Raid Programmers's Manual

Bharat Bhargava

Purdue University, bb@cs.purdue.edu

Report Number:
87-709
RAID Programmer's Reference Manual *

Edited by
Bharat Bhargava
Paul Noll

Technical Report Number CSD-TR-709
Department of Computer Sciences
Purdue University
W Lafayette, IN 47907

September, 1987

*This research is supported by NASA and AIRMICS under grant number NAG-1-976, by U.S. Department of Transportation contract number DTRS-57-85-C-00099, and by a David Ross fellowship.
CONTENTS

Contents

1 Preface

2 Introduction to RAID

3 RAID Communications
   3.1 The Name Space
   3.2 The RAID Oracle
   3.3 RAID Communications Facilities

4 RAID Server Interface
   4.1 Transaction Execution Flow

5 Data Description
   5.1 RAID Query Language Grammar
   5.2 RAID Directory Structure
   5.3 Commit Request Message Format
   5.4 Transaction Specific File Formats

6 Processing Details
   6.1 Processing by the Action Driver
      6.1.1 AD Data Structures
      6.1.2 AD to AM Interface
   6.2 Replication Control
   6.3 Access Manager

7 Using RAID
   7.1 Installation
   7.2 Example Test Session
   7.3 Changing RAID

8 Software Guide
1 Preface

This manual is intended to serve as an introduction and guide for programmers who are working with the RAID distributed database system. It is a collection of the documentation and writings of the many people who have worked on developing the RAID system. A primary source for this manual is the RAID report [3].

2 Introduction to RAID

RAID is a robust and adaptable distributed database system which serves as a test bed for conducting scientific experiments. The experiments are designed to provide empirical evaluation of design and implementation of replicated copy control algorithms, site failure and network partitioning management protocols, dynamic reconfiguration techniques, adaptable architectures, communications software, and transaction processing support in operating systems. These experiments identify principles that are necessary for reliable, high-performance transaction processing in a reliable manner.

Our goal is to build a prototype system with all the support for transaction processing in a laboratory environment, so that experiments can be conducted. A principal design goal of RAID is adaptability. For this reason, RAID is a very modular system and allows for different configurations, via switching of transaction processing algorithms. For example, one can run either a two-phase or a three-phase commit protocol. A variety of concurrency control and recovery algorithms can also be used. Indeed, each site in a distributed database system can have different optimal facilities for recovery and concurrency.

RAID provides complete support for transaction processing, including transparency to concurrent access, crash recovery, distribution of data, and atomicity. An instance of RAID can manage any number of virtual sites distributed among the available physical hosts. Each virtual site consists of several servers necessary for transaction processing. The site is virtual since its servers can reside on one or more hosts and since the site is not tied to any particular host on the network. For example, by distributing the servers that communicate via the communication package, one can easily run a ten-site RAID instance on five workstations. Furthermore, two or more separate instances of RAID can run independently.

Each site implements facilities for query parsing and transformation to the transaction format, access management with stable storage, transaction execution, concurrency control, replicated copy management, site failure and network partitioning management, naming, etc. The name server, called the oracle, provides location independent addressing for all servers. The oracle keeps track of sites leaving (say, due to failures) and/or joining the RAID instance.

Each site uses the RAID communications package, which is a library of high-level routines called by the RAID servers to communicate. High-level calls include reliable multicast, necessary for distributed transaction commitment.
3 RAID Communications

3.1 The Name Space

A RAID virtual site consists of a number of different servers and a copy of a fully replicated database. The servers are the atomicity controller, the concurrency controller, the access manager, the action driver, and the replication controller, which are described separately. RAID servers need to communicate both with servers at the same virtual site and with servers at other virtual sites within their instance of RAID. We can uniquely identify each RAID server with the tuple (RAID instance number, RAID virtual site number, server type, server instance). The last tuple element is necessary because some server types can have multiple instances active simultaneously. Figure 1 shows a possible distribution of the servers among several physical hosts. Note that a single host can support multiple virtual sites, and that different servers of a single virtual site can reside on different hosts.

A server address consists of the pair (machine name, port number). Servers wishing to communicate with each other must know each other's address. This can be done very simply by assigning each RAID server a fixed machine and port number. However, this scheme is inflexible and impractical because well-known ports cannot easily change once they have been allocated. Moreover, well-known ports are usually handed out sparingly by the central authorities, so a large number of well-known ports would be difficult to get in an environment not solely dedicated to RAID.

Our solution uses a mechanism called an oracle to give each server at startup time a random port to listen on, and to make the name of this port available to other servers.

3.2 The RAID Oracle

An oracle is a server process listening on a well-known port for requests from other servers. The two major functions it provides are registration and lookup. A server registers by telling the oracle its name (i.e., the tuple which uniquely identifies it). Any server can ask the oracle for the lookup service to determine the location of any other server.

Once a server obtains an address for another server, it stores it in its cache of addresses. Thus, the oracle is needed only at server startup time or relocation time. An entry within the cache of addresses has the following format:

```c
struct sv {
    int factive;      /* have we heard from this entity? */
    int wRAID, wVS, wType, wS;    /* RAID identification tuple */
    struct sockaddr_in sa;    /* server's socket address */
    struct sv *next;       /* next server in chain */
};
```
Figure 1: A possible distribution of the RAID servers among physical hosts. These servers represent four sites from a single RAID instance.
There are four routines which must be called by a RAID server that wishes to communicate with other servers. The routines are available by linking the server object files with the RAID communications library. The routines are Setup(), RegisterSelf(), FindAll(), and FindPartner().

The oracle also provides other services, namely server deletion notification and new server notification. The latter means that whenever a server registers itself, all servers are notified of that fact. Note that server relocation can be done by deleting an existing server, and creating a new one on the desired site. The oracle will automatically notify the affected sites. We have also implemented facilities to allow a new server to present to the oracle a notification set along with its registration request. Then, each time a server disappears or is created, those servers that include that server in their notification set are informed. This facility will make the FindAll() and FindPartner() routines obsolete. With these new facilities, notification of server deletion and creation are handled automatically. The RAID programmer does not have to handle server notification events.

The oracle currently is not fault-tolerant. Oracle failure, except at system startup time, is not a disaster. It only means that the system becomes static, i.e., servers cannot be registered or relocated relocate themselves (the only type of server relocation happens when a server crashes and recovers: it may not reappear at the same physical address). However, we have considered methods to make the oracle more robust. Some possibilities are to have a backup oracle, make the oracle decentralized, or to arrange for oracle reelection if sites detect an oracle failure.
Setup

NAME
Setup - initialize RAID communications

SYNOPSIS
int Setup(wRAID, wVS, wType, wS)
int wRAID, wVS, wType, wS;

DESCRIPTION
To initialize its communications, a server must call Setup. It takes four integer arguments, namely the RAID instance wRAID, the virtual site number wVS, the server type wType, and the server instance wS. The actual value of the server instance number is not important, but two instances of a server type should have different instance numbers.

Setup performs the following functions:
- Initialize UDP communications by finding a datagram socket for the server to listen at.
- Locate the RAID oracle through the network service database, send it a message, and wait for a reply from the oracle to confirm that this server can communicate with the oracle.

EXAMPLE
The AC for virtual site (2,1) (RAID number 2, virtual site number 1) would make the following call:
Setup(2, 1, TYPE.AC, 0)

DIAGNOSTICS
If all functions are successfully completed, a value of OK is returned. Otherwise, a value of ERROR is returned.
RegisterSelf

NAME
  RegisterSelf - register a server with the oracle

SYNOPSIS
  int RegisterSelf()

DESCRIPTION
  A server needs to do RegisterSelf if it wants to make itself known to other RAID servers. This routine first creates an oracle message which contains a service name based on the identification tuple (RAID instance, virtual site number, server type, server instance) and the UDP socket address on which this server is listening. Then, this message is sent to the oracle and the routine waits for an oracle reply to acknowledge the service registration.

DIAGNOSTICS
  A value of ERROR is returned if unable to send to or receive from the oracle or if the oracle indicates that it was unable to register the service. Otherwise, a value of OK is returned.
**FindAll**

**NAME**
FindAll - query the oracle for all servers of specified type

**SYNOPSIS**

```c
int FindAll(wType)
int wType;
```

**DESCRIPTION**

FindAll is used by certain server types (e.g. ACs) to find every other server of a certain type that is now running in the current instance of RAID. This routine takes a symbolic server type (e.g. TYPE_AC) as an argument. This routine first creates an oracle message which contains a service name based on the identification tuple (RAID instance, WILDCARD, wType, WILDCARD). This message is then sent to the oracle, which replies by sending one message for each service which matches the specified name. Each oracle reply message contains the full service name and the UDP socket address on which the service is listening. FindAll adds this information to its cache of server addresses.

**NOTES**

RegisterSelf must be called before this routine.

**DIAGNOSTICS**

A value of ERROR is returned if any errors are detected in oracle communications. Otherwise, a value of OK is returned.
FindPartner

NAME
  FindPartner - find a companion RAID entity

SYNOPSIS
  int FindPartner(wType)
  int wType;

DESCRIPTION
  FindPartner is used by a server to find the address of another server of type wType
  in the current virtual site.

EXAMPLE
  An AC would find its partner CC by the following call:
    FindPartner(TYPE_CC)

  This routine first creates an oracle message which contains a service name based on
  the identification tuple (RAID instance, virtual site number, wType, server instance).
  Then, a loop is entered and messages are exchanged with the oracle until the specified
  partner has registered itself. When the partner is found, the loop is broken and
  the partner's service name and UDP socket address are added to the cache of server
  addresses.

DIAGNOSTICS
  A value of ERROR is returned if any errors are detected in oracle communications.
  Otherwise, a value of OK is returned.
3.3 RAID Communications Facilities

RAID servers communicate with each other using high-level operations. These operations are implemented as subroutines that understand and manipulate RAID data structures. For instance, the routine used by an AC to send transaction histories to a CC is `SendCCO()`. A CC has a `SendACO()` routine to send commit/abort messages to an AC. Any sending routine uses the `SendPacket()` operation to interface with the low-level RAID transport protocol. Also, the `ServActive()` and `ServAddr()` operations are available to assist in transmission preparation. Any server can use the `RecvMsg()` operation to wait for messages from other servers. It is important to note that all packets sent across the network are ASCII strings, although machine-dependent representations are used on each site.

The above high-level facilities are implemented on top of the low-level RAID transport protocol. We call this protocol `LUDP`, where L stands for “Long.” This protocol is identical to UDP except that there is no restriction on packet sizes (all implementations of UDP restrict packet sizes to some maximum length). LUDP is built on top of UDP. Each LUDP packet is fragmented if necessary, and then sent using UDP. At the destination, fragments are collected and reassembled; each time a fragment arrives a check is made to see if any packet is complete; if so, that packet is passed to the waiting receive.

The interface to LUDP consists of two routines `sendto_ludp()` and `recvfrom_ludp()` that correspond to the UNIX UDP interface `sendto()` and `recvfrom()`. Address binding is done as for UDP. The high-level facilities use LUDP in exactly the same way that they would use UDP, but without worrying about packet size restrictions.

Since the LUDP is built on top of the UDP/IP layer, it may be more expensive. On the other hand a reasonable implementation ought to be able to make small packets get through in about the same time as UDP, with a minimal cost of one more copying of a buffer. Some of the advantages of our approach are:

- easy and quick
- no changes to the kernel
- portable (VAX and 3b2)
- good interface is provided for applications software

In terms of reliability, the LUDP should perform essentially like UDP (i.e. packets should almost always get through; if they get through they should be uncorrupted.) As in UDP, LUDP does not make any guarantee about the order in which packets get through. LUDP deals with the possibility of partial packets (fragment loss) in the following manner: when the first fragment of a packet arrives start a timer; when the timer goes off, if the entire packet has not arrived, drop it. This forces each fragment packet to have the following information:
RAID Communications

1. packet id
2. length of total packet
3. its offset in the packet.

Each packet has the following format:

| RAID instance | virtual site number |
| server type id | site session number |
| packet id | packet length (does not include packet or fragment headers) |

. fragment . . | fragment offset |
header . . | fragment length |

fragment data

Packet Transmission Design

if packet size <= max permissible
    build packet header
    send out packet with UDP
else (packet size > max permissible)
    build packet header
while (remaining packet size > 0)
    build fragment header
    copy min(max permissible, packet remnant) to buffer
    send packet
    decrement remaining packet size

Packet Reception Design

set alarm for timeout interval (actually higher level does this)
parse packet header
if packet length = fragment length
    copy data to user buffer and return with packet header info
find entry for fragment in partial packet list
    if first fragment
check that users buffer is big enough for packet
initialize partial packet info
set interval timer
set packet arrival time
if last fragment copy data to user buffer and
return packet header
if timeout alarm sounds, return with TIMEOUT (but keep unfinished packets around)
every now and then garbage collect to remove packet fragments that
have been around too long but aren’t complete

Figure 2 shows the layered structure of the RAID communications package. This design
allows lower layers to be replaced without affecting the interface used by the rest of the sys­
tem. For instance, the LUDP could be replaced by a different protocol for large datagrams
such as VMTP [4], to improve performance, portability, or reliability.
Figure 2: The RAID communications package.
SERVER LOOKUP

NAME
ServActive, ServAddr - server lookup operations

SYNOPSIS
int ServActive(wRAID, wVS, wType, wS)
int wRAID, wVS, wType, wS;

struct sockaddr_in *ServAddr(wRAID, wVS, wType, wS)
int wRAID, wVS, wType, wS;

DESCRIPTION
ServActive and ServAddr look up the specified server in the cache of server addresses.
Server identification is based on the tuple (wRAID, wVS, wType, wS).

DIAGNOSTICS
ServActive returns a value of TRUE if the specified server is found in the cache.
Otherwise, a value of FALSE is returned.
ServAddr returns a socket address for the specified server if it is found in the cache.
Otherwise, a value of NULL is returned.
SendPacket

NAME
SendPacket - send a RAID packet

SYNOPSIS
int SendPacket(sbPkt, wRAID, wVS, wType, wS)
char *sbPkt;
int wRAID, wVS, wType, wS;

DESCRIPTION
SendPacket provides an interface to lower level communication routines in order to
send the RAID packet sbPkt to the RAID address (wRAID, wVS, wType, wS)

EXAMPLE
PingAC(wVS, sbSeq)
int wVS;
char *sbSeq;
{
    sprintf(sbTran, "%c%s", PING_REPL, sbSeq);
    return(SendPacket(sbTran, RAIDNum, wVS, TYPE_AC, 0));
}

DIAGNOSTICS
A value of ERROR is returned if the specified server is inactive or the call to sendto1dg
fails. Otherwise, a value of OK is returned.
**RecvMsg**

**NAME**

RecvMsg - receive a packet

**SYNOPSIS**

Unique RecvMsg(sbMsg, pRAID, pVS, pType, pS)
char **sbMsg;
int *pRAID, *pVS, *pType, *pS;

**DESCRIPTION**

RecvMsg provides an interface to lower level communication routines in order to wait for and receive a packet arriving from the network. When the packet arrives, it may be an oracle packet or a packet from another RAID entity. In the former case, the routine handles the information from the oracle. In the latter case, the packet buffer is handed up to the calling routine.

**EXAMPLE**

GetReply(pmsg, wRAID, wVS, wType, ws, Reply)
char **pmsg;
int wRAID, wVS, wType, wS;
Unique Reply;
{
    int wRAIDTmp, wVSTmp, wTypeTmp, wSTmp;
    Unique dgType;
    wRAIDTmp = wVSTmp = wTypeTmp = wSTmp = ERROR;
    while((dgType = RecvMsg(pmsg, &wRAIDTmp, &wVSTmp, &wTypeTmp, &wSTmp))
        != ERROR)
        /* check who the message is from and what it is */
    if (wRAIDTmp == wRAID && wVSTmp == wVS && wTypeTmp == wType &&
        wSTmp == wS && dgType == Reply) return(OK);
    return(ERROR);
}

**NOTES**

This routine won't always return when a network packet arrives. It only returns when a RAID packet is received.

**DIAGNOSTICS**
The type of message is returned after successful reception along with the identity of the RAID entity. The identity is passed up to the calling routine through the parameters pRAID, pVS, pType, and pS. If necessary, the RAID packet is passed up through the parameter sbMsg.

If any type of error is detected, a value of ERROR is returned.
NAME
recvfrom.ldg - receive a datagram of arbitrary length

SYNOPSIS
int recvfrom.ldg(s, pbuf, len, flags, from, fromlen)
int s;
char **pbuf;
int len, flags;
struct sockaddr_in *from;
int *fromlen;

DESCRIPTION
This routine allocates space for and builds an arbitrary size packet, and then sets buf
to point to the packet.
The parameters, return values, and semantics of recvfrom.ldg are identical to those
of the BSD-Unix routine recvfrom with the following exceptions:

- pbuf is declared (char **) instead of (char *)
- len is ignored

SEE ALSO
recv(section 2 of BSD-Unix PRM)
NAME
sendto_lgd - send a datagram of arbitrary length

SYNOPSIS
int sendto_lgd(s, msg, len, flags, to, tolen)
int s;
char *msg;
int len, flags;
struct sockaddr_in *to;
int *tolen;

DESCRIPTION
This routine sends the packet msg of length len to the destination to.
The parameters, return values, and semantics of sendto_lgd are identical to those of
the BSD-Unix routine sendto with the exception:
• to must be declared as (struct sockaddr_in *) instead of (struct sockaddr *
).

SEE ALSO
send(section 2 of BSD-Unix PRM)
4 RAID Server Interface

Each of the RAID modules is implemented as a separate server, communicating with other modules via datagrams. The servers are currently implemented as single Unix processes.

Messages in RAID are null-terminated Ascii strings. Using Ascii incurs a conversion overhead, but provides a standard network data type that is suitable for all machines with eight-bit bytes. The text of the message consists of alphanumeric characters. Since the text of most messages is human readable, debugging is easier. The message header includes the type of message, which determines how it should be parsed.

An important design consideration is the fact that RAID servers must not expect synchronous messages. In particular, a server waiting for a reply to one message should at the same time be able to handle both requests from other servers and replies to earlier messages. In order to provide this property, the servers are implemented as stateless. A stateless server maintains a queue of the outstanding requests that it is processing. Whenever it receives a reply message, it checks the queue the outstanding request the reply refers to, and processes it accordingly. Whenever it receives a request message, it does the local processing needed to handle the request and replies to it.

As previously stated, a RAID virtual site consists of several servers necessary for transaction processing. Figure 3 depicts the organization of a RAID site. The functions of each server are defined on the following pages.
USER INTERFACE

NAME
User Interface (UI) - a RAID server

SYNOPSIS
raid RAIDNum VSNum

DESCRIPTION
User Interface is a front-end invoked by a user to process relational calculus (QUEL-type) queries on a relational database. The RAID instance number is specified by RAIDNum and the virtual site is specified by VSNum.

UI initiates its own Action Driver (AD), parses queries submitted by the user (for the time being, the UI only accepts a query file name) and gives each parsed query to the AD. It then waits to hear from its AD whether the query committed or failed, and notifies the user of this fact. Multiple users can use a virtual site simultaneously, hence for each user in the site there will be one UI/AD pair.

User Interface terminates when its user exits. The user indicates this by entering "endraid".

The stateless property of the UI is manifested in its ability to accept and parse any user input.
ACTION DRIVER

NAME
Action Driver (AD) - a RAID server

SYNOPSIS
act RAIDNum VSNum SNum

DESCRIPTION
Action Driver executes database commands. The RAID instance number is specified by RAIDNum, the virtual site is specified by VSNum, and the server instance is specified by SNum.

Action Driver accepts a parsed query in the form of a tree of actions from its User Interface and executes the transaction reading data from the local copy of the database. Currently, the database is fully replicated at each site. AD assigns the transaction an identifier that is unique among all transactions in the system. Transaction execution produces a transaction history consisting of read and write actions. After execution completes, the transaction history is sent to Atomicity Controller to determine whether it is globally serializable\(^1\). If AC replies "abort," then AD informs its User Interface that the query failed. If the decision is "commit," then AD determines all the data copies that need to be updated for that transaction, and sends it to every Access Manager in the system (since the database is fully replicated). AD then informs its UI that the query has succeeded. The AD lives as long as it is needed, i.e. until the user exits.

The stateless property of the AD is based on its ability to process any command in the query language.

\(^1\)Note that RAID currently uses this validation (or optimistic) approach to concurrency control. Validation has performance advantages in distributed systems, since it communicates the entire read/write-set in one message. All concurrency control algorithms are modified to fit the validation mold.
ATOMICITY CONTROLLER

NAME
Atomicity Controller (AC) - a RAID server

SYNOPSIS
acd -r RAIDNum -s VSNum

DESCRIPTION
Atomicity Controller guarantees that a transaction is uniformly committed or aborted on all sites in the system. The RAID instance number is specified by RAIDNum and the virtual site is specified by VSNum. Note that these are required arguments and are not optional.

After a transaction requests a commitment decision from its local AC, AC sends the read-set and write-set to all other ACs. Each AC sets special commit locks for the items in the write-set. Since RAID uses validation concurrency control, these locks are necessary during the commit phase only, to prevent modifications while the concurrency controller is making its decision. The ACs pass on the information to their CCs, which make the local concurrency control decisions. Then, the originating AC collects votes from all other ACs, makes the commit decision, and distributes the decision to the other sites.

MESSAGE PROCESSING STATES

ERROR A bad datagram is received and dropped.
PING_REPL A message is sent to this AC in reply to a ping request.
PING_REQ A message is sent from another RAID server to request a reply to a ping packet. AC should respond with a PING_REPL message.
TIMEOUT A message reception times out. The AC uses pings to determine if another server has failed.
RC_RECOVERED A message is sent by the local RC to indicate that this site has completely recovered (i.e., all data copies are up-to-date). The AC does not send any more read sets to the RC for validation.
AC_WRITESESVECT A message containing an updated session vector is sent by another AC. This AC updates its own session vector.
AD_REQUEST A message is sent by a local AD to request transaction verification. If the site is not completely recovered the AC passes the transaction to the RC for read set validation (msgtype = AC_CHKFLOCKREQ).
RC_REPLY A message is sent by the local RC with the the results of read set validation. If the read set is invalid, the transaction is aborted and the AC informs the requesting AD. This case is ignored if the site has all data copies up-to-date.
CONTROL_TRANS At this point the AC receives a control transaction or the read set of a new transaction is verified. The AC adds the transaction to its list of transactions and forwards it to the CC for concurrency control validation (msgtype = CC_TRANS_CODE).

AC_REQUEST A message is sent by another AC to request transaction verification. This AC adds this transaction to its list of transactions and forwards it to the CC for concurrency control validation (msgtype = CC_TRANS_CODE).

AC_REPLY A message is sent by another AC to answer this AC's request for transaction verification. The AC determines if the transaction can now be committed.

CC_REPLY A message is sent by the local CC with an indication of the concurrency control validation of a transaction. The AC determines if the transaction can now be committed.

ABORT A message is sent by the AC on the originating site that a transaction is being aborted.

COMMIT A message is sent by the AC on the originating site that a transaction is being committed.

RC_TYPE1_REPLY A message is sent by a remote RC with a reply for this AC's recovery announcement. The AC gets the session vector from the reply and forwards the rest of the reply to the local RC (msgtype = AC_NEWFLOCKS).
NAME
Replication Controller (RC) - a RAID server

SYNOPSIS
```
rc -r RAIDNum -s VSNum
```

DESCRIPTION
Replication Controller solves the replicated copy control problem, i.e., it maintains consistency of the replicated copies of the database even in the event of multiple site failures. The RAID instance number is specified by RAIDNum and the virtual site is specified by VSNum. Note that these are required arguments and are not optional. The RC tracks the availability of up-to-date data within the system through the use of fail-locks [1]. When a site has failed, a fail-lock is set on behalf of the failed site by the remaining operational sites for each data item which is updated during the failure period. This approach ensures the replication of fail-locks and provides more fault-tolerance for the system. When a site recovers, it collects fail-locks from the other operational sites. The fail-locks are used by the recovering site to distinguish outdated data items from up-to-date data items. Up-to-date data items are made immediately available for transaction processing. When an attempt is made to read an outdated data item, RC issues a copier transaction to refresh the data. A copier transaction reads an up-to-date copy on an operational site and writes the up-to-date copy on all sites. The good copy is written to all sites to clear the fail-locks for the data item to be cleared on all sites. Replication Controller is long-lived.

MESSAGE PROCESSING STATES

**AC_REPLY** A message is sent by the local AC to inform the RC that a transaction has been committed or aborted. After commitment, the RC updates fail-locks for data item in the write set of the transaction. The RC also checks if the committed transaction was a copier transaction initiated from this site. If the copier transaction aborted, the RC informs the AC that the original transaction cannot be completed (msgtype = RC.REPLY). If the copier transaction committed and it was the last copier generated for a user transaction, then the RC informs the AC that the database transaction can now be restarted.

**AC_CHKFLCKREQ** A message is sent by the local AC to request verification of the data-items in the read set of a transaction. The RC checks for a fail-lock on each data-item and generates copier transactions to refresh out-of-date copies. If all data-items in the read set are up-to-date on this site then the RC informs the AC that the transaction can proceed with a RC.REPLY message.
AC_TYPE2TORC A message is sent by the local AC when it detects or is informed of a site failure (via a control transaction). A message of this type causes the RC to read the updated session vector and to set fail-locks for the written data items in the last committed transaction.

AC_TYPE1TORC A message is sent by the local AC after it has received a control transaction for recovery announcement (type 1) from some site which is coming up. The RC reads the updated session vector, formats a RC_TYPE1REPLY message containing the session vector and the current set of fail-locks, and sends the message to the recovering site.

AC_NEWFLOCKS A message is sent by the local AC when this site is recovering and it has received a message containing a new session vector and the set of fail-locks in response to a recovery announcement. The RC reads the new session vector and extracts the set of fail-locks from the message. The RC can now process any messages which were enqueued during the window of vulnerability (i.e., the time between the startup of the RC and the reception of fail-locks).

AC_READSESVECT A message is sent by the local AC after it has updated the session vector. The RC reads the session vector.
NAME
Concurrency Controller (CC) - a RAID server

SYNOPSIS
ccd -d debug -m method -r RAIDNum -s VSNum

DESCRIPTION
Concurrency Controller checks if a transaction history is locally serializable at a given site. The debugging level is specified by debug. The concurrency control method is specified by method. Current values are
1. One User System
2. Simple Locking
3. Read/Write Locking
4. Two Phase Locking
5. Cyclic Graph Detection

The RAID instance number is specified by RAIDNum and the virtual site is specified by VSNum. Note that these are required arguments and are not optional.

The CC receives transaction histories from its local AC. It maintains a list of all relevant previously committed transactions, and uses this list to check if the new transaction is serializable [5]. A transaction T can be purged from this list when there is no currently executing transaction that started before T ended. CC is adaptable, and can be set to use one of several concurrency control methods [2]. Adaptability will be implemented for other servers in the future. Concurrency Controller is a long-lived server, and implements validation (optimistic) concurrency control.

MESSAGE PROCESSING STATES
CC.PING_CODE A message is sent by another RAID server to verify the existence of the CC. The CC responds with a PING.REPL message.
CC.TRANS_CODE A message has been sent with a transaction packet. The CC adds it to its list of transactions and performs concurrency control validation. A response is sent by the CC with a CC.REPLY message.
ABORT A message is sent when a transaction has been aborted. The CC should remove it from its list of transactions.
COMMIT A message is sent when a transaction has been committed. The CC should add the transaction to its commit list.
'r' or 'R' A message consisting of a recovery packet is sent.
"s" or "S" A message consisting of a shutdown packet is sent.

"f" or "F" A message consisting of a flush packet is sent. The CC deletes all transaction lists and initializes the concurrency control method.

"m" or "M" A message consisting of a change method packet is sent. The CC changes its concurrency control method to the type indicated in the packet.

"P" A message is sent to request the PID of the CC. The CC replies with a message containing its PID.
NAME
Access Manager (AM) - a RAID server

SYNOPSIS
am RAIDNum VSNum

DESCRIPTION
Access Manager provides write access to the local database, and works with AC to ensure that updates are posted atomically. The RAID instance number is specified by RAIDNum and the virtual site is specified by VSNum.
Note that read requests go directly from ADs to the local database to permit parallel access to the fully replicated database, although the same access program is used by both AM and AD. Every time a transaction is deemed committable, the AM hears about it directly from the AD which originated the transaction. The message includes any data that needs to be written. AM performs the writes, and informs the AC after their completion. Access Manager is a long-lived server (that is, it lives as long as its RAID instance lives).

MESSAGE PROCESSING STATES
ERROR A bad datagram is received and dropped.
TIMEOUT A timeout occurs but the AM should take no action.
AD REQUEST A message is sent by a local AD to indicate that a transaction has been committed. The AM writes the write set of the transaction into the database.
CLOCK SERVER

NAME
Clock Server - a RAID server

SYNOPSIS
cs RAIDNum VSNum

DESCRIPTION
The RAID Clock Servers provide approximately synchronized timing services to the local servers [6]. The RAID instance number is specified by RAIDNum and the virtual site is specified by VSNum.

An important goal in a distributed system is the synchronization of asynchronous processes. In the presence of lost or delayed messages this synchronization is difficult. Since RAID servers are stateless this asynchrony is not a problem most of the time. However, it is essential that the concurrency controllers be able to determine the relative order of actions that access the database. The ordering information is currently maintained in the form of timestamps that record the time at which database accesses occur. This requires that the local clocks used by the CCs be synchronized.

A clock server takes as parameters the approximate network delay, the maximum clock drift, and the maximum permissible clock difference. It maintains a difference between the local clock and the imaginary global clock. Periodically, the clock servers on each site communicate among themselves their current view of the global time. The period is determined from the input parameters. Whenever a local clock server receives from another clock server a time that is further in the future than its current time, it sets its clock ahead to that time. This method has the advantage of keeping the clocks approximately synchronized without requiring a leader, but has the disadvantage that the fastest clock in the RAID instance will determine the time for the entire system.
4.1 Transaction Execution Flow

Figure 4 depicts the relationships between the RAID servers during transaction processing. The numeric labels in the figure refer to the phases in the life of a transaction as follows:

0. UI gets a query from the user, parses it, and passes the parsed query on to AD.

1. AD assigns a unique transaction ID, and records the read-set and write-set as it processes the transaction using the local database. Updates are preserved in a log or a differential file.

2. AD forms a commit request and sends it to the local AC. This request contains the transaction ID, a list of identifiers of items read, along with the time at which the read occurred, and the list of identifiers of items written. No timestamps are available for the writes since they have not yet taken place.

3. AC sends transaction history to RC for read-set validation if AC considers this site to still be recovering (i.e., fail-locks are still set for copies on this site). RC checks for a fail-lock on each data item in transaction's read-set. Copier transactions are generated for any out-of-date items which are found in the read-set.

4. RC responds to AC with indication of read-set validity after completion of necessary copier transactions.

5. If read-set is valid (no fail-locks), the AC acquires special commit-locks for the items in the write-set. If some commit-locks are already set, it may choose to wait for them to be released, in which case it must use some method for avoiding or breaking deadlocks. AC then sends the transaction history to CC and remote ACs. If the read-set is invalid, the AC aborts this transaction.

6. CC and remote ACs reply to AC with a commit/abort decision for the transaction.

7. Once all votes are recorded from the local CC and the remote ACs, AC informs AD of the commit/abort decision.

8. AD sends the log or differential file to all AMs and tells them to commit the transaction, if the transaction was deemed globally serializable.

9. AM writes all data of the committed transaction to the database

10. AM informs AC that the transaction's data was successfully written. In response, AC releases the commit-locks belonging to the transaction, and informs the local CC and all other ACs.
11. AC sends the transaction write-set to RC. RC clears fail-locks for items in the write-set. Fail-locks are set for any sites which are perceived to be down.

12. AM tells AD that write was successful. AD informs user that the transaction committed.
Long-lived servers are denoted by italic type.

Figure 4: Transaction processing on a RAID site.
5 Data Description

5.1 RAID Query Language Grammar

\[
\begin{align*}
<xact_file> & ::= <db_stmt> <query_list> \\
<db_stmt> & ::= database <db_list> ; \\
<db_list> & ::= <id> | <db_list>, <db_list> \\
<query_list> & ::= <query> ; | <query_list> <query> ; \\
<query> & ::= <id> <in_from_exp> : <exp_list> <in_from_exp> <where_exp> | \\
& ::= <id> <in_from_exp> <in_from_exp> <where_exp> \\
<in_from_exp> & ::= <into_exp> <from_exp> | <from_exp> <into_exp> \\
<into_exp> & ::= <id> <into_list> \\
<from_exp> & ::= from <from_list> \\
<into_list> & ::= <id> | <into_list>, <id> \\
<from_list> & ::= <from_id> | <from_list>, <from_id> \\
<from_id> & ::= <id> | <id> <id> \\
<br_list> & ::= <exp> | {<exp>}+, <exp> \\
<br> & ::= <id> | <ident> | <temp_ident> \\
<query> & ::= ( <query> ) | {<query>} | | <const> | \\
& ::= <b_exp> <operator> <exp> | <id> {<exp_list>} | \\
& ::= <id> <br_list> | <id> [<br_list>] | <operator> <exp> \\
<b_exp> & ::= ( <b_exp> ) | not <b_exp> | <b_exp> | <b_exp> | \\
& ::= <b_exp> xor <b_exp> | <b_exp> and <b_exp> | \\
& ::= <b_exp> xand <b_exp> | <exp> in <query> | \\
& ::= <query> contains <query> | <exp> \\
<int> & ::= {<digits>}+ \\
<float> & ::= {<digits>}+. {<digits>}+ \\
<string> & ::= " {<chars>}+ " \\
<qtrue> & ::= true \\
<qfalse> & ::= false \\
<ident> & ::= {<letters>}+{<letters> | <digits>}+ \\
<dot_ident> & ::= {<letters>}+. {<letters>{<letters> | <digits>}+}+ \\
<temp_ident> & ::= #<ident> \\
<letters> & ::= a | b | ... | z | A | B | ... | Z \\
<digits> & ::= 0 | 1 | ... | 9 \\
<chars> & ::= <letters> | <digits> | | | @ | ...
\end{align*}
\]
5 DATA DESCRIPTION

5.2 RAID Directory Structure

The RAID directory structure is based on the distribution of data over a RAID instance, a virtual site, and a specific database.

- Top Level: /usr/raid/sites/
  - Contains subdirectories for each RAID instance:
    0/ 1/ 2/ ....

- Each instance directory contains subdirectories for each virtual site:
  0/ 1/ 2/ ....

- Under a virtual site subdirectory: 0/0/
  - The high-level directory for databases on this site:
    database/

- Under a database directory: 0/0/database/
  - Site name list and a subdirectory for each database:
    .sitename  bball/  csdept/

- Under a defined database subdirectory: 0/0/database/csdept/
  - Subdirectory for buffered write actions:
    .bfwd/

  - Files used by the DBMS (present in every database subdirectory):
    .schema  .tblanm  .tblrmn  DIRECTORY

  - Data files for relations in database csdept:
    classpreq  fall87  instructor  pnum
class  cnum  iname  name
NAME
.sitename - site name list for RAID instance

DESCRIPTION
Each RAID virtual site has its own version of this file. This file contains the site names for all other sites in the RAID instance with the virtual site.

EXAMPLE
Let RAID instance 0 have three virtual sites (0, 1 and 2). The .sitename file for virtual site 0 contains:

/usr/raid/sites/0/1/database
/usr/raid/sites/0/2/database

The .sitename file for virtual site 1 contains:

/usr/raid/sites/0/0/database
/usr/raid/sites/0/2/database

The .sitename file for virtual site 2 contains:

/usr/raid/sites/0/0/database
/usr/raid/sites/0/1/database

NOTES
When the tool dbcreat is run on a particular virtual site, the file .sitename is used to define the other virtual sites where data is to be replicated. The tool creates a database at every site listed in the file.

SEE ALSO
dbcreat
DATABASE DEFINITION FILES

NAME
database definition files - files required by the RAID DBMS
.tblrnm - table of relation names
.tblanm - table of attribute names
.schema - schema for a RAID database

DESCRIPTION
Each RAID database has three files which contain database definition information. These files are created by the tool dbcreat.
The file .tblrnm contains the relation name table for the database. The first relation name is always DIRECTORY. This is a special relation used by the database software. The rest of the table consists of the user defined relation names.
The file .tblanm contains the attribute name table for the database. The first two attribute names are always rname and rsite. These are the attributes of the relation DIRECTORY. The rest of the table consists of the user defined attribute names.
The file .schema contains blank-separated ASCII numbers with a group of numbers for each defined relation. Each group has the following format:

  relation id
  attribute id - repeated for maximum possible attributes for a relation

A relation id is an index into the relation name table and an attribute id is an index into the attribute name table. If a relation has less than the maximum possible number of attributes, the extra attribute id positions in a group are filled by the value -1.

EXAMPLE
Consider a database with two user defined relations: rel1 and rel2. Let relation rel1 have the attributes a1, b1, and c1 and the relation rel2 have the attributes a2 and b2. For this example, let | represent a string delimiter.

The file .tblrnm contains:
  DIRECTORY | rel1 | rel2 |

The file .tblanm contains:
  rname | rsite | a1 | b1 | c1 | a2 | b2 |

The file .schema contains:
  0 0 1 -1 -1 -1 1 2 3 4 -1 -1 2 5 6 -1 -1 -1

To clarify the contents of the .schema file, there are three groups of numbers here.
0 0 1 -1 -1 -1 This group represents the relation DIRECTORY. The first number, 0, is the index of the name DIRECTORY in the relation name table. The next five numbers are the indices of the names of DIRECTORY’s attributes in the attribute name table. Relation DIRECTORY has only two attributes, name and rsite (names 0 and 1 in the table).

1 2 3 4 -1 -1 This group represents the relation rel1. The name rel1 has an index of 1 in the relation name table and the attributes a1, b1, and c1 have indices of 2, 3, and 4 respectively in the attribute name table.

2 5 6 -1 -1 -1 This group represents the relation rel2. The name rel2 has an index of 2 in the relation name table and the attributes a2 and b2 have indices of 5 and 6 respectively in the attribute name table.

SEE ALSO
dbcreat
NAME

database data files - files for storing data

DESCRIPTION

A RAID database consists of one or more relations. Each relation is given a name by the creating user. The relation name is used to identify a file which contains all tuples for the relation.

A data file has no particular organization and tuple records are stored one right after the other. A tuple record has the following format:

```c
struct d_rec {
    int d_tplid; /* tuple id */
    char d_att[MXDATT][MAXAL] /* attribute values */
}
```

Note that a tuple id of -1 represents a deleted tuple.
5.3 Commit Request Message Format

A commit request message is used to request a vote on transaction commit/abort. It is created by the AD associated with the UI in which the transaction was started. The message is sent by the AD to the local AC, which forwards it to the local RC for read set validation if this site is recovering from a failure. The message is then given to the local CC and to remote ACs which give it to their local CCs. More details on the passing of this message can be found in the section entitled “Transaction Execution Flow”.

The message primarily consists of blank-separated numbers in ASCII. The format is the following:

\[
\begin{align*}
&<\text{transaction id}> <\text{site id}> <\text{number of actions (A)}> \\
&<\text{transaction start time}> <\text{transaction end time}> \\
&<\text{action start time}> \\
&<\text{number of read elements (R)}> \\
&<\text{number of write elements (W)}> \quad \text{repeated for each action:} \\
&<\text{relation id}> <\text{tuple id}> - \text{repeated R times} \\
&<\text{relation id}> <\text{tuple id}> - \text{repeated W times}
\end{align*}
\]

5.4 Transaction Specific File Formats

Transaction processing in RAID uses files to store transaction specific information. These files are examined in this section.
NAME
.dif - differential file

DESCRIPTION
The RAID system currently uses a differential file to store the contents of the transaction buffer before any merge takes place with the physical database. This file is maintained separately from the main database. All changes that would be made to the database as a result of transactions performed are registered in the differential file, and a merge action takes place at either a given time period or after some given number of transactions have been completed.

The differential file is stored in a file called .dif under the database's directory. A record in the differential file has the following format:

```c
struct df_rec {
    int df_act;     /* action: insert, delete, or update */
    int df_rid;     /* relation id */
    int dfTplid;    /* tuple id */
    char df_att[MXDATT][MAXAL]  /* attribute values */
}
```

To read a record of a given relation, the differential file handler first attempts to retrieve the record from the differential file. If the record is not found, the handler retrieves it from the main data file. In either case, the handler returns the record to the transaction buffer.

To write a record of a given relation, the differential file handler creates a differential file buffer with a header of the information. The buffer is then put into the differential file instead of the main file and a counter is used to keep track of the number of transactions.
NAME
.log - transaction log file

DESCRIPTION
Each transaction requires several actions which are processed in a transaction buffer. The file .log is used to keep an ongoing record of the contents of the transaction buffer.
A record in the log file has the following format:

```c
struct log_rec {
    int log_tranid;          /* transaction id */
    int log_act;             /* action */
    int log_rid;             /* relation id */
    int log_tplid;           /* tuple id */
    int log_time;            /* time at which action occurred */
    char log_olddatt[MXDATT][MAXAL]; /* old attribute values */
    char log_newatt[MXDATT][MAXAL]; /* new attribute values */
};
```
6 Processing Details

6.1 Processing by the Action Driver

Queries are specified in a transaction file using an editor. A transaction is first passed to the User Interface, which invokes the parser. The following is a list of the primitive action commands produced by the parser along with their formats. Note that < > denotes a required command parameter while [ ] denotes an optional command parameter (with possible repetition).

begin
database <db.name>
cbegin
load <rel.name> [<rel.name>]
project <rel.name>.<att.name> [<rel.name>.<att.name>]
select <predicate>
join <rel.name> <rel.name>
update <equation> <predicate>
insert <rel.name>
delete <rel.name> <predicate>
user output is <filename>
end

The following list explains the meaning of these keywords.

begin - indicates the beginning of a transaction.
database - specifies that the name of the database to be manipulated will follow.
cbegin - indicates the beginning of a query command.
load - tells the system to load the following relation or relations.
project - specifies a projection to be performed on a relation or relations.
select - specifies a selection to be performed on a relation or relations.
join - specifies a join of two relations.
update - specifies an update of a relation.
insert - specifies an insertion into a relation.
delete - specifies a deletion from a relation.
user - indicates that the name of an output target will follow.

end - indicates the end of a transaction.

The parser feeds the primitive action commands to the AD. When AD receives the tokens, it builds certain structures to simplify the execution of the queries. The structures are mainly linked lists created dynamically, with the predicate of a query stored in a traditional tree structure. While building the structures, the AD performs type checking, semantic checking and operator overloading. After the whole transaction file is checked, AD executes the query as a transaction. For any query set, the mandatory first action is loading of the database schema. The AD does not directly write to the physical database. Instead, it alters a buffer that is created for each transaction.

6.1.1 AD Data Structures

Some of the major data structures used by the AD are now discussed:

**rnm**

```c
char rnm[MXDREL][NMSIZE];
```

Table of database relation names. The first entry is the string DIRECTORY.

**anm**

```c
char anm[MXANM][NMSIZE];
```

Table of database attribute names. The first two entries are rname and rsite.

**wksch**

```c
struct wkschma {
    int ws_rid;
    int ws_wrid[MXWATT];
    int ws_waid[MXWATT];
};

struct wkschma wksch[MXWREL];
```

The database working schema. It contains an entry for each relation in the database and entries for temporary schemas needed to handle queries. For each relation, the wksch entry records the rid (index into rnm) and the rid and aid (index into anm) for each relation attribute.
6 PROCESSING DETAILS

wb

struct work_buf {
    int w_flag;          /* write flag */
    int w_tplid;          /* tuple id */
    /* begin read set */
    int r_flag;
    int r1;
    int r2;              /* end read set */
    char w.att[MXWATT][MAXAL];    /* attributes */
};

struct work_buf wb[MXWB];

Working buffer. Stores the tuples for all the relations currently being processed. Also stores the tuple id, a tuple flag and all tuple attributes.

wbp

struct wkbf_ptr {
    int wbp_rid;          /* relation id */
    int wbp_loc;          /* start location */
    int wbp_length;       /* length */
    int wbp_ishead;       /* TRUE or FALSE */
    int wbp_next;         /* physically, working buffer of a relation
                          * is not contiguous (e.g., more than one
                          * insert to a relation in one transaction). */
};

struct wkbf_ptr wbp[MXWBP];

Working Buffer Pointer For each relation (temporary or permanent) in the wb, an entry of the wbp array will contain the relation rid, the starting location (in the wb) of the relation's tuples, and the length of the subarray in wb where the tuples for this relation are located. Each entry also contains a flag called “ishead”. This marker indicates whether this wbp entry is the first for this relation. In addition, each entry has a “next” pointer. Next will point to the next entry in the wbp for the current relation.
actptr

    int actptr;

Points to the entry in the wbp array corresponding to the last action performed.

Processing with AD Data Structures  A relation load results in the allocation of an entry in the wbp and (for each tuple loaded) an entry in the wb. For the get query, the tuples to be retrieved in the wb array are marked "USE". Setting the "USE" flag indicates the tuple should be displayed in the query output.

In the case of a get which involves more than one relation, a join of the relations is performed. This is accomplished by allocating a new wksch entry and a new wbp entry. These entries will point to the cartesian product of two relations which will have already been loaded into the wb table. A select is performed on the cross product (marking valid tuples "USE"). A project is then performed by copying the appropriate attributes of the marked tuples into a new set of wb slots pointed to by a new wbp.

In the case of a get into, the tuple set created by the project described above will be targeted for insertion into a new relation. The action driver will create this relation and add it to the schema file after the entire query file has been processed.

To process an insert, the action driver will store the new tuples in the wb, flag them as "USE", and connect them via wbp entry "next" pointers to the previously loaded relation to which they are to be added. These tuples will be written to the log file by a "bufwrite" routine with an "INS" switch set. This is also true of the "get into" tuples.

For an update, a similar procedure will be followed except that the tuples in the wb will be updated and marked "USE". They will then be sent to the bufwrite procedure with a switch of "UPD".

Tuple deletion proceeds as follows. The tuples to be deleted are marked "DEL" in the wb. They are written by the bufwrite routine using a switch of "DEL". After placement in the physical relation file, the deleted tuples will still be marked "DEL". In all subsequent processing, they are ignored.

An important feature of this system is that upon completions of each query in a query file, any resultant temporary relations will be discarded from the wb, wbp, and the wksch arrays. This is done to conserve space within the system.

6.1.2 AD to AM Interface

Each message from an AD to its AM contains a code in the first byte of that message. This code is used to tell the AM how to process this message. The following is the list of codes together with their meanings.
6 PROCESSING DETAILS

u the line (terminated with
  n) following this code should be passed to the user process for display.

t the line following this code is the timestamp of the current query file, thus should be
  passed to AC for concurrency validation.

r the following line contains the relation name file entries. (the array rnm).

a the following line contains the attribute name file entries (the array anm).

s the following line contains the schema file entries (the array sch).

e an empty line follows. It signals the current query file process is complete. Thus END­
  TRAN flag should be sent to the user (by AM).

6.2 Replication Control

Session Vector Implementation Each site maintains its session number as an integer. The session number is kept in the file ActualSessionNumber in the working directory of the AC. This file is read by the AC each time it is restarted and a modified copy is written by the AC when it is prepared to begin recovery.

The structure for the session vector is defined in the header file “sv.h”:

```
struct site_rec {
  int vs_num;
  int session_num;
};

struct session_vector_type {
  int num_existing_sites;
  struct site_rec site[wVSMax];
};
```

The virtual site number and the session number of each virtual site in this RAID instance is kept in a site’s session vector. A current copy of the session vector is kept in the file SessionVector in the working directory of the AC. This file is written by the AC and read by the RC.

Fail-Lock Implementation The implementation of fail-locks allocates memory as necessary. Information is organized in the following manner:
6 PROCESSING DETAILS

Relation #1 <-> Relation #2 <-> ... <-> Relation #n

Tuple Block #1 <-> Tuple Block #2 <-> ... <-> Tuple Block #m

A doubly linked list of relations with fail-locks is kept. Each relation has a doubly linked list of tuple blocks. Each tuple block contains a count of the number of tuples in the block and a list of tuple entries. Each tuple entry contains the tuple identifier and a bit map which has a bit for each site in the RAID instance. A bit has a value of zero for a site with an up-to-date copy and a value of one for a site with an out-of-date copy. The tuple entries in a tuple block are kept sorted by tuple identifier.

The data structures are defined in the header file "flock.h":

```c
struct relation {
    RELATION_ID_TYPE rel_id;
    struct tblock *first_tblock;
    struct relation *next_relation;
    struct relation *previous_relation;
};

struct tuple_rec {
    TUPLE_ID_TYPE tup_id;
    BIT_MAP_UNIT flock[NUM_SITE_BM_UNITS];
};

struct tblock {
    int num_tuples;
    struct tblock *next_tblock;
    struct tblock *previous_tblock;
    struct tuple_rec tuples[MAX_TUPLES_IN_TBLOCK];
};
```

6.3 Access Manager

An AM to AM message is assumed to be in the following format:

```plaintext
<CODE><database name><space><file contents>
```

CODE is a single character with the following meanings:

d .diff file follows
After receiving such a message, AM updates the corresponding file in local site with the contents in the message.
7 Using RAID

This section explains the steps necessary to install and experiment with RAID.

7.1 Installation

Installation requires the configuring of RAID software (see the Software Guide section). Actual processing requires the creation of RAID databases through the dbcreate program. Other tools to manipulate a RAID database include the dbdestroy and dbstat programs.
NAME
dbcreat - create a new RAID database

SYNOPSIS
dbcreat RAIDNum VSNum dbname

DESCRIPTION
dbcreat creates a new RAID database where dbname is the name of the database the user wants to create, RAIDNum is the RAID instance number of the RAID system the database will be located, and VSNum is the site id of the virtual site the user wants to work at.
A file named .sitename is created in each site in a RAID instance. This file contains all the names of other sites in the RAID instance, with each name mapping to the path of the directory corresponding to that site.
In the current implementation, the system is fully replicated. Thus, dbcreat will create a directory /dbname at local site /usr/raid/RAIDNum/VSNum/database, and create the same directory at other sites whose names are in the file .sitename.

NOTES
This version of dbcreat does not communicate with other entities of the RAID system (AM, AC, AD, or CC) and it is assumed that the database is created before the system is running. This implementation was chosen because the main objective of the current version of RAID is to test the concurrency control and replicated file facilities of the system. New databases should be added by just stopping all RAID entities in the system and then invoking dbcreat.

SEE ALSO
dbdestroy, dbstat
NAME

dbdestroy - destroy an existing RAID database

SYNOPSIS

dbdestroy RAIDNum VSNum dbname

DESCRIPTION

This program deletes the files associated with the database dbname for RAID instance
RAIDNum. Initially, destruction occurs at virtual site VSNum followed by the other sites
listed in the file .dbname.
Since a destroyed database cannot be recovered, the user of this program is asked to
confirm the destruction before any action is taken.

SEE ALSO

dbcreat, dbstat
NAME
   dbstat - examine a RAID database

SYNOPSIS
   dbstat RAIDNum VSNum dbname [-f] [-d rel_name] [-s] [-l]

OPTIONS
   -f    output the differential file
   -d    output information for specified relation
   -s    output RAID schema

DESCRIPTION
   This program outputs information for the database dbname at RAID instance RAIDNum,
   virtual site VSNum. All output is directed to standard output.
7.2 Example Test Session

A simple test session is now described to illustrate the execution of RAID. User input is italicized. Assume that the user is on RAID instance 0 and virtual site 0 and that there are two other virtual sites, 1 and 2, in this RAID instance. Also assume that the database bball is going to be referenced and that it is desirable to initialize this database.

To initialize the database bball, first destroy any existing version with dbdestroy and then create a new version with dbcreat.

```
% dbdestroy 0 0 bball
Are you sure you want bball deleted(y/n)?  y
Now attempting to delete the database at local site /usr/raid/sites/0/0/database. 
Database bball has been deleted at local site /usr/raid/sites/0/0/database. 
Database bball has been deleted at site /usr/raid/sites/0/1/database. 
Database bball has been deleted at site /usr/raid/sites/0/2/database. 
%
% dbcreat 0 0 bball
making directory bball at site 0. 
making directory bball at site /usr/raid/sites/0/1/database. 
making directory bball at site /usr/raid/sites/0/2/database. 
making directory .bfwd at local site 0. 
```

---

Schema loaded:
number of relation names is : 0
number of attribute names is : 0
number of relations defined is: 0

---

Relation file 'DIRECTORY' has been opened

Please type the relation name:CR_i.
Relation 1 -- > players
Please type the attributes associated with the players relation
  Attribute 0 -- > lname
  Attribute 1 -- > fname
  Attribute 2 -- > homecity
  Attribute 3 -- > homestate
  Attribute 4 -- > bdate
DB: Maximum number of attributes 5 has been reached.
DB: No more attributes can be added
Relation file 'players' has been opened.

Please type the relation name: players.
Relation 2 - - - > teams
Please type the attributes associated with the teams relation
Attribute 0 - - - > franchise
Attribute 1 - - - > city
Attribute 2 - - - > year
Attribute 3 - - - >
DB: No more attributes can be added for the relation: teams
Relation file 'teams' has been opened.

Please type the relation name: teams.
Relation 3 - - - > positions
Please type the attributes associated with the positions relation
Attribute 0 - - - > posname
Attribute 1 - - - > posnumber
Attribute 2 - - - >
DB: No more attributes can be added for the relation: positions
Relation file 'positions' has been opened.

Please type the relation name: positions.
Relation 4 - - - > plays
Please type the attributes associated with the plays relation
Attribute 0 - - - > lname
Attribute 1 - - - > fname
Attribute 2 - - - > posname
Attribute 3 - - - >
DB: No more attributes can be added for the relation: plays
Relation file 'plays' has been opened.

Please type the relation name: plays.
Relation 5 - - - > season
Please type the attributes associated with the season relation
Attribute 0 - - - > lname
Attribute 1 - - - > fname
Attribute 2 - - - > franchise
Attribute 3 - - - > year
Attribute 4 - - - > ba
DB: Maximum number of attributes 5 has been reached.
DB: No more attributes can be added for the relation: positions
Relation file 'season' has been opened.

Please type the relation name CRi.
Relation 6 -- >

dbcreat bball completed.

Now that the database is prepared, the components of RAID can be brought up, starting with the communications oracle.

% /usr/raid/oracle/oracle &

After the oracle, the AM, CC, AC and RC for each virtual site can be brought up. Here are the commands for virtual site 0.

% /usr/raid/db/bin/am 0 0
% /usr/raid/cc/ccd -d 0 -m 2 -r 0 -s 0
% /usr/raid/ac/acd -r 0 -s 0
% /usr/raid/rc/rcd -r 0 -s 0

The RAID instance is now running and database transactions can begin.
NAME
raidtool - Control and monitor RAID in the SunWindows environment

SYNOPSIS
raidtool [-i] instance

OPTIONS
-i Make the RAID system for this raidtool have instance number instance.

DESCRIPTION
raidtool is a tool for manipulating a RAID system. A RAID system is composed of a
number of virtual RAID sites, each having the same RAID instance number. The main
control panel of raidtool consists of four parts. The top section contains buttons
that control the entire RAID system and raidtool. The second section contains
variables to some of the control functions. The third section contains a button for
each of raidtool functions which allow individual RAID sites to be controlled and
monitored. The bottom section displays raidtool messages.

Global Control Buttons
Quit Quit raidtool. Any RAID sites currently running must be stopped externally
to raidtool. The window associated with raidtool will not disappear until all
servers are stopped (not including the oracle).
Done Close the main raidtool window into an icon.
Start New Site Start a new site based on the host and part variables.
Add Existing Site Inform raidtool that a site has been started externally. The
number of the site is taken from the existing site number variable.
Stop Site Stop all or part of a site based on the site and part variables. Currently
this does not actually stop a server. Rather, it informs raidtool that the server
crashed or has been stopped.
Restart Site Restart all or part of a site based on the site, host, and part variables.

Functions raidtool provides several classes of functions. An instance of each
class may be started by pressing the appropriate button in the "Raidtool Functions"
portion of the main control panel. Each function instance is control by a separate
control panel. The panel can be removed from the display by pressing the "Done"
button on the panel. It can be redisplayed by pressing the button on the main control
panel again.
Transaction Generation  Transactions may be generated for each site. The site is selected based on the site variable. The transactions that are generated depend on the variables which are controlled by items in the transaction generation panel. Random variable values are determined by three items, a base value, a distribution function, and a distribution value. A value for the random variable is the sum of the base value and a value based on the distribution function and distribution value. The “One Shot” button generates a single transaction. A non-zero value for the transaction rate will cause transactions to be automatically generated.

FILES
/usr/raid/newtalk/control

NOTES
raidtool consists of several major modules. These are the main raidtool functions [raidtool.c], the RAID system support operations [rsys.c], the raidtool function support operations [rtf.c], raidtool functions [transgen.c, commirate.c], and other support operations [randvar.c, array.c]. Several include files are used for common declarations. These are raidtool.h, rsys.h, randvar.h, and array.h. Comments within these files explain the role of each of the structures.

raidtool is designed so that additional raidtool functions may be added easily. The rtf.c module provides the basic structure for a raidtool function. It takes care of creating the frame, the title, and the done button for the function. Additional features that are common to all raidtool functions should be implemented in this module. The rtf.c file also explains what procedures are required for each class of raidtool functions. A new raidtool function can be modelled after the existing transaction generation function.

The set of parts (or servers) that are in a raid site is defined and used in the raidtool.c file. To add support for an additional part, the part name must be added to the RaidPartList, the NUMPARTS definition must be changed in raidtool.h, and the start-part and stop-part procedures must be changed to handle the new part.

SEE ALSO
The following document is useful for anyone wishing to modify raidtool: ‘‘The SunView Programmers Manual’’

BUGS

• The pipe from control is not read correctly.
• The functionality of raidtool should be increased.
• All types of servers cannot be stopped by raidtool.
• The RAIDSITE type should be an abstraction.
7.3 Changing RAID

Since RAID is a collection of communicating, stateless servers, changes can be made to the system in a mostly transparent manner. For example, the Replication Controller was added to RAID in August, 1987. The incorporation of the RC proceeded along the following lines:

1. The functions of the RC were analyzed.

2. A determination was made of the external communications necessary for the RC to perform its functions. As it turned out, the only existing RAID server to require modifications was the AC.

3. Communication states were added to the AC. This was relatively easy because of the stateless property of the AC.

The steps performed for the addition of the RC can be considered a model for any changes in the RAID system.
8 Software Guide

This section contains a guide to the software written for the RAID system.

Compilation All of the RAID software is contained in subdirectories under the directory /usr/raid. These subdirectories are defined by function. For example, the software for the Atomicity Controller is located in /usr/raid/ac. Each subdirectory also contains a file named Makefile for use with the Unix make utility. A file named Makefile also exists in /usr/raid to generate all of the RAID programs. It does this by invoking the make files in the subdirectories.

Description The following pages describe the location and contents of the files which contain RAID software.
NAME

Header Files - for RAID software

DESCRIPTION

Many header files have been written for RAID. These are listed here.

FILES

/usr/raid/include

ac.h Definitions for atomicity controller code.
am.h Definitions for access manager.
ASN.h Definitions for code which uses actual session number.
ccd.h Definitions for concurrency controller code.
comm.h Definitions used by the RAID communications package.
common.h Definitions which are common information between the communications library and server code.
cs.h Definitions for clock server.
file.h Definitions for file names of the RAID system.
flock.h Definitions for code which manipulates fall-locks.
get_trans.h Definitions used in the extraction of a transaction from a message buffer.
hosts.h Definition of RAID hosts.
log_levels.h Definition of priority levels for debug and log messages.
netutils.h Definitions for network utilities.
omutils.h Definitions for onm utilities.
oracle.h Definitions for code which communicates with oracle server.
raid_comm.h Definitions for RAID high-level communication routines.
rc.h Definitions for replication control server code.
rdg.h Definitions for network addressing.
slave.h Definitions for concurrency control validation code.
states.h Definitions for unique constants.
sv.h Definitions for code which manipulates the session vector.
utils.h Definitions for utilities.

/usr/raid/db/h

amconnect.h Definitions for AD and AM communications.
HEADER FILES

bfw.h Definitions for buffering a write action.
buffer.h Definitions for working buffers.
ccdefn.h Definitions for putting transaction information into concurrency control format.
const.h Global definitions for database commands.
ctree.h Definitions for the condition tree used in selection and update.
defn.h Global definitions for database storage formats.
mutex.h Definitions for a mutex semaphore.
schema.h Definitions for a database schema.
update.h Definitions for a relation list.
wkbfwd.h Definitions for a buffered write action.
wkcc.h Definitions for a concurrency control information buffer.
wchk.h Variable declarations for concurrency control information.
wkctr.h Variable declarations for a condition tree.
wkdata.h Global variables for the action driver.
wksch.h Variable declarations for schema data structures.
NAME
Communications Software - code for the RAID communications package

FILES
/usr/raid/comm
  comm.c Routines which provide low-level communication functions.
  comm_utils.c Utility routines used by the RAID communications package.
  common.c Used to declare variables which are common to the comm library as well
           as other modules.
  ldg_recv.c Routines for the receiving end of the long datagram package
  ldg_send.c Routines for the sending of long datagrams.
  raid_comm.c Routines which provide interfaces to low-level communications.
  sbconv.c Routines to convert various structures to strings and vice-versa.
  site.c Routine to set directory path names for a RAID virtual site.
NAME
Clock Server Software - code for the RAID Clock server

FILES
/usr/raid/cs
GetRaidTime.c Driver program to get RAID time.
cs.c The main program for clock synchronization.
initCShosts.c Routine to initialize communications with Global Clock servers on other RAID virtual sites.
USER INTERFACE SOFTWARE

NAME
User Interface Software - code for the User Interface RAID server

FILES
/usr/raid/db/src
raid.c The main program for the UI server.

SEE ALSO
Shared Database Software
ACTION DRIVER SOFTWARE

NAME
Action Driver Software - code for the Action Driver RAID server

FILES
/usr/raid/db/src
act.c The main program for the AD server.
actbfw.c Routine to write a tuple.
actcb.c Routine to process command "begin".
actee.c Routine to record the read set.
actcon.c Routines to manipulate a condition tree.
actde.c Routine to process command "delete".
acteva.c Routine to evaluate a predicate tree.
actgnm.c Routine to get a name from standard input.
actin.c Routine to process command "insert".
actjn.c Routine to process command "join".
actld.c Routine to process command "load".
actout.c Routine to process command "output".
actpj.c Routine to perform projection.
actse.c Routine to perform selection.
actup.c Routine to process command "update".
cc.c Routines to put transaction information in the format required by concurrency control.
dumpecc.c Routine to dump out CC-formatted transaction information.
ccesp.rt.c Routines to support transaction information formatting.
dbtplid.c Routine to assign an identifier to a relation tuple.
print.xxx.c Routines to provide interface with other RAID servers.
rload.c Routines to load a relation from the database.
rqry.c Routine to find the end of a relation in the working buffer.
sqry.c Routines to get schema information such as relation id and attribute id.
wkrid.c Routine to generate a new working relation id.

SEE ALSO
Shared Database Software
NAME
Atomicity Control Software - code for the Atomicity Controller RAID server

FILES
/usr/raid/ac
  acd.c The main program for the AC server.
  autogen.c Routine to return the string equivalent for a constant. This file is automatically generated by ./makeautogen.
  list.c Routines to manipulate the transaction list.
  trans.c Routines for transaction processing.
NAME

Replication Control Software - code for the Replication Controller RAID server

FILES

/usr/raid/rc

asnio.c Routines to read and write the actual session number.

flgen.c Routines to initialize fail-locks and to insert fail-locks into messages and
extract them from messages. The acronym flgen means fail-lock general routines.

flintf.c Routines to control read set verification and the updating of fail-locks for write
set elements. The acronym flintf means fail-lock interface routines.

fllow.c Routines to actually check, set, and clear fail-locks. The acronym fllow means fail-lock low-level routines.

rc.c The main program for the RC server.

sv.c Routines to perform operations on session vectors.

svio.c Routines to read and write the session vector.
CONCURRENCY CONTROL SOFTWARE

NAME
Concurrency Control Software - code for the Concurrency Controller RAID server

FILES
/usr/raid/cc
2PL.c Routine for two-phase locking method.
ced.c The main program for the CC server.
ccpgrms.c General routines for use by concurrency control checker.
check.c Main routine for concurrency control validation.
cyclic.c Routine to determine if transaction is committable under cyclic conflict method.
delete.c Routines to remove data structures.
housekeep.c Routines for startup and cleanup.
lists.c Routines to manipulate transaction lists.
locking.c Routine to determine if transaction is committable under locking method.
oneuser.c Routine to determine if transaction is committable under single user method.
rwlocks.c Routine to determine if transaction is committable for read/write locking method.
slave.c Routine for transaction validation.
status.c Routine to accept final abort/commit decision for transaction.
ACCESS MANAGER SOFTWARE

NAME
Access Manager Software - code for the Access Manager RAID server

FILES
/usr/raid/db/src
am.c The main program for the AM server.
amDstor.c Routine to update the DIRECTORY file.
amRemoteRstor.c Routines to store transaction file to differential file, apply data integrity block to differential file, and merge differential file with database for a transaction which originated at another RAID virtual site.
amRstor.c Routines to store transaction file to differential file, apply data integrity block to differential file, and merge differential file with database for a transaction which originated at this RAID virtual site.
amslload.c Routine to load information for the schema table, relation name table, and attribute name table.
amssstor.c Routine to store information for the schema table, relation name table, and attribute name table.
lbypass.c Routine to perform differential file recovery for data of local origin.
lstor.c Routine to apply data integrity block to log file for data of local origin.
merge.c Routines to merge differential file and relation file.
rbypass.c Routine to perform differential file recovery for data of remote origin.
ropenrw.c Routine to open a relation file in the database.
rstor.c Routine to apply data integrity block to log file for data of remote origin.
undo.c Routines to undo a transaction by the audit trail recovery method.
update.c Routines to update a log file and relation file.

SEE ALSO
Shared Database Software
NAME
Shared Database Software - common code used by UI, AD, and AM RAID servers

FILES
/usr/raid/db/src
  cvtdata.c Routines to encode and decode data for commitment.
  mtxgr.c Routines to gain and release a mutex semaphore.
  openrw.c Routine to open a file.
  sload.c Routine to load the schema table, relation name table, and attribute name table.
  ssstor.c Routine to store the schema table, relation name table, and attribute name table.
NAME
Database Configuration Software - code used for configuring RAID databases

FILES
/usr/raid/db/src
  dbcreate.c Program to create a RAID database.
  dbdestroy.c Program to remove a RAID database.
  dbstat.c Program to output status of a RAID database.
  mergemain.c Program to merge a differential file and relation file.
DEBUG SOFTWARE

NAME
  Debug Software - code to assist in debugging RAID

FILES
  /usr/raid/debug
  debug.c Routines to assist in debugging.
References


