Fault Detection in L2000 Storage-Replication Service

6.199 Project Report
MIT Laboratory of Computer Science - Library 2000
Supervisor: Jerome H. Saltzer
by Komkit Tukovinit
May 8, 1994

Abstract

This report is to fulfill the last part of the 6.199 Advanced Undergraduate Project requirement. The project's objective is to build mechanisms that detect the faults expected in the Library 2000 (L2000) system. The mechanisms include a local integrity checker and a remote integrity checker. The main accomplishments of the project are:

- Design of the detection mechanisms
- Analysis of the design in relation to the fault models
- Partial implementation of the detection mechanisms

Table of Contents

6.199 Project Report
MIT Laboratory of Computer Science - Library 2000

Supervisor: Jerome H. Saltzer

by Komkit Tukovinit

Fault Detection in L2000 Storage-Replication Service

This report is to fulfill the last part of the 6.199 Advanced Undergraduate Project requirement. The project's objective is to build mechanisms that detect the faults expected in the Library 2000 (L2000) system. The mechanisms include a local integrity checker and a remote integrity checker. The main accomplishments of the project are:

- Design of the detection mechanisms
- Analysis of the design in relation to the fault models
- Partial implementation of the detection mechanisms

This report is geared toward the readers interested in continuing the replication research with Library 2000. The document is organized as shown in Table 1 and in the table of contents. Note that sections 1 and 2 provide the general introduction to the project, while sections 3 through 8 are the real substance of the project. Sections 9 and 10 wrap up the report.

Sections	Comments
1	• briefly introduces Library 2000 project and its replication research • a refresher of the general concepts used in building a fault-tolerant system
2	• bullet-points general criteria for design that influence the choices made in the project
3	• provides the motivation for the project
4	• introduces and sketches the scope of the 6.199 project
5	• discusses the faults that are being addressed by the report
6	• shows the overall system design and the made assumptions about the system
7	• specifies the detection scheme used to detect the modeled faults
8	• argues why the detection scheme would detect the modeled faults
9	• bullet-points and discusses the issues not addressed in the project
10	• gives credit to whom the credit belongs

Table 1: Organization of the 6.199 Report

1. Introduction

"The technology of on-line storage, display, and communications will, by the year 2000, make it economically possible to place the entire contents of a library on-line, accessible from computer workstations located anywhere." [Saltzer1] Many engineering issues arise when such a library system is built. One of them is how to preserve the digital data stored in the system for several decades: a period of time longer than the storage's lifetime.

What could go wrong with the data in a storage system? Something "goes wrong" (a fault occurs) when the stored data cannot be accessed or when the data returned by the system is incorrect. There are many causes of faults, but they can be broadly categorized as follows [Gray1]: environments, operations, maintenance, hardware, software, and process. Examples for each category are listed in Table 2 for concreteness.

Category	Examples
Environments	Power failure, weather, earthquakes, fires.
Operations	Crucial processes killed, system configured improperly.
Maintenance	Good disks with data taken out to be repaired instead of bad disks.
Hardware	Disks fail, Processor fails.
Software	Programs maliciously mutate or destroy data.
Process	Shutdown due to administrative decision.

Table 2: Fault Categories And Their Examples

How can faults be tolerated or masked by the system? To mask the faults, the system must be able to detect the faults. Detecting the faults is the topic of this report which will be discussed in the later sections. To mask the faults, the system must also be able to correct the detected faults.

The system must have redundant data to correct faults because redundant data must be used to reconstruct lost data. Data can be made redundant by replicating it using different mechanisms including tape backup and RAID. However, the two mechanisms have undesired properties that can be fixed if the data are replicated on replicas located in different geographical locations. For tape backup, it is hard to verify that the stored data can be retrieved when needed, and that the backed-up data include everything that needs to be backed up. RAID cannot tolerate environmental faults such as a hurricane or an earthquake since all redundant data are located in one geographical location.

Replicating only the data does not eliminate all faults. The mechanisms that detect and repair the faults must also be replicated to avoid a single-point failure.

What is the most important characteristic of the replicas housing the replicated data and software? The answer is the probability that the replicas will fail simultaneously to the point that the data cannot be recovered must be low. Therefore, the replica failure modes must be as independent as possible. Independence can be maximized by minimizing the replicas' commonalties. For example, replicas should be located in different geographical areas so that a natural disaster would not destroy all the replicas.

When the replica failure modes are independent, it is unlikely that the data will fail simultaneously. However, without failure corrections, the data will fail serially. A piece of data may fail on a replica, then the same piece fails on another replica, and so on, up to the point that the data is irrecoverable. Hence, it is imperative that a single fault be detected and corrected as soon as possible. In another word, the replicas' window of vulnerability—the period when additional faults can exceed the threshold of recoverability—should be as small as possible.

Besides detecting and correcting faults, the L2000 replication service must also properly propagate legitimate data changes to the replicas. This can be problematic: the replicas might view legitimate changes as faults, and the replicas will fix the changes; hence, the data in the system cannot be modified. Moreover, since maximizing the replicas' independence means locating them in a wide geographical area, the replication service—detecting and repairing faults, and propagating legitimate changes to the replicas—must be done efficiently to offset communication cost and unreliability.

2. General Criteria for Design

The general criteria for design, as mentioned in [Library2000-1], permeate the design choices made in this report. The criteria are as follows:

- Works on standard operating system, file system, disk storage system, and networking protocol.
- Replicas' data may be weakly consistent [Golding1, Golding2, 6.826-1] and updates can be unavailable for a bounded time.
- Works with system with at least 1,000,000 Gbytes of data.
- Should be made simple by comprising a small number of independent mechanisms whose correctness can be verified.
- Lifetime of data exceeds the lifetime of any storage and exceeds the mean time between major disasters.

3. Motivation

Why do we design a new storage-replication service when a few [Liskov1, Satyanarayanan1, Page1, Hisgen1, Golding1, Golding2] have already been designed? This is because the demands on the replication service in L2000 are different from the systems that researchers have considered. There are four main differences:

- Difference in frequencies of expected events
- Length of data life
- Need for simple implementation
- Ability to layer the scheme over available file systems without modification

The first reason is the most important.

First, the frequencies of expected events in the L2000 system are different, given the volume and the lifetime of data. Table 3 contrasts the relative frequencies of expected events in other file systems with those in L2000. As shown in the table, in the L2000 system, the rate of data decay is expected to exceed the rate of data update, and to be in the same magnitude as the rate of data addition. Therefore, we must emphasize discovering and repairing data decay, while other systems emphasize, for example, high performance in obtaining consensus on data update and addition.

Events	Other File Systems	L2000
Reading data	Most frequent	Most frequent
Updating data	Second most frequent	Rare
Adding Data	Less frequent	Less frequent
Data spontaneously decayed	Rare	Less frequent

Table 3: Relative Frequencies of Expected Events

Second, the data on the L2000 system must last for decades and survive the upgrades and replacements of storage devices. The data on the old device must be copied—correctly and automatically—to the new device when the new device is installed. In contrast, other file systems usually require manual copy when a new device is installed.

Third, the L2000 design should be simple so that the correctness of the design is obvious or can be verified easily. This property gives confidence that the data on the system will survive unaltered through decades.

Fourth, the L2000 design must layer on top of the available file systems. Modifications to system programs are avoided. Not all the works cited, [Liskov1, Satyanarayanan1, Page1, Hisgen1, Golding1, Golding2], have this need.

4. 6.199 Project

The above three sections discussed the goals of the L2000 replication research, so how does this project contribute to the research? To correct or mask faults on a system, the faults must be detected. The project provides some thoughts about how to detect faults. Because there are many faults that can occur on the system, but there are only a few things that can be accomplished in the project, the fault that has the highest probability of occurring—data mutation—is focused upon.

The following four sections, sections 5 through 8, are the substance of the project. The flow of the sections roughly follows Figure 1 [Dally1]. The fault models in section 5 discuss how faults occur. Section 6 describes overall designs and assumed environments of the system. The detection scheme is discussed in section 7 and finally, the analysis of the scheme is discussed in section 8.

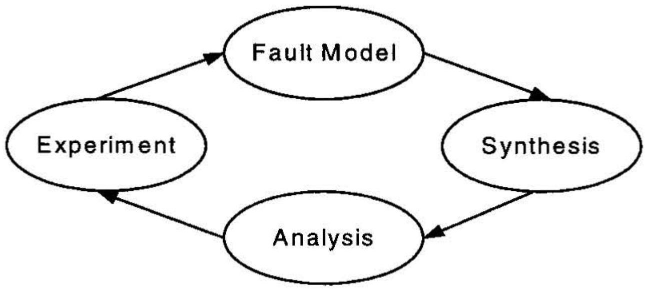


Figure 1: Steps in Engineering Reliability

5. Fault Models/Threat List

As in any synthesis of a fault-tolerant system, the ways the system fails are considered by synthesizing a fault model or a threat list. To ease the analysis, two fault models are used. The two models—static and dynamic—have similar fundamental failure modes but different data dynamics.

To discuss the dynamic model, a model of how the data change in the system is also considered.

Section 5.1 discusses the failure modes common to the static and the dynamic systems. Section 5.2 differentiates the two systems. Section 5.3 discusses how the data changes in the dynamic system. Finally, section 5.4 suggests how changes in the dynamic system are propagated.

5.1 Failure Modes/Threat List

A threat is an occurrence that causes the system to fail. Many threats can cause system's failure, but only a few have probabilities significant enough to be worrisome. Hence, it is sensible to worry the most about a few threats with the highest probabilities of occurring.

In this project, only one threat is addressed: data mutation. Since L2000's documents are stored on file systems, the threat to the documents has the same characteristics as the threat to the files in a file system. Data mutation in a traditional file system means file (bit) corruption, file disappearance, and file addition. File corruption means the bits in a file were illegitimately changed. File disappearance means data files illegitimately disappear. File addition means files that are not part of the data are illegitimately added.

How often do faults occur? The upper bound of the frequency cannot be computed since there are no frequency statistics for all the causes (Table 2) of data mutation. However, the lower bound can be estimated by using the failure characteristic of the storage device.

The frequency of disk failures per year can be calculated as in Equation 1 [Patterson1]. 10,000 is the approximate number of hours per year. Note that the equation relies on the assumption that disks fail independently. Also note that disks' MTTF is in hours.

$$Failures\_Per\_Year = \frac{\#\_of\_disks \times 10{,}000}{Disk\_MTTF}$$

Equation 1: Frequency of Disk Failures

According to [Library2000-1], a typical data center has 1,000 disks. According to the specification, the Mean Time To Failure (MTTF) of 1992's disks is 500,000 hours. Therefore, the lower-bound of failure rate is 20 failures per year. For a typical 2000's data center, the number of disks is the same, but the disks' MTTF is more than 1,000,000 hours. Therefore, the lower-bound failure rate is 10 failures per year.

We speculate that the rate of data failures (decay) will exceed the rates of data update and deletion, and will be in the same magnitude as the rate of data addition. The rate of failures is higher than the computed lower bound because of the following reasons:

- the computed value does not include other failure causes
- disks located at the same site often do not fail independently
- manufacturer's specification of disks' MTTF is doubtful: 30 years of experience shows that the number is unreliable
- the indirection table that is used for convenience inherently cannot be replicated in a distributed way

The indirection table will be described in section 6.3.

5.2 Dynamics of the System

The last section discusses the general failure modes of the system. This section discusses the system dynamics—how the data change over the system's lifetime. We analyze systems with two dynamics:

- static
- dynamic

The static system is a system on which the data does not change: the data stays the same from the beginning to the end of system's lifetime. The dynamic system is a system on which the data changes: the data can be added, deleted, and updated. The following paragraph elaborates on the static and the dynamic systems.

Figure 2 shows the state progress of a replica. In the following explanation, we assume that there are no illegitimate changes applied to the system, or they have been repaired properly. In the static system, since there is no change in the system, the replica's states—the replica's documents and the checksum data—are equal in all periods. For example, the states in period N+2 would be equal to those in period 1. In the dynamic system, this is not the case. It is possible that the states will differ. For example, the states in period N+2 might be different from the states in period 1.

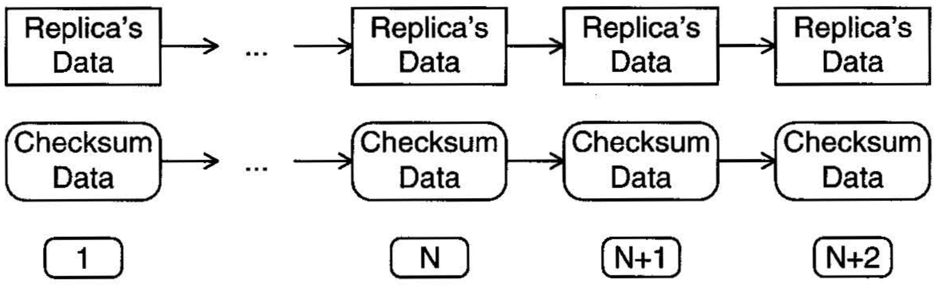


Figure 2: Progress of Replica's State

5.3 Change Model

We would like to discuss how faults are detected in the static and the dynamic systems. To discuss how legitimate changes are detected in the dynamic system, a model of how the data changes is needed. This subsection discusses the model.

In this proposed change model, valid changes occur when the changes are applied to the majority of the replicas. Valid changes include file addition, file deletion, and file update. For example, in Figure 3, the changes to replicas BC in the replica group ABC are valid because the changes are applied to the replica majority.

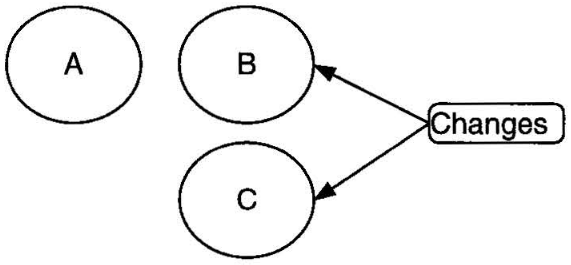


Figure 3: Example of Valid Changes to System

5.4 Suggestion for Change Propagation Scheme

What do we do with the legitimate changes once they are detected? In this subsection, a propagation scheme that might work in the system is suggested. However, since the scheme is not part of the project, it will not be analyzed. The suggestion is to let the repair mechanism propagate changes seen on the replica majority to the rest of the replicas.

In the detection scheme being proposed, when changes are applied to the replica majority, the minority replicas will detect the changes. The minority replicas will list the changes as items that need to be repaired, and the changes will be propagated to the minority replicas as repair items.

For example, in Figure 3, changes are applied to replicas BC, a majority of the replica group. A detection mechanism on A will detect the changes. Since the changes are on the majority of the replicas, A will see the changes as legitimate and will retrieve the changes from either B or C.

The propagation scheme has a flaw. If the changes applied to the replica majority decay before they are propagated, the changes will be in no-majority states, and will not be propagated. For example, in Figure 3, if the changes on B are deleted, then only C has the changes making C a minority. C will eventually detect its differences from the replica majority—which says that there have been no changes—and treat the changes as faults. C then gets rid of the changes. This is a case where a single fault causes a valid change on the system to disappear; the changes that have not been propagated to all the replicas are not fault-tolerant.

Hence, the legitimate changes to the replicas are said to be in two stages: fragile and fault-tolerant [Library2000-1]. Changes that are in the fragile stage have not been propagated to all the replicas, and might disappear or be marked as irrecoverable before they are propagated. Changes that are in the fault-tolerant stage have been propagated to all the replicas and are as fault-tolerant as any other data in the replicas.

6. Interfaces and Assumed Environments

Figure 4 shows a design of each replica in the system. *Each* replica consists of mechanisms that are run as daemons, and of interfaces through which the daemons and L2000 clients access the replica's data. The mechanisms shown in Figure 4—local integrity checker (LIC) and remote integrity checker (RIC)—are detection mechanisms that detect data changes on the replicas. The detected changes can be legitimate or illegitimate. The mechanisms are not described in this section: LIC is discussed in section 7.1 and RIC in 7.2.

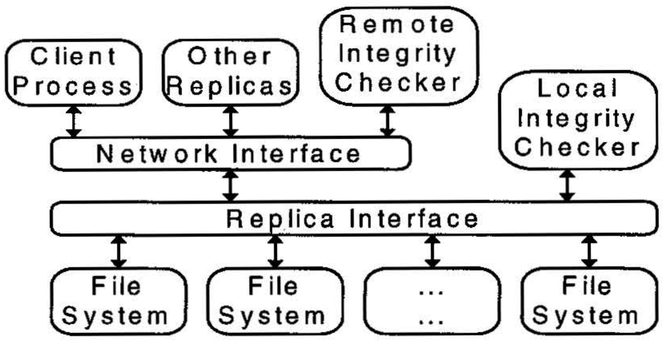


Figure 4: Interfaces and Mechanisms on a Replica

L2000 clients and the detection mechanisms access the replica's data in a well-defined way. They use the shown interfaces—the network interface and the replica interface—to access the replica's data. The network interface is described in section 6.1 and the replica interface in 6.2.

Besides the interfaces, other environments assumed in the project are also described in this section. Section 6.3 describes the UID used as a handle to the replica's data, its advantages, and its disadvantages. Section 6.4 discusses the property of the checksum algorithm needed by the detection mechanisms. Section 6.5 discusses other assumptions made in the project.

6.1 Network Interface

The network interface is where L2000 clients request services from a replica. The interface is also used by the replication service in two ways. First, the other replicas need to use the interface to check and retrieve the data on the replica for repair. Second, the RIC uses the interface to check for data divergence among the replicas. All users of the interface—the replication service mechanisms and the clients—are assumed to talk to the interface through a communication protocol such as TCP/IP.

Table 4 lists the primitives required to support the replication service. Note that UID is the unique identifier used to uniquely identify documents stored on the L2000 system.

Primitives	Descriptions
get(UID)	Get the data identified by the UID
cksum(UID)	Calculate and return the checksum of the data identified by the UID
getcksumrecord()	Get the checksum file last produced by the LIC.

Table 4: Network Interface Primitives

For the replication service, replicas need "get" to retrieve good data so damaged data can be repaired. The RIC needs "getcksumrecord" to retrieve the checksum file, as will be discussed in section 7, needed to find state differences among the replicas. The voting procedure, used by both the LIC and the RIC, uses "cksum" to check for the existence and to get the checksum calculation of a UID.

For client services, clients need "get" to retrieve the data on the replica. They might also need "cksum" to check if the transmission of data started by "get" is uncorrupted. There might be other unmentioned primitives that the clients need. For example, since the clients are external users of the server resources, the network interface might need to check if the clients are authorized users of the resources. Note that the project does not try to identify all the primitives needed by the L2000 clients.

6.2 Replica Interface

The replica interface's primary function is to translate a UID into a data location, e.g., file system and path name. Also, since the replica interface supports the network interface, the replica interface must support all the primitives listed in Table 4. However, since the replica interface is assumed to be used by the users internal to the replica, it does not need to support all the functionalities supported by the network interface. For example, it does not need to check if the users of the interface are authorized users.

Besides supporting the network interface's primitives, the replica interface must support primitives that the LIC requires. The LIC needs the enumeration of all UIDs and their associated checksums; the functionality is supported by "enum&cksum." Also, the system's checksum file must be updated when there are valid changes to the system. Hence, "putcksum" allows the system's checksum file to be replaced. Table 5 shows the replica interface's primitives.

Primitives	Descriptions
get(UID)	Get the data identified by the UID
cksum(UID)	Calculate and return the checksum of the data identified by the UID
getcksumrecord()	Get the checksum file last produced by the LIC.
enum&cksum()	Enumerate all UIDs and calculate their checksums
putcksum(data)	Replace the system's checksum file with a supplied data

Table 5: Replica Interface Primitives

6.3 UID to Location Translation Table (ULTT)

What is a UID in the L2000 system? A UID is the token used to uniquely and universally identify a document stored on the L2000 systems and related electronic library systems [Library2000-1]. The UID is a result of the need to identify documents uniquely and universally across multiple library systems.

In today's system, data is stored using file systems. Hence, UID needs to be translated into the location of the data. Table 6 shows the structure of the UID-Location Translation Table (ULTT) and a sample entry of the table. Note that the real entry of the L2000 system might not look like the sample entry in the table. For example, the location of the data needs to include a server's name (one of the twenty servers in a replica), a file system's name, and the file's path name.

UID	Location	Other Information
102456798	/afs/athena.mit.edu/user/k/o/kom/.cshrc	File's ACL, Entry's checksum, etc.

Table 6: UID-Location Translation Table And a Sample Entry

The advantages and the disadvantages of using a ULTT are discussed in the following subsections.

6.3.1 Advantages of Using a ULTT

There are two main advantages of using a ULTT:

- Flexibility of replica management
- Abstraction used by the integrity checkers

The first advantage of using a ULTT—flexibility of replica management—is a classical advantage of using an indirection table. The file system and virtual memory system all use indirection tables that allow the systems to relocate, hide the internal representations of, and control the access to system resources without the user's explicit awareness.

With a ULTT, the replica can relocate the location of its data without letting the client of the interfaces know. The replica also hides how the data is represented in the system. For example, a replica might keep its documents using the UNIX file system, while another replica might use a new object-oriented file system.

The ULTT is also a logical place to store information that are associated with the UIDs. For example, the replicas may need to associate with the documents ACLs of the documents.

The second advantage of using a ULTT is the integrity checkers now can deal with the documents as an abstract object rather than as files. If the documents have to be treated as files, the checkers would need to store the locations of the files—the way files are uniquely identified—in the checksum file. For example, they will need to store the servers' names, the file systems' names, and the files' path names to be able to identify the data. The checkers might also need to check the integrity of the objects used as parts of the locations, e.g., the inodes in the file systems.

Moreover, since the checksum files used by the detection mechanisms are exchanged among the replicas, the files must have similar entries so the files can be compared. If the file names are used instead of UIDs, the replicas must have exactly the same configurations: they must have the same servers' names, the same file systems' names, the same files' path names, etc. The replicas' management must be tightly coordinated and thus, is cumbersome and inconvenient.

6.3.2 Disadvantages of ULTT

The ULTTs of the replicas might not be the same. For example, if the replicas have different servers' names, then the ULTT entries, which map the UIDs to the file locations which servers' names are part of, must be different. Hence, another replica cannot provide a local replica with a good copy of ULTT if the local replica's ULTT is corrupted. There is no redundant information to fix the corruption.

There are at least two ways this problem can be dealt with:

- Provide redundancy for the replica's ULTT
- Treat the corruption of a ULTT entry as a fault and correct it appropriately

The first approach requires that multiple copies of the ULTT are kept on a replica. Care should be taken to minimize the common failure modes of the copies. For example, the copies should be stored on different servers within the replica. When a corrupted entry is found, redundant entries can be used to repair the damage.

The second approach treats a corrupted ULTT entry as a corruption of the document associated to the ULTT entry. The corruption is repaired by retrieving a redundant document from another replica, and by overwriting, if possible, the corrupted ULTT entry.

In the second approach, note that if a ULTT entry consists of only a UID and a location, the proposed detection mechanisms will detect an entry corruption. A corrupted entry means the mapping from the entry's UID to the data location is incorrect. Therefore, the data identified by the UID will be incorrect, or a valid UID will disappear, or an invalid UID will appear. Hence, the detection mechanisms, which verify the replica's data using the UID mapping, would detect the corruption. If an entry consists of other information not related to location mapping, such as an ACL, the entry needs to contain the checksum of the information so that the corruption of the information can be detected.

In this project, the second approach is assumed. Note that taking the second approach increases the failure rate of the data since a corruption to the table entry is considered a failure of the associated data. This point was discussed earlier in section 5.1.

6.4 Property of Checksum Algorithm

In the project, the detection mechanisms treat the checksum values as the pseudo contents of the file: the checksums of the files are used for comparison rather than the files' contents. Hence, the checksum algorithm should have a property as follows. The probability of having files with different contents being mapped to the same checksum value, as shown in Figure 5, is very low.

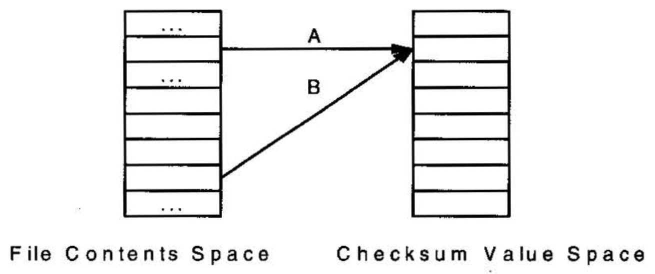


Figure 5: Mapping From File Contents to a Checksum Value

6.5 Other Assumptions

This section describes two assumptions made to ease the synthesis and the analysis of the detection scheme.

The first assumption is, the network used by the replicas does not partition. The detection mechanisms do not have to deal with being unable to talk to other replicas because of a network partition.

The second assumption is, although the replicas can sometimes be down, the occurrence is so rare that the replicas will always be able to talk to one another. This assumption again simplifies the detection scheme: the detection mechanisms do not have to deal with being unable to talk to other replicas.

7. Detection Scheme

Given the ULTT discussed in section 6.3, the detection scheme does not have to treat L2000 documents as files. However, since the implementation done in this project uses path names as UIDs, it is helpful to discuss UIDs as if they were files. Hence, the discussions from here on talk about files instead of UIDs. The reader should keep in mind that the files' path names are really the documents' UIDs.

By the fault model discussed in section 5.1, the detection scheme must detect illegitimate changes applied to the replicas' data: bit corruption, file addition, and file deletion. By the change model discussed in section 5.3, the scheme must also detect legitimate changes applied to the replicas knowing that sometimes the changes are not applied to all replicas. To detect the changes—both legitimate and illegitimate—two mechanisms are used: local integrity checker (LIC) and remote integrity checker (RIC).

The LIC checks for illegitimate changes (faults) occurring on a replica by comparing the replica's states captured at two time points. It is discussed in section 7.1. The RIC checks for legitimate changes not applied to all the replicas by comparing the checksum files of two replicas. It is discussed in section 7.2. Section 7.3 portrays another view of the detection scheme.

7.1 Local Integrity Checker (LIC)

The LIC detects faults that occur on a replica between two time points. The LIC does three things:

- detects changes that are applied to a replica's data between two time points
- checks if the detected changes are legitimate
- reports all and only illegitimate changes applied to the replica

The general idea is as follows. Every T_{LOCAL} minutes, the checker generates a checksum file F_{N+1} collecting the UIDs identifying all the documents stored on a replica and collecting their associated checksums. The checker then compares F_{N+1} with the checksum file F_N, the checksum file that was generated and validated in the previous period N. Since F_N reflects the replica's state at the period N, the differences F_{N+1} has from F_N must be the changes applied to the replica's data during N and N+1. Since the changes can be either legitimate or illegitimate, the checker checks for the legitimacy by conferring with other replicas. Once the legitimacy of the changes is determined, the LIC reports all and only the illegitimate changes (faults).

The checker follows two steps. First, it generates the local difference report listing the changes applied to the replica's data between periods N and N+1. How the report is generated is detailed in section 7.1.1. Second, the checker checks if the changes are legitimate by confirming the changes with other replicas. How the changes are confirmed is detailed in section 7.1.2. The faults detected by the LIC should be repaired. Section 7.1.3 suggests how the faults should be repaired.

7.1.1 Local Difference Report

The local difference report lists the changes applied to a replica's data during the periods N and N+1. The report can be generated in the following way.

Every T_{LOCAL} minutes, the checker

1. calculates the checksums of all files on the replica, and saves the files' names and checksums to a "checksum" file F_{N+1}
2. calculates and records the checksum of F_{N+1} so F_{N+1}'s corruption can be detected.

3. appends to F_{N+1} the finishing time (Greenwich Mean Time)
4. calculates the checksum of the previous checksum file F_N, excluding the timestamp, and compares the calculated checksum with F_N's recorded checksum. If the checksums mismatch, F_N has been corrupted; a flag is raised and the checker stops. If the checksums match, F_N is good and the checker continues.
5. compares F_{N+1} with F_N and generates the differences listing the deleted files, the added files, and the changed files as shown in Table 7.

Previous Checksum File	Current Checksum File	Inconsistencies
Has file F	Does not have file F	F has been deleted
Does not have File F	Has F	F has been added
File F has checksum C_N	File F has checksum C_{N+1}	F has been changed

Table 7: Changes Listed by Local Difference Report

In this step, the LIC discovers all changes that were applied to the replica since last checking period. We cannot tell if the detected changes are legitimate or illegitimate. The confirmation through voting, discussed in the next section, checks for changes' legitimacy.

7.1.2 LIC Confirmation through voting

Because the changes that are applied to a replica can be either legitimate or illegitimate, the changes listed in the local difference report should be categorized by their legitimacy. The changes' legitimacy is determined by having the replicas vote on the changes. If the replica majority has the changes, then the changes are legitimate. Otherwise, the changes are faults that should be repaired.

This step deletes from the local difference report all legitimate changes. The final output of the LIC lists all and only illegitimate changes that occurred on the replica between the periods N and N+1. The checker follows the following steps:
for each change listed in the local difference report

1. asks all the replicas for the status of the change

2. if the replica majority has the change, the change is legitimate and its entry is deleted from the local difference report. Otherwise, the change is a fault and its entry stays in the report.

After the above steps, the remaining entries of the local difference report are faults that occurred on the system. The LIC returns this report.

Table 8 shows the conditions that each detected change must have to be confirmed as a fault. For example, in the replica group ABC, if the local difference report on A lists the file F as having been deleted, and either B or C does not have F, then the majority of the replicas (AB, or AC) agrees that F does not exist. F has been deleted from the replicas; A does not need to fix F's deletion.

Cases	Majority State	Change Cases	Action
File F deleted	Has F	Deletion Fault	Confirmed as deleted
File F deleted	Does not have F	Deletion Change	Deleted from report
File F added	Has F	Addition Change	Deleted from report
File F added	Does not have F	Addition Fault	Confirmed as added
File F changed	F's checksum is C_{N+1}	Update Change	Deleted from report
File F changed	F's checksum is C_N	Change Fault	Confirmed as changed
File F changed	No-majority state.	Change Fault	File F added to a "possibly irrecoverable" report

Table 8: Conditions for Changes to be Confirmed as Faults

Note that file existence is a boolean value; it cannot be ambiguous: a replica either has a file or it does not. So file existence always has a majority state. On the other hand, checksum value is a multi-value number and can be ambiguous. It is possible that there is no replica majority agreeing on the same checksum value. Hence, checksum values can have no-majority states. A no-majority state cannot be tolerated because having it implies that enough faults have occurred to push the replicas' state over the threshold of recoverability.

To calculate if there is a majority, Equation 2 is used. n is the total number of replicas.

$$\#\_of\_agreeing\_replicas \geq \left\lfloor \frac{n}{2} + 1 \right\rfloor$$

Equation 2: Majority Equation

All legitimate changes detected by the LIC are changes that were applied to the replica being checked. If the changes were applied to the replica majority but not the replica being checked, the LIC will not detect the changes. Hence, we need another mechanism, remote integrity checker, to deal with the situation. The remote integrity checker is discussed in section 7.2.

7.1.3 LIC Repair Model

This subsection suggests how the faults reported by the LIC should be repaired.

The aim of the repair is as follows. After the local replica is repaired, the replica should have the state that is reflected by F_N, the validated checksum file of the last period, plus the detected legitimate changes. Figure 6 shows the state progression of a replica when there are changes applied to the replica. During the periods N and N+1, legitimate and illegitimate changes are introduced into the replica. At N+1, the LIC detects the changes and creates a repair order listing the illegitimate changes. After the repair mechanism is run, the state of the replica is as shown in the figure's last box—a state identical to the state at N plus the legitimate changes introduced during N and N+1.

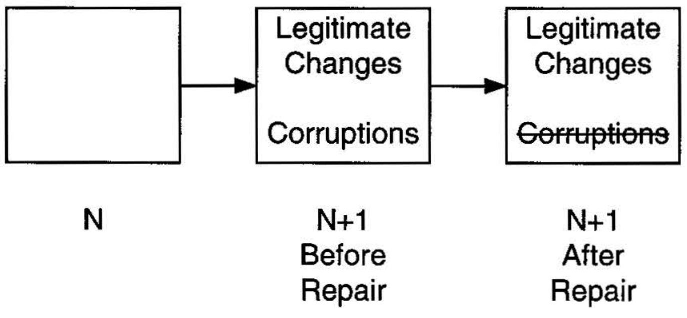


Figure 6: Replica's State Progression With Legitimate Changes and Corruptions.

Note that not only the faulty data need to be repaired, the checksum file must also be updated to reflect the legitimate changes. This is so that when the LIC checks for changes in the next period, all legitimate changes already detected will not be seen as changes. Table 9 lists the cases and the fixes. The most interesting case listed in the table is the first—when F_N is corrupted or deleted. We will analyze why the suggested fix works in section 8.2.3.

Cases	Suggested Fix
F_N is corrupted or deleted	A validated checksum file is retrieved from another replica and used as the validated checksum file, then the algorithm is rerun.
A reported change is confirmed by majority state	The change is incorporated into F_N which is used as the next validated checksum file.
A reported change is not confirmed by majority state	The change is eliminated by getting redundant data from other replicas or by deleting the change. F_N is used as the next validated checksum file

Table 9: Fixes Needed For Faults Reported by Local Integrity Checker

7.2 Remote Integrity Checker (RIC)

The RIC checks for the divergence of replicas' data from the replica majority. A replica's data can diverge because the replica might not have the changes that other replicas do. This can happen because valid changes might not be applied to all the replicas.

The remote integrity checker does three things:

- detects differences of the local replica's checksum file and a partner replica's
- determines from the differences which replica has the most recent change and which replica does not
- produces the repair list for the replicas that need to get changes

The RIC's algorithm is similar to the LIC's except that the checksum files being checked belong to two replicas. The general idea is as follows. Every T_{REMOTE} minutes, the checker on the local replica A randomly picks a partner replica B and retrieves from B B's most recent checksum file F_B. The checker then compares F_B with A's most recent checksum file F_A. The differences between F_A and F_B are changes that need to be applied to either A or B. By having the replicas vote on the differences, the replicas that need to be updated can be determined.

The checker follows two steps. First, it generates the remote difference report listing the differences that the two checksum files have. How the report is generated is discussed in section 7.2.1. Second, the checker determines—by conferring with other replicas—which of the two replicas being checked needs to be updated. How the differences are confirmed by other replicas is discussed in section 7.2.2. Section 7.2.3 suggests how the detected changes should be applied to the replicas.

7.2.1 Remote Difference Report

The remote difference report lists the differences of the replicas' checksum files. The report is generated as follows.

Every T_{REMOTE} minutes, the checker process:

1. Retrieves the latest checksum file F_A of the local replica A
2. Checks the timestamp and the checksum of F_A. If F_A is older than T_{LOCAL} minutes, or if F_A's checksum does not match its recorded checksum, a flag is raised and the check is terminated; otherwise, the check is continued.
3. Randomly picks a partner replica B and retrieves its most recent checksum file F_B.
4. Checks the timestamp and the checksum of F_B. If F_B is older than T_{LOCAL} minutes, or if F_B's checksum does not match its recorded checksum, a flag is raised and the check is terminated; otherwise, the check is continued.
5. Generates the differences between F_A and $F_{B.}$

Table 10 shows all the difference cases

Replica A	Replica B	Difference Code
Has file F	Does not have file F	IR1
Does not have File F	Has F	IR2
File F has checksum C_A	File F has checksum C_B	IR3

Table 10: Differences Listed by Remote Difference Report

Note that each difference detected in this step can be one of the followings:

- change applied to A but not B
- change applied to B but not A

The cases cannot be differentiated without conferring with other replicas. Hence, the next step, confirmation through voting, is needed.

7.2.2 RIC Confirmation through voting

Each difference listed in the remote difference report can be one of the following categories:

- change applied to A but not B
- change applied to B but not A

By having the replicas vote on a difference, the difference can be categorized.

The final output of the RIC lists all the changes that A needs, and all the changes that B needs. The checker follows the following steps:

for each difference listed in the remote difference report

1. asks all the replicas for the status of the difference
2. categorize each difference as listed in Table 11

Table 11 shows how the detected differences are categorized. For example, if replica A has a file F, replica B does not, and the majority of the replicas does, then a missing file entry is generated for replica B. As already discussed in section 7.1.2, it is possible to have a no-majority state.

Difference Type	Majority State	Replica Needing Update	Category
IR1	Has F	B	Missing File
IR1	Does not have F	A	Added File
IR2	Has F	A	Missing File
IR2	Does not have F	B	Added File
IR3	File F has checksum C_B	A	Changed File
IR3	File F has checksum C_A	B	Changed File
IR3	No majority state	At least one	F might be irrecoverable

Table 11: Categories of the Detected Differences

Note that changes are propagated to the replicas through the normal repair mechanism. Hence, the repair order generated for the legitimate changes detected by the RIC looks like a repair order generated for illegitimate changes. The repair mechanism follows the same steps to fix illegitimate changes and to propagate legitimate changes.

7.2.3 RIC Repair Model

This subsection suggests how legitimate changes are propagated to the replicas using the information generated by the RIC.

The aim of the repair is as follows. After the two replicas are repaired, they should have the changes that were on the other replica but were not originally on the replica.

For example, Figure 7 shows a replica group ABC. When change1 and change2 are applied to the replicas, change1 is applied to A and C, and change2 to B and C. Then A's RIC picks B to compare their states. After confirming the checksum files' differences with ABC, the RIC reports that B has change2 which A should get and A has change1 which B should get. After the repair mechanism is run, AB both have change1 and change2.

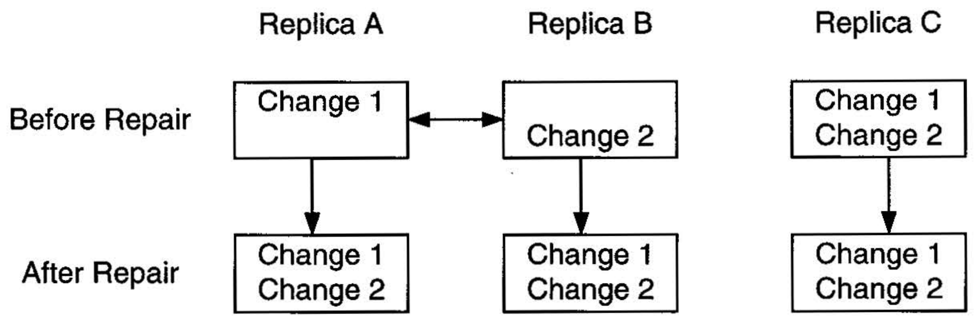


Figure 7: Progress of Replicas' States

Table 12 lists the cases and their fixes. The most interesting case listed in the table is the first—when the checksum file is corrupted or deleted. We will analyze why the suggested fix works in section 8.1.3. Note that A's and B's validated checksum files do not have to be updated in this case: the task of updating the checksum files can be delegated to A's and B's LICs.

Replicas	Conditions	Suggested Fix
Either	Checksum file is corrupted	Do not fix anything; let the LIC handle it
Either	Changes that was not originally applied to either replica	The changes are retrieved and applied to the replica.

Table 12: Fixes Needed For Changes Reported by the RIC

7.3 Another View of the Detection Scheme

The above discussion describes what the LIC and RIC do to detect changes in the system. What is not explicitly described is how the LIC and the RIC are related to the replicated system, the fault model, and the change model. The discussion in this section will alleviate that.

The proposed detection scheme detects the divergence of the replicas' data from the view of what the system's data should be. We have categorized the divergence: illegitimate changes and legitimate changes. According to the fault model, the divergence caused by illegitimate changes happens because of the disk characteristic. According to the change model, the divergence caused by legitimate changes happens because the replicas do not have a single view of the legitimate changes: the changes might not be applied to all replicas.

What is the system's view—the view that sees all and only the valid data in the system—in our model? In our model, the system's view is all the data that can be confirmed by the majority of the replicas—confirmed by referring to the real data and not the checksum files' entries. For file existence, only files that exist on the replica majority are valid system data; files that do not exist on the replica majority, but might exist on some replicas, do not belong to the system's data. For file contents, contents whose value is agreed on by the replica majority is the system's file contents. The values of contents that do not agree with the majority are divergence.

How are the checksum files on the replicas related to the system's view? In the static system, the checksum files on the replicas are identical to the system view.

In the dynamic system, the checksum files on the replicas are subsets of the system's view. Why are they subsets of the system's view? When a replica's checksum file is refreshed by an LIC, the checksum file contains all past (correct) history of the replica's data up to the refreshing point—without the legitimate changes that are applied to other replicas. The system view, however, has all past history of the replicas with all legitimate changes to the system. Hence, the checksum files are subsets of the system view.

Table 13 shows the system view and the checksum files as "sets." Note that putting deletion and update into the sets are dependent on the existence of prior addition: a file cannot be deleted or updated unless it exists.

Set Descriptions	Explanation
V	System's View
$V+Add_{UID1}$	Updated system view with file addition
$V+Del_{UID1}$	Updated system view with file deletion
$V+Update_{UID1}$	Updated system view with file update
$CHECKSUM_A$	Replica A's checksum file

$\mathrm{CHECKSUM}_A$+ $\mathrm{Add}_{\mathrm{UID1}}$	Updated A's checksum file with file addition
$\mathrm{CHECKSUM}_A$+ $\mathrm{Del}_{\mathrm{UID1}}$	Updated A's checksum file with file deletion
$\mathrm{CHECKSUM}_A$+ $\mathrm{Update}_{\mathrm{UID1}}$	Updated A's checksum file with file update

Table 13: System's View and Checksum File as Sets

The LIC and the RIC use checksum files to detect changes on the replicas. However, the checksum files are subsets of the dynamic system's view. Hence, in the dynamic system, differences or changes detected by the LIC and RIC must be confirmed with the system's view—by confirming with majority state of the replicas.

8. Analysis

The objectives of the analysis in this section are to show that the synthesized scheme detects two types of occurrences:

- all faults occurring in the fault models as discussed in section 5
- all legitimate changes discussed in the change model also in section 5

The analysis has two parts. One is for the static system, and the other for the dynamic system. Although the L2000 system is dynamic, the static system is analyzed because the analysis is simpler and a part of the analysis applies to the dynamic system.

The are two types of changes. Legitimate changes—which happen only in the dynamic system—include file addition, file deletion, and file update. Table 14 lists the faults, or illegitimate changes, that occur in both the static and the dynamic systems. E1 is when a replica's data file disappears. E2 is when a file is added to the replica. E3 is when a data file's contents is changed illegitimately. E4 is a special case of E1—the checksum file that captures a correct state of the replica disappears. E5 and E6 are when the contents of the checksum file is updated.

Events	Explanations
E1	Missing file
E2	Added file
E3	Changed file
E4	Checksum file missing
E5	Checksum file updated inconsistently.
E6	Checksum file updated consistently.

Table 14: Fault Scenarios

What is the difference between E5 and E6? Remember that when the LIC finishes recording the replica's UIDs and their checksums to a checksum file, it also records the checksum of the checksum file. If the checksum file is changed and the recorded checksum is not, then when the file's checksum is recomputed, the recomputed checksum will not match the recorded checksum, and vice versa. This is an "inconsistent" change. The "consistent" change is when the checksum file and the recorded checksum are both changed, and the recorded checksum is changed in such a way that the recomputed checksum matches the recorded checksum.

We first discuss the static system and follow with the dynamic system.

8.1 Static System

All changes to the data are illegitimate in the static system. Hence, all changes to the data—file's contents change, file disappearance, file appearance—all need to be detected and recognized as faults.

We claim that the local integrity check—without the confirmation through voting—will detect all faults. We first discuss why the detection scheme works for events E1, E2, and E3 in section 8.1.1. We then discuss E4 and E5 in 8.1.2, and then finish with E6 in 8.1.3. We will conclude this subsection in 8.1.4 with three general observations: why confirmation through voting is unneeded in the static system, why the RIC is unneeded in the static system, and how the checksum file can always be correct if it too can be corrupted or deleted.

8.1.1 E1, E2, and E3

In the cases of E1, E2, and E3, we assume that all the checksum files done in the 1st period through the Nth—F_1 through F_N—are correct. Also, there are no faults in the system in those periods, or else the occurring faults are repaired as discussed in section 7.1.3. The checksum file done in the previous period is the Nth checksum file F_N. The checksum file done in the current period is the (N+1)th checksum file F_{N+1}. If there are differences between F_N and F_{N+1}, the changes to the storage system's state must account for the differences since F_N reflects the previous system's state.

All changes in the static system are faults and therefore, the detected changes can only be E1, E2, or E3. As listed in Table 7, E1, E2, and E3 would obviously be detected if F_N is correct.

In the above discussion, we assume that F_1 through F_N are correct; operationally, how can this assumption be true? The critical assumption here is that F_1 is correct. We justify the assumption by saying that we want to ensure that the initial state of the system does not change throughout the system's lifetime. The initial state must be correct by definition, and F_1 is a part of the initial state; hence, F_1 is correct.

We can easily verify that F_2 through F_N are correct: they must be equal to F_1. If we calculate F_1's checksum and record it, when we compare F_2 through F_N to F_1, we know that the real F_1 is being used because we can detect changes in F_1 by recomputing F_1's checksum and comparing the checksum with F_1's recorded checksum. If we know that F_N is correct, then when we compare F_N to F_{N+1}, the difference <u>must</u> be the changes done to the replica's files.

8.1.2 E4 and E5

For E4 and E5, the correctness of the detection mechanism is clearer. The local detection does not work if the checksum file was deleted; therefore, E4 will obviously be detected. Since we record the checksum of the checksum file, E5 is detected: the checksum of the changed checksum file is different from the checksum of the original checksum file.

8.1.3 E6

A checksum file can be changed consistently in two ways:

- the file bits and the recorded checksum are updated consistently
- outdated but consistent checksum file replaces the current checksum file.

In the first case, we assume that the checksum algorithm is good enough, as discussed in section 6.4, that the probability of the bits and the recorded checksum being changed simultaneously and consistently is so low that we can ignore it.

In the second case, E6 is not a problem in the static system. Figure 2 shows the progress of the replica's state. If the system is static, and all faults on the system are repaired properly, then the checksum data should be the same from period 1 through period N. Hence, it does not matter which of the previous period's checksum file replaces the stored checksum file. The checksum files are identical.

8.1.4 Observations

There are three observations: why confirmation through voting is unneeded in the static system, why the RIC is unneeded in the static system, and how the checksum file can always be correct if it too can be corrupted or deleted.

The first observation is, in the static system, it is sufficient for the LIC to stop after generating the local difference report: confirmation through voting is unneeded. This is because, as discussed in section 7.1.2, confirmation through voting is a mechanism used to differentiate faults in the system from the legitimate changes. Since static system does not have legitimate changes, all changes listed in the local difference report are illegitimate. Therefore, the reported changes do no need to be confirmed by the replica majority.

The second observation is, it is unnecessary to run the LIC as daemons in the static system. This is because the LIC's function is to detect the legitimate changes not applied to all the replicas. In the static system, there is no legitimate changes, and therefore, the LIC is unneeded.

The third observation is, the LIC's correctness heavily depends on the checksum file being correct. So how can we make sure that the checksum file is always correct when it too can be corrupted or deleted? As suggested in section 7.1.3, when the checksum file of a replica is corrupted, another replica's checksum file is retrieved to replace the corrupted checksum file. Since the system is static, and all the validated checksum files have the same view—the system view, all replicas' checksum files are identical. So the corruption or the deletion of a replica's checksum file does not effect the integrity checkers' correctness because replacing the checksum file and running the LIC on the replacement file restores the checksum file to the correct state.

8.2 Dynamic System

What has changed in the dynamic system from the static system? The files that are added, deleted, or changed now may be legitimate changes to the system. Also, legitimate changes to the system might not be reflected by all the replicas.

What are the implications of the changes? There are three. First, we can no longer take the output of the check as in the static system—local integrity check without confirmation through voting—as the correct repair list. The changes might be legitimate: the files were added, deleted, or changed deliberately. The local difference report needs to be confirmed with other replicas, i.e., by confirmation through voting. Second, because the checksum files are not the same through all the periods, without confirmation through voting, previous checksum file replacing the current checksum file is a problem: the local difference report might generate inaccurate repair list. Finally, we now need to employ the RIC to check for changes that are not reflected by all the replicas.

In the following subsections, the LIC and the RIC will be discussed in the context of the dynamic system in sections 8.2.1 and 8.2.2 respectively. Section 8.2.3 offers an observation of why the corruption or the deletion of the validated checksum file is handled properly in the dynamic system. Section 8.2.4 offers case by case analysis of fault detection in the dynamic system.

8.2.1 Local Integrity Check

As in the static system, we claim that the LIC's local difference report contains all changes that occur during the last checking period N and the current checking period N+1. The argument is the same as in the static system which was discussed in section 8.1.1.

In the static system, all changes listed in the local difference report are illegitimate: the static system does not allow changes. However, the changes in the dynamic system can be illegitimate or legitimate. Therefore, the changes in the report must be checked for legitimacy through confirmation through voting. Table 8 lists the cases how the changes are confirmed.

Because a replica majority represents the system view, it is obvious that the legitimacy of the changes can be determined by voting except in the case of no-majority state. We do not consider the no-majority state because it is an intolerable event. Thus, LIC's report lists all and only faults that occurred on the replica during N and N+1 periods: E1, E2, and E3 are detected.

As in the static system discussed in 8.1.2, E4 and E5 are detected.

In the static system, E6 does not cause a problem because all periods' checksum files are the same. In the dynamic system, the checksum files are not the same: they cannot be because the system's data change. Hence, if an outdated checksum file is used in a comparison, the local difference report will list changes that are valid on the replica but were not listed in the outdated checksum file as part of the replica's data. However, when the changes are confirmed with the replica majority, all legitimate changes will be categorized as legitimate changes and all illegitimate changes as faults. Therefore, although the LIC does not explicitly detect E6, E6 does not cause the LIC to generate an inaccurate repair list.

Another way to view why E6 is handled correctly requires a reference to section 7.3. All validated checksum files—checksum files that do not incorporate faults as all the replicas' checksum files used by the LICs and the RICs are—are subsets of the current system view. Hence, when an old validated checksum file is used in comparison, the local difference report will list all legitimate and illegitimate changes that occurred on the replica since the time the checksum file was validated. No illegitimate changes can escape the detection because the validated checksum file only contains legitimate data. When the changes are verified with the system view by conferring with other replicas, the changes' legitimacy will be categorized correctly since the majority view—the system view—reflects the true state of the system. Hence, illegitimate changes—E1, E2, and E3—are the only changes listed by the LIC in the dynamic system.

8.2.2 Remote Integrity Check

The RIC checks for the data divergence of a replica from the replica majority by comparing replicas' checksum files. Why is it that the RIC will check for all changes that are not applied to all the replicas?

As discussed in section 7.3, the checksum files on the replicas are subsets of the system view. If the system remains static after a change, the union of the validated checksum files on all replicas becomes the system view because the LICs updates the checksum files using the data that reflects the system's view. Hence, when an RIC detects differences of a replica's checksum file from other replicas' and the repair mechanism updates the checksum file with the appropriate changes, the checksum file will eventually reach the system view—given that the system view has not changed since the change. Hence the RIC will eventually detect all changes that were applied to other replicas but not to the local replica.

8.2.3 Observation

A dynamic replica relies on the LIC to detect all faults occurring on the replica, and the RIC to detect legitimate changes that are not applied on the replica. The LIC and the RIC both rely on the correctness of the replica's checksum file, so how can the checksum file be always correct if it too can be corrupted or deleted?

In the static system, when a replica's corrupted checksum file is replaced with another replica's validated checksum file, it is not a problem since all validated checksum files are the same. In the dynamic system, the checksum files of different replicas might not be the same because changes might not be applied to all the replicas: some replicas' views will be closer to the system view than the others. Hence, replacing the corrupted checksum file with another replica's validated checksum file does not achieve the same effect as in the static system.

To argue that replacing a corrupted checksum file with another replica's validated checksum file is correct, we again refer to section 7.3. All validated checksum files on the replicas are subsets of the system's view. When another replica's checksum file replaces the local replica's checksum file, the replacement file does not include any faults: faults are not part of the system view and all validated checksum files are subsets of the system view. Hence, the LIC will detect all faults as long as the checksum file used for comparison is a subset of the system view. With the replacement checksum file, the LIC will again bring the checksum file in sync with the replica's view of the data.

8.2.4 Case by Case Analysis

The above analysis discusses the cases in abstract: details are grossed over. In this section, a few samples where faults occur on the system, or where changes are applied to the system, will be discussed. Readers who have already understood the detection scheme might want to skip over this section.

8.2.4.1 Faults on a Replica

This is the case where addition, deletion, or update faults occur on a replica. We start with an addition fault, follow with a deletion fault, and finish with an update fault. In all cases, we suppose that the events happen on the replica group ABC.

In the addition case, we suppose that a file D illegitimately appears on the replica A during the periods N and N+1. When A's LIC generates the checksum file F_{N+1}, F_{N+1} will have D as an entry. Therefore, when the LIC compares the previous checksum file F_N with F_{N+1}, the addition of D will be detected because F_N does not have D. When the LIC confers with other replicas about D, it will determine that D's addition is illegitimate because only A has D. The repair mechanism then will eliminate D. F_N, with the updated time stamp, is used as the next validated checksum file.

In the deletion case, events symmetric to those in the addition case happen. We suppose that a file D illegitimately disappears from the replica A during the periods N and N+1. When A's LIC generates the checksum file F_{N+1}, F_{N+1} will not have D as an entry. Therefore, when the LIC compares the previous checksum file F_N with F_{N+1}, the deletion of D will be detected because F_N has D. When the LIC confers with other replicas about D, it will determine that D's deletion is illegitimate because the replica majority has D. The repair mechanism then will reinsert a good copy of D using a copy from another replica. F_N, with the updated time stamp, is used as the next validated checksum file.

In the file change case, we suppose that a file D's contents is illegitimately updated on the replica A during the periods N and N+1. When A's LIC generates the checksum file F_{N+1}, D of F_{N+1} will have a different checksum than D of F_N. Therefore, when the LIC compares the previous checksum file F_N with F_{N+1}, the update of D will be detected. When the LIC confers with other replicas about D, it will determine that D's contents reflected in F_{N+1} is illegitimate because the replica majority's D has the same checksum as D in F_N. The repair mechanism then will replace D with a good copy of D from another replica. F_N, with the updated time stamp, is used as the next validated checksum file.

8.2.4.2 Changes Reflected on a Replica

This is the case where addition, deletion, or update changes occur on a replica. We start with an addition change, follow with a deletion change, and finish with an update change. In all cases, we suppose that the events happen on the replica group ABC.

In the addition case, we suppose that a file D is legitimately created on the replicas A and B during the periods N and N+1. When A's LIC generates the checksum file F_{N+1}, F_{N+1} will have D as an entry. Therefore, when the LIC compares the previous checksum file F_N with F_{N+1}, the addition of D will be detected because F_N does not have D. When the LIC confers with other replicas about D, it will determine that D's addition is legitimate because A and B have D. The repair mechanism then will incorporate D into F_N's entry and updates F_N's timestamp. The updated F_N is used as the next validated checksum file.

In the deletion case, we suppose that a file D is legitimately deleted from the replicas A and B during the periods N and N+1. When A's LIC generates the checksum file F_{N+1}, F_{N+1} will not have D as an entry. Therefore, when the LIC compares the previous checksum file F_N with F_{N+1}, the deletion of D will be detected because F_N has D. When the LIC confers with other replicas about D, it will determine that D's deletion is legitimate because A and B do not have D. The repair mechanism then will strike D's entry from F_N and updates F_N's timestamp. The updated F_N is the next validated checksum file.

In the file change case, we suppose that a file D's contents is legitimately updated on replicas A and B during the periods N and N+1. When A's LIC generates the checksum file F_{N+1}, D of F_{N+1} will have a different checksum than D of F_N. Therefore, when the LIC compares the previous checksum file F_N with F_{N+1}, the update of D will be detected. When the LIC confers with other replicas about D, it will determine that D's contents reflected in F_{N+1} is legitimate because the replica majority's D has the same checksum as D in F_{N+1}. The repair mechanism then will update D's entry in F_N with the new checksum and updates F_N's timestamp. The updated F_N is used as the next validated checksum file.

8.2.4.3 Changes not Reflected on a Replica

This is the case where addition, deletion, or update changes occur on other replicas but not on the local replica. We start with an addition change, follow with a deletion change, and finish with an update change. In all cases, we suppose that the events happen on the replica group ABC.

In the addition case, we suppose that a file D is legitimately created on the replicas B and C. A's LIC does not detect the addition because D is not on A. When A's RIC picks B as the partner and compare A's and B's checksum files, the difference with D is detected because B's checksum file has D but A's does not. When the RIC confers with other replicas about D, it will determine that D's addition is legitimate because B and C have D. The repair mechanism then will put D on A using D from B or C. When A's LIC runs next time, D's entry will be incorporated into A's validated checksum file.

In the deletion case, we suppose that a file D is legitimately deleted from the replicas B and C. A's LIC does not detect the addition because D is still on A. When A's RIC picks B as the partner and compare A's and B's checksum files, the difference with D is detected because B's checksum file does not have D but A's does. When the RIC confers with other replicas about D, it will determine that D's deletion is legitimate because B and C do not have D. The repair mechanism then will delete D from A. When A's LIC runs next time, D's entry will be deleted from A's validated checksum file.

In the change case, we suppose that a file D is legitimately changed on the replicas B and C. A's LIC does not detect the addition because D is still the same on A. When A's RIC picks B as the partner and compare A's and B's checksum files, the difference with D is detected because D in B's checksum file has a different checksum than D in A's checksum file. When the RIC confers with other replicas about D, it will determine that B's D is legitimate version of D because D's on B and C are the same. The repair mechanism then will update D on A using D from either A or B. When A's LIC runs next time, D's new checksum will be incorporated into A's validated checksum file.

9. Issues Not Addressed in this Project

The issues that are important, but are not addressed in this project include:

- What's new mechanism [Library2000-1].
- The mapping from UIDs of documents to the data [Library2000-2]. The proposed system uses path names as UIDs.
- Algorithm that calculates good checksums of objects that contain large amount of data.
- Group membership protocol. The proposed system will have fixed group membership.
- Property that updates are either ignored or become reliable in a bounded time. The design has the property that, if the updates are reflected on the majority of the replicas, it is highly probable that the updates will become reliable within a bounded time.
- Comparison to or usage of update propagation schemes that rely on rumor mongery [Shroeder1] or anti-atrophy sessions [Golding1, Golding2] which may be more effective than the proposed design.
- The detection scheme works in the narrowly defined model. However, the scheme is supposed to outlast technologies, but if the technologies change enough that the defined model no longer applies, would the scheme still work? Do we aim to have it work?
- Fault detection technique that reads all the data on a system does not scale well when the system scales. Disk striping technique might need to be incorporated into the discussed scheme.

Out of all the issues pointed out here, the last one is possibly the most urgent. Let us expand on the problem.

9.1 I/O Throughput and Striping

If our system reads the data serially, and we can assume that the I/O throughput is the limiting factor in reading the data, then Equation 3 shows the time needed to do local integrity check.

$$Time\_taken\_to\_check\_data = \frac{\#\_of\_disk\_read \times size\_of\_disk}{I/O\_Thruput}$$

Equation 3: Time Needed For Local Integrity Check (Serial Read)

If our system reads the data by striping, and we can assume that the bus throughput is the limiting factor in reading the data, then Equation 4 shows the time needed to do local integrity check.

$$Time\_taken\_to\_check\_data = \frac{\#\_of\_disk\_read \times size\_of\_disk}{Bus\_Thruput}$$

Equation 4: Time Needed For Local Integrity Check (Striping)

Table 15 shows a server's configuration in the year 1992 and the year 2000. Table 16 shows the time needed to complete local integrity check—without confirmation by voting—doing serial read and striping.

Year	Server System Configuration
1992	• 50 disks; each with 2 Gbytes • I/O throughput: 5 Mbytes/Sec (SCSI-II) • Bus throughput: 133 Mbytes/Sec (32-bit PCI)
2000	• 50 disks; each with 100 Gbytes • I/O throughput: 10 Mbytes/Sec (?) • Bus throughput: 266 Mbytes/Sec (64-bit PCI?)

Table 15: Server Configuration

Year	Serial Read	Striping
1992	5.56 hours	12.53 seconds
2000	5.79 days	5.22 hours

Table 16: Time Needed To Complete Local Integrity Check

We can deal we the scaling problem in two ways:

- by using striping
- by scaling up the T_{LOCAL} and T_{REMOTE}—the time intervals the LIC and the RIC operate

Both have disadvantages. If we use striping, the design, the analysis, and the implementation of the detection scheme might be so complex that we cannot be sure the scheme is correct. By scaling up the checking periods, we lengthen the window of vulnerability of the data. So it is unclear at this point which approach is more advantageous.

10. Acknowledgment

My contribution to the L2000 replication research has been mostly the analysis of the scheme. Most ideas are from [Library2000-1] paper, Mitchell Charity, and Jerry Saltzer. Contributions to the report were made by Thomas Lee, Jeremy Hylton, and Ali Alavi.

I especially thank Mitchell and Jerry for patiently explaining to me issues that I needed to think about—from embarrassingly simple issues to the most subtle ones. I thank Mitchell for spending a great deal of time with me exploring different designs, and in helping me set up the environments that I needed to get things going.

Bibliography

[6.826-1] 6.826 Class Handout #47. *Replication Techniques.* MIT Laboratory of Computer Science. November 22, 1993.

[6.826-2] 6.826 Class Handout #42. *Consensus.* MIT Laboratory of Computer Science. November 15, 1993.

[Dally1] Dally, W. J. 6.823 Lectures Note #14. MIT Laboratory of Computer Science. Spring 1994.

[Golding1] Richard A. Golding. *A weak-consistency architecture for distributed information services.* Technical Report UCSC-CRL-92-31 (6 July 1992). Concurrent Systems Laboratory, University of California at Santa Cruz.

[Golding2] Richard A. Golding and Darrell D. E. Long. *The Performance of Weak-consistency Replication Protocols.* Technical Report UCSC-CRL-92-30 (6 July 1992). Concurrent Systems Laboratory, University of California at Santa Cruz.

[Gray1] Jim Gray and Andreas Reuter, “Fault Tolerance,” from *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993, pp. 93-156.

[Hisgen1] Hisgen, Andy, et al. “Granularity and Semantic Level of Replication in the Echo Distributed File System,” *Proceedings of the Workshop on Management of replicated Data*, Houston, Texas (November 8, 1990), pp. 2-4.

[Library2000-1] Storage Service Replication Design Thoughts, Ideas from Mitchell Charity, Win Treese, and Jerry Saltzer, Library 2000, Draft of 1/7/93. /afs/athena.mit.edu/user/other/Saltzer/library/work-in-progress/think-pieces/storage-replication.txt.

[Library2000-2] Semantics of the Library 2000 Network Storage Service, Version of March 9, 1993, Notes from meetings and discussion attended by Mitchell Charity, Quinton Zondervan, Manish Mazumdar, Rob Miller, Thomas Lee, Win Treese, Ron Weiss, and J. H. Saltzer. /afs/athena.mit.edu/user/other/Saltzer/library/work-in-progress/think-pieces/storage-server.txt

[Liskov1] Barbara Liskov, et al. *Replication in the Harp File System*, MIT LCS Technical Report MIT/LCS/TM-456. August 1991.

[Page1] Page, Thomas W., Jr., et al. *Architecture of the Ficus Scaleable Replicated File System.* UCLA Computer Science Department Technical Report CSD-910005, March 1991.

[Patterson1] David A. Patterson, Garth Gibson, and Randy H. Katz. *A Case for Redundant Arrays of Inexpensive Disks (RAID).* ACM SIGMOD Conference, pages 109-116, June 1988.

[Saltzer1] Jerome H. Saltzer. LIBRARY 2000, A research prototype of the on-line electronic library of tomorrow. /afs/athena.mit.edu/user/other/Saltzer/library/work-in-progress/prospectus.txt. October 31, 1991

[Satyanarayanan1] Mahadev Satyanarayanan, et al. "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers* 39, 4 (April 1990), pp. 447-459.

[Schroeder1] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with Grapevine: the growth of a distributed system. *ACM Transactions on Computer systems,* **2**(1):3-23 (February 1984).

Appendix

11. Implementation

This section describes some of the implementation details done so far in the project. I must admit that I have not done much in implementing the design except laying out the ground works so that I could complete my MEng thesis, which will not be pursued, more easily.

11.1 Pathname as UID

We have not yet agreed upon the UIDs that will be used in the system. Since this is a pilot project that studies the feasibility of the algorithm, we decide to use path names as the documents' UIDs. Moreover, we do not want to associate any particular information to the documents. Hence, a ULTT is not needed.

11.2 Network Interface

Network interface's functionalities are described in section 6.1. The implementation is located in /r/development/projects/kom/server.pl. The interface can be accessed by creating a TCP/IP connection (such as telnet) to reading-room-3.lcs.mit.edu using port 1. The connection creation will wake up MUXD daemon which listens to port 1. To initiate one of the network interfaces, "rep1", "rep2", and "rep3" can be given to the MUXD daemon which will run the specified program. 3 interfaces are provided because we are trying to simulate 3 replicas on one machine.

Table 17 shows the primitives supported by the implemented interface. "enum" primitive is not needed by the RIC, but was implemented for an earlier design of the RIC. cksumrecord has not been implemented.

Functions	Descriptions
enum	Enumerates all UIDs on the systems.
get(UID)	Gets the checksum and the data of the specified UID
cksum(UID)	Returns the checksum of the UID
bye	Quits the interface.

Table 17: Network Interface Implemented

Suppose a process needs to retrieve the data with the UID "1/2" from the replica 1 (out of 3) using the interface, the process follows the following procedure:

1. telnet reading-room-3.lcs.mit.edu 1\n
2. rep1\r\n
3. get\n
4. 1/2\n
5. bye\n

The interface does not give a response until step 4 is completed when the interface returns 4 data items: the normal completion code "normal", the checksum for "1/2", the number of lines in "1/2", and the contents of "1/2".

Other primitives listed in Table 17 have similar usage.

11.3 Replica Interface

Replica interface's functionality is described in section 6.2. By design, the network interface, as described in Appendix 11.2, should be implemented on top of the replica interface. The network interface is initiated by MUXD daemon to satisfy the interface user's request. To satisfy the request, the network interface should invoke the functionalities of the replica interface.

Since the replica interface needs to service the network interface and other detection mechanisms, it can be conveniently implemented as Perl scripts that can be run separately from other parts of the system. When the network interface or the detection mechanisms use the replica interface, they execute the associated Perl scripts to access the functions. If a UID is specified through the interface, the replica interface first translates the UID to a file location and then satisfies the particular operation.

The above two paragraphs describe the ideal implementation of the replica interface. However, the design of the system comes after an exploratory implementation. Hence, the priori implementation is somewhat different from the design.

The implementation of the replica interface's functionality is embedded in the network interface as well as a few executable Perl scripts. All primitives listed in Table 17 are embedded in the network interface program /r/development/projects/kom/server.pl. The enum&cksum primitive is implemented in the checksum file generator as will be described in Appendix 11.4.3. The other primitives not mentioned have not been implemented.

11.4 What was Completed

Table 18 lists the tasks that have been completed in the project. The first column lists the areas that I wanted to address. The second column lists the tasks that have already been done in that area. All Perl codes can be found in /r/development/projects/kom on reading-room-3.lcs.mit.edu.

Areas	Completion
Environments	• Network Interface • File system structures set up for testing
Local Integrity Checker	• Checksum file generator • Checksum file comparator
Remote Integrity Checker	• Checksum file comparator • Routines that interface with network interface

Table 18: Tasks Completed

Each of the completion in Table 18 is described briefly as follows. Note that the checksum file comparator is used by both the LIC and the RIC.

11.4.1 Network Interface

What was done to implement the network interface was described in Appendix 11.2.

11.4.2 File System Structures

reading-room-3.lcs.mit.edu is set up to simulate 3 replicas. The programs for each replica and its data are located in /r/development/projects/rep. Each "replica" has its own program and data directories.

11.4.3 Checksum File Generator

The checksum file generator is a program that lists all the files under a specified starting directory and their checksums. The program requires one parameter: the starting directory. The program, located in /r/development/projects/kom/ckfiles.pl, can be used to generate a checksum file used in local integrity check.

11.4.4 Checksum File Comparator

The checksum file comparator is a program that compares two checksum files generated by the checksum file generator and lists the differences between the files. The outputs are composed of three files: the add files (*.add), the delete file (*.del), and the update file (*.upd). The program takes three parameters: the new file, the old file, and the output files' prefix. The program checks what has changed from the old file in the new file, and output to the files with the specified prefix. The program is located in /r/development/projects/kom/repfiles.pl.

11.4.5 Local Integrity Checker Skeleton

This is the skeleton of the LIC daemon. It incorporates the checksum file generator and the checksum file comparator to do the local integrity check. When the program is started, it will delay for a random period of time before it starts executing. Once it finishes executing, it sleeps for T_{LOCAL} minutes before it executes again. The program is located in /r/development/projects/kom/localck.pl. Note that the step to categorize the legitimacy of the changes has not been implemented.

11.4.6 Remote Integrity Checker Skeleton

This is the skeleton of the remote integrity checker. The skeleton is the implementation of the previous design of the remote integrity checker. Although it does the wrong thing, meaning it does something else rather than the proposed design, it contains the random delay, as in 11.4.5, and other miscellaneous functions needed to implement the designed RIC. The program is located in /r/development/projects/kom/remoteck.pl.

11.4.7 Routines That Interface With Network Interface

The routines are built as part of the original design of the RIC. They are scattered throughout /r/development/projects/kom/remoteck.pl. The routine initiates a TCP/IP connection to MUXD daemon as well as starting the network interface.

11.5 Cache and Detecting Faults on Disk

The question is how do we make sure, when a file integrity is checked, that the file on the physical disk, not a copy of the file in the cache, is checked? Our AIX system does not have a utility that flushes the cache to disks. Therefore, we are never sure if a file being checked is in the cache. To simplify things, we assume that the data size on our system is much larger than the cache size. Hence, we can be reasonably sure that every time the LIC is working, the check is being done on the physical file bits instead of the cached file bits.

11.6 Order of Listing in Checksum File

We assume the UIDs have a complete order and hence, they can be sorted unambiguously. In the implementation of the integrity checker, we list the files and their checksums by the sorting order implemented in Perl's sort utility. The order makes checking the differences of the files easy: we can use a finite-state machine to do the job. Interested readers should check the detail in /r/development/projects/kom/repfiles.pl.

MEMO

To: Jerry Saltzer

From: Komkit Tukovinit

Subject: My Possibly Last Contribution to Library 2000

Attached is the paper that I wrote up on another fault detection scheme. The scheme is based on what we discussed last week.

I am not sure if the scheme meets your expectation. From what you responded to what I last wrote to you by email, it seems as if you are expecting a scheme that has two checking steps. The first step checks for the differences in the checksum files of the periods N and N-1. The second step checks the differences of the checksum files from all the replicas. As I showed to you last week, a scheme that uses checksum files containing no faults for global comparison is not software fault-tolerant. This is the case for both the scheme in my 6.199 project and the scheme that you suggested earlier.

The scheme that is proposed in this paper does not have two checking steps: it checks for faults and changes to the replicas globally. There is no notion of the local difference check or global difference check.

I cannot work any more on this paper. So I hope it will be helpful to the research effort some way or another.

KT

Attachments:

The paper.

6.199 Project Continuation
MIT Laboratory of Computer Science - Library 2000
Supervisor: Jerome H. Saltzer
by Komkit Tukovinit

Fault Detection in L2000 Storage-Replication Service

This paper is a continuation of the 6.199 report completed on May 8, 1994. The paper suggests another fault detection scheme. The scheme is different from the one proposed in the 6.199 report in the following ways: it requires less hardware resources to achieve the same level of fault tolerance, and it is software fault-tolerant.

The hardware requirement is reduced by using weak replicas in place of some of the normal (strong) replicas. A weak replica requires orders of magnitude less hardware resources than a strong replica. Software fault-tolerance is achieved by combining the local integrity checker and the remote integrity checker—the mechanisms used in the 6.199 project. The two mechanisms—detecting data faults and data divergence of the replicas—are now one mechanism, and thereby, eliminating single-point failures caused by having only the local integrity checkers check for the replicas' faults.

The detection scheme being proposed is conceptually simple. An integrity checker asks all the replicas—strong and weak—for their states and compares the states. The differences that the replicas have from the majority states are faults to be corrected or legitimate changes to be propagated.

The organization of the paper is shown in Table 1.

Sections	Descriptions
1	provides the motivation for writing this paper.
2	shows how the configuration and the number of the replicas are chosen.
3	shows how data are legitimately added and deleted from the replicas.
4	details the mechanisms used by the detection scheme.
5	shows how the detection mechanisms work.
6	wraps up the paper.

Table 1: Organization of the Paper

1. Motivation

If the detection scheme suggested in the 6.199 report works, why do we need to come up with another scheme? There are at least two problems with the 6.199 scheme:

- the scheme is not software fault-tolerant
- the scheme requires more hardware than required to detect and correct faults

First, the scheme is not software fault-tolerant; there is only one process per replica—a local integrity checker (LIC)—that checks for faults on the replica. Hence, if the LIC of a replica dies, the faults on the replica will go undetected. Also, the death of the replica's LIC will go undetected.

Second, the scheme requires too much hardware for the level of fault-tolerance it provides. For example, to detect and correct one fault, the scheme requires three strong replicas. However, only two strong replicas and one weak replica, requiring less hardware than three strong replicas, are needed to detect and correct one fault. Strong replicas and weak replicas are defined in the next section.

2. Increasing Redundancy by Using Weak Replicas

The system used to discussed the detection scheme in this paper is a (n,n-1) system. This means that the system has n strong replicas and n-1 weak replicas. This section shows how the numbers are selected. The section is organized into two subsections: the first defines weak and strong replicas, and the second shows how the numbers n and n-1 are selected.

2.1 Definitions of Weak and Strong Replicas

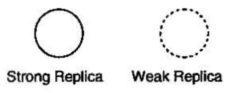
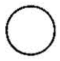
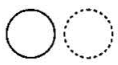
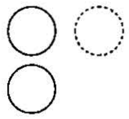
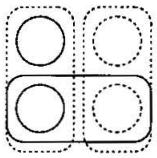
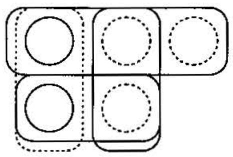
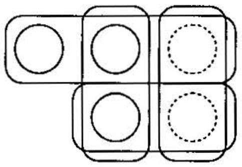
What are a strong replica and a weak replica in the system? A strong replica stores real data associated with the UIDs; a weak replica only has the checksums associated with the UIDs. From a strong replica, a client can retrieve the UIDs that the replica stores, the checksums associated with the UIDs, and the data associated with the UIDs. From a weak replica, a client can retrieve the UIDs that the replica has, the checksums associated with the UIDs, but not the data associated with the UIDs.

2.2 How n and n-1 are Chosen

In the paper's introduction, we claimed that we can replace some of the strong replicas with weak replicas and still have the same level of fault-tolerance that the system with only strong replicas provides. The question that we should ask next is: can we add more weak replicas to make the system even more fault-tolerant since adding more replicas usually provides more fault-tolerance? In the following paragraphs, we will show that we can. However, we will also show that there are complications caused by using weak replicas beyond a certain point. The complications will diminish the return from providing more weak replicas. Looking at the complications, we will conclude that the system with n strong replicas and n-1 weak replicas is the best for our situation.

We start by showing how the fault-tolerance of a system is increased by increasing the number of replicas. Figure 1 shows what is being described here. The first row shows how the replicas are shown in the figures: strong replicas are shown as solid circles and weak replicas as dotted circles. In the later figures, there are lines drawn around the replicas to show what the system can detect and correct. A solid line around replicas means that the simultaneous faults occurring on the enclosed replicas can be detected and corrected. A dotted line around replicas means that the simultaneous faults occurring on the enclosed replicas can be detected but cannot be corrected.

Figure 1: Replicas And Their Tolerance

Weak and Strong Replicas	
(1,0) System	
(1,1) System	
(2,1) System	
(2,2) System	
(2,3) System	
(3,2) System	

The first system shown—one strong replica and zero weak replica (1,0)—cannot detect or correct a fault if we are assuming that the signals given by the disk controllers cannot be trusted. To detect a fault, we add a weak replica to make a (1,1) system.

A (1,1) system can detect a fault occurring on either replicas, but cannot correct the detected fault because the replica with the fault cannot be determined. Moreover, if the fault occurs on the strong replica, the fault cannot be corrected because there is no redundant data. To be able to correct the fault, we add another strong replica to make a (2,1) system.

A (2,1) system can detect and correct a fault that occurs on any of the three replicas. Which replica is faulty can be determined by majority voting, and there always is redundant data to repair the fault. The question then becomes, can adding a weak replica increase the fault-tolerance? So we add a weak replica making a (2,2) system.

A (2,2) system can detect and correct all faults that a (2,1) system can. Moreover, it can detect and correct some of two simultaneous faults. Specifically, it detects and corrects faults that occur simultaneously on a strong replica and a weak replica. Because the faulty replicas have different data characteristics—one has the real data while the other the checksum data—the replicas most likely failed differently. Hence, if the checksums reported by the other strong replica and the other weak replica match, we can say with a high confidence that the data on the other strong replica is good.

Note that the system cannot correct simultaneous faults that occur on two replicas of the same type, e.g., both strong replicas are faulty. This is because we cannot say which sets of the replicas—strong-strong or weak-weak—are right. Since the replicas of the same types have the same data characteristics, the replicas of either set can fail simultaneously. Can adding another weak replica increase fault tolerance? So we add another weak replica making a (2,3) system.

A (2,3) system can detect and correct all faults that a (2,2) system can. Moreover, it can detect and correct two simultaneous faults that occur on two weak replicas. This is because when two weak replicas fail, there is one good weak replica left. The good weak replica can confirm that the data on the two strong replicas are correct. The system also detects simultaneous faults on the two strong replicas; there are three weak replicas that indicate what the correct checksum of the data is. However, the fault cannot be corrected because there is no redundant data. What about adding a strong replica to the (2,2) system to make a (3,2) system?

A (3,2) system can detect and correct faults as a (2,2) system does. Moreover, it can detect and correct two simultaneous faults that occur on two replicas of any types. It can also deal with some of the three simultaneous faults. Specifically, it will detect and correct faults that occur simultaneously on two strong replicas and one weak replica. This is because the other good strong replica's data will agree with the other weak replica's data. The agreement can be trusted to provide good information because the different replicas are not likely to fail the same way.

What is the point of the above descriptions? There are two. First, adding weak replicas to the system can raise the fault tolerance of the system. This is evident in the progression of the above discussion. However, the addition might not be worth the conceptual complication detailed in the second point.

Second, exploiting the redundancy provided by adding weak replicas can be complicated. We need to analyze exactly what is going on within the system instead of just relying on the majority voting. For example, in the (2,2) system, we needed to think about the failure characteristics of the weak and the strong replicas in relation to each other. We also need to differentiate the weak replicas and the strong replicas when they vote. This is evident in the (2,2) system: not all combinations of two replicas can be corrected in the system.

We would like to argue that adding weak replicas works best when we limit to the (n,n-1) system. This is because we can use a simple majority voting scheme to determine the valid view of the system. The system will be able to detect and correct n-1 faults if the replicas fail independently. The simplicity will help us in seeing if the detection scheme will detect all expected faults and changes.

3. Updated Change Model

The change model is different from the one proposed in the 6.199 report. This section describes the change. The section is organized as follows. Section 3.1 says what are legitimate changes and why they are different from the 6.199 report. Section 3.2 discusses how changes become legitimate on the system. Section 3.3 concludes section 3 discussing why changes to the system need to be categorized into the fragile and robust states.

3.1 Legitimate Changes

The change model in the 6.199 report says that legitimate changes can be file addition, file deletion, or file update. For this paper, the legitimate changes can be only file addition or file deletion. All changes to the files' contents are faults in the system.

There are two reasons why this change model is selected. First, in the L2000 system, we might want a UID of a data to uniquely identify the data. Hence, once a UID is generated for a data, the UID is for that data only, and the data should not be changed.

Second, for all the data deletion, we might want to move the data to another archival device rather than removing the data from the devices completely. This allows mistakes to be recovered: data deletion does not have to be permanent.

Note that this model is similar to the model that libraries are now using. Books can be added or removed from a library's collection, but they cannot be updated. To "update" a book with a newer version of the book, the old version is removed from the library, and a new version is added. Or else the new version of the book is added to the collection without removing the old version.

3.2 How Changes Become Legitimate

Changes become legitimate when they are applied to the majority of the replicas. The replicas where changes are applied can be weak or strong replicas. Note that since the system has a (n,n-1) configuration, changes applied to the replica majority means at least one of the strong replicas has the change. Hence, the system has enough information to propagate the changes to the rest of the replicas.

When a change is applied to a weak replica, the replica updates its records that store the UIDs stored by the replica and their associated checksums. In the addition case, a new UID and its checksum will be added to the record. In the deletion case, an existing UID and its checksum will be deleted from the record.

When changes are applied to a strong replica, the replica updates its records of the UIDs and their data. In the addition case, a new UID and its data will be added to the record. In the deletion case, an existing UID and its data will be deleted from the record. Moreover, the checksum file, a state of the replica captured by the checksum generator, as will be discussed in 4.1, is updated. In the addition case, a new UID and its checksum will be added to the file. In the deletion case, an existing UID and its checksum will be deleted from the file.

3.3 Fragile and Fault-Tolerant States

As discussed in the 6.199 project, the changes applied to the system can be in the fragile or the robust (fault-tolerant) state. Changes that are applied to the majority of the replica but have not propagated to the rest of the replicas are in the fragile states. If the changes decay beyond the point of recoverability before all the replicas receive the changes, the changes will be eliminated from the system or marked as irrecoverable since they will be seen as faults. When the changes' states become fault-tolerant, the changes have been applied to all the replicas. The fault-tolerant changes are as fault-tolerant as any other data on the system: it is highly unlikely that the data will decay beyond the point of recoverability.

4. Detection Scheme

This section describes the detection scheme. The general idea is as follows.

For each of the strong replicas in the (n,n-1) system, there is a checksum generator process. The checksum generator does nothing but generating a checksum file that lists all the UIDs on the replica and their associated checksum calculations every $T_{CHECKSUM}$ minutes, the interval that we want the integrity of the files checked. The checksums are calculated by reading the data from the storage devices.

To check the integrity of the system, $N_{CHECKER}$ integrity checker processes are used. Every random($T_{CHECKSUM}$ x $N_{CHECKER}$) minutes, a checker wakes up and requests the latest checksum files, generated by the checksum generator processes on the strong replicas, from all replicas. The checker then compares the differences among the files, checks for the majority state of those differences, and generates reports for each of the replicas that needs to be fixed or updated. The differences that each replica has from the majority states are faults that occurred on that replica or changes that the replica has not gotten.

The checksum generator is detailed in section 4.1 and the integrity checker in 4.2. Section 4.3 offers a few observations about the detection scheme.

4.1 Checksum Generator

The checksum generator generates the UIDs and checksum calculations of the UIDs for all the data located on a strong replica. There is one checksum generator for every replica; the generator does not have to be running on the replica. Every $T_{CHECKSUM}$ minutes, the checksum generator does the following:

1. Enumerates all UIDs for the data located on a strong replica
2. Produces checksum calculations for those UIDs by reading the data
3. Saves the UIDs and the checksums to a file
4. Attaches to the end of the file the current time (Greenwich means time)
5. Attaches to the end of the file the checksum of the file being written up to and including the time-stamp
6. Tells the replica to use the checksum file as the replica's checksum file

In effect, the checksum generator captures the states of a replica so that the integrity checker can compare the states of all the replicas by using the states captured by the checksum generators.

4.2 Integrity Checker

There are $N_{CHECKER}$ integrity checkers for the replica group (n,n-1). The number $N_{CHECKER}$ can be chosen to make the checkers as fault-tolerant as desired. The fault-tolerance of the checkers will be discussed in section 4.3. Each of the checkers does the following. Every random($T_{CHECKSUM}$ x $N_{CHECKER}$) minutes, the checker

1. Requests the latest checksum file from each replica
2. Checks the time-stamp of the checksum file. If the time-stamp is older than $T_{CHECKSUM}$, raise a flag and terminate.
3. Checks the checksum of the checksum file. If the recalculated checksum of the checksum file—calculation of everything up to and including the time-stamp—mismatches the recorded checksum, raise a flag and terminate.
4. Compares the checksum files from the replicas and generates a repair order for each of the replica as shown in Table 2.

This Replica	Majority State	Causes of Differences	Repair Order
Has file F	Has F	N/A	N/A
Has file F	Does not have F	Fault or Change	Delete F
Does not have file F	Has F	Fault or Change	Get F
Does not have file F	Does not have F	N/A	N/A
F has checksum C_A	F has checksum C_A	N/A	N/A
F has checksum C_A	F has checksum C_B	Fault	Replace F
F has checksum C_A	No majority state	Fault	cannot be automatically recovered.

Table 2: Repair Order Generated by the Integrity Checker

As discussed in the 6.199 report, it is possible that the file checksum has a no-majority state since checksum values are not boolean. However, a no-majority state can only occur after the number of faults has gone beyond the threshold of recoverability: an occurrence that must be made highly unlikely in our system.

In effect, the integrity checker checks for the differences that each replica has from the replica majority and generates reports about those differences. Since the states of the replicas are captured by the checksum generators that are run at different times, the compared states may be stale. However, since the integrity checker checks the time-stamps of the checksum files, the differences generated by the checkers are guaranteed not to be more than $T_{CHECKSUM}$-minute stale.

4.3 Observations

We make two observations here:

- why we have checksum generator and integrity checker instead of just the integrity checker
- how the software can be made fault-tolerant

The most peculiar thing about the scheme proposed here is why we have two different mechanisms doing two separate things. We can just have the integrity checkers capture the states of the replicas and then compare the states. The reason we have two mechanisms is so that the states of all the replicas will not have to be captured at the same time. Since the data can be large (up to 1,000,000 Gbytes), we want to make sure all the data on the replicas are not read at the same time since the reading can be time-consuming.

In the beginning of this paper, we claimed that the proposed scheme will be more fault-tolerant than the scheme proposed in 6.199 report. How is this additional fault-tolerance achieved?

Note that if an integrity checker dies, the rest of the integrity checkers will still detect the faults and changes that occur on the replicas, albeit with a larger time interval than we would like. In the 6.199 system, if the LIC of a replica dies, the faults on that replica will not be detected by other replicas. Also, the LIC's death will not be detected.

Still, there is a problem with the current scheme: a checker's death is not detected, and hence not corrected. By the Markov model, as shown in Figure 2, all the checkers will eventually fail. Note that P_{FAIL} is the probability that a checker fails. Hence, we need detection and correction mechanisms for the checker's failures to ensure that the integrity checkers will not all die. Note that the correction mechanism does not have to be automatic: we can just detect the checkers' failures and notify the system operators to restart the checkers.

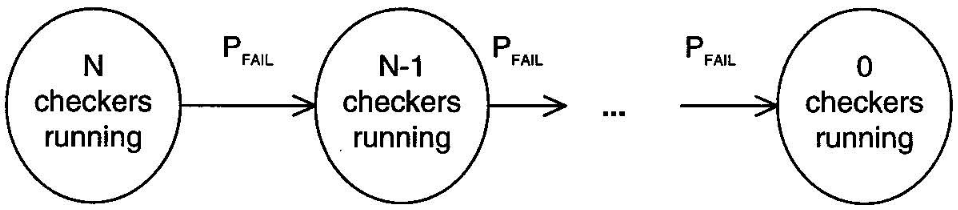


Figure 2: Markov Model of Integrity Checker's Failures

How can we detect the checkers' failures? One way to do this is to have independent processes checking if the checkers are still alive. The problem with this approach is we then have the problems of ensuring that the checkers that check the liveliness of the integrity checkers do not all die. This problem suggests that we use another scheme.

What we can do is to have the integrity checkers check on one another. This can be done when a checker wakes up and checks the integrity of the replicas. Before the checkers compare the states of the replicas, it asks other checkers if they are still alive. With this scheme, we will always know when a checker dies.

There is yet another problem with this scheme. The checker must be able to operate asynchronously. In Figure 3, an integrity checker A goes to sleep after it completes a check. When another integrity checker B wakes up and checks if other checkers are alive, the checker A will be asleep. In order for the checker A to acknowledge that it is alive, it must wake up and execute another instruction thread that is not part of the normal thread. Perl, the language that is used to partly implement the scheme in 6.199, does not provide this capability.

```
Loop:
        Is everybody alive?
        Start Checking.
        Sleep
        Goto loop
```

Figure 3: Integrity Checker Process A

5. Analysis

This section shows how the detection scheme detects all faults and changes on the system. We first show how faults are detected in section 5.1. How changes are detected is shown in section 5.2. We conclude the section by giving an observation in section 5.3 why the checksum file of a strong replica needs to be updated as well as the data when a change is applied to the replica.

5.1 Faults

This section shows how faults are detected. There are two types of faults that we worry about in this paper: data faults and software faults. The data faults include file addition, file deletion, and file update. The software fault includes the deaths of the detection processes. We first discuss the data faults in section 5.1.1, and then the software fault in 5.1.2.

5.1.1 Data Faults

There are three types of data faults: file addition, file deletion, and file update. When a fault occurs on a replica, the replica's checksum file—generated by a checksum generator—will capture this fault. Since a fault is not likely to occur simultaneously on the majority of the replicas, when the integrity checker compares the replica's checksum file to the majority state, the majority state will not have this fault. Hence, the fault will be detected. We discuss the addition fault, the deletion fault, and the update fault in order. We conclude section 5.1.1 with the discussion of the situations where a checksum file suffers a fault.

In the deletion case, suppose a file F is deleted illegitimately from a replica A. When the checksum generator of A enumerates A's UIDs and the associated checksums, F will not be listed in the checksum file. However, the checksum files from other replicas—constituting the majority—will have F. Therefore, the integrity checker will detect that F is missing on A.

In the addition case, suppose a file F is added illegitimately to a replica A. When the checksum generator of A enumerates A's UIDs and the associated checksums, F will be listed in the checksum file. However, the checksum files from other replicas—constituting the majority—will not have F. Therefore, the integrity checker will detect that F has been added to A.

In the update case, suppose a file F is updated illegitimately on a replica A. When the checksum generator of A enumerates A's UIDs and the associated checksums in the next check point, F will be listed in the checksum file with the updated checksum. However, the replica majority will list F with a different checksum. Therefore, the integrity checker will detect that F has been changed on A.

The above three paragraphs detail the situations where the data files of a replica suffer faults. What happens when the checksum file of a replica—the file we used to detect faults on the replica—suffers faults? A checksum file can have two types of faults: file corruption or file deletion.

In the file corruption case, the integrity checker will detect that the file has been corrupted. This is because after the checksum generator finishes enumerating the replica's UIDs and their associated checksums, it calculates the checksum of the checksum file and attaches the checksum to the end of the file. Hence, if the file is corrupted, when the integrity checker recalculates the checksum of the checksum file, the recalculated checksum will mismatch the recorded checksum.

In the file deletion case, the integrity checker will also detect that a checksum file has been deleted. This is because if the checksum of a replica has been deleted, when the integrity checker asks the replica for its checksum file, the replica will not have the file. Hence, that event that the replica's checksum file is missing will be detected.

5.1.2 Software Fault

Besides the data fault, there can be faults with the detection software as well. What we worry the most about software fault is some of the processes used in the detection scheme might die. The scheme will detect the processes' deaths.

Suppose a checksum generator on a replica A die. When an integrity checker wakes up and asks for the checksum file from A, A will return a checksum file that was last produced by the dead checksum generator. Eventually, the file will be older than $T_{CHECKSUM}$ minutes. An integrity checker would notice this fact because the checksum file contains the time-stamp when the file was generated.

Suppose an integrity checker dies. When another integrity checker wakes up, it will ask all the integrity checkers if they are alive. Since the dead integrity checker cannot answer, its death will be detected.

5.2 Changes

How can legitimate changes be detected? There are two types of legitimate changes: file deletion and file deletion. This section will show that both types of changes will be detected.

Suppose that a file F is legitimately added to the majority of the replicas. Because our system's configuration is (n,n-1), this means that at least one of the strong replicas has the file F. When a file is added to the weak replicas, their checksum records are updated. When a file is added to the strong replicas, their data records are updated as well as their checksum records (as discussed in section 3). Hence, in the normal case where the addition has not decayed, the majority of the checksum files will have F. Therefore, the integrity checker will treat the replicas that do not have F as if they sustained F deletion fault.

In the abnormal case, F might be deleted or updated in one of the replicas before an integrity checker is run. In case of deletion, if the deletion causes F to exist only on the replica minority, the existence of F on the minority replicas will be seen as faults and deletion repair orders will be issued for those replicas. In case of update, if the update causes the existence of F to be in a no-majority state, F will not be propagated and a flag will be issued saying that the file F cannot be automatically recovered.

Suppose that a file F is legitimately deleted from the majority of the replicas. When a file is deleted from the weak replicas, their checksum records are updated. When a file is deleted from the strong replicas, their data records are updated as well as their checksum records (as discussed in section 3). Hence, in the normal case where the deletion has not decayed, the majority of the replicas will not have F. Therefore, the integrity checker will treat the replicas that have F as if they sustained F addition fault.

The abnormal case for file deletion is when a file deleted is restored. If the restoration pushes the number of the replicas who still have F to become the majority replicas, then the minority replicas that do not have F will be treated as if they sustained F deletion faults.

In this section, we show that in the normal case, the changes applied to the majority of the replicas will be detected. The change repair orders are symmetrical to those in the fault cases. An addition change will result in the integrity checker issuing a deletion fault repair order. A deletion change will result in the integrity checker issuing an addition fault repair order. We also show that in the abnormal case where the changes decay before they are propagated, the replicas states will revert to the state prior to the changes or the changes will cause the state to become ambiguous.

5.3 Observation

Note that it is essential that the checksum file of a strong replica—generated by the checksum generator—be updated besides updating the replica's data when a change applied to the replica. This is because checksum files are used to detect the changes that are applied to the system. If the checksum files are not updated, there are situations where the changes applied to the system will be removed from the system although the changes have not decayed.

This is an example of the said situation. Suppose a file F is added to the n-1 weak replicas and one strong replica A, and A's checksum file is not updated. When an integrity checker wakes up and asks for A's checksum file, the checksum file will not have F listed. The only checksum files that have F listed will be the n-1 checksum files from the weak replicas. Since F is not listed by the replica majority, the addition of F will be treated as an addition fault. If the n-1 weak replicas are fixed appropriately, then F will only exist on A. In the next checking period, the A's checksum file will list F. However, since it is only listed on A, F will be treated as an addition fault on A. If A is fixed appropriately, F will be deleted from A. Hence, the addition change applied to the system is eliminated since A's checksum file is not updated along with the data addition.

6. Epilogue

Since I have run out of time to finish this paper, I cannot think of things to say for this section except one. This "design" is one that somewhat agrees with what Jerry thinks what the algorithm should be. I believe that the scheme works in the extent of detecting the expected faults and changes. If I were to get an MEng thesis out of this project, I would take this scheme as the detection scheme, and come up with a repair mechanism that works with the detection scheme. Since the legitimate changes on the system cannot be differentiated from the faults using this scheme, a separate change propagation mechanism is not needed. Hence, with a working repair mechanism, the replication service for L2000 would be completed.