
<td>raid9.cs.purdue.edu/uraid9/databases/DC</td>
</tr>
<tr>
<td>account</td>
<td>raid8.cs.purdue.edu/uraid8/databases/DC</td>
</tr>
</tbody>
</table>
(QC-RSW), or read-all-write-one policy (QC-RAWO), to name just a few of the possibilities. QC-ROWA requires that any read quorum must contain at least one site, and any write quorum must involve all sites. Similarly, QC-RAWO requires that any read quorum must be able to access all sites, but only one site is required to perform a write operation. QC-RSW requires both read and write quorums to be comprised of a majority of sites. Since QC-RSW does not require all sites to be operational in order to form its quorums, it provides a greater degree of availability than QC-ROWA or QC-RAWO. The RC also implements a version of read-one-write-all (ROWA) that does not employ the quorum mechanism. This allowed us to test the performance cost of our quorum implementation.

The AC implements centralized two-phase commit (2PC) and centralized three-phase commit (3PC). Transactions in the AC are independent of each other, so the selection of a commit protocol can be performed on a per-transaction basis. In practice, this selection is done by the RC, which may elect to utilize the AC default protocol.

4.2 Benchmark Data

Several benchmarks for database systems exist [BDT83, A+85]. However, for distributed database systems there are no well-accepted benchmarks. The data and the workload can be distributed among the sites in many different ways, especially in systems that support data replication. Distributed systems vary widely in their model of transactions, including support for concurrency control, reliability, and replication. Designing general benchmarks for different systems presents a difficult problem for benchmark developers.

DebitCredit Benchmark The DebitCredit (or TP1 or ET1) benchmark is described in [A+85]. DebitCredit is intended to be the simplest realistic transaction processing benchmark. There is only one form of DebitCredit transaction, representing a simple banking transaction. This transaction reads and writes a single tuple from each of three relations: the teller relation, the branch relation, and the account relation. In addition, a tuple is appended to a special write-only sequential history file describing the transaction. Figure 4 shows a DebitCredit transaction. The benchmark requires that the entire transaction be serializable and recoverable [BHGS7].

The teller, branch, and account tuples are 100 bytes long, and the history tuples are 50 bytes long. Each teller, branch, and account tuple must contain an integer key and a fixed-point dollar value. Each history tuple contains a teller, branch, and account id as well as the relative dollar value of the transaction.

Extensions to the DebitCredit Benchmark The DebitCredit benchmark is designed for a variety of sizes, with the database size decreasing with the strength of the transaction processing system. Table 1 summarizes the sizes of the various relations for different transaction processing speeds. DebitCredit is designed so the branches and, to a lesser extent the
begin
  update teller <teller-id> by <value>
  update branch <branch-id> by <value>
  update account <account-id> by <value>
  insert history <teller-id> <branch-id> <account-id> <value>
end

Figure 4: The basic DebitCredit transaction.

Table 1: Sizes of DebitCredit relations.

<table>
<thead>
<tr>
<th>TPS</th>
<th>branches</th>
<th>tellers</th>
<th>accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 10</td>
<td>1000</td>
<td>10,000</td>
<td>10,000,000</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>1,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
<td>100,000</td>
</tr>
<tr>
<td>&lt; 1</td>
<td>1</td>
<td>10</td>
<td>10,000</td>
</tr>
</tbody>
</table>

tellers, form a hot spot in the database. In our experiments we wanted to vary the hot-spot size, so we chose to make all relations the same size and explicitly choose a hot spot size for each experiment. Since the experiments were performed on low-end machines we use the database size for one transaction per second, and make all three relations 100 tuples.

The DebitCredit benchmark specifies that the database system should perform data-entry from a field-oriented screen. In the laboratory environment, we instead simulate transactions by randomly generating four values: teller id, account id, branch id, and relative value.

In order to obtain a greater variety of transaction streams, we extended the DebitCredit benchmark to support changes in transaction length, percent of accesses that are updates, and percent of accesses that are to hot-spot items. Each transaction consists of some number of actions, each of which accesses a random tuple of one of the three relations. The access is either an update of the balance field or a select on the key field. Some percentage of the updates are directed to a hot-spot of the relation, which is the first few tuples of that relation. Each transaction that performs at least one update ends with an insert to the history relation.

We built a transaction generator to generate a stream of random DebitCredit transactions based on the following input parameters:

- transactions: number of transactions to generate.
• branches: number of tuples in branch relation.
• tellers: number of tuples in tellers relation.
• accounts: number of tuples in accounts relation.
• average length: average number of actions in a transaction.
• probability long: probability that average length is used for a particular transaction. Otherwise one-fifth average length will be used. The default for the probability is one, which creates a unimodal distribution around average length. Transaction length is a normal distribution, with standard deviation 1/3 the length.
• update percent: percent of the actions that are updates rather than just selects.
• hot-spot size percent: percent of the database comprising the hot-spot. Each action is checked to see if it should be a hot-spot action. If so, it accesses tuples number \([0, \ldots, \text{hot-spot size percent} \times \text{n.tuples}]\) for the chosen relation. Note that all relations have the same hot-spot size.
• hot-spot access percent: the chance that an action on a relation will access the hot-spot of that relation.

The transaction length is bimodal, in an attempt to reflect real systems with a mix of large and small transactions. The standard deviation was chosen so that zero is three standard deviations away from the average, to decrease the number of times the normal distribution has to be truncated to avoid zero-length transactions. The hot-spot is over a fixed number of tuples across three relations. Since RAID does tuple-level locking, this is the same as having a single hot-spot in one of the relations.

This modified version of the DebitCredit benchmark is no longer restricted to a real banking application as in the original benchmark. However, it uses the same relations and the same types of actions as the original benchmark, and has the advantage of supporting transaction streams with heterogeneous characteristics.

Data Distribution RAID supports a wide range of distributions of data among sites, with differing transaction processing characteristics. Since the benchmarks are created for single-site database systems we measured a variety of different data and workload distributions.

The key in the data distribution is the degree of replication. The range is from full replication, in which each site has a copy of all data, to no replication, in which only one copy of each item is shared among the sites. Most real systems are likely to use something in between, balancing the need to have copies in case of site failures with the performance cost of performing operations on multiple sites. In our experiments we used both partial and full replication.
Verb | arguments
--- | ---
update | <select col> <select key> <update col> <update value>
insert | <new tuple value>
delete | <select col> <select key>
select | <select col> <select key>
begin | none
dern | none

Figure 5: Simple benchmarking language

4.3 Action Driver Simulator

The AD simulator is used in place of the RAID AD to provide an easily controllable workload. It processes transactions in a special benchmark language. The commands in the language all consist of a single line starting with a verb and ending with a number of arguments. The verbs are begin, end, update, select, insert, update-rel, and delete. Figure 5 shows the arguments for the verbs. Each verb except for begin and end takes a database id and a relation id as its first two arguments, so these are left out of the table. The meaning of the verbs are:

- **begin**: begin a transaction.
- **end**: begin commit processing for a transaction.
- **update**: reads tuples from the database that match a key value in a particular column. It then changes the value of the attributes of those tuples in some other column. **update** and **update-rel** are only supported for integer and float columns.
- **update-rel**: (relative update) same as update, except that the value is added to the attribute in the update column rather than just replacing the old value.
- **insert**: insert a tuple into the relation.
- **delete**: retrieve all tuples from a relation that match a key value in a particular column, and delete them.
- **select**: retrieve all tuples from a relation that match a key value in a particular column.

Transactions are specified by enclosing a number of other actions in a begin/end pair. All of the transactions from the DebitCredit benchmark can be expressed in this language.
The AD simulator accepts a parameter $\beta$ that specifies the inter-arrival rate of the transactions. $\beta$ is used as the average for an exponential random variable. When an arrival occurs the AD parses the next transaction from the input file, creates an AD transaction data structure for the new transaction, and begins executing it by issuing commands to the RC. When a transaction completes, statistics are compiled on its execution profile and its data structure is returned to a common pool. When the file is empty the AD simulator waits for the active transactions to complete, and prints its statistics.

The AD runs a timer for each transaction. It maintains a delta list of alarms of various types. Depending on the state of the transaction, the timer can be of type life-over, restart, or ignore. Life-over timeouts cause a transaction to abort and restart. These timeouts are used to resolve deadlocks and to handle lost messages. Restart timeouts cause a previously aborted transaction to restart. Ignore timeouts are used to safely disable the alarm that is currently active. Removing the alarm from the head of the alarm queue is not safe, since the alarm signal may already be on its way. Arrivals are also handled with special alarms of type arrival that are not associated with a transaction. The life-over alarms represent a transaction timeout, presumably because of deadlock or lost messages. The timeout interval is a constant number of milliseconds per action, chosen to maximize system throughput.

**Control Relation**

Some of the experiments require that algorithms be changed dynamically while RAID is processing transactions. To support dynamic adaptability, each RAID database has a special control relation. This relation contains one tuple for each site, containing one column for each server type. Each attribute is an integer representing the state of a particular server on a particular site. The interpretation of the integer is different for each server type, but in general the integer specifies the algorithm being executed by the server. The control relation is write-only, and is updated by special control transactions issued by the AD. Each of these control transactions accesses only the control relation.

The control transactions are processed like normal transactions until they are committed. At this point, each server checks to see if the transaction is a control transaction for that server. If so, the server selects the integer from the column corresponding to the server's logical site id, and interprets it independently. The control relation is fully replicated, and is set up by the replication controller so that writes occur on all sites. Since control transactions are serialized just like other transactions, there is automatic protection against multiple operators introducing an inconsistent state. Furthermore, control transactions synchronize the adaptation in serialization order.

---

8 A delta list is a list of times in increasing distance from the present [Com84]. The time for an element tells how long after the preceding element an alarm should occur.

9 If the transaction is already in commitment it may not be abortable. In this case, the AC takes the abort request as a timeout and retries the commit request for the transaction.

10 Actually, the relation can be read to learn the state of the server during debugging, but its value is not used by the servers.
Control transactions originate in the AD simulator by normal update transactions in the input file that use the tuple id instead of a key to select tuples. Usually each control transaction writes all tuples in the control relation, changing the value for one server on all sites.

4.4 Open versus Closed Experiments

The AD simulator is set up to run two basic types of experiments. In open experiments the transaction inter-arrival gap is varied to control the system load. In closed experiments the multiprogramming level is fixed. When one transaction completes another is started. Open experiments are more representative of the type of load found in on-line transaction systems. Arrivals are separated by an exponential random variable, representing, for instance, arrivals of customers to a teller. Actual applications probably fall somewhere in between these models, behaving like an open system when the load is low, and behaving like a closed system when the load is high.

We ran a series of open and closed experiments on concurrency controller and compared the information returned, to the accuracy of the confidence intervals. The results of the closed experiments were consistently easier to understand and interpret than the results of the open experiments. The problem is that at a high degree of concurrency an open system is very unstable, and at a low degree of concurrency the choice of concurrency controller does not matter since there are very few concurrency conflicts [CS84]. In summary, it is difficult to maintain a high degree of concurrency over a range of independent variable values in an open experiment, which makes it difficult to gather experimentally meaningful results.

For the concurrency control experiments using a closed system makes maintaining a high degree of multi-programming easier, which allows a better exploration of the differences between concurrency controllers over a wide range of parameter values. For the atomicity control and replication control experiments using a closed system makes it easier to maintain a constant low multi-programming level without running as many pilot experiments to establish a reasonable inter-arrival gap.

4.5 Experimentation with Restart Policies

In most applications, the successful completion of transactions is required. In such applications, transactions aborted by the transaction manager must be retried until they succeed. In order to model such behavior in our experiments, transactions aborted by the system were restarted by the AD.

Performance is sensitive to the restart policy used, since restart occurs most often during high periods of conflict, and restarting transactions raises the degree of multiprogramming. Also, if transactions are restarted too quickly they are likely to again conflict with the same transactions that caused the original restart. In RAID we delay restarts to improve the
chance that the conditions that caused the restart will have changed when the transaction restarts.

We studied eight different restart policies based on three binary attributes: rolling average versus total average response time for the mean restart delay, exponential random versus constant delay, and ethernet backoff\(^\text{11}\) versus non-increasing backoff. The total average used the average response time since the system was started as the mean restart delay. By contrast, rolling average estimated the average response time of the last few transactions. Exponential random delay used the average response time as the mean of an exponential random variable used to compute the actual restart delay. Constant delay uses the average response time directly. The ethernet backoff policy doubles the restart delay after each time an individual transaction is aborted.

We found that using the combination of non-increasing backoff, rolling average and exponential random delay methods resulted in a restart policy that was responsive and that maintained system stability. This is the restart policy that we used for all of our other experiments.

4.6 RAID Experimental Procedure

All experiments are run early in the morning, when network activity is low. All of the RAID machines are first rebooted to ensure that the experiments will run on a “clean” system. Shortly after a machine reboots a user cron\(^\text{12}\) job runs a shell script\(^\text{13}\) that sets up the experiments. This shell script reads a special directory and executes any benchmark files there. Each line of a benchmark file represents a complete invocation of RAID to process transactions from a temporary transaction file. After a RAID instance terminates, the data from the run is collected and stored. Figure 6 shows the logic of the experiment script, figure 7 shows the logic of the script that runs a single experiment, figure 8 shows the logic of the script that runs dynamic adaptability experiments, and figure 9 shows the logic of the script that runs a single instance of RAID.

The raw data is processed each morning with an AWK program [AKW]. This program scans the directory where the data files are stored, building tables of information containing counts of all messages sent between the servers. These tables are checked by the program for consistency to detect anomalous behavior, such as lost messages or excessive numbers of aborts. Finally, one line is printed for each experiment. This line summarizes the performance characteristics of the system as a whole and the interesting statistics for each of the RAID subsystems.

\(^{11}\)We call this ethernet backoff rather than exponential backoff to avoid the name conflict with exponential random.

\(^{12}\)Cron is a Unix service that arranges for processes to be invoked at specified times.

\(^{13}\)A shell script is a program in the Unix command interpreter’s (the shell’s) language.
• Try to start an oracle on the designated oracle site. If there already is an oracle on that site, the new oracle will find it and terminate during startup.

• Unmount all remote-mounted file systems. This step reduces the possibility of remote machine failures affecting the experiments.

• Mount the two file systems needed for the experiments.

• Find the benchmark directory for this machine on this day. If it is empty or non-existent, terminate.

• Look for an initialization file (name "Initialize") in the benchmark directory. If there is such a file, run it. These files initialize the database, usually by running dbreset to clear all relations and then loading freshly generated tuples.

• For each benchmark script in the benchmark directory (recognized as a file name with a ".bm" suffix):
  - For each line in the file:
    * invoke the DebitCredit script with the specified arguments.

• Re-mount the remote file systems.

Figure 6: Start Experiment Script Logic

• Generate a transaction benchmark according to the parameters of the experiment, using the transaction program.

• Write a command file for the AD simulator that sets its parameters (arrival gap, timeout, maximum concurrency, restart backoff method, and open/closed experiment type).

• Invoke the RAID script, handing it additional parameters and the name of the command file for the AD simulator.

Figure 7: DebitCredit Script Logic
• Generate a short transaction benchmark according to the parameters of the experiment, using the transaction program.

• Prepend a control transaction to convert to the initial concurrency control method to the transaction benchmark file.

• Append a control transaction to convert to the final concurrency control method to the transaction benchmark file.

• Append another short transaction benchmark to the end of the transaction benchmark file to keep the load steady while conversion is occurring.

• Write a command file for the AD simulator that sets its parameters.

• Invoke the RAID script, handing it additional parameters and the name of the command file for the AD simulator.

Figure 8: Converting DebitCredit Script Logic

• Choose a unique RAID instance number for this experiment.

• Clean oracle entries for the chosen RAID instance.

• Start the RAID instance.

• Busy-wait until two consecutive oracle list commands report the same number of registered servers. (An alternative would be to check the configuration file for the database to find out how many sites are involved.)

• Check the server log files to make sure they all initialized correctly.

• Run the AD simulator using the specified benchmark transactions.

• Terminate the RAID instance, and clean up the oracle.

• Move the server log files to an archive directory, timestamping them with the date and time of the experiment.

Figure 9: RAID Script Logic

31
All I/O activity during an experiment is directed to the local disk. Such activity consists primarily of database accesses and updates, and writes to the transaction log. Pilot experiments in which some data were directed to a shared file server were successful when only about half the machines (4-5) were involved in experiments, but had poor confidence intervals when more machines were active. Each server also keeps a log of statistics which is written during system startup and system termination. Server logs do not impact transaction processing performance and therefore are directed to the file server.

On the Sun 3/50s care was taken to make sure that the screen was blanked when the experiments were run. The video monitor uses the same bus as the CPU to access memory, resulting in approximately a 25% slowdown when the screen is not blanked.

We scanned the system log files to learn about automatic system activity that might disturb the experiments. The experiments were scheduled to avoid nightly and weekly distribution of software to the workstations, and to be after the nightly file system backups.

Each experiment involved running 250 transactions on the system. 250 was chosen as a reasonable number that yielded approximately steady-state measurements despite the start-up and tail-off times. There was little difference between running 250 transactions and running 300 transactions. We did not run higher numbers of transactions to economize on computing resources.

Experiments were run on the extended Debit-Credit benchmark using five independent variables:

- transaction length: the number of actions in a transaction.
- fraction updates: the fraction of the actions that are writes.
- inter-arrival gap: the delay between the arrival of one transaction and the arrival of the next for open experiments.
- multi-programming level: the number of transactions running at a given time.
- hot-spot access fraction: the fraction of accesses that access the hot-spot.
- hot-spot size fraction: the fraction of the database that comprises the hot-spot.

For each of these independent variables, we measured the performance of the system in terms of the dependent variable throughput, expressed in transactions per second.

Unless otherwise indicated, experiments were run on a system that used a timestamp order concurrency controller, a read-one-write-all replication controller, and a two-phase commit protocol. All experiments are "closed" with a fixed degree of multiprogramming. The degree of multiprogramming was set to a small number (3) to minimize serialization conflicts. Aborted transactions were restarted after a delay that was computed using an exponential random distribution with the rolling average of transaction response time as the
mean. In each experiment the workload was provided by a single AD running on one of the sites. Multiple workload experiments would also be interesting, but are more difficult to synchronize and parameterize.

5 Conclusion

The implementation of Raid-V2 system has been completed. The system is being used for experimentation and for building the O-Raid distributed object-oriented database system [DVB89]. This is a draft version of the report.

References


