Efficient Availability Mechanisms in Distributed Database Systems

Bharat Bhargava
Dept. of Computer Sciences
Purdue University
W. Lafayette, IN 47907
bb@cs.purdue.edu

Abdelsalam Helal
Computer Science Engineering
University of Texas at Arlington
Arlington, TX 76019
helal@cse.uta.edu

Abstract

The resiliency of distributed database systems can be realized through a collection of integrated fault-tolerance mechanisms. These include data replication techniques, failure detection, failure isolation through reconfiguration and adaptability, and non-blocking atomic commitment. Collectively, these mechanisms enhance the availability and operability of the system in the presence of various types of site and communication failures. In this paper, we focus on mechanisms for data replication, failure detection, and reconfiguration. We present the implementation details of each of these mechanisms along with their integration within the RAID system developed at Purdue. Data replication is implemented through the partial replication of data relations, and through the use of a library of replication control methods. An on-line replication control server (RC) provides highly available database operations through the adaptable use of these methods. Failure detection is implemented via a reliable surveillance facility that monitors the changes in system connectivity. Such failures include site and communication failures as well as network partition. Repairs and network merges are also detected by this facility, thus leading to the automatic initiation of recovery. We will show how failure isolation is achieved through data and server reconfiguration and by the adaptable use of replication methods.

1 Introduction

RAID [10, 6] [1] is a distributed database system that has been implemented to explore new fault-tolerant schemes and adaptability policies that can achieve high levels of availability and performance. In this paper, we discuss the implementation of the RAID fault-tolerance mechanisms, including data replication, failure detection and surveillance, and data reconfiguration. The empirical evaluation of these mechanisms can be found in [8, 7, 19].

The immediate goal of replication in RAID is to increase data, user, and system availability in presence of failures. By maintaining multiple copies of data relations, some copies of the database remain available even though the system has suffered site or communication link failures. Both detectable and predictable failures are accounted for through a failure detection facility and a data reconfiguration scheme. A surveillance facility is used to detect failures and their subsequent repairs. Once a failure is detected, the view of the system is updated and new transactions are executed in the new view. This way the failure is isolated. Once a failure is predicted, the database administrator may issue a control transaction to reconfigure and redistribute the data relations so as to temporarily avoid future access to parts of the system that are anticipated to fail. Our design of the replication controller is targeted toward tolerating multiple occurrences of combinations of site and communication failures as well as network partition.

The surveillance protocol monitors changes in the connectivity of the RAID sites. Changes in connectivity are treated as *view hints* that trigger failure (repair) detection exceptions which, in turn, trigger reconfiguration (recovery) actions. In transaction processing systems, surveillance can be responsible for controlling performance degradation during failures. For example, without failure detection, transactions that are issued during periods of failure may suffer delayed negative response that indicates their abortion. This delay consists of the time wasted in executing transactions till the point where remote access to failed copies is attempted (this could possibly be the commit point), plus the time awaited for until a timeout exception occurs. Not only do these transactions suffer delays before they get aborted, but they also waste system resources (CPU cycles and com-

[1] The Purdue RAID and RAID-V2 systems are not related to the concept of Redundent Array of Inexpensive Disks.

CIKM '93 - 11/93/D.C., USA

munication bandwidth) and contribute to unnecessary data contention with other transactions that are not affected by the failure. Consequently, all transactions (the ones that get aborted and the ones that go through) experience response time degradation. The view hints that are exported by the surveillance facility makes it possible to completely avoid any delays before a transaction is aborted due to the failure. This avoidance spares system resources and eliminate unnecessary data contention.

Adaptability and reconfigurability are used to cope with the changing performance and availability requirements. Adaptability can improve both performance and availability by allowing data and system reconfiguration. *System reconfiguration* aims at isolating parts of the system whose failure has been detected or predicted, or merging isolated parts of the system whose repair has been confirmed. Such reconfiguration, which is adapted through the use of the surveillance facility, avoids the cost of executing 'living in the past' transactions. *Data reconfiguration* is another form of adaptability where replication and distribution specifications of data relations can be changed dynamically. Data reconfiguration can be used to adapt to variations in transaction access patterns in order to improve transaction response time and system throughput. Redistributing a data object so as to create a local copy and hence avoid remote access is an example of such adaptation. When failures are predicted, data reconfiguration can be used to temporarily relocate copies whose sites are anticipated to fail. In RAID, data reconfigurability is achieved by employing a technique whereby copies can be made potent/impotent, dynamically.

The paper is organized as follows. The rest of this section is devoted to an overview of the second version of the RAID system (RAID-V2). Section 2 elaborates on the various RAID replication mechanisms. Sections 3 gives the details of the RAID surveillance facility. Adaptability and data reconfiguration is discussed in section 4. Finally, summary and conclusions are given in Section 5.

1.1 An Overview of the RAID-V2 System

RAID-V2 is the second version of the RAID distributed database system [10, 6]. RAID-V2 is a server-based, relational, distributed database system that is being developed on Sun workstations under the Unix operating system. In this section, we give a brief description of the RAID-V2 system and its performance. Details of the design and implementation of the system can be found in [6].

Each database site in the RAID system consists of seven servers, each of which encapsulates a subset of the functionality of the system. The seven servers are the User Interface (UI), the Action Driver (AD), the Access Manager (AM), The Concurrency Controller (CC), the Atomicity Controller (AC), the Replication Controller (RC), and the Surveillance Controller (SC). The user interface is a front-end that allows a user to invoke QUEL-type queries on a relational database. The action driver translates parsed queries into a sequence of low-level read and write actions. The access manager is responsible for the storage, indexing, and retrieval of information on a physical device. The concurrency controller checks that read and write actions of different transactions do not conflict. The atomicity controller is responsible for ensuring that transactions are committed or aborted atomically across all sites. The replication controller manages multiple copies of data objects to provide system reliability and mutual consistency of replicated data. The Surveillance controller collects connectivity information about RAID sites, and advertises view changes to the replication controller. Figure 1 illustrates the paths of communication in RAID.

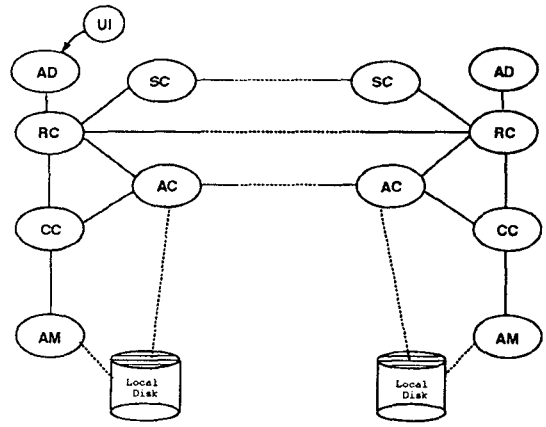

Figure 1: The RAID-V2 System Architecture

Servers in RAID communicate solely through the exchange of messages [9]. All inter-server actions consist of a request and a reply. Once a request is issued for a transaction, further progress on that transaction is blocked until a reply is received.

2 Replication Mechanisms

Replication control is the part of transaction management that is responsible for ensuring one-copy serializability [5]. Many replication control methods have been introduced in the literature [4, 2, 26, 15, 1, 20, 16, 22]. Usually, replication control is built on top of a concurrency control component that guarantees serializability of the equivalent hypothetical one-copy database. Therefore, in a layered implementation of a transaction manager, replication control comes higher in the hierarchy than concurrency control [6]. A log-

ical operation on a database object is tested for one-copy equivalence by the replication controller and is then transformed into physical operations on available replicas. The replication control then passes the physical operations to the concurrency controllers for serializability check. An implementation of a replication control method usually requires the use of a data directory as well as view information in order to determine which sites to involve in performing a certain logical operation.

Replication and object fragmentation has been implemented in several research projects [25, 14, 27, 24]. Replication in RAID is designed to meet and help achieve five main objectives. These are increased fault-tolerance, replication and location transparency, data reconfigurability and adaptability, and controlled performance degradation during failures. To achieve these objectives, we have implemented a variety of replication mechanisms. Specifically, we have implemented off-line replication management tools, a stand-alone replication control server (RC), quorum selection heuristics, kernel-level support for quorum operations, and a RC-interface to a surveillance facility.

The off-line replication management tools are used for creating, administrating, and reconfiguring replicated databases under RAID. The on-line RC server transforms logical operations–parsed by an SQL interpreter– into physical operations on quorums. The distinctive feature of the RC is its quorum-based interface to a library of replication control methods. The interface facilitates the adaptable use of a variety of these methods. Quorum selection heuristics are used in RAID to reduce the message traffic overhead associated with the quorum methods. To further minimize quorum overhead, a kernel-level multicast facility is implemented. Details of the multicast implementation and performance can be found in [9]. The RC-interface to the RAID surveillance facility provides useful hints for selecting available quorums and for controlling performance degradation during failures.

In this section, we present the design and implementation details of the RAID replication mechanisms.

2.1 Off-line Replication Management

A RAID replicated database is created, off-line, according to a specification of its relation schema and replication information. Each database is given a unique logical name. To use the database, an instance of the RAID system is created under this logical name. Once an instance is activated, users can attach to it through user interfaces. User transactions are parsed into read and write operations and are directed to the on-line RC server.

The construction and replication of a RAID relational database is done off-line and can be described

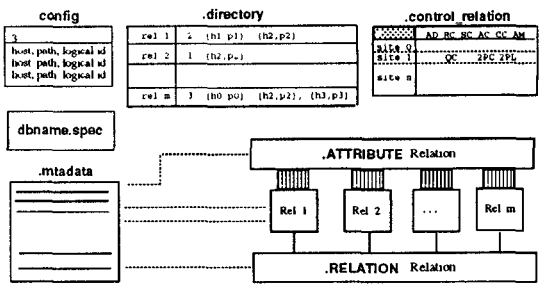

Figure 2: RAID Database Layout

by the following process. The relations 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 directory that contains a database copy on the associated host. For example, "/uraid10/raid/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 initialized using the *dbcreate* command. The dbcreate command takes two arguments. The spec file and the logical 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 representation of the replication information found in the spec file. 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 the instantiation of RAID. In addition to the directory and the configuration file, dbcreate creates user and meta relations. The meta relations contains schemas information of users and meta relations. User relations are optionally initial-

ized by inserting tuples from a specified input file. After creating all these files in a local temporary directory, *dbcreate* 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 RAID directory. Figure 2 depicts the RAID database layout.

In addition to dbcreate, RAID provides commands to remove, reset, and extend its databases. 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 destroies 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, adjust the directory, and update the meta relations. Finally, the *raid* command takes a database path name, where a copy of the database is s tored, and uses it to start up an appropriate RAID instance. From the configuration file, which can be found in the database path name, the *raid* command learns of all the Host-Path pairs and their mapped unique logical id's. For each Host-Path pair (H_i, P_j) with logical id k, the command remotely creates a replication controller, an atomicity controller, a concurrency controller, a surveillance controller, and an access manager on the host H_i. The *raid* command passes the logical id k, As a command-line argument, to all the instantiated servers at host H_i. In addition, the replication controller and the access manager servers are passed the path P_j. 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 servers.

2.2 The On-line Replication Control Server (RC)

The replication controller (RC) features highly available database operations through the use of quorum-based methods. RC is, in general, tolerant to network partition and can tolerate up to $\lceil \frac{n+1}{2} \rceil$ site failures, in an n-site system. A distinctive feature of the RC is its quorum-based interface to a library of replication control methods. The interface hides the details of a particular replication method from the RC server. This resulted in a clean implementation of an infrastructure that can opt or adapt to use certain existing replication method or other methods that can be added in the future. Currently, RC can adapt to use the read-one-write-all, the quorum consensus, or the general quorum assignment methods. Another distinctive feature of the RC is its experimental-based policies that govern the proper use of replication methods through adapting the degree of replication and through dynamic data reconfiguration. In general, the RAID policies aim at maximizing the availability of fault-intolerant methods, and minimizing performance penalties of methods which are highly fault-tolerant. In addition, the RC features replication and relocation transparency, and controlled performance degradation during failures.

2.2.1 The *RC* Interface

The RC maps logical actions from a local action driver (AD) into physical actions on available copies. The mapping is done through the 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 the local copy of a fully-replicated directory to locate all existing copies of that relation. It then uses the view hint vector that is exported by the surveillance facility to identify which copies are currently available. The interface then passes the set of sites where copies are currently available, along with the action needed to the replication control method. The latter decides whether available copies 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 such quorum, some quorum selection heuristics are used. Section 2.3 explain the use of these heuristics.

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 and 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. If one or more of the replies never arrived, the AD will eventually timeout and will send an abort message to the RC to flush the transaction out of the system.

Similar to the Read requests, the RC handles StartCommit requests by finding a quorum for each Write

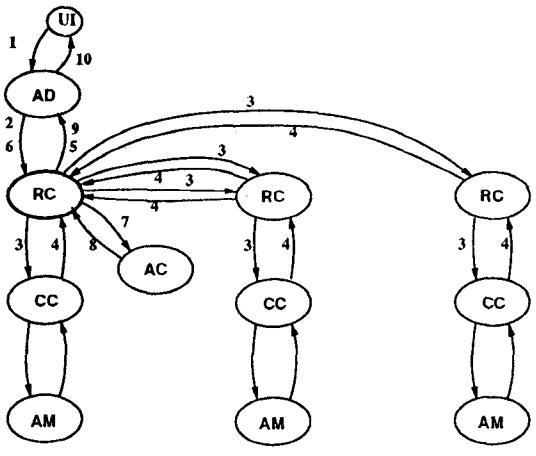

Figure 3: Control Flow in RAID Replication Controller (RC)

operation in the StartCommit request. The RC constructs 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 wishes 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.

Figure 3 depicts the communication paths between RC and other servers in RAID. The labels on the pathes of Figure 3 are explained below.

1. Transaction arrives at a AD from its UI.
2. AD processes transaction into read and write operations, reading through the RC. Write operations are saved in the AD until commit time.
3. The RC reads by communicating with its CC and with remote RCs.
4. Tuples are returned from the CC or remote RCs to the originating RC, ...
5. and from the RC to the AD.
6. When the AD has completed the read phase of transaction processing it passes the list of tuples that should be updated to the RC.
7. 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.
8. The AC sends a positive or negative acknowledgement to the RC.
9. The RC passes the acknowledgement on to the AD.
10. AD returns the acknowledgement to the UI.

2.2.2 The RC Server Implementation

The RC server consists of approximately 2500 lines of code written in C. In addition to the library of replication methods and the quorum interface, the code for the RC server consists of three sections: initialization, main loop, and termination and statistics dump.

Initialization includes setting up communication with the RAID name server which we call the ORACLE [11]. The RC set up communication by sending a registration request to the ORACLE. The RC request includes a *Tell_Me_About* list that specify which other servers the RC wishes to know of their whereabout. Members of the list can be marked synchronous or asynchronous. In the first case, the registration request will block until the awaited synchronous member has contacted the ORACLE. In the second case, the request will not block, and RC will be notified with the address of the awaited asynchronous member as soon as the latter register with the ORACLE. The RC's *Tell_Me_About* list includes as synchronous members, other RC's, local CC, local AC, and local SC; and as asynchronous members, all local AD's.

Once communication is initialized, RC reads the RAID directory from the database path name passed by the *raid command*. It also tries to read a communication cost matrix. The cost can be determined by the number of gateways the remote request has to go through, or by the inverse of the computation power (MIPS) of the remote sites. The RC then initializes the transaction table which maintains state information of local transactions and remote requests; initializes its view hint which keeps connectivity and reachability information of other RC's; and finally initializes the replication control method that is specified, as an argument, to be used.

The main loop consists of receiving a high-level RAID message, decoding and processing the message, and blocking to receive another. When a message is received, its header is decoded in order to extract the message type and the id of the transaction to which the message is destined. The transaction table is then searched for a matching transaction id. If the transaction is found, its state information is used to process the message.

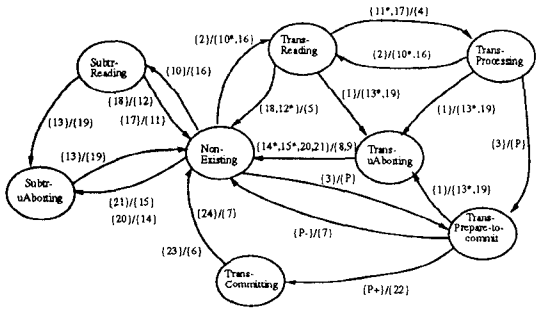

Figure 4: The RAID Replication Control Automaton

Figure 4 depicts a simplified version of the RC server automaton. The automaton specifies all possible sequences of state transitions a RAID transaction may go through. From *Non-Existing*, a new transaction moves to *Trans-Reading* by sending out read requests to a quorum of sites. When all replies arrive, the transaction moves to *Trans-Processing* state. To process another read operation, the transaction sends another set of read requests and moves to *Trans-Reading* state again. The transaction loops between these two states until it gets aborted by the user issuing the transaction, in which case it moves to TransuAborting; until one of the replies that arrived is negative (due to concurrency conflict), in which case it moves to *Non-Existing* state; or until it reaches its commit point, in which case it moves to *Trans-Prepare-to-Commit* state. At this state, the update list of all deferred writes of the transaction is built. In the case of quorum consensus, version numbers are packed into the update list. Immediately after the update list is ready, the transaction moves into *Trans-Committing* state where it stays there until it receives the result of the commitment and moves to *Non-Existing* state. On the other hand, a remote read request creates a subtransaction that sends a read request to its local concurrency controller and then moves immediately to *Subtr-Reading* state. The subtransaction stays on that state until it either receives a reply, in which case it forwards it to the home site issuing the request, or gets aborted by the user issuing the home transaction in the home site.

When a transaction or a remote request reaches the *Non-Existing* state, it is removed from the transaction table. When in any waiting state, one of the replies never arrives, RAID end-to-end timeout mechanism aborts the transaction after a timeout value by flushing an abort message throughout the whole system. More details can be found in Figure 4 by following each single transition.

Termination of the RC server happens when a kill signal sent by the ORACLE is received. When the kill message is received, the RC aborts all pending transactions and dump RC statistics into well known files. The statistics include number of RC aborts which are aborts due to quorum unavailability, average update list size, and average read and quorum size.

2.2.3 The RC Quorum Interface

The RC quorum interface is shown in figure 5. The interface consists of six high-level function calls. For each read request, there is a *Single_READ_Quorum* invocation. The parameters passed are transaction id, RC method name, and relation descriptor. *Single_READ_Quorum* passes the relation descriptor on to the *Available_Sites* function call. The latter indexes the directory to determine the set of sites where the relation is replicated. This set is then filtered by the view hint returning the set of available copies for that relation. *Single_READ_Quorum* then passes the set of available sites along with the RC method name to the replication control methods library where the appropriate routine is invoked. For methods where more than one quorum is available, the library routines select a particular quorum using either the *Random_Permutation* or the *Min_Site_Permutation* heuristics or both. The *Single_READ_Quorum* invocation ends by returning a read quorum (set of sites from which the RC requests the relation). When all requests have been replied, RC invokes *Construct_Value* to determine the most-up-to-date relation. *Construct_Value* returns a relation that consists of tuples that have the highest version numbers. For all the writes, there is an *ALL_WRITE_Quorum* invocation. The parameters passed are transaction id, RC method name, and the update list. For each element in the update list, *ALL_WRITE_Quorum* invokes the *Single_WRITE_Quorum* function call. The latter proceeds analogous to the *Single_READ_Quorum* and returns a write quorum for the passed update element. The *ALL_WRITE_Quorum* ends by returning the union of the write quorums of all the relations included in the update list. The RC then invokes the *Pack_Update* function call in order to include the union of the write quorums in the update list. For some methods, the *Pack_Update* function call modifies the version numbers of the relation tuples. When the packed update list is returned by *Pack_Update*, RC hands the list to its local AC in order to start commitment. Finally, when the RC server is started, it invokes *Init_RC_Protocol* to do any necessary initialization. For example, reading the quorum parameters relation in the case of the quorum consensus method. *Init_RC_Protocol* always invokes *Recover* for possible recovery procedures required by certain repli-

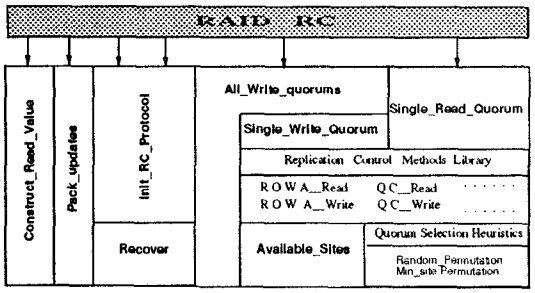

Figure 5: RC Quorum Interface

cation methods.

2.3 Quorum Selection Heuristics

In the quorum consensus method, an operation can usually be performed by more than just one quorum. In order to select a quorum, the RC quorum interface uses one of three heuristics. The RANDOM heuristic generates a random permutation out of the set of sites where data is replicated and returns the smallest prefix of the permutation that constitutes a quorum. The MINSITE heuristic descendingly sorts the set of sites according to their weights, and returns the smallest prefix of the sorted set that constitutes a quorum. The RAND-MINSITE heuristic generates two quorums form the RANDOM and the MINSITE heuristics respectively, and returns the RANDOM quorum only if it has the same size as the MINSITE quorum. Otherwise, it returns the MINSITE quorum. Intuitively, random selection of quorums produces uniform message traffic which, in turn, helps load-balancing the RAID sites. Random selection, however, can result in large-size quorums and hence can increase the volume of message traffic. On the other hand, smallest-size selection of quorums incurs optimal volume of message traffic. Smallest-size quorum selection, however, results in a skewed load distribution among the RAID sites. The performance of these heuristics were studied in [18].

3 The RAID Surveillance Mechanism

Surveillance in RAID is implemented as a separate server named the Surveillance Controller, or SC. The SC servers detect both site and link failures as well as subsequent repairs. Whenever a change in the system connectivity is detected, the SC servers export a new view hint to the Replication Control servers (RC) to reflect the change. In response, RC's adapt to the new view hint and discard quorums that span failed or unreachable sites. This way, transactions' operations are directed only to sites that are reachable.

The design of the RC and the SC servers does not contain or create functional dependency between the two servers. This separation was realized by the following requirements.

- The RC does not require the surveillance facility, although it highly benefits from it. This allows for instantiating RAID without the SC servers, if so desired. It also allows transaction processing to continue despite SC server failure.
- RC treats the connectivity information that it imports from the SC server as a view hint [23] and not as a synchronized view. This way, the correctness of the replication methods used by the RC server does not depend on the surveillance facility.

The main idea behind our protocol is to periodically send out *I_am_alive* broadcast messages, and to periodically check for received *I_am_alive* messages. We use interrupt signals to create this periodicity. Each SC maintains a private timer for each participant SC. A timer for an SC participant at site j reflects the duration of time that has passed since the last *I_am_alive* message was received from site j. Whenever an *I_am_alive* message from site j is received, the SC at the receiving site resets the timer corresponding to the SC st site j. If the *I_am_alive* message from site j is not received in a certain period of time, its corresponding timer will expire indicating that either a site or a link failure has occurred. The protocol can be described as follows.

- As soon as an SC starts up, it sets the alarm to be interrupted every fixed interval of time. We call the duration of this time the *delta* time. SC then blocks waiting to receive messages or periodic alarm signals.
- When SC is interrupted, it wakes up and does one of two chores, in alternation. In the first alternation, it broadcasts an *I_am_alive* message to remote SC's.
- Each SC uses a timer for every other SC. Each timer includes the number of *ticks* left before it goes off. Whenever an *I_am_alive* message is received from an SC on site i, the *ticks* in the timer of site i are reset to a positive constant that we call *TimeoutTicks*.
- When SC is interrupted and it is in the second alternation, it checks the validity of its view. It does so by decrementing one *tick* from all the timers. It then builds a temporary view by including sites that have non-zero positive number of ticks remaining after the decrement. Sites that have not sent an *I_am_alive* message within (*TimeoutTicks*

× *delta*) seconds are the ones whose timers contain zero number of *ticks*. The *TimeoutTicks* count guards against lost or extremely delayed messages. If the new temporary view is found different from the old view, SC installs the temporary view as permanent and notifies its local RC with that view. If, by the subsequent second alternation, SC does not receive an acknowledgement message from its local RC, it re-notifies the RC with its current view, which could possibly be different from the one originally sent.

The protocol takes *delta* and *TimeoutTicks* as parameters. The choice of these parameters imposes a compromise between increased communication overhead (in case of small values of *delta*) and increased abort rate of living-in-the-past transactions (in case of large values of *delta*). The default values of these parameters are set to 30.0 seconds, and 3 ticks, respectively.

The RAID surveillance protocol has the following features:

- Symmetry: the protocol is decentralized and does not require a coordinator. This avoids running an election protocol in the case of a coordinator failure.
- Asynchronous coordination: the protocol participants are allowed to be completely out of synchrony. This avoids clock drift problems and solutions.
- Reliability: the protocol makes no assumptions on the timeliness or reliability of message delivery. Instead, it guards against message loss or delays.
- versatility: the protocol does not need to be restarted when adding or removing new participants. This accommodates for dynamic system reconfiguration.

The performance and effectiveness of the surveillance protocol was examined experimentally in [19]. Several other surveillance protocols have been proposed in the literature [28, 3, 21, 13, 12, 17].

4 Data Reconfiguration

Data reconfiguration is used in RAID to improve the performance and to guard against anticipated failures. To improve transaction response time and system throughput, RAID can adapt to variations in the transaction read/write mix by re-distributing the data in order to optimize the quorum size of the dominant operation. Redistribution can also aim at creating a local copy in order to avoid remote access. When failures are predicted, data reconfiguration can be used to temporarily relocate copies whose sites are anticipated to fail. To achieve data reconfiguration in RAID, we use a technique whereby copies of an object can be regenerated or made invalid. Our reconfiguration technique requires that objects be physically fully replicated. Each replica can, however, be made invalid, or can be regenerated. An invalid copy is not part of the database until it is regenerated. In addition to redistributing a data object, our technique can –to some extent– reconfigure the replication method that is used with that object. This is done by reconfiguring the object's read and write quorums.

To implement reconfiguration, RAID uses a fully-replicated *Quorum Relation* that encapsulates the database distribution and quorum information. Table 1 shows the quorum relation of a four-relation DebitCredit database in an 8-site RAID. For each relation, the size of the read and write quorums and the weights of the copies are specified. Copies with zero weight are *invalid* copies. In order to read(write) a relation, a number of copies with total weight greater than or equal to R_Quorum(W_Quorum) is required. As an example, consider the Branch and Account relations. The Branch relation is configured so that it has three copies at site 0, 1, and 2. It is also configured so that it must be accessed through the read-one-write-all method (ROWA). The Account relation is fully replicated and is configured to be accessed through a quorum consensus read-same-as-write method (QC-RSW).

RAID uses data reconfiguration to implement adaptability policies. For instance, one of the RAID adaptability policies prescribes the appropriate degrees of replication to use under different transaction characteristics and failure conditions. The policy is implemented by updating the quorum relation to reflect the new validation/invalidation of the replicas. The definition of this policy is based on an integrated experimental study of availability and performance in RAID. In this study, the effect of varying the degree of replication on the performance and availability of replication methods is examined both experimentally and analytically. The degree of replication that incurs the minimum compromise of the performance and availability (called the *practical degree of replication*) is found and is prescribed for use under a given transaction characteristics and failure conditions.

Figures 6, and 7, demonstrate two experiments in this study in a 9-site RAID system. Figure 6 shows the response time and availability of the ROWA method for read-only transactions and for workstation reliability of 0.90. The practical degree of replication is 2 copies. Higher degrees of replication do not improve availability or impair the response time. Figure 7 shows the response time and availability of the QC-RSW method for 50 update percent and for worksta-

Table 1: The RAID Quorum Relation

Relation	R_Threshold	W_Threshold	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8
Teller	4.00	6.00	0	1	1	0	1	1	1	1
Branch	1.00	3.00	1	1	1	0	0	0	0	0
Account	5.00	5.00	1	1	1	1	1	1	1	1
History	3.00	5.00	1	1	1	1	1	1	0	0

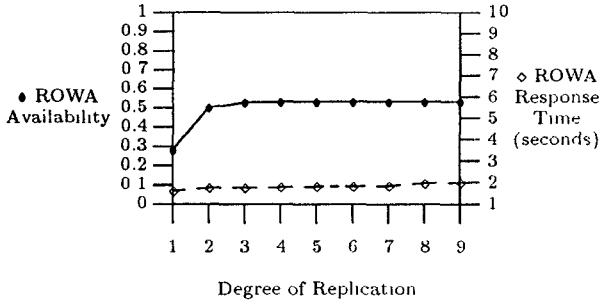

Figure 6: Read-one-write-all: 0% Updates, 0 90 Reliability

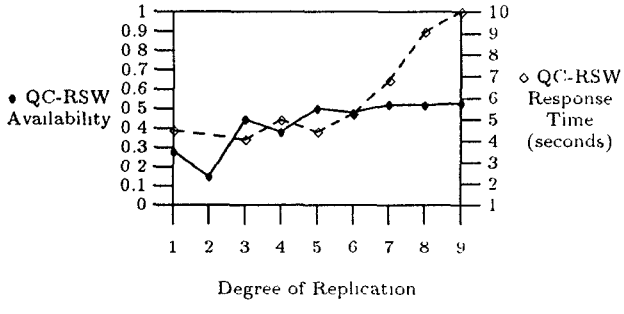

Figure 7: Quorum Consensus(Read-same-as-write): 50% Updates, 0 90 Reliability

tion reliability of 0.90. The practical degree of replication is 5 copies. Higher degrees of replication severely impairs response time and, in the same time, does not increase availability significantly.

Table 2 lists a sample of the practical degrees of replication for both the ROWA and the QC-RSW, for 0, 20, and 50 update percents, and for 0.90, 0.95, and 0.99 workstation reliability. The degree of replication of the database relations can be adapted to the practical values once estimates of the transactions' update percent or the workstation reliabilities are obtained. The adaptability is accomplished by updating the quorum relation in accordance with table 2.

5 Conclusion

This paper describes the design and implementation of some fault-tolerance mechanisms in the distributed database system RAID. These mechanisms include adaptable data replication, failure detection, and failure isolation through reconfiguration. Replication in RAID has been implemented in terms of a stand-alone replication controller, off-line replication management, quorum selection heuristics, and an interface to a surveillance facility. Failure detection is implemented via a reliable surveillance facility that maintains network-independent reachability information called view hints. We have shown how the surveillance facility is integrated with replication control, to increase fault-tolerance and to control performance degradation during failures. Finally, we have demonstrated the use of data reconfiguration and replication adaptability to achieve failure isolation.

References

[1] Amr El Abbadi and S. Toueg. Availability in partitioned replicated databases. In *Proc. Fifth ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 240–251, March 1986.

[2] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. *Proceedings of the 2nd Int'l Conference on Software Engineering*, pages 562–570, October 1976.

[3] J. F. Bartlett. A nonstop operating system. *Proc. of Hawaii Intn'l Conference on System Sciences*, January 1978.

[4] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596–615, December 1984.

[5] Philip Bernstein and Nathan Goodman. The failure and recovery problem for replicated databases. *Proceedings of the 2nd ACM Symp. on Principles of Distributed Computing*, pages 114–122, August 1983.

[6] Bharat Bhargava, Karl Friesen, Abdelsalam Helal, Srinivasan Jagannathan, and John Riedl. Design and implementation of the RAID V2 distributed database system. Technical Report CSD-TR-962, Purdue University, March 1990.

[7] Bharat Bhargava, Karl Friesen, Abdelsalam Helal, and John Riedl. Adaptability experiments in the raid distributed database system. *Proceedings of the 9th IEEE Symposium on Rehability in Distributed Systems*, pages 76–85, October 1990.

Table 2: Practical Degrees or Replication of the ROWA and the QC-RSW

Update Percent	ROWA			QC-RSW		
	r=0.90	r=0.95	r=0.99	r=0.90	r=0.95	r=0.99
0%	2	2	1	3	3	1
20%	2	1	1	5	-	-
50%	1	1	1	5	3	1

[8] Bharat Bhargava, Abdelsalam Helal, and Karl Friesen. Analyzing availability of replicated database systems. *International Journal of Computer Simulation*, 1(4):393–418, December 1991. A special issue on distributed file systems and database simulation.

[9] Bharat Bhargava, Enrique Mafla, and John Riedl. Communication in the Raid distributed database system. *Computer Networks and ISDN Systems*, 1991.

[10] Bharat Bhargava and John Riedl. A model for adaptable systems for transaction processing. *IEEE Transactions on Knowledge and Data Engineering*, 1(4):433–449, December 1989.

[11] Bharat Bhargava, John Riedl, and Andrew Royappa. The raid distributed database system. Technical Report CSD-TR-691, Purdue University, August 1987.

[12] Kenneth Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. *ACM 11th Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[13] Sally Bruso. A failure detection and notification protocol for distributed computing systems. *Proc. of IEEE 5th Intn'l Conference on Distributed Computing Systems*, pages 116–123, May 1985.

[14] P. Dasgupta and M. Morsi. An object-based distributed database system supported on the clounds operating system. Technical Report GIT-ICS-86/07, Georgia Tech, 1986.

[15] D. K. Gifford. Weighted voting for replicated data. *Proceedings of the 7th Symposium on Operating System Principles*, pages 150–162, December 1979.

[16] Abdelsalam Abdelhamid Heddaya. *Managing Event-based Replication for Abstract Data Types in Distributed Systems*. PhD thesis, Harvard University, October 1988. TR-20-88.

[17] C. Hedrick. Routing information protocol. *Network Working Group, RFC 1058*, June 1988.

[18] Abdelsalam Helal. *Experimental Analysis of Replication in Distributed Systems*. PhD thesis, Purdue University, May 1991.

[19] Abdelsalam Helal, Yongguang Zhang, and Bharat Bhargava. Surveillance for controlled performance degradation during failures. In *Proc. 25th Hawaii International Conference on System Sciences*, pages 202–210, Kauai, Hawaii, January 1992.

[20] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[21] W. Kim. Auditor: A framework for highly available DB-DC systems. *Proc. of IEEE Second Symposium on Reliability in Distributed Software and Database Systems*, pages 116–123, July 1982.

[22] Akhil Kumar. Hierarchical quorum consensus: A new class of algorithms for replicated data. *Proceedings of the 10th IEEE symposium on Distributed Computing Systems*, May 1990.

[23] Buttler W. Lampson. Hints for computer system designers. *ACM 9th Symposium on Operating System Principles, Bretton Woods*, October 1983.

[24] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost. Computation and communication in R*: A distributed database manager. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[25] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[26] T. Minoura and G. Wiederhold. Resilient extended true-copy token scheme for a distributed database system. *IEEE Transactions on Software Engineering*, 9(5):173–189, May 1982.

[27] Calton Pu. *Replication and Nested Transactions in the Eden Distributed System*. PhD thesis, University of Washington, May 1985

[28] Bernd Walter. Network partitioning and surveillance protocols. *Proc. of IEEE 5th Intn'l Conference on Distributed Computing Systems*, pages 124–129, May 1985.