

© Copyright by Jan Marcin Zymla, 2003

AN ARMOR-BASED FAILOVER MECHANISM
FOR OFF-THE-SHELF APPLICATIONS

BY

JAN MARCIN ZYMLA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

To my family and friends.

ACKNOWLEDGMENTS

My first thanks go to Professor Ravi K. Iyer, my thesis advisor. I would also like to thank Dr. Zbigniew Kalbarczyk, who gave me guidance and help. Thanks to my colleagues at the Center for Reliable and High-Performance Computing, especially Keith Whisnant, with whom I worked on parts of this project.

But above all, thanks to my Mom, my Dad, my whole family, and my friends who gave me their love and support during my life so far.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION	1
CHAPTER 2 TARGET SYSTEM	5
2.1 Messaging Middleware	5
CHAPTER 3 ARMORS	8
3.1 ARMOR Architecture	8
3.2 Error Detection and Recovery	9
3.2.1 Node failure	10
3.2.2 ARMOR failure	12
3.2.3 Application failure	12
3.3 Configuration for Application Failover	12
CHAPTER 4 IP ADDRESS MIGRATION	14
4.1 Concepts	15
4.1.1 Hardware and logical addresses	15
4.1.2 ARP IP-to-hardware mapping	15
4.2 ARP Spoofing	16
4.2.1 IP aliasing	18
4.3 IP Failover Setup	20
4.3.1 Primary node	20
4.3.2 Backup node	20
4.3.3 Failover	21
4.3.4 Switching roles	22
CHAPTER 5 DATABASE MIGRATION (DIRECTORY REPLICATION TOOL)	24
5.1 Motivations and Requirements	24
5.1.1 Motivations	24
5.1.2 Requirements	25
5.2 Replication Library	26
5.2.1 Interposition	26
5.2.2 Statelessness	28
5.2.3 File integrity	28
5.2.4 Error recovery	32
5.3 Local Intermediary Process	32
5.3.1 Communication with the replication library	34
5.3.1.1 Receiving information about function calls	34

5.3.1.2	Blocking during MD5 checksum computation	34
5.3.2	Communication with the remote server	35
5.3.2.1	Forwarding function call information	35
5.3.2.2	Wakeup call	37
5.3.2.3	File transfer	37
5.3.3	Error recovery	37
5.4	Remote Server	39
5.5	Supporting Applications	40
5.5.1	Adding wrapper functions	40
5.5.2	Stand-alone configuration	40
5.6	Portability and Limitations	40
CHAPTER 6 EXPERIMENTAL RESULTS		42
6.1	Performance	42
6.1.1	Performance overhead directory replication tool	43
6.1.2	Failover	47
6.2	Reliability	48
6.2.1	Fault injections to the directory replication tool	49
6.2.1.1	Fault injections to the application	49
6.2.1.2	Fault injections to the local intermediary process	49
6.2.1.3	Fault injections to the remote server	50
6.2.1.4	Directory replication tool summary	51
6.2.2	Fault injections to the middleware	51
6.2.2.1	Fault injections to the watchdog	52
6.2.2.2	Fault injections to the application	55
6.2.3	Fault injections to ARMORs	58
6.2.3.1	Node failures	58
6.2.3.2	Embedded ARMORs failures	58
6.2.4	Summary of the reliability analysis	59
CHAPTER 7 CONCLUSIONS		60
REFERENCES		62
APPENDIX A DIRECTORY REPLICATION TOOL MANUAL		64
A.1	Local Intermediary Process	64
A.2	Remote Server	64
A.3	Replication Library	64
APPENDIX B DIRECTORY REPLICATION TOOL ALGORITHMS		66
B.1	Replication Library	66
B.2	Local Intermediary Process	68
B.3	Remote Server	70
APPENDIX C DIRECTORY REPLICATION TOOL LISTINGS		72
C.1	Replication Library	72
C.1.1	Source code of <code>config.h</code>	72
C.1.2	Source code of <code>replib.h</code>	74

C.1.3 Source code of `replib.c` 76
C.1.4 Source code of `Makefile` 98

LIST OF FIGURES

Figure 1.1	IP address migration tool.	2
Figure 1.2	Directory replication tool.	3
Figure 1.3	System hierarchy.	3
Figure 2.1	Middleware processes.	5
Figure 2.2	Database access.	7
Figure 3.1	ARMORs configuration example.	10
Figure 3.2	Node failure and recovery.	11
Figure 3.3	Configuration for application failover.	13
Figure 4.1	Client-server application with primary and backup nodes.	14
Figure 4.2	Address resolution protocol (ARP).	17
Figure 4.3	ARP spoofing.	18
Figure 4.4	IP aliasing.	20
Figure 4.5	Primary and backup nodes.	21
Figure 4.6	Switching roles between primary and backup nodes.	23
Figure 5.1	System configuration.	25
Figure 5.2	Architecture of the directory replication tool.	26
Figure 5.3	Function interposition.	27
Figure 5.4	Wrapper function for <code>write()</code>	30
Figure 5.5	Sequence of events to ensure same order on both local and remote hosts.	31
Figure 5.6	Detection of library crash inside a <code>write()</code> wrapper function.	33
Figure 5.7	Sequence of events for a <code>write()</code> call.	35
Figure 5.8	Sequence of events for the transmission of a <code>write()</code> call between the local intermediary process and the remote server.	36
Figure 5.9	Sequence of events for the transfer of files between the local intermediary process and the remote server.	38
Figure 6.1	Motorola CPX8216.	42
Figure 6.2	Slowdown results.	45
Figure 6.3	Mean time to recover (MTTR).	47
Figure 6.4	Error detection and recovery hierarchy.	48
Figure 6.5	Middleware fault injection results for the watchdog module.	54
Figure 6.6	Middleware fault injection results for the watchdog module (activated faults).	54
Figure 6.7	Middleware fault injection results for module <code>test_mod</code>	55

Figure 6.8	Middleware fault injection results for the database server.	57
Figure 6.9	Fault injection results for application modules.	58

LIST OF TABLES

Table 6.1	Initialization time results.	43
Table 6.2	Slowdown results.	46
Table 6.3	Mean time to recover (MTTR) results.	48
Table 6.4	Application crash effects on the directory replication tool.	49
Table 6.5	Local intermediary process crash effects on the directory replication tool.	50
Table 6.6	Remote server crash effects on the directory replication tool.	51
Table 6.7	Outcome categories of fault-injection experiments.	53
Table 6.8	Middleware fault injection results for the watchdog module.	54
Table 6.9	Middleware fault injection results for module <code>test_mod</code>	55
Table 6.10	Middleware fault injection results for the database server.	56
Table 6.11	Fault injection results for application modules.	57

CHAPTER 1

INTRODUCTION

Reliability and high availability are essential to critical applications, especially in the field of public safety communications. At the same time these applications are usually commercial off-the-shelf (COTS) software designed to run on COTS hardware, and often error detection and recovery support is added at the end of the development cycle when all the functionality is in place. The objective of this work to provide a software solution to improve the reliability of COTS (or legacy) applications without the necessitating the source to be modified.

A hierarchical reliability solution. Errors can originate at different system levels including hardware, operating system, and application. Mechanisms to cope with these errors vary depending on error type and origin. In addition to error detection and recovery mechanisms directly built into the application, external techniques for application or node failures detection and recovery prove to be necessary. Application failures are recovered by clearing the system of remainders of the failed application and by restarting the application. Node failures are recovered by restarting the application on a backup node. This last procedure is known as *failover*. The objective of this work is to provide high-level external mechanisms for application and node failure detection and recovery. An application, part of a public safety telecommunication network, is used as a target for demonstrating and evaluating the proposed solution.

The higher-level reliability services added are based on three main components:

- **ARMORs:** ARMORs (adaptative, reconfigurable, and mobile objects for reliability) [1], [2] provide a customizable and self-checking reliability infrastructure for fault management.

ARMORs support detection and recovery from the application failures and any misbehavior of failover supporting components.

- **IP address migration tool:** This component maintains connectivity between clients and the application in case of node failure. When a node failure is detected, the backup node takes over the IP (Internet protocol) address of the primary node and all IP communications are rerouted to the backup node, as illustrated in Figure 1.1.

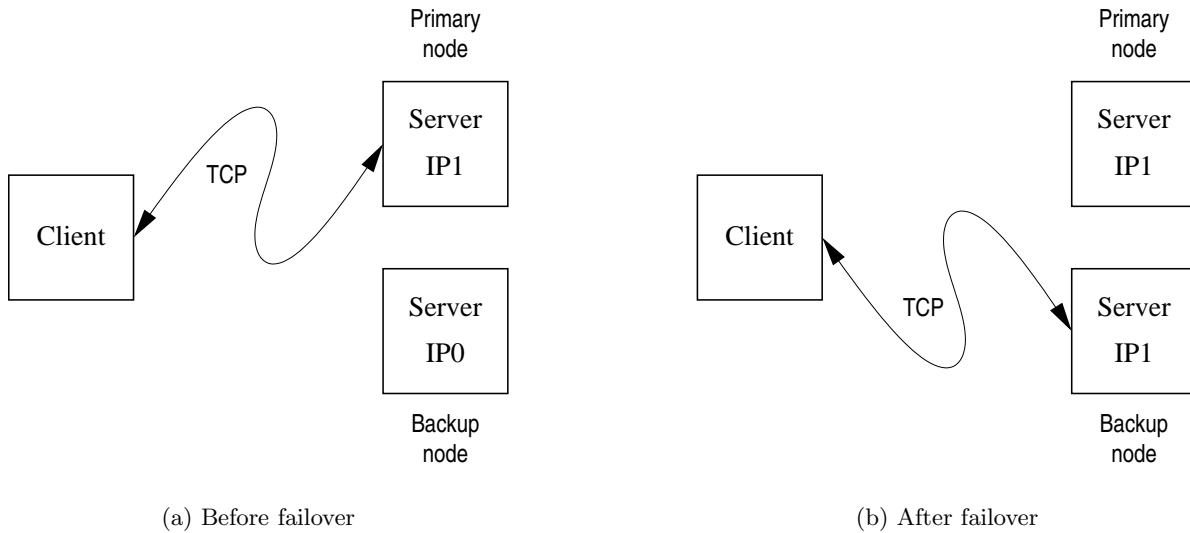


Figure 1.1 IP address migration tool.

- **Directory replication tool:** This component maintains updated copies of the data files used by the application. Whenever a file is modified on the local node, the same modification is done by the directory replication tool to the copy of the file on the backup node, as illustrated in Figure 1.2.

A middleware provides reliability services to the application, e.g., reliable message passing and recovery of application processes after a process crash. ARMORs, the IP address migration tool, and the directory replication tool provide an additional error detection and recovery layer to this middleware as illustrated in Figure 1.3, which shows the hierarchical reliability solution. Error detection and recovery inside the application processes are done by the middleware. The directory replication tool and the IP address migration mechanism, which provide failover services to the application, are controlled by ARMORs. The IP address migration mechanism is directly built-in

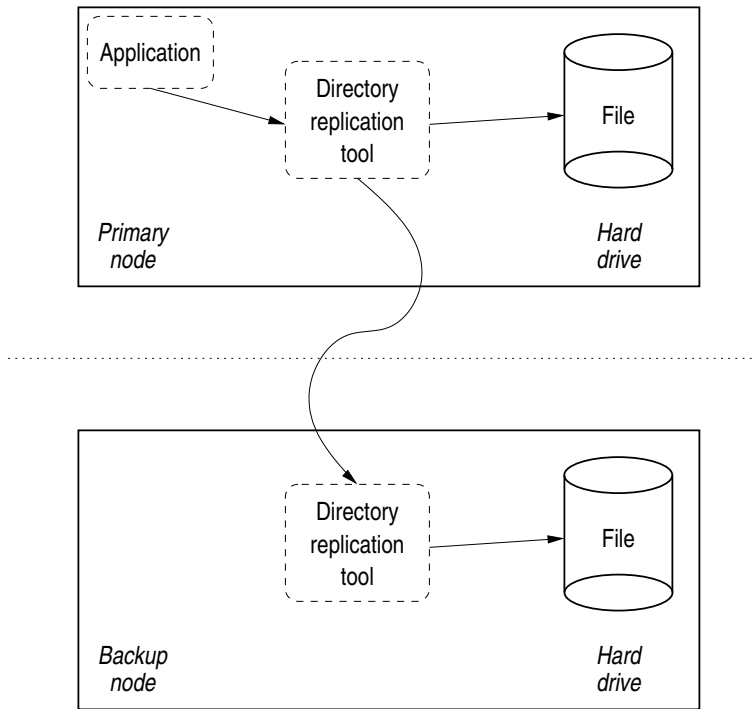


Figure 1.2 Directory replication tool.

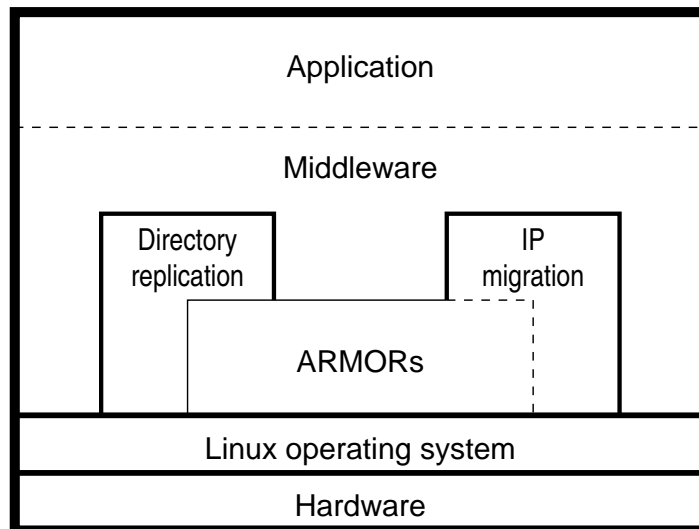


Figure 1.3 System hierarchy.

to the ARMOR architecture. The application and the directory replication tool are embedded ARMOR processes. An embedded ARMOR is an application to which core ARMOR elements (e.g., heartbeat replier) are added. The ARMOR architecture provide error detection and recovery for the application and the directory replication tool.¹

Contributions. The major contribution of this work is the design and implementation of the directory replication tool to maintain an updated copy of files on a remote node. Other contributions include (1) improvements to previous work [3] to create an IP address migration mechanism allowing a failed primary node to automatically become a new backup node, (2) integration of the application and the IP address migration tool into ARMORs,² and (3) thorough evaluation of performance and reliability of the baseline application and of the proposed solution.

Related work. Failover was investigated in previous work. Different mechanisms were proposed for IP address migration. They either require special hardware [4] or do not allow a failed backup node to become a new backup node after it is restarted [3], [5]. The interposition mechanism used by the directory replication tool to intercept function calls was used previously to detect intrusions (libsafe [6], [7]), audit application security [8], and profile libraries with no access to their source code [9]. ARMORs are presented in [1]. A more thorough description of ARMORs and examples of ARMOR configurations for different applications are given in [2].

Thesis outline. The remaining chapters of this thesis are organized as follows. Chapter 2 introduces the application and middleware for which the failover support was developed. The reliability infrastructure used in this work, ARMOR, and its configuration for the application failover support are presented in Chapter 3. The IP address migration mechanism used to maintain connectivity between clients and the application after a failover is explained in Chapter 4. Chapter 5 describes the directory replication tool which is used for the migration of the application database to the backup node. Experimental performance and reliability results of the failover mechanism are presented in Chapter 6. Finally, Chapter 7 states concluding remarks and discusses future work.

¹A more detailed discussion on different levels of application support offered by ARMORs can be found in [2].

²In collaboration with Keith Whisnant.

CHAPTER 2

TARGET SYSTEM

The objective of this work is to provide ARMOR-based (adaptative, reconfigurable, and mobile objects for reliability) failover support for legacy applications. This chapter presents details of the target system.

2.1 Messaging Middleware

The messaging middleware is used to connect distributed applications running across both wireline and wireless networks and allows data to be moved across different applications regardless of the network protocol they use. This middleware is composed of a system watchdog, multiple modules, and a database server. The system watchdog starts, monitors, and stops the modules. It also detects and restarts crashed modules. Figure 2.1 shows the system configuration, with the watchdog (`watchd`) and its child processes, the middleware modules, as well as the database server (`dbserver`). The modules provide system services (e.g., tracking, table services, etc.), data communication,

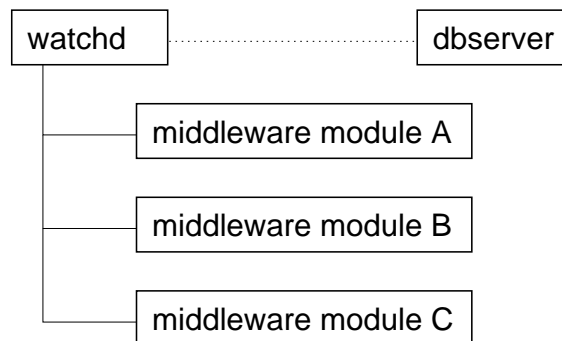


Figure 2.1 Middleware processes.

and other higher level services. The messaging middleware also provides an API (application programming interface) for the modules to communicate with the database server and use database services.

System watchdog (watchd). The system watchdog (`watchd`) first creates shared resources (e.g., shared memory) used by all modules and then starts the modules. The `watchd` process detects crashed modules by using the Unix `waitpid()` function call, which notifies the watchdog whether a module has crashed or has simply exited.

System services. System services may be implemented as separate modules or provided through the API library functions linked to object code of the modules. Example services provided by the messaging middleware include:

- Address management: Each object (module, database, etc.) is assigned a 16-byte address. This service provides a consistent handling of the objects.
- Configuration service: This service permits adding or removing modules.
- Database service: This service provides a database functionality.
- Memory management: This service provides global memory that can be accessed by multiple modules and/or the system watchdog.
- Memory table service: This service provides indexed lists of data.
- Tracking service: This service provides a way to follow the execution progress of a module.

Database server (dbserver). The database server is started as a normal child process of the system watchdog (`watchd`), not as a module. The modules access the database indirectly through the database server as illustrated in Figure 2.2. The modules use TCP connections to send requests to and receive the requested information from the database server. The database server stores the database on the hard drive.

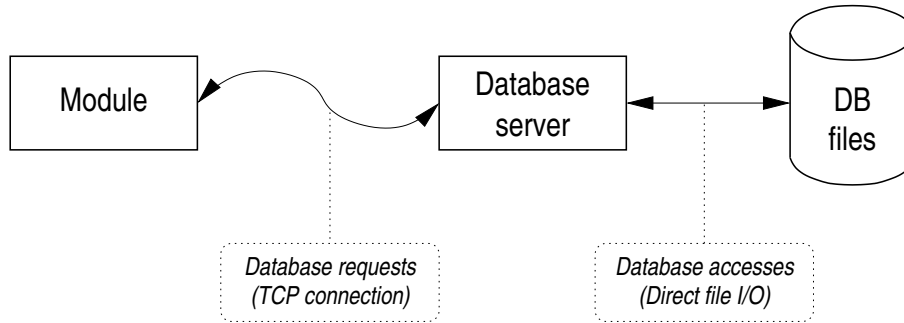


Figure 2.2 Database access.

Modules. Modules provide user-defined functionalities and services and are implemented as Unix processes started and monitored by the watchdog. An example of a user-specific module is the test module (`test_mod`) used in our experiments. The test module imposes a high workload on the database server and on the watchdog. This is essential in creating a high activation rate during fault injection (see Chapter 6 for experimental results).

There are also system modules required for key services to work. For example, the *tracking module* allows source modules to send text messages (e.g., about errors or exceptions) to destination modules that can then store or print those messages. The tracking module routes those messages from the source module to the destination module. Another example of a service module is the *memory table server*, which centralizes all memory table accesses.

Application. The target application used in this study constitutes the system supporting voice and data communications across wireless and wireline networks. The application runs on a Motorola MCP750 PowerPC CPU card within a CPX8216 cPCI chassis.

CHAPTER 3

ARMORS

To provide reliability to the application (described in Chapter 2), we use the ARMOR framework. ARMORs (adaptative, reconfigurable, and mobile objects for reliability) are multithreaded highly configurable processes that provide customizable levels of dependability to user applications [1], [2].

This chapter presents the ARMOR architecture. After introducing the different kinds of ARMORs and describing the error detection and recovery, we present the ARMOR configuration that is used for this work.

3.1 ARMOR Architecture

ARMOR-based infrastructure is designed to provide reliability to COTS (commercial off-the-shelf) distributed applications running on a heterogeneous network of COTS non-fault-tolerant hardware. ARMORs are self-checking processes that provide a wide range of reliability mechanisms (heartbeats, checkpointing, process monitoring, etc.) to applications. ARMORs allow run-time system reconfiguration in the case of failure. For example, in the event of a node failure, ARMORs originally hosted on the failed node can be migrated to another node. An ARMOR consists of an ARMOR microkernel, which manages a set of ARMOR elements. Elements are basic building blocks providing all ARMOR-supported functionalities, e.g., TCP connections to daemons on different nodes, interprocess communications to other ARMORs on the same node, and detection and recovery from application and ARMOR failures. There are four major types of ARMORs:

- **Fault tolerance manager (FTM):** The FTM is in charge of controlling the other ARMORs and recovering from failures.

- **Daemons:** On each node there is a daemon that provides communication between ARMORs residing on different nodes on the network. The daemon starts the other ARMORs on its node and serves as a gateway for communication between local and remote ARMORs. The daemon also monitors the local ARMORs by means of periodic heartbeats. In the case of an ARMOR failure, the daemon notifies the FTM, which initiates recovery.
- **Execution ARMORs:** An execution ARMOR is in charge of providing multiple detection and recovery techniques to an external application. Different levels of application support can be used depending on the ARMOR-awareness of the application and the reliability needs. For instance, ARMOR-aware applications can use progress indicators or text-segment signature checking (see [2] for more details).
- **Embedded ARMORs:** ARMORs can also be integrated with the application and form a single process. Such ARMORs are called embedded ARMORs. They allow for a better application support. For the purpose of this work, a version of the application that is integrated with an embedded ARMOR was created.¹

Figure 3.1 shows an example ARMORs configuration consisting of three nodes: Node 0 running FTM, Node 1 executing an application supervised by Execution ARMOR 1, and Node 2 executing an application with an embedded ARMOR.

3.2 Error Detection and Recovery

In ARMOR-based infrastructure error detection and recovery responsibilities are distributed hierarchically to different ARMORs. The FTM is responsible for the detection of failures of daemons and nodes and for their recovery. In turn, daemons are responsible for the detection of failures of execution ARMORs running on their nodes and for the recovery of those ARMORs. Finally, execution ARMORs and embedded ARMORs are responsible for the detection of failures of the application they monitor. This section describes the different failure scenarios that are handled.

¹The integration of embedded ARMORs to the application and the directory replication tool (see Chapter 5 for a presentation of the directory replication tool), and ARMOR configuration scripts for application failover have been written in collaboration with Keith Whisnant, one of the lead developers of the ARMOR architecture.

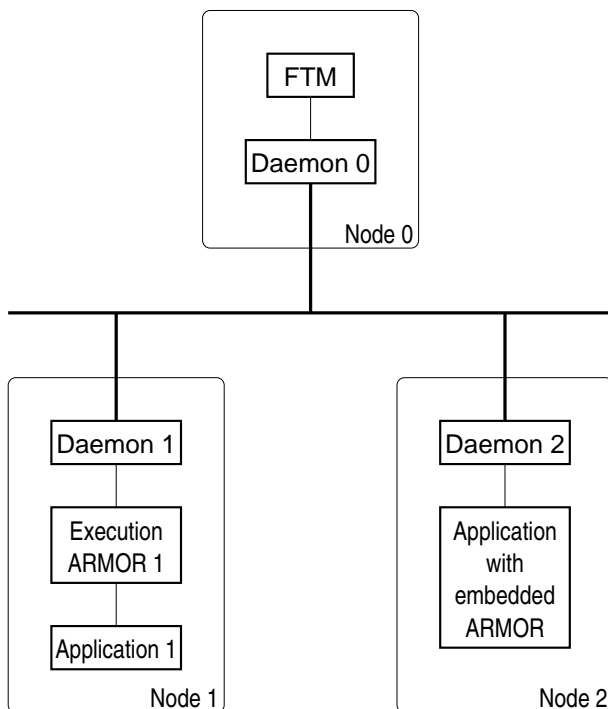
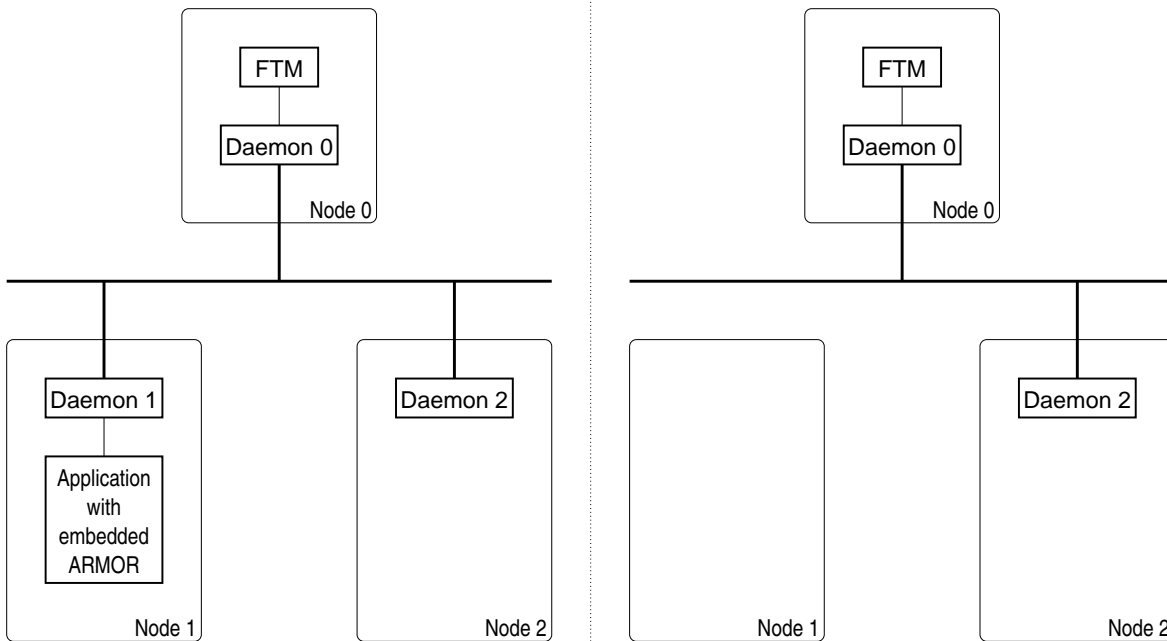


Figure 3.1 ARMORs configuration example.

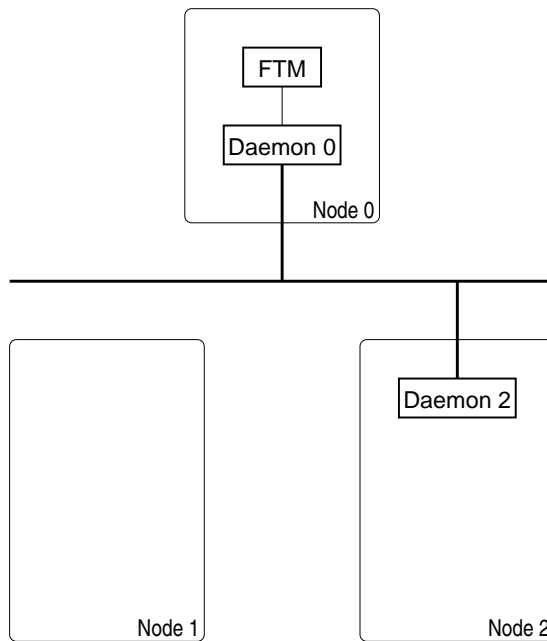
3.2.1 Node failure

In the case of a node failure, all communication is lost between the FTM and the failed node. A node failure is thus detected by the FTM if the daemon on that node does not respond to the heartbeats sent by the FTM. Note that since all communications between the FTM and the ARMORs on another node go through the daemon, a failure of that daemon is considered by the FTM as a node failure. When a node failure is detected, the FTM chooses a spare node to restart the ARMORs previously hosted by the failed node. A daemon must already be running on that spare node for the recovery to take place.

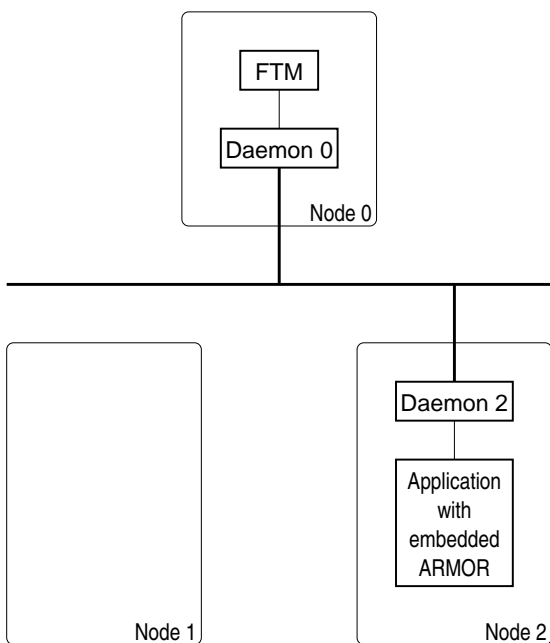
An example of node failure and recovery is shown in Figure 3.2. In the first step, before the failure, a daemon and an application with an embedded ARMOR are running on Node 1. A second daemon is running on a spare node, Node 2. Both daemons are communicating with the FTM. In the second step, Node 1 fails and the communication between the daemon on Node 1 and the FTM is lost. The FTM detects the node failure because it does not receive responses from Daemon 1 to the heartbeats sent. In the third step, the FTM recovers on Node 2 the application with the embedded ARMOR. Daemon 2 starts the application with an embedded ARMOR.



Step 1: Before the failure



Step 2: Node 1 failed



Step 3: Recovery on node 2

Figure 3.2 Node failure and recovery.

3.2.2 ARMOR failure

ARMOR failures are detected by the daemons through two techniques. In the first, the daemon calls `waitpid()`. This Unix function call returns status information about the child processes of the daemon (i.e., the other ARMORs running on the same node). Crashes and other process terminations are detected this way. In the second technique, the daemon sends heartbeats to the other ARMORs on its node to detect process hangs. After an ARMOR failure is detected, the FTM is notified and can recover the failed ARMOR either on the same node or on a spare node.

3.2.3 Application failure

Application failure can be detected by multiple techniques. The available techniques depend on the ARMOR-awareness of the application. Only process crashes can be detected for non-ARMOR-aware COTS applications. Process hangs and other application errors can be detected for applications more tightly integrated with ARMORs.

3.3 Configuration for Application Failover

The objective of this work is to provide ARMOR-based failover capability to the application (including its middleware). The hardware configuration consists of two identical PowerPC nodes, Node 1 and Node 2. The application initially executes on the primary node (Node 1). Node 2 is a backup node. Figure 3.3 shows the ARMOR configuration. On the primary node, Daemon 1 starts the application with an embedded ARMOR. Daemon 1 also starts the local intermediary process of the directory replication tool² with an embedded ARMOR. The IP address migration tool³ is an element of the daemon.⁴ On the backup node (Node 2), another daemon (Daemon 2) is running. In the case of a primary node failover, FTM instructs Daemon 2 to initiate the recovery of the embedded ARMORs on the backup node. Components of this configuration, including performance and reliability assesment, are described in the following chapters.

²The local intermediary process (LIP) of the directory replication tool is running on the primary node. The directory replication tool is described in detail in Chapter 5.

³The IP address migration tool is described in detail in Chapter 4.

⁴Another possible configuration would be to have an execution ARMOR overseeing standalone application and local intermediary process.

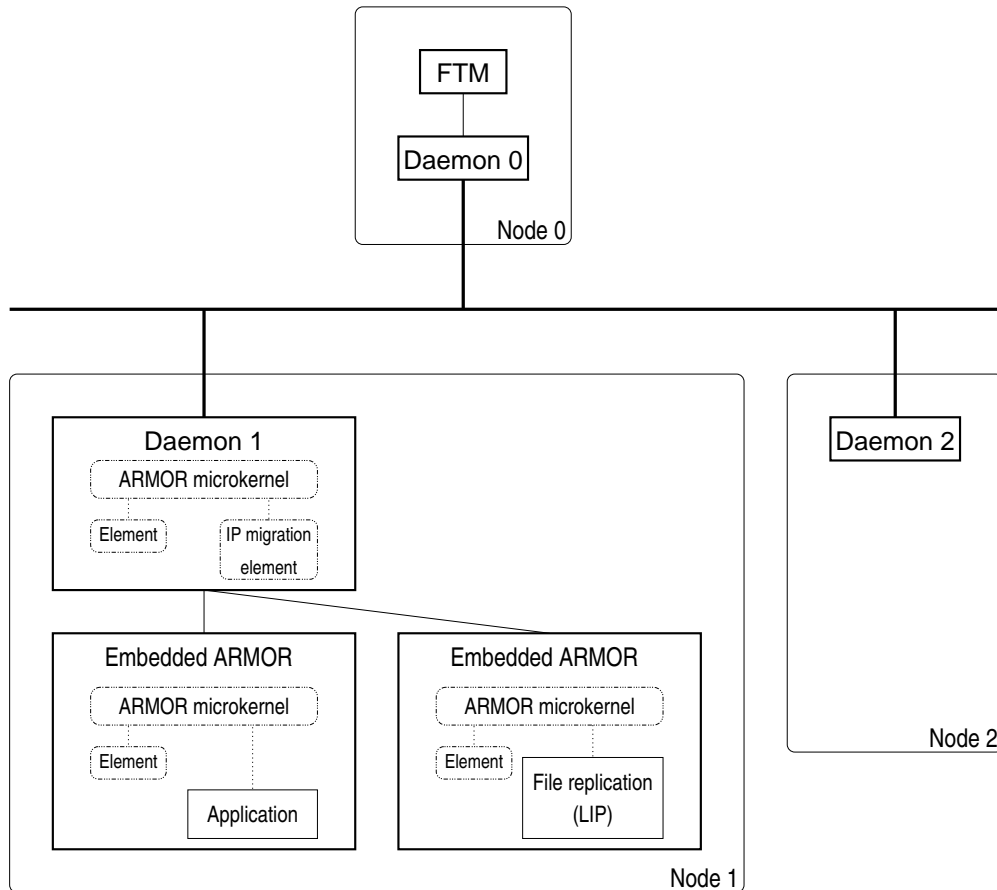


Figure 3.3 Configuration for application failover.

CHAPTER 4

IP ADDRESS MIGRATION

In client-server applications, a client opens a TCP (transmission control protocol) connection or sends a UDP (user datagram protocol) datagram to a server by using the server's IP (Internet protocol) address and TCP or UDP port number. The node (computer) that is initially (before any failure occurs) running the server application is called the *primary node*. The node (computer) that takes over in the case of primary node failure is called the *backup node*. The test bed is composed of two server nodes (the primary node and the backup node) and of one or more client nodes as shown in Figure 4.1.

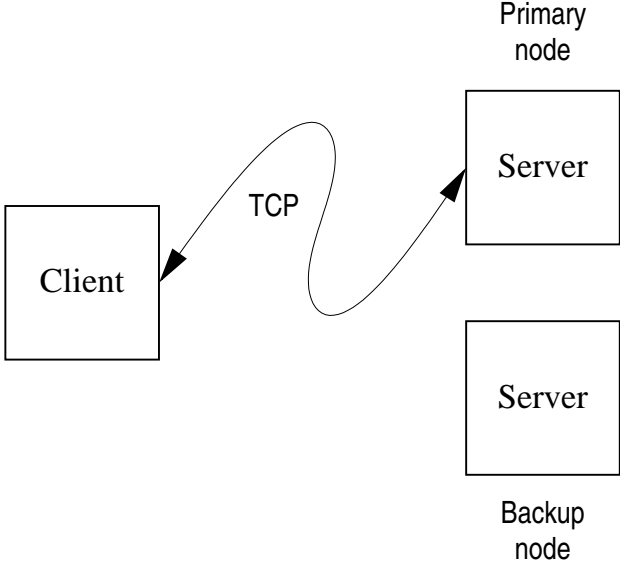


Figure 4.1 Client-server application with primary and backup nodes.

In order to provide a failover capability to the server, without making the client aware of that, we need the backup server to respond to the same IP address as the primary server. This goal

is achieved by having the backup server take over the IP address of the primary server by means of ARP (address resolution protocol) spoofing (see Section 4.2). As a result, all communications between the client and the server are redirected to the backup server. Note that the proposed mechanism does not recover TCP or other higher-level connections. It relies on the application to do so.

IP address migration is envisioned to handle the failure scenario. This scenario involves the following steps:

1. A client connects to the primary server.
2. If the primary server fails, the client detects the failure and tries to reconnect to the server using the same IP address as before.
3. The backup server takes over the IP address of the failed primary server.
4. The client reconnects to the backup server while still using the same IP address.

4.1 Concepts

4.1.1 Hardware and logical addresses

In the Internet layering model [10] each node has a corresponding hardware (Ethernet in our case) address and a logical (IP) address. TCP and UDP communications are based on logical addresses. Hosts and routers map IP addresses to hardware addresses by using the address resolution protocol (ARP) [11].

4.1.2 ARP IP-to-hardware mapping

When a node needs to map an IP address to a hardware address, it broadcasts a *request* message to all nodes on the network. The node with the requested IP address sends a *reply* message to the requesting node. To prevent constant broadcasting, each node keeps a cache of recently resolved IP-to-hardware address pairs. All exchanged messages contain the IP and hardware addresses of the sender. An IP-hardware pair is added to, or updated in, the cache of a node in the three following cases:

- a) When a node receives an ARP *reply* message from a host to which it has sent a *request* message, it adds the IP-hardware pair to its cache.
- b) If a node receives an ARP *request* message asking for the hardware address corresponding to its IP address, the IP-hardware address pair is updated or merged into the cache.
- c) If a node receives an ARP message from a sender whose IP address is already in the local cache, the IP-hardware address pair is updated into the cache.

In the example in Figure 4.2 there are three nodes. The hardware addresses for nodes 1, 2, and 3 are H1, H2, and H3, respectively, and their IP addresses are IP1, IP2, and IP3. At the beginning, all nodes' caches are empty.

Step 1. Node 1 broadcasts a *request* message asking for the hardware address corresponding to IP3.

This message contains Node 1's IP address (IP1) and hardware address (H1).

Step 2. Node 2 ignores this message. Node 3 adds the IP-hardware pair for Node 1 (IP1–H1) to its cache because the message was a *request* for Node 3's hardware address (Case b). Node 3 also sends a *reply* message to Node 1 giving its hardware address (H3).

Step 3. Node 1 adds the IP-hardware pair for Node 3 (IP3–H3) to its cache because it has previously sent a *request* message asking for the hardware address associated with IP3 (Case a).

4.2 ARP Spoofing

In order for the IP address migration to take place, the backup node must be able to accept a connection and send messages using the primary node's IP address as its own. Also the nodes on the local area network (LAN), in this case the local Ethernet network, must send to the backup node all Ethernet frames destined for the primary node. To achieve the latter goal we alter the way the IP-hardware address matching is done by changing the data in each node's ARP cache.¹ This is done by broadcasting a *gratuitous* ARP reply message (i.e., reply messages to nonexistent request

¹Another way would be to set the network interface card (NIC) Ethernet address of the backup node to the address of the failed node [4]. However, this would require special reconfigurable NICs.

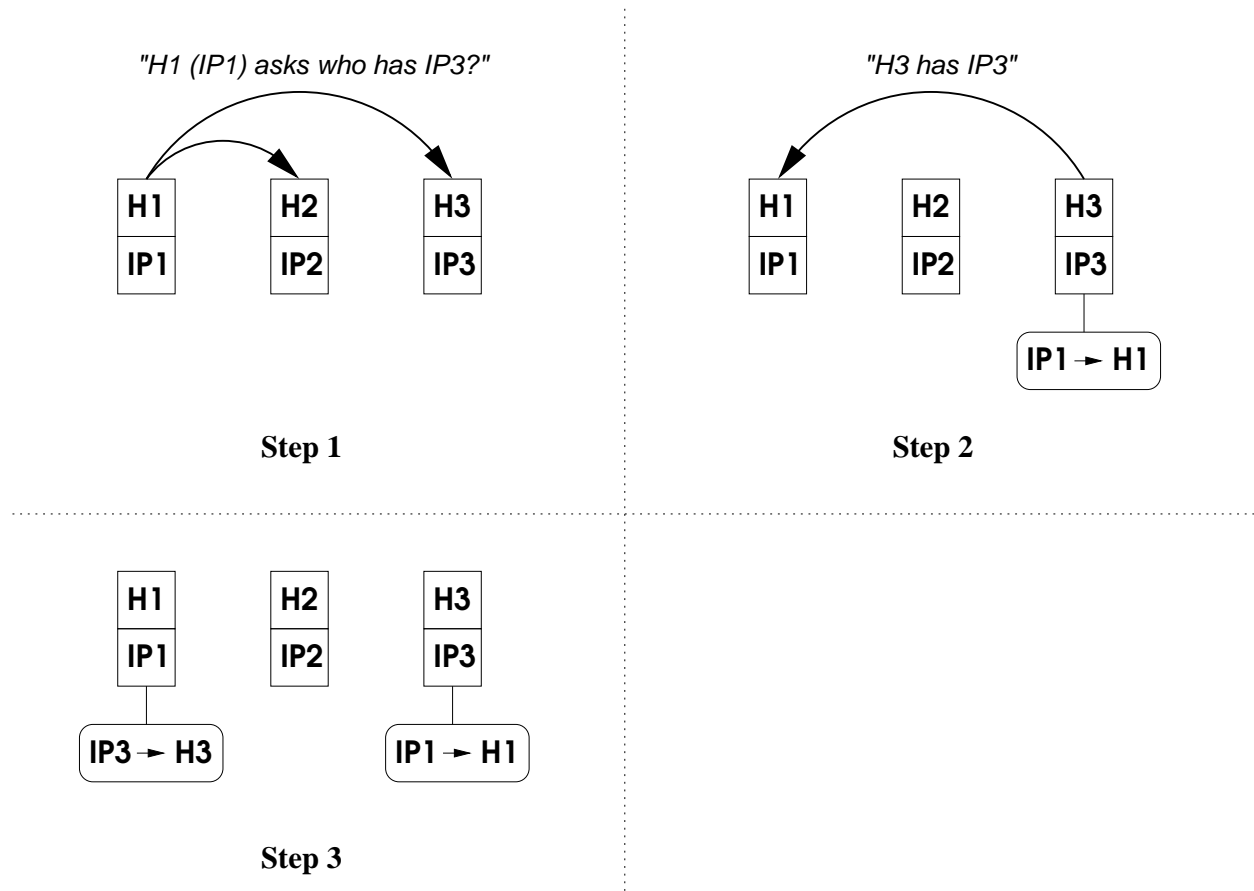


Figure 4.2 Address resolution protocol (ARP).

messages) which associates the IP address of the (primary) node with the hardware (Ethernet) address of the backup node. All nodes that have been communicating with the server will therefore update their ARP cache and send all frames destined for the server to the backup node. Also, the backup node responds to ARP request for server IP address. This method is called ARP spoofing [3], [5].

Figure 4.3 gives an example of how ARP spoofing works:

Step 1. At the beginning, Node 1 has in its cache the IP-hardware pair for Node 3 (IP3–H3). All messages directed to IP3 are sent to the hardware address H3 (i.e., to Node 3).

Step 2. Node 2 initiates the procedure to take over IP3. It sends a gratuitous ARP reply message stating that its own hardware address, H2, corresponds to IP3 (the IP address of Node 3).

Step 3. Node 1 receives the gratuitous ARP message. Because Node 1's cache contains an IP-hardware pair for IP3 (Case c in Section 4.1.2), Node 1 updates the pair with the new hardware address, H2. As a result, when Node 1 sends a packet to IP3, the packet is sent to the hardware address H2, i.e., to Node 2.

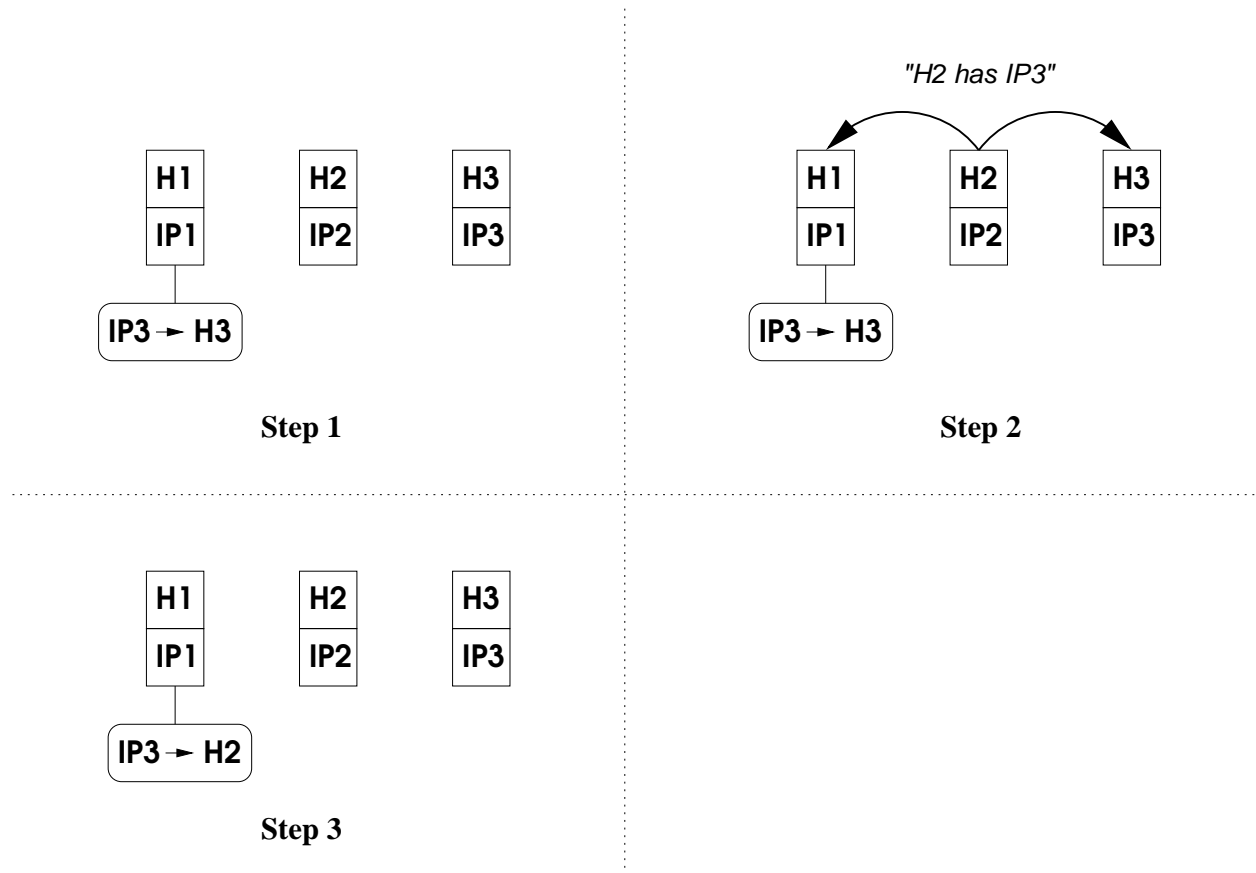


Figure 4.3 ARP spoofing.

Since the primary node address resolution mechanism (part of the kernel in Linux) might still be working even though the server application is down, gratuitous ARP messages are sent at regular intervals (e.g., 1 s). This way a mishap would not last long, even if the primary node has won a race (i.e., responded last) against the backup node to reply to an ARP request for its IP address.

4.2.1 IP aliasing

In order to preserve a constant channel of communication to both the primary and backup nodes, without disconnections during the failover. Each node needs to have its own constant IP address

in addition to the IP address used by the client to connect to the server. This means that the primary node has two IP addresses, and the backup node will also have two IP addresses after the IP takeover.

We use IP aliasing to assign two IP addresses to a node with only one physical network interface card (NIC) [3], [5], [12], [13]. IP aliasing allows one physical NIC to respond to two or more different IP addresses. This is done by creating a virtual NIC (e.g., `eth0:1`) associated with the physical NIC (e.g., `eth0`). The virtual NIC can be configured (with `ifconfig`) to respond to a different IP address than the physical NIC.

Two new terms are introduced for further discussion:

- **Server IP address:** This is the IP address used by the client to connect to the server. The *server IP address* is initially used by the primary node and, after the failover, it is taken by the backup node. This IP address is not “physically” attached to a particular node, but can move from one node to another depending where the server application is running.
- **Auxiliary IP address:** This is the IP address used solely by one node. The *auxiliary IP address* is “physically” attached to the node and it is used for communication between the primary and backup nodes and for maintenance.

When a node is running the server application and, thus, uses both the server IP address and an auxiliary IP address, it is important to pay attention to which IP address is the default IP address, i.e., the local IP address most applications use for their communications if they do not specify another local address when beginning new connection. Most applications, both servers and clients, do not specify any local IP address when calling `bind()` and instead let the kernel choose which local IP address to use. In such a situation, the kernel chooses the IP address attached to the outgoing interface. Unless the routing table specifies otherwise, the outgoing interface is the main interface (e.g., `eth0` on Linux), not the virtual interface (e.g., `eth:1` on Linux), and the default IP address is the IP address attached to the main interface. In order for the server application to work properly, the IP address that is used by the clients to communicate with the server application (the *server IP address*) is therefore attached to the main interface, and the auxiliary IP address is attached to the virtual interface.²

²Another solution to this default IP address problem would have been to add new entries in the routing table, choosing the virtual interface as the outgoing interface for communications with the clients of the server application.

In the example described in Figure 4.4, Node 1 has a single physical NIC, `eth0`, with hardware address HW1. IP address IP1 is assigned to this interface. This is the default IP address. Node 1 has also a virtual interface, `eth0:1`. The IP address IP10 is an IP alias associated with this virtual interface.

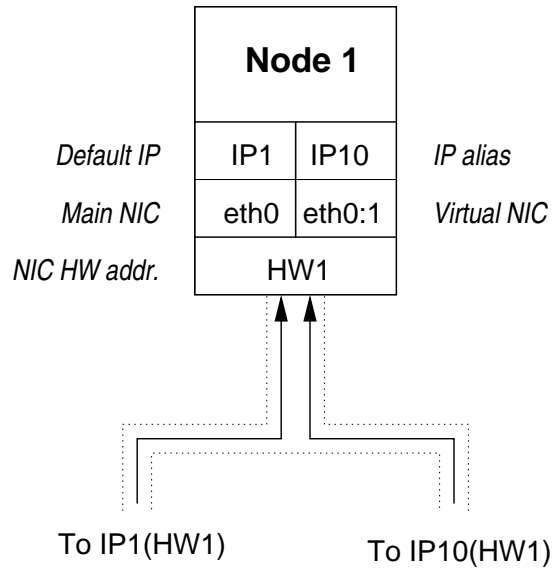


Figure 4.4 IP aliasing.

4.3 IP Failover Setup

The testbed is made of two server nodes (primary and backup) and an outside network (in our experiments a PC running a workload emulator for the application).

4.3.1 Primary node

The primary node is the node on which the server application runs at first. As shown in Figure 4.5(a), this node uses address IP3, the *server IP address*, as its main IP address, and IP1, the *internode IP address*, as its IP alias.

4.3.2 Backup node

The backup node is the node that is on standby to take over in case the primary node fails. This node uses IP2, the *auxiliary IP address*, as its main IP address. While the backup node is on

standby, it has only one IP address, as illustrated in Figure 4.5(b).

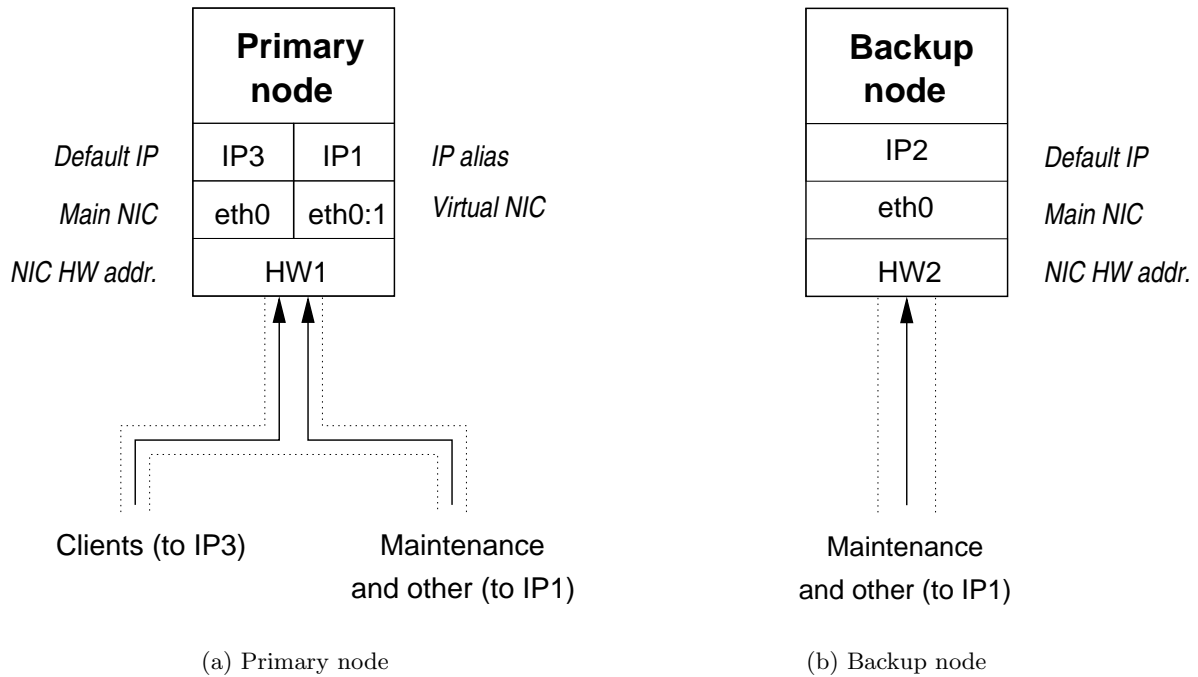


Figure 4.5 Primary and backup nodes.

4.3.3 Failover

When the failure of the primary node is detected (see Chapter 3), the backup node executes the following steps:

1. Set its new main IP to IP3.
2. Set IP2 as an IP alias (on `eth0:1`).
3. Start an ARP spoofing daemon that sends gratuitous ARP messages every second.
4. Start a daemon that waits for a signal of a reboot of the primary node to terminate the ARP spoofing daemon (see Section 4.3.4).
5. Start the directory replication tool (see Chapter 5).
6. Start the server application.

After this procedure is completed, all communications addressed to IP3 are directed to the backup node. Clients can connect to the server application by using the same server IP address as before (IP3). From the client perspective, it looks like the server was just restarted on the same node.

4.3.4 Switching roles

After the failover has been completed, the primary node must be able to take the role of the backup node. For the purpose of this discussion we will call the primary node that failed *Node 1* and the node that took over after the primary node failure *Node 2*.

With Node 2 still spoofing ARP messages, any attempt by Node 1 to take over IP3 will result in a race condition that neither of them is assured to win. In order for Node 1 to be able to successfully play the role of a backup node, Node 2 must stop ARP spoofing. When Node 1 reboots, it is configured as a backup node, with only one IP address: IP1. Node 1 also signals its reboot to Node 2 by sending a message to a “wait” daemon running on Node 2. Upon receiving this message, the wait daemon shuts down the ARP spoofing daemon. At this point the system is back in the same state as before the failover process began but with reversed roles. Node 2 is the primary node and Node 1 is the backup node. This scenario is illustrated in Figure 4.6.

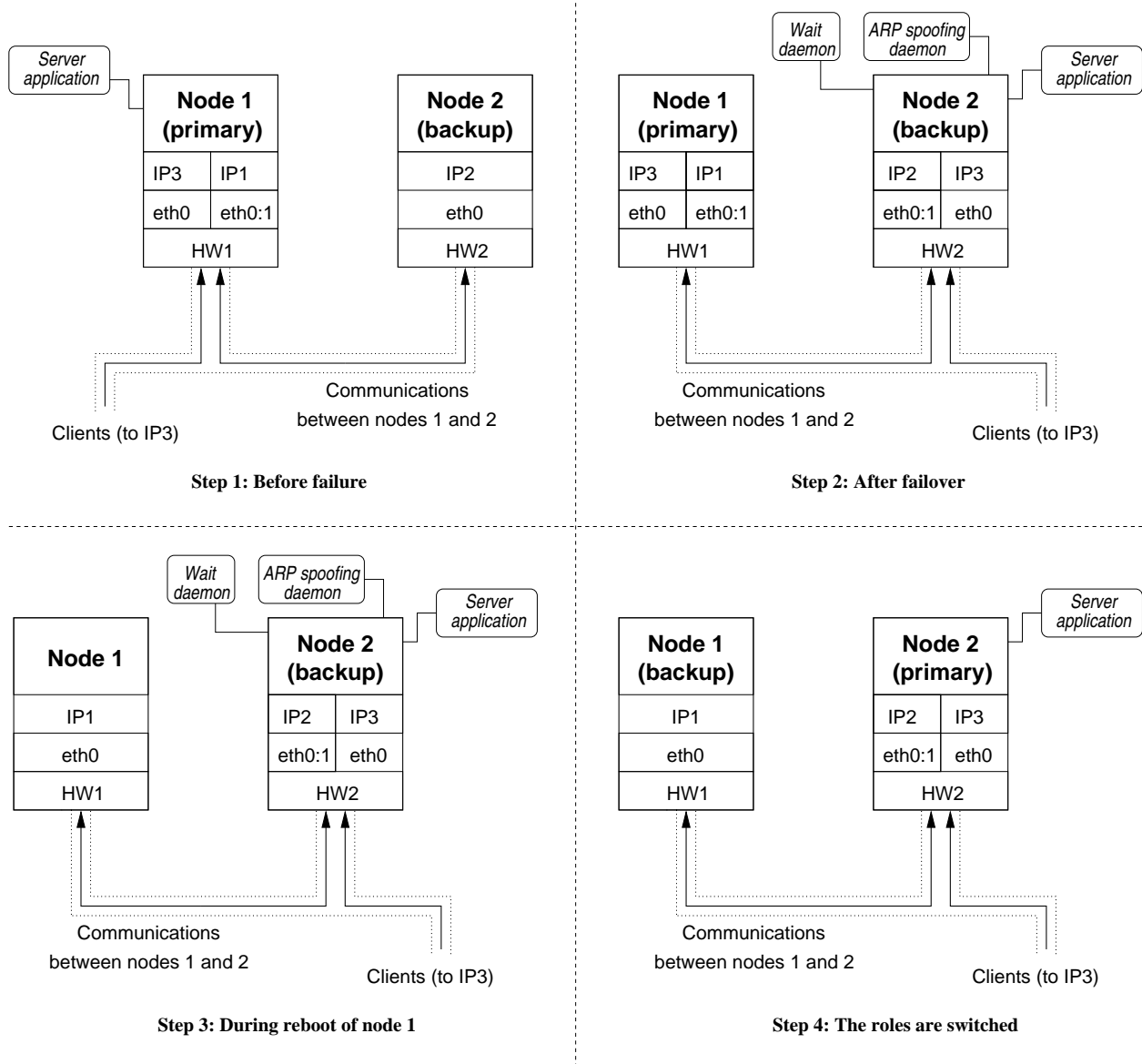


Figure 4.6 Switching roles between primary and backup nodes.

CHAPTER 5

DATABASE MIGRATION (DIRECTORY REPLICATION TOOL)

A typical server application, upon receiving a client request, produces and communicates a response back to the client. In the application scenario considered in this study, computations performed by the server require manipulation of a database stored on the disk. In the previous chapter, we described the server failover transparent to the client (IP address migration). In this chapter we present how, in the case of failover, the backup server obtains access to the data stored by the primary server.

In the configuration shown in Figure 5.1, the primary and backup nodes have their private hard drives, i.e., they cannot access each other's hard drive. This chapter introduces a software tool that replicates data on the backup node. (See Appendix A for the user manual on how to use the directory replication tool, Appendix B for pseudocode of the algorithms constituting the tool, and Appendix C for the source code.)

5.1 Motivations and Requirements

5.1.1 Motivations

For the proper work of the server application, a copy of the database must exist on the backup node. To achieve this goal the directories containing the database files on the local disk must be replicated on a remote computer. We call the files in those directories (and by extension, the

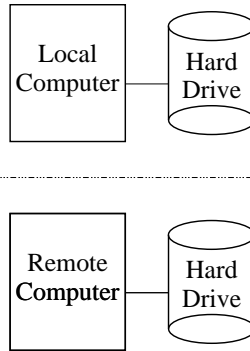


Figure 5.1 System configuration.

directories themselves) *watched* files (or directories). The copies of those files (or directories) on the remote computer are called *backup* files. The tool developed to support the file/directory replications is called the *directory replication tool* and is composed of the following:

- A **replication library** that intercepts I/O related system calls made by the **watched application** on the local system,
- A **local intermediary process** (LIP) with which the *library* communicates,
- A **remote server** (RS) which is in charge of writing files to the remote hard drive.

In the example given in Figure 5.2, two processes (Process 1 and Process 2) execute on the local node. When any of the two processes calls the `write()` function, the call is intercepted and the wrapper function for `write()` implemented in the replication library is executed. This wrapper function calls the `write()` function from `glibc` [14], the standard C library for Linux. The wrapper function also informs the local intermediary process about the `write()` function call being executed. The local intermediary process forwards the information to the remote server on the remote node. The remote server executes the same call as the one performed by the local node to the `write()` function from `glibc`. As a result, the same file modifications are done on both nodes.

5.1.2 Requirements

Key requirements for the directory replication tool include: (1) to maintain the integrity of the replicated files, (2) to support the restart of either the local intermediary process or the remote server in case of failure.

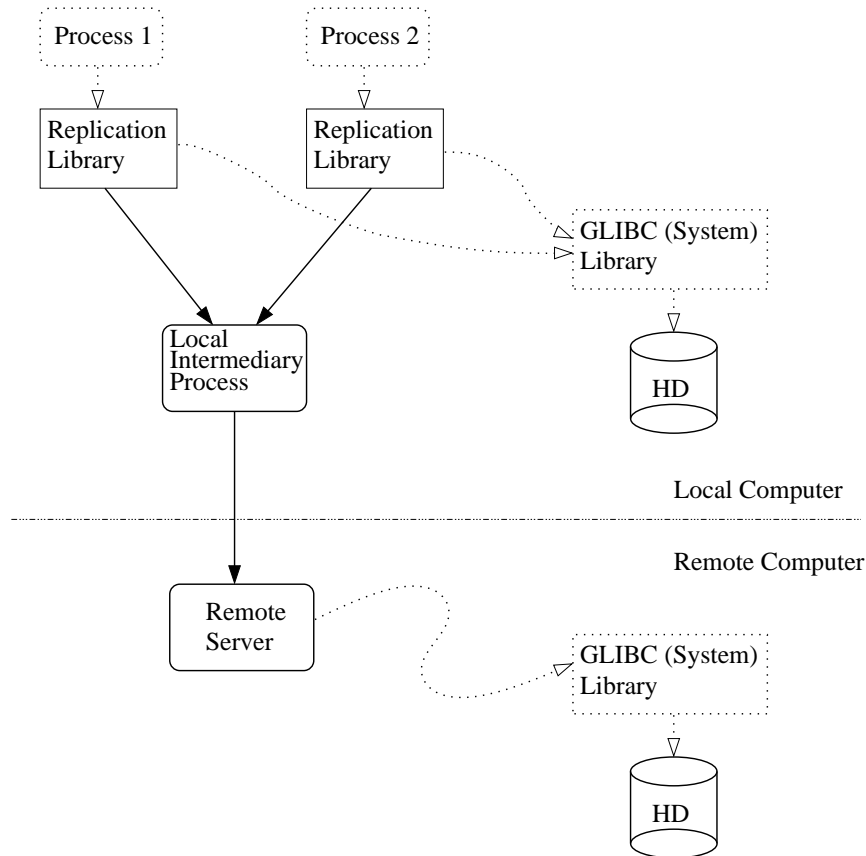


Figure 5.2 Architecture of the directory replication tool.

5.2 Replication Library

The wrapper functions implemented in the replication library intercept I/O-related calls to `glibc`. They then proceed to the actual function call and transmit all arguments, data, and return value to the local intermediary process. In the current version, `write()` and `open()`¹ are the two function calls that are intercepted.

5.2.1 Interposition

The mechanism used to catch function calls to `glibc` is called interposition. It consists of replacing runtime calls to specified functions from the standard C library by wrapper functions in order to gain information or take over those function calls (see Figure 5.3). Interposition is a well known concept used, for example, for profiling of graphic libraries without access to source code [9], application security audit [8], or intrusion detection (libsafe [6], [7]).

¹We also intercept `_open()`, which is another name for `open()`.

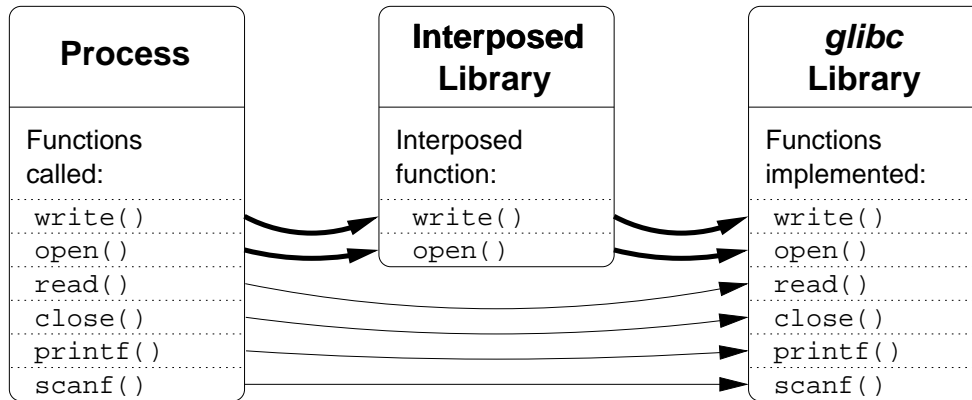


Figure 5.3 Function interposition.

The replication library is a shared library that contains functions having the same names as the functions we want to intercept (e.g., `write()`). When a process is started on a Linux system, a loader (`/lib/ld.so`) loads the shared library used by the process (e.g., `/lib/libc.so`, which is the location of `glibc`). There is a provision in the loader for overriding certain functions from shared libraries. By specifying to the loader that the replication library is to be *preloaded*, the replication library can be loaded before any other shared library, and the functions the library contains get precedence over the ones with the same name contained in the other shared libraries. For instance, if a preloaded library provides a function named `write()`, and if another, not explicitly preloaded, library (e.g., `glibc`) also provides a function named `write()`, the `write()` function provided by the preloaded library is executed when a process calls the `write()` function. It is still, however, possible for the `write()` function of the preloaded library to call the `write()` function provided by the nonpreloaded `glibc` library by using the `dlsym()` function. Preloading a library does not require any modifications to the object code of the application. It does however require the application to be dynamically linked (see Section 5.6). Preloading a library can be done by adding the name of the replication library to the configuration file `/etc/ld.so.preload`, or to the environment variable `LD_PRELOAD`. For set user ID (SUID) programs,² the loader does not take into account the `LD_PRELOAD` environment variable,³ and the name of the replication library has to be added

²SUID programs execute with the user ID (UID) of the owner of the executable file, not the UID of their parent process.

³Taking the `LD_PRELOAD` environment variable into account for SUID programs would be a security breach. A normal user could create a preloaded library with a wrapper function overriding a common function (e.g., `printf()`), and execute an SUID program with UID set to superuser (e.g., `/bin/passwd`). This would result in its wrapper function being executed with superuser privileges.

to the `/etc/ld.so.preload` file.⁴ In this work, a set of wrapper functions is implemented for the function calls we want to intercept: `write()` and `open()`. By intercepting I/O and file system related functions, we know what *write* accesses are done to the local files and so the same sequence of operations can be replicated on the remote node.

5.2.2 Statelessness

To detect which files are being written to, the replication library gets the name of the file being accessed by invoking `readlink()` on `/proc/self/fd/<file descriptor for the file>`. The `/proc` filesystem stores a symbolic link to the corresponding file for each open file descriptor. By using `readlink()` on this link, we can get the name of the corresponding file. The `readlink()` function call also calls the `glibc` function `fstat()`, which returns the attributes of a file. If the file is not a regular file⁵ in one of the watched directories, then the local intermediary process is not contacted and `libc`'s `write()` is called directly.

The information sent to the local intermediary process (see Section 5.3.1) by the replication library for each `write()` is sufficient to proceed to the same operation on the related file on the remote node. No state data needs to be stored either by the local intermediary process or by the remote server. Therefore, both the local intermediary process and the remote server can be started at any time and restarted at will.

5.2.3 File integrity

To maintain the integrity of the version of files on the remote node, strict ordering of all `write()` calls must be imposed. All *write* operations on files on the remote node must happen in the same order as on the local node. If a Process A is calling a system function to write-access a file, no other Process B can call any write-access system function before Process A's system function call has returned. With such a strict ordering, we are guaranteed that the files are identical on both nodes. The only discrepancy that might happen is that the last *write* operations may not have been executed before the crash of the local node. But even in such a scenario, the state of the files

⁴This file has write permission restricted to the superuser. Therefore a normal user cannot add a library name to this file and create a security breach.

⁵I.e., not a directory, a character or block special (device) file, a pipe special file, a socket, etc.

on the remote node is guaranteed to be the same as it was at some point in the past on the local node.

Strict ordering is achieved by having the wrapper function send a *request* message to the local intermediary process and wait for a response before proceeding to the actual `glibc write()` function call. The local intermediary process does not respond to a request before the previous operation has been completed (i.e., all arguments, data, and return value for the previous `write()` have been received and acknowledged). This works as a semaphore allowing only one `write()` at a time.

The sequence of events as illustrated in Figure 5.4 is as follows:

1. A call to `write(fd, ...)` is intercepted, where `fd` is the file descriptor for the file being written.
2. The function `fstat(fd)` is called. If the file is not a regular file, no communication is done with the local intermediary process, the actual `glibc write()` is directly executed, and the wrapper function returns.
3. The function `readlink("/proc/self/fd/<fd>", ...)` is called to resolve the actual file name. The returned file name is then checked to see whether it is a file to be watched or not. If the file is not to be watched, the actual `glibc write()` is directly executed, and the wrapper function returns.
4. A *request* message is sent to the local intermediary process.
5. If the local intermediary process is not dealing with any other `write()`, it responds to the request message. Otherwise, it waits until it has finished dealing with the other `write()` before responding.
6. If a response has been received, or if an error has occurred (e.g., the connection is broken, timeout, etc.), it proceeds to the actual `glibc write()` call. In the latter case, it subsequently returns.
7. Information about the `write()` call (arguments, data, and return value of the `glibc` call) is sent to the local intermediary process.
8. The local intermediary process can now deal with a new `write()` call.

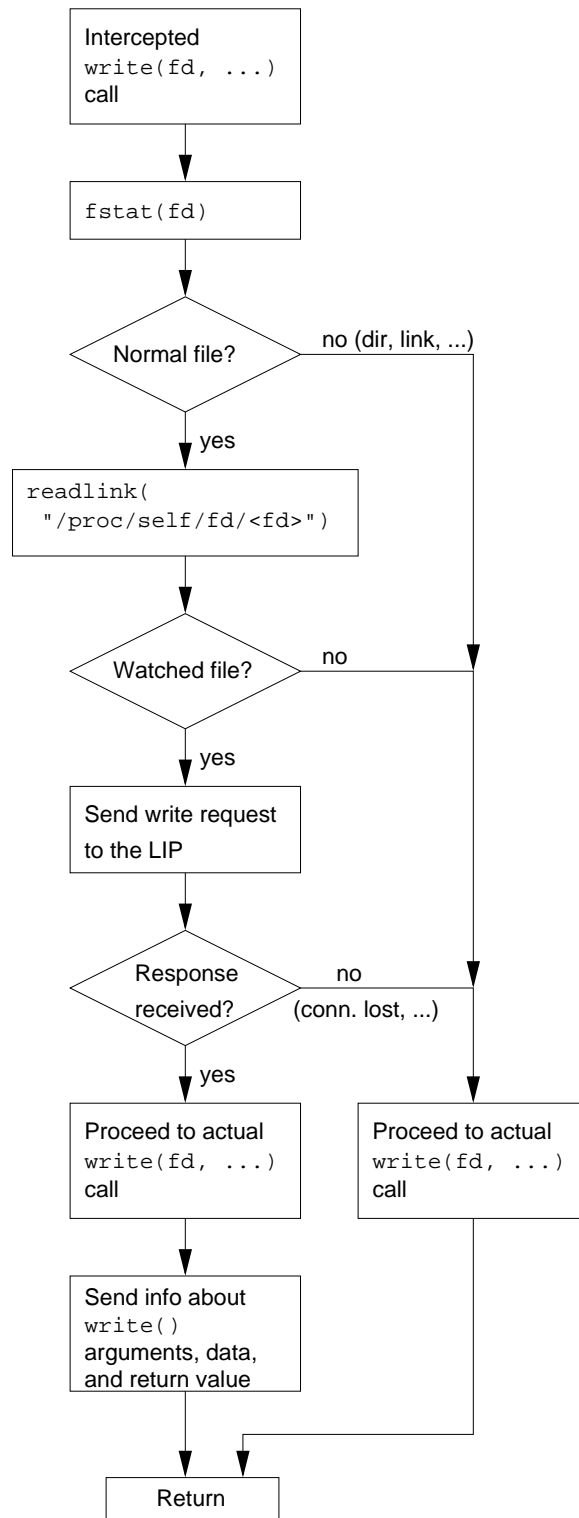


Figure 5.4 Wrapper function for `write()`.

5.2.4 Error recovery

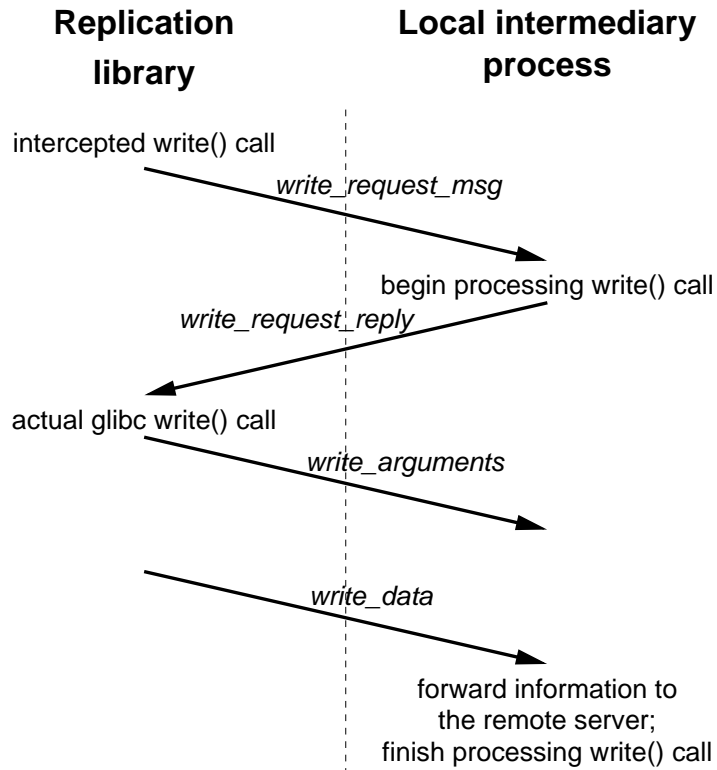
The wrapper functions, as well as the `glibc` functions, are part of the process using these functions. This means that any segmentation fault (or illegal instruction, etc.) that happens in these functions results in a process crash, and the crashed process must be restarted. When a crash occurs inside a wrapper or `glibc` function, two cases must be distinguished, depending on whether the crash occurs before or after the actual `glibc write()` call. If the crash happens before the call to `glibc write()`, then none of the files is modified, and consistency is maintained. If the crash happens after a call to `glibc write()` on the local node, but before having sent all the information about the `write()` to the local intermediary process, the local file is modified whereas the remote file is not. Figure 5.6 shows how such a crash can be detected by the local intermediary process. Since the replication library has sent a *write_request* message⁶ to the local intermediary process, but has not sent the information about the file, and the connection between the replication library and the local intermediary process has been broken, the crash is detected by the local intermediary process. After detecting such a crash, the local intermediary process retransmits the whole modified file to the remote server, thereby maintaining consistency across the nodes.

5.3 Local Intermediary Process

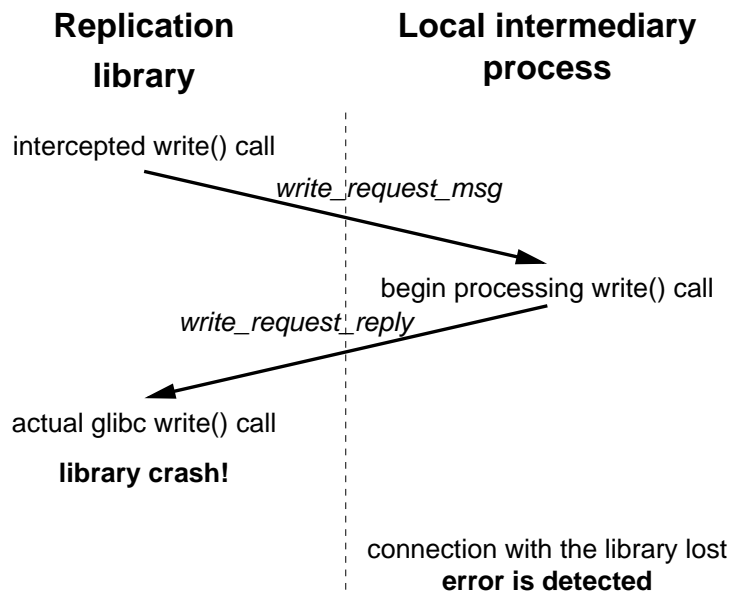
The role of the local intermediary process is to keep the replication library simple and to reduce the overhead due to the strict ordering of write requests (see Section 5.2.3). Any crash inside a wrapper function provided by the replication library results in a crash of the process using the library. Any crash of the local intermediary process can be transparently recovered, as is explained in Section 5.3.3. Therefore, from a reliability standpoint, it is beneficial to have a replication library as simple and robust as possible even at the cost of a more complex local intermediary process. Another reason for having a local intermediary process is that the request-and-response procedure required for strict ordering of write requests is faster when using intranode communications than internode TCP connections.

When the remote server starts, the local intermediary process provides a copy of all the files in the watched directories to the remote server (see Section 5.3.2.3). The local intermediary process

⁶The messages exchanged by the replication library and the local intermediary process are explained in detail in Section 5.3.1.



Case with no error



Crash after actual glibc write() call

Figure 5.6 Detection of library crash inside a write() wrapper function.

also receives all connections from the replication library. The local intermediary process subsequently orders and forwards the received information about function calls to the remote server (see Section 5.3.2.1).

5.3.1 Communication with the replication library

The local intermediary process listens to a known TCP port for incoming connections from the replication libraries. We further detail the procedure in the case of an intercepted `write()` call.

5.3.1.1 Receiving information about function calls

The first message sent by the wrapper function (provided by the replication library) to the local intermediary process after a function call has been intercepted contains the name of the intercepted function, `write()`, the PID (process ID) of the calling process, the filename and working directory, and a request ID. If no other `write()` call is being processed, the local intermediary process responds by sending a *write_request_reply* message. Otherwise, the local intermediary process waits until it finishes processing previous `write()` calls before responding to the new incoming request.

After receiving the *write_request_reply* message, the wrapper function sends a new message with the arguments of the intercepted `write()` call and its return value, indicating how many bytes have actually been written to the disk. Then, the wrapper sends the actual data that has been written. This sequence is illustrated in Figure 5.7.

After the local intermediary process receives the data, two scenarios are possible: (1) the remote server has not started yet—in this case, nothing happens—or (2) the remote server is started and has sent the *wakeup call* (see Section 5.3.2.2). In this case, the local intermediary process proceeds with forwarding the information about the `write()` call to the remote server as described in Section 5.3.2.1.

5.3.1.2 Blocking during MD5 checksum computation

When the remote server starts and sends the *wakeup call*, the local intermediary process starts transferring the files in the watched directories to the remote server. (This procedure is described in detail in Section 5.3.2.3.)

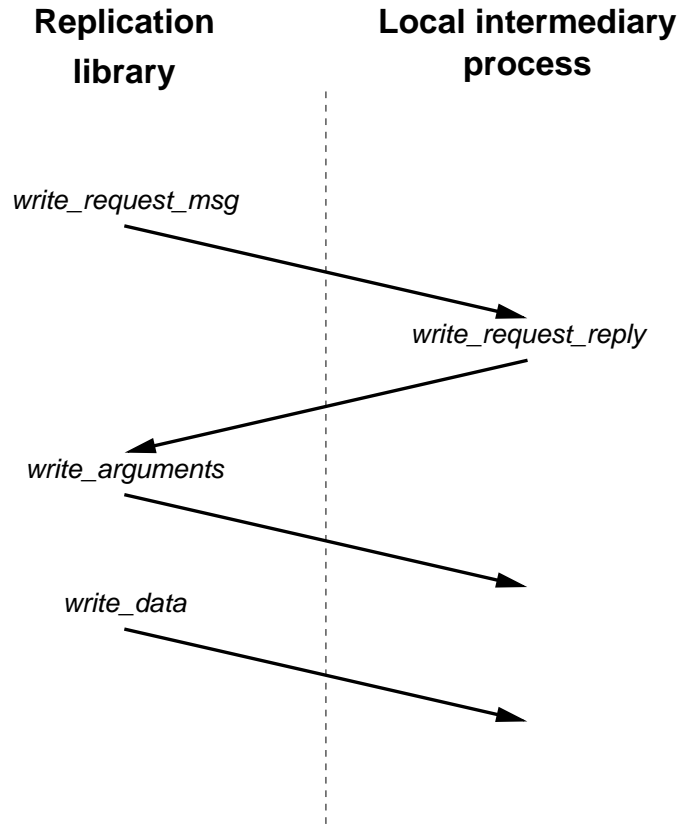


Figure 5.7 Sequence of events for a `write()` call.

During this procedure, in order to prevent any inconsistency between the files on the local and the remote nodes, the local intermediary process does not send any `write_request_reply` message, and all `glibc write()` calls are delayed. The performance impact is discussed in Chapter 6.

5.3.2 Communication with the remote server

The role of the local intermediary process is to be an intermediary between the replication library and the remote server. After receiving information about an intercepted `glibc` function call, the local intermediary process forwards this information to the remote server.

5.3.2.1 Forwarding function call information

The communication between the local intermediary process and the remote server is done through TCP connections. After the local intermediary process has received information about intercepted function call from the replication library, the local intermediary process connects to a known TCP

port on the remote node. The first message sent to the remote server contains information about the function call, arguments, return value, and the working directory. All file names are resolved to their canonical versions⁷ by the wrapper functions provided by the replication library. The local intermediary process then waits for an acknowledgment from the remote server. After the arrival of the acknowledgment, and in the case of a `write()` call, the local intermediary process sends the data being written. The remote server sends a second acknowledgment. The local intermediary process can now process new `write()` requests from the replication library. The strict ordering of `write()` calls is thus maintained in the communication with the remote server. This is illustrated in Figure 5.8.

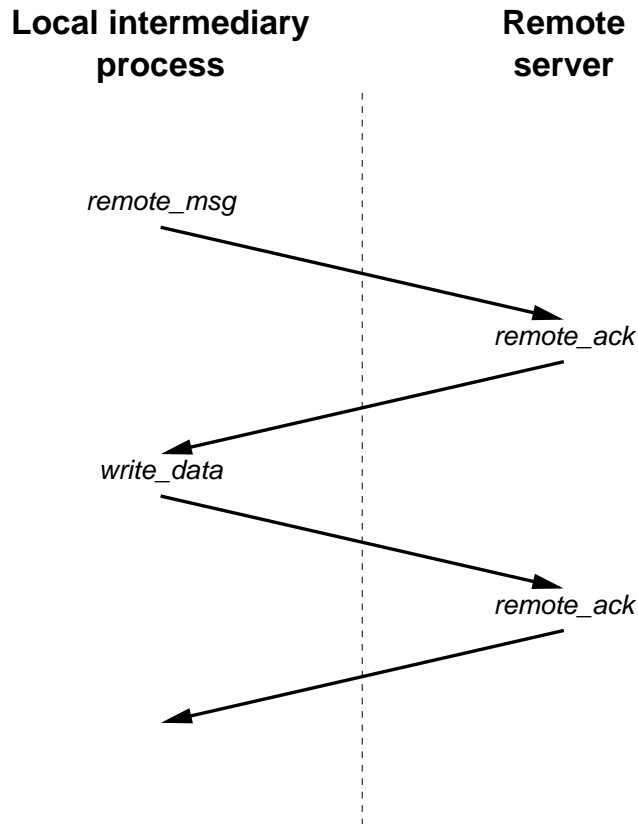


Figure 5.8 Sequence of events for the transmission of a `write()` call between the local intermediary process and the remote server.

⁷A canonical file name is the absolute name of the file, with no symbolic links, no links to the same directory (`.`) or parent directory (`..`) and no repeated path separators (`/`).

5.3.2.2 Wakeup call

The local intermediary process listens at a known TCP port (different than the one used to communicate with the processes using the replication library) for a wakeup call sent by the remote server upon startup (see Section 5.4). The reception of this message triggers the forwarding of information about intercepted `glibc` function calls to the remote server. There is no acknowledgment *per se* sent to the remote server, but since the local intermediary process starts the transfer of files just after the reception of the wakeup call, the remote server can know whether the wakeup call has been received or not.

5.3.2.3 File transfer

Once the *wakeup call* has been received by the local intermediary process, it can proceed to transfer the files in the watched directories to the remote server. The local intermediary process first creates a list of these files and computes their MD5 sums⁸ [15]. Then, for each of the files the local intermediary process sends an *advertise* message to the remote server which contains the file's name, length, mode, and MD5 sum. The remote server sends back a *request* message stating whether it wants the file to be transferred or skipped. The remote server asks to skip a file if it already has the same file (same location, name, length, and MD5 checksum). If the remote server requested the file to be transferred, the local intermediary process sends the file through a reliable protocol developed for the directory replication tool. After the file has been successfully transferred, or if the remote server requested the file to be skipped, the local intermediary process advertises the next file. When there are no more files to advertise, the local intermediary process sends a *no_more_files* message to the remote server. This sequence of events is illustrated in Figure 5.9.

5.3.3 Error recovery

In the case when the local intermediary process crashes, the local intermediary process is restarted but some `write()` calls might have been lost during the interval the local intermediary process was not active. To remedy such a situation, the local intermediary process initiates a file transfer to the

⁸MD5 stands for message digest algorithm version 5. This algorithm is used to produce a 128-bit checksum of files. We use the MD5 implementation by L. Peter Deutsch of Alladin Enterprises.

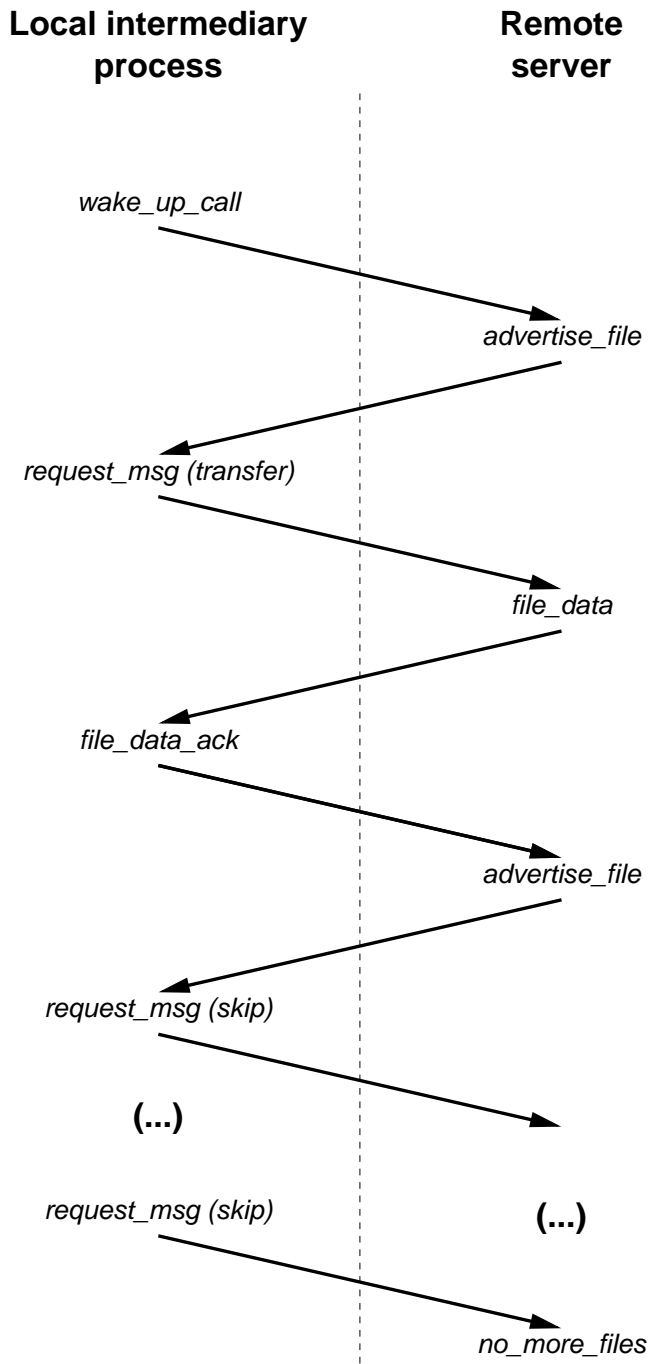


Figure 5.9 Sequence of events for the transfer of files between the local intermediary process and the remote server.

remote server as described in Section 5.3.2.3.⁹ Upon startup, the local intermediary process sends upon startup a *lip_start* message to the remote server.¹⁰ When the remote server receives *lip_start* message, it replies with a wakeup call and thus starts the file transfer procedure.

5.4 Remote Server

The remote server is responsible for writing the files received from the local intermediary process to the disk on the remote node.

Wakeup call Upon startup, the remote server sends a *wakeup call* message to the local intermediary process as described in Section 5.3.2.2

File transfer Before sending the *wakeup call*, the remote server computes the MD5 checksum for all *backup* files. After having sent the *wakeup call*, the remote server receives *advertise_file* messages from the local intermediary process. If the MD5 checksum of the backup file and the one transmitted in the *advertise_file* message are different, or if there is no backup file with the same name as the advertised file, a *request_msg (request)* is sent and the file is transferred. Otherwise a *request_msg (skip)* is sent, and the file is not transferred. If there are any extra backup files (files for which there were no corresponding advertised files), they are deleted on the remote node.

Function call information After receiving information about a system call as described in Section 5.3.2.1, the remote server proceeds to the same `glibc write()` call on the remote node file copy. The `write()` calls are done in the same order on both nodes.

Error recovery If the remote server crashes, it is restarted and therefore it sends a new *wakeup call* to the local intermediary process. This begins the process of transferring the files.

⁹In the normal situation (not following a crash of the local intermediary process), the file transfer is initiated by the remote server sending a wakeup call to the local intermediary process.

¹⁰This message is sent every time the local intermediary process is started. However, since the local intermediary process is normally started before the remote server, this message is not actually received by the remote server. It is only received by the remote server if the remote server was started before the local intermediary process.

5.5 Supporting Applications

5.5.1 Adding wrapper functions

In the application we studied there were only two relevant function calls to be watched: `write()` and `open()`. The directory replication tool can be easily adapted for applications requesting other function calls to be monitored and replicated. This requires new wrapper functions to be added into the library, and the corresponding handlers in both the local intermediary process and the remote server. The replication library, local intermediary process, and remote server have been designed to facilitate the addition of such function wrappers and handlers. Adding a new function to watch should be straightforward.

There are, however, some pitfalls worth noting, especially in the case of input/output on streams (e.g., `fwrite()`). One could rely on the lower-level equivalents (e.g., `write()`), since they will be called by the higher-level functions. However, due to the way `glibc` is linked, it is not possible to intercept these lower-level functions when they are called by their higher-level equivalents. It could be done, but at the cost of patching the `glibc` library [16]. A better option would be to handle those functions directly, by writing new wrapper functions for the higher-level I/O functions.

5.5.2 Stand-alone configuration

With no modifications, the directory replication tool can also be used outside of the ARMOR environment as a stand-alone tool.

5.6 Portability and Limitations

All experiments have been done on a PowerPC running Linux 2.2.17. The present implementation is believed to be working on any platform running Linux, as long as both the local and the remote nodes have the same endianness and byte-size for integers. It also should be easily ported to Sun Solaris OS, which allows library preloading using the environment variable `LD_PRELOAD`, just as Linux does [9].

The main limitation of the proposed directory replication tool is that it cannot work with statically linked object code. The interposition mechanism we use in the replication library works

only with dynamically linked applications. The executable file of an application dynamically linked against a shared library will not contain the object code for the library functions it calls. Instead, when the program is loaded, the loader loads the library into memory. This requires that there be an accessible copy of the shared library (usually in `/lib`, `/usr/lib`, or `/usr/local/lib`). The executable file of an application statically linked against a static library will contain the object code for the library functions that are used. It is not possible in this case to use an alternate implementation of any function or library. However, only very few commercial applications are statically linked against the standard C library. Another limitation is that processes linked to the replication library are ignoring the SIGPIPE signal. This signal is ignored in order to prevent this process from crashing due to a SIGPIPE signal that could be raised in the case of a crash of the local intermediary process. This may cause problems if the application protected by the replication library expects this signal not to be blocked.

CHAPTER 6

EXPERIMENTAL RESULTS

This chapter presents experimental evaluation results of the performance and reliability of the ARMOR-based failover support for the application. All measurements have been conducted on a Motorola CPX8216 cPCI chassis with two PowerPC CPU Cards,¹ shown in Figure 6.1, running Linux version 2.2.17. These two CPU cards are the primary and backup nodes.

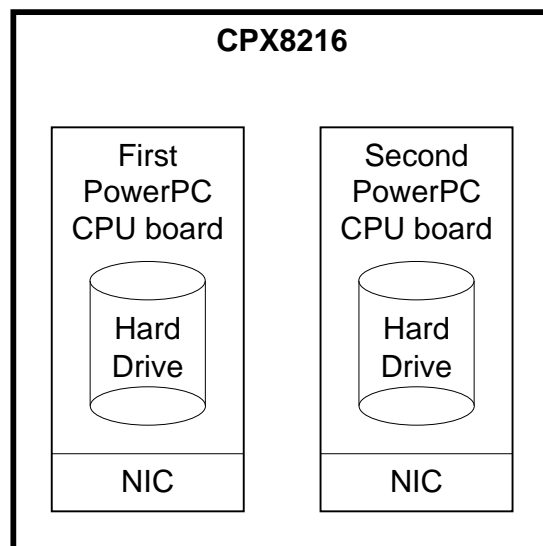


Figure 6.1 Motorola CPX8216.

6.1 Performance

This section presents the performance overhead due to the directory replication tool (see Chapter 5) and discusses performance of the failover mechanism.

¹PreP Mesquite cPCI (MCP750) with HAC CPU modules with 350 MHz PowerPC CPUs, 128 MB of RAM.

6.1.1 Performance overhead directory replication tool

The performance overhead due to the directory replication tool consists of: (1) *initialization time*, the time to transfer files from the primary node to the backup node at startup (see Section 5.3.2.3), and (2) *application overhead*, the overhead introduced by each call to the new versions of `write()` and `open()` functions.

Initialization time. As described in Chapter 5, after receiving the *wakeup call* from the remote server (running on the backup node), the local intermediary process blocks all write operations made by the application until the local intermediary process finishes the file transfer to the remote server.

The initialization time has been measured assuming that there is no copy of the watched files on the backup node. The measurements are conducted for three sets of files:

1. *resume.ps* (781 kB).
2. *glibc-2.3.tar* (87 MB).
3. *db/* directory and all the files therein (1678 kB), which is the database for the middleware and the application.

For each set, we measure the time between the start of the remote server and the completion of the file transfer. The *throughput* is obtained by dividing this initialization time by the size of the transferred files. The results are shown in Table 6.1.

Table 6.1 Initialization time results.

File	Transferred data	Initialization time (s)	Throughput (kB/s)
resume.ps	781 kB	0.591	1338
glibc-2.3.tar	87 MB	106.2	819
db/	1678 kB	1.196	1403

Application slowdown. Use of the replication library slows down the execution of the application. There are multiple reasons for this slowdown.

- **Overhead due to loading the replication library.** The application loader must load the library each time a new process starts. This is a “static” overhead that is the same for all processes. This overhead has been measured using a simple C program: `int main(int argc, char *argv[]){}`. This program is executed 1000 times without the replication library and then executed with the replication library being loaded at each execution.² The measured “static” overhead is of the order of 2.1 ms which is negligible for any real application.
- **Overhead due to wrapper functions.** The second source of overhead is the execution of the wrapper function for `write()`. The wrapper function needs to identify the file being accessed, verify that this file is being watched, and if so, communicate with the local intermediary process by sending the information about the `write()` function call (filename, position, number of bytes to write, data, etc.).
- **Overhead due to maintaining strict ordering.** The last source of overhead is that only one `write()` call can be processed at a time by the local intermediary process. Any subsequent `write()` calls by the application are blocked. This is explained in detail in Section 5.2.3.

To measure the overhead resulting from calls to `write()`, `gzip` (a compression-decompression utility that uses extensively `write()` calls) has been used as a benchmark application. In the tests, we have measured the time to successively compress and decompress a file. The files used were *glibc-2.3.tar* (source code for the GNU `glibc`, 87 MB uncompressed, 18 MB compressed) and a Postscript file, *resume.ps* (781 kB uncompressed, 547 kB compressed). We have also measured the time to execute an application using the middleware. For the `gzip` benchmark, a run consisted of a single compression of a file (*resume.ps* or *glibc-2.3.tar*) immediately followed by the decompression of the compressed file. For the middleware-based application benchmark, a run consisted of starting the database server, starting the watchdog with a test module (`test_mod`), and waiting for that test module to terminate. These measurements were conducted for five different scenarios:

1. Baseline, without the replication library (*baseline*).
2. With the replication library but without the local intermediary process (*no_lip*). In this

²The `LD_PRELOAD` environment variable is set to force the loader to load the replication library before executing the application.

configuration, no failover is provided, and this measurement shows the performance impact of a failure of the local intermediary process.

3. With the replication library and the local intermediary process but without the remote server (*no_rs*). In this configuration, no failover is provided and this measurement shows the performance impact of a failure of the remote server.
4. With the system working in full configuration: replication library, local intermediary process, and remote server (*dir_rep_tool*). This is the primary operation mode.
5. Without the replication library, but storing the files on the remote node using NFS (network file system) (*nfs_baseline*). Note that the middleware cannot use files stored on NFS. Although NFS and the directory replication tool cannot be directly compared due to differences in functionality (e.g., NFS does not replicate files, does not allow for dirty disconnections, etc.), both NFS and the directory replication tool transfer files over the network invisibly to the application. Therefore, NFS can be considered as a performance baseline for the directory replication tool.

The results are shown in Figure 6.2 and Table 6.2. One can see that the overhead due to

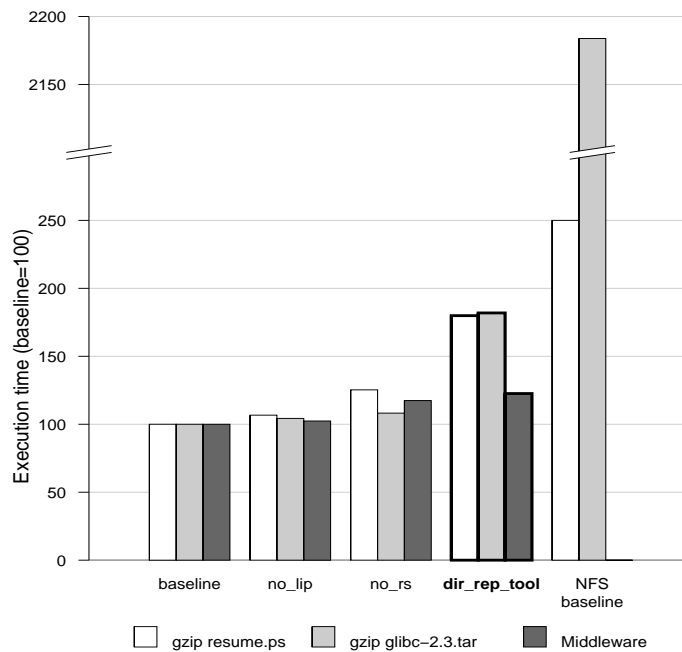


Figure 6.2 Slowdown results.

the replication library alone (*baseline*, 2.4% to 6.7%), and to both the replication library and the local intermediary process (*no_lip* and *no_rs*, 8.2% to 25.3%) are relatively small. The overhead with the full configuration of the directory replication tool working is high for *gzip* (*dir_rep_tool*, 79.9% to 81.9%), but lower for the middleware-based application (22.6%). The smaller overhead for the middleware-based application is due to less intensive I/O. However, the overhead with the directory replication tool is much lower than the overhead when using NFS (*nfs_baseline*, 150% to 2083%). This is mainly due to the fact that the directory replication tool does not require network communication for inputs (e.g., `read()`).

Table 6.2 Slowdown results.

Scenario	Benchmark	Execution time* (s)	Overhead (%)	Runs
baseline	gzip resume.ps	0.636 ± 0.019 (0.618–0.850)	n/a	1754
no_lip	gzip resume.ps	0.679 ± 0.018 (0.659–0.959)	6.7	1754
no_rs	gzip resume.ps	0.797 ± 0.023 (0.771–1.340)	25.3	1754
dir_rep_tool	gzip resume.ps	1.144 ± 0.627 (0.989–12.318)	79.9	1754
nfs_baseline	gzip resume.ps	1.590 ± 0.128 (1.335–2.432)	150.0	1754
baseline	gzip glibc-2.3.tar	109.5 ± 7.6 (97.5–134.5)	n/a	300
no_lip	gzip glibc-2.3.tar	114.2 ± 5.4 (99.5–135.1)	4.3	300
no_rs	gzip glibc-2.3.tar	118.5 ± 6.0 (108.1–138.7)	8.2	300
dir_rep_tool	gzip glibc-2.3.tar	199.2 ± 12.3 (171.1–248.5)	81.9	300
nfs_baseline	gzip glibc-2.3.tar	2390.4 ± 7.1 (2377.9–2401.2)	2083.8	14**
baseline	middleware	7.231 ± 0.911 (6.809–11.702)	n/a	694
no_lip	middleware	7.399 ± 1.102 (6.878–11.719)	2.4	693
no_rs	middleware	8.483 ± 1.160 (7.943–13.309)	17.4	693
dir_rep_tool	middleware	8.861 ± 1.363 (8.086–15.142)	22.6	693
nfs_baseline	middleware	n/a	n/a	n/a

* Execution time ± standard deviation (minimum–maximum). ** Even though the number of runs is smaller than for other scenarios, the mean execution time remains reliable because of the small standard deviation and difference between the maximum and minimum observed execution times.

6.1.2 Failover

In the previous section, we measured the performance impact of the directory replication tool on the execution of the server application. In this section, we measure the time it takes for the failover of the server application from the primary node to the backup node to complete. We measure the time elapsed from the node failure until the server application is started and working on the backup node. This time is called mean time to recover (MTTR). The MTTR is the time when the server application is unavailable for clients.

The MTTR is composed of two parts. The first is the time elapsed between the occurrence of the failure and the time at which the server application is started on the backup node. It includes the time to detect the failure, to notify the ARMOR daemon on the backup node, to finish the IP address migration (see Chapter 4), and to start the local intermediary process (see Chapter 5). It does not depend on the server application. The second part is the time elapsed between the time at which the server application, with the replication library being preloaded (see Chapter 5), is started and the time at which the application is operational and able to receive client connections. In the following discussion, we call these two parts respectively MTTR1 and MTTR2 (see Figure 6.3).

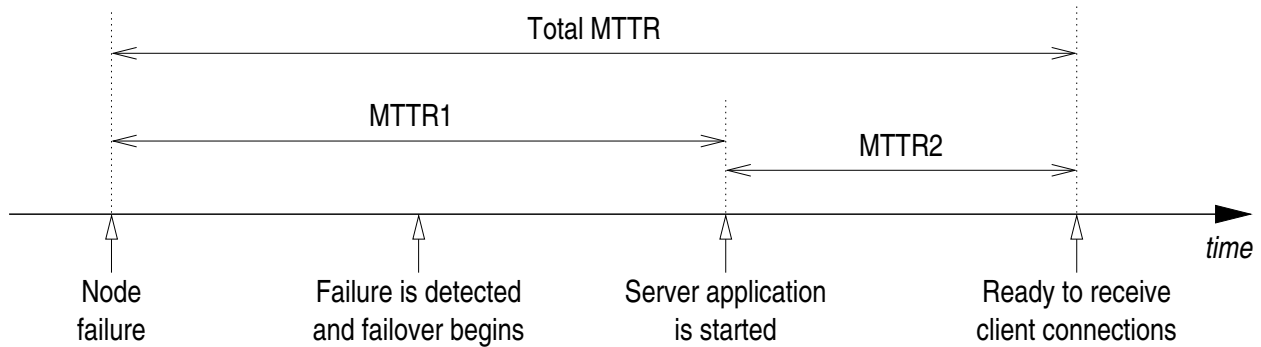


Figure 6.3 Mean time to recover (MTTR).

The measured MTTR1 is of the order of 15 s. It is mainly due to the time needed to detect the node failure. The detection mechanism is based on heartbeats. The node failure is detected when a timeout occurs after the last heartbeat was sent. MTTR1 is directly dependent on the interval between two heartbeats and on the timeout.³ It can be reduced at the price of a slightly higher network load and possibly a higher number of false-positives. Once the failure is detected,

³Statistically, it is equal to half the interval between two heartbeats plus the timeout period.

the ARMOR daemon on the backup node is notified and completes the IP address migration in less than 1 s. The measurements for MTTR2 and total MTTR are shown in Table 6.3.

Table 6.3 Mean time to recover (MTTR) results.

Server Application	MTTR2 (s)	Total MTTR (s)	Runs
test_mod	5.7	22.1	100
Full application	58	72	50

6.2 Reliability

Reliability is provided to the middleware-based applications through a multilevel ARMOR-based error detection and recovery hierarchy illustrated in Figure 6.4. Each level is responsible for detecting errors to the components directly below. ARMOR FTM and daemons reside at the highest level—Level 3. They are responsible for detecting errors in Level 2 processes and for their recovery. ARMOR FTM and daemons are also responsible for error detection and recovery among the two. The middleware watchdog and the local intermediary process reside at Level 2. The middleware watchdog is responsible for error detection and recovery in middleware-based modules and processes. Middleware-based modules and applications (e.g., application modules, test module, database server, etc.) are in the last level. This section presents the results of fault injections to these elements of the error detection and recovery architecture. It begins with the directory replication tool, follows with the middleware, including middleware-based applications, and concludes with ARMORs.

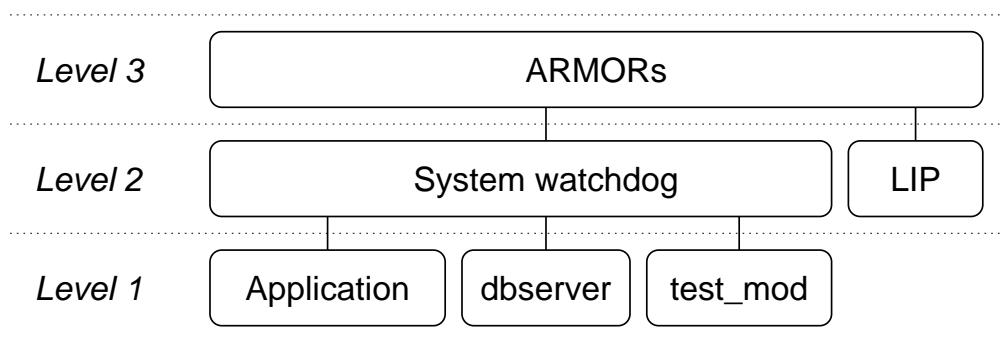


Figure 6.4 Error detection and recovery hierarchy.

6.2.1 Fault injections to the directory replication tool

Three test campaigns were conducted to test the robustness to crashes of the directory replication tool. The first two campaigns consisted of 300 fault injections (SIGKILL) each to the application process and the local intermediary process. (The SIGKILL signal causes immediate process termination. It can be obtained by the UNIX command `kill -9 <pid>`, where `<pid>` is the PID of the process to terminate. The SIGKILL signal simulates a process crash.) The third campaign consisted of 600 fault injections (SIGKILL) to the remote server using two different scenarios. The application process used in this experiment was `gzip`.

6.2.1.1 Fault injections to the application

In this campaign, the main objective was to test the error recovery mechanism described in Section 5.2.4. Each run consisted of a SIGKILL signal delivered to the `gzip` process during the compression of a Postscript file of 781 kB. Table 6.4 shows that all injected faults were successfully recovered from, including the 12.3% of them which occurred inside the `write()` wrapper function provided by the replication library and which required the retransmission of the file being accessed at the time of the crash. These results prove the robustness of the mechanism for application error recovery implemented by the local intermediary process and the remote server.

Table 6.4 Application crash effects on the directory replication tool.

	Faults injected	Complete file retransmission required	Successful recoveries
Number of cases	300	37	300
As % of injected faults	100.0%	12.3%	100.0%

6.2.1.2 Fault injections to the local intermediary process

The objective of this campaign was to measure the effects of a crash of the local intermediary process on the application using the replication library. Each run consisted of a SIGKILL signal delivered to the local intermediary process during the compression of a postscript file of 781 kB by `gzip`.

The results in Table 6.5 show that all crashes of the local intermediary process were successfully recovered from. In 7.7% of the crashes (23 occurrences), error propagated and caused the remote server to exit prematurely, but in all these cases the remote server was successfully restarted. Such cases are due to the local intermediary process crashing while transferring information to the remote server and the TCP connection being broken. Any crash of the local intermediary process requires the retransmission of all files to the remote server. Consequently, even premature exit of the remote server due to propagated errors does not add additional overhead beyond what would be occurred anyway. If the remote server had not crashed, the procedure would have required the local intermediary process to force the remote server to start its initialization procedure again (which requires sending the wakeup call to initiate the retransmission of files). When the remote server crashes and is restarted, the same file retransmission procedure is initiated by the remote server.

Table 6.5 Local intermediary process crash effects on the directory replication tool.

	Faults injected	Remote server restart required	Successful recoveries
Number of cases	300	23	300
As % of injected faults	100.0%	7.7%	100.0%

6.2.1.3 Fault injections to the remote server

The objective of this campaign was to measure the effects of a crash of the remote server on the local intermediary process. In the first scenario (under workload), each run consisted of a SIGKILL signal delivered to the remote server during the compression of a postscript file of 781 kB by gzip. In the second scenario (without workload), each run consisted of a SIGKILL signal delivered to the remote server with no application running. The results in Table 6.6 show that all crashes of the local intermediary process were successfully recovered from. In the first scenario (under workload), all crashes resulted in the remote server exiting prematurely, but being successfully restarted. Again, note that the overhead due to the local intermediary process restarting (which requires retransmission of all files to the remote server) would have been incurred anyway, even if

the local intermediary process had not exited and been restarted. A file retransmission between the local intermediary process and the remote server is the normal procedure after the crash of either the local intermediary process or the remote server.

Table 6.6 Remote server crash effects on the directory replication tool.

	Faults injected	Local intermediary process restart required	Successful recoveries
Scenario 1: under workload			
Number of cases	300	300	300
As % of injected faults	100.0%	100.0%	100.0%
Scenario 2: without workload			
Number of cases	300	0	300
As % of injected faults	100.0%	0.0%	100.0%

6.2.1.4 Directory replication tool summary

In these three campaigns, a total of 1200 faults have been injected into the directory replication tool. The major results can be summarized as follows: (1) none of the faults injected into the local intermediary process or the remote server propagated to the application process, (2) all faults were successfully recovered from, and (3) in some cases, a fault in the local intermediary process propagated to the remote server, or vice-versa, but without much higher overhead or lasting effects, and both the local intermediary process, and the remote server were successfully recovered in such cases.

6.2.2 Fault injections to the middleware

To measure the reliability of the middleware, three fault injection campaigns were conducted. The system watchdog, the database server, and the test module were targeted. The injected faults were transient single-bit flips in the text segment, one fault being injected per experiment. NFTAPE (Network Fault Tolerance and Performance Evaluator) [17] and its light-weight fault injector DBFI

(Debugger Based Fault Injector) were used to inject the faults. Modifications to the middleware code were made to detect the effect of faults⁴ and to allow fault injections.⁵ Each experiment consisted of creating clean database files, starting the middleware and the database server, and injecting a fault in either the system watchdog, the database server, or the test module. After the execution of the test module, the middleware and the database server were stopped and started again to check if the database files had been corrupted. After the experiment was finished, the output of the two consecutive runs were compared to a baseline output to detect the result of the fault injection. The observed outcome categories are described in Table 6.7. Rates of activated and not activated faults are given relative to the number of injected faults. Rates of all categories of manifested faults (CR, EX, AD, HG, DCR, and FSV) and rates of not manifested faults are given relative to the number of activated faults.

6.2.2.1 Fault injections to the watchdog

In the first campaign, 2000 faults were injected into the watchdog. The results of this campaign are shown in Figures 6.5 and 6.6, and Table 6.8. Out of the 2000 injected faults, only 104 were activated (5.2%) and 62 manifested (59.6% of activated faults). The low number of activated faults is due to the fact that most of the watchdog code is rarely used, and some parts are dedicated to recovery or a different configuration than the one we used. Those parts were therefore never executed during this campaign. However, 99%⁶ of activated faults did not manifest, or caused a crash or an abnormal exit. Only on one occasion (1% of activated faults) did a fail-silence violation occur.⁷ This result shows that a very high proportion of errors in the watchdog could be detected by using an external crash and abnormal exit detection system as the one implemented by the ARMOR infrastructure.

⁴Output of text messages.

⁵A `wait(2)` was added at the beginning of `main()` functions of both the watchdog and the database server to allow for synchronization with the fault injector.

⁶This number is derived by adding the number of faults that did not manifest (42), or resulted in a crash (54) or an abnormal exit (7), and dividing this sum (103) by the number of activated faults (104).

⁷Erroneous output was produced by the text module as a result of this fault injected into the watchdog.

Table 6.7 Outcome categories of fault-injection experiments.

Outcome Category	Description
Not Activated	The program never reached the corrupted instruction. No failure can occur.
Activated	The program reached and executed the corrupted instruction.
Not Manifested (NM)	The fault was activated but did not cause an abnormal and visible impact on the application. The output produced by the application was correct and the database was not corrupted.
Manifested	The fault was activated and caused an abnormal and visible impact on the application.
Crash (CR)	The injected process crashed. This may be due to the process attempting to execute an illegal instruction, to access to a memory location not allocated to this process, to write to a read-only memory location (segment violation), or other invalid memory access (e.g., bus error). The process exits and sends one of the following signals: SIGILL (illegal instruction), SIGSEGV (segment violation), SIGBUS (bus error). In the case of a crash of the test module, the module was always restarted by the watchdog
Abnormal exit (EX)	The process exited before completion of its normal execution. In the case of an abnormal exit of the test module, the module was always restarted by the watchdog
Application detection (AD)	An error was detected by application internal check.
Hang (HG)	The process ceased to make progress.
Database corruption (DCR)	The database was corrupted. All subsequent database accesses gave incorrect results. The database needed to be manually reinitialized.
Fail silence violation (FSV)	The error propagated to another module or the output produced by the process was incorrect (excluding database corruptions).

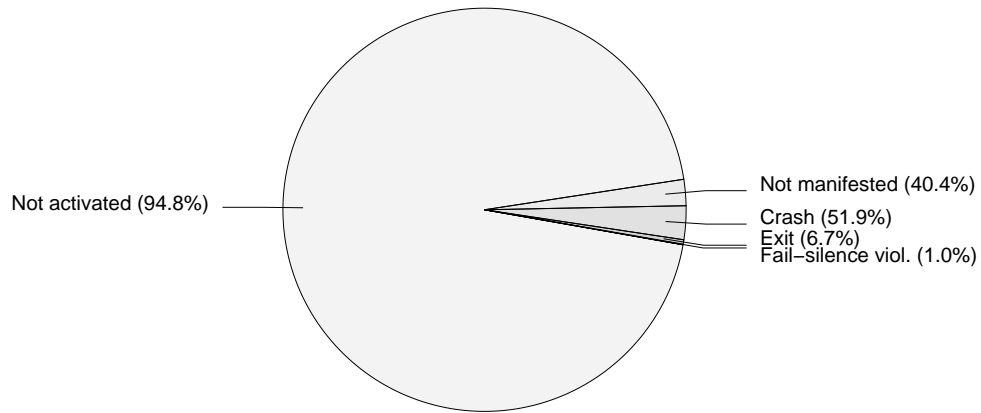


Figure 6.5 Middleware fault injection results for the watchdog module.

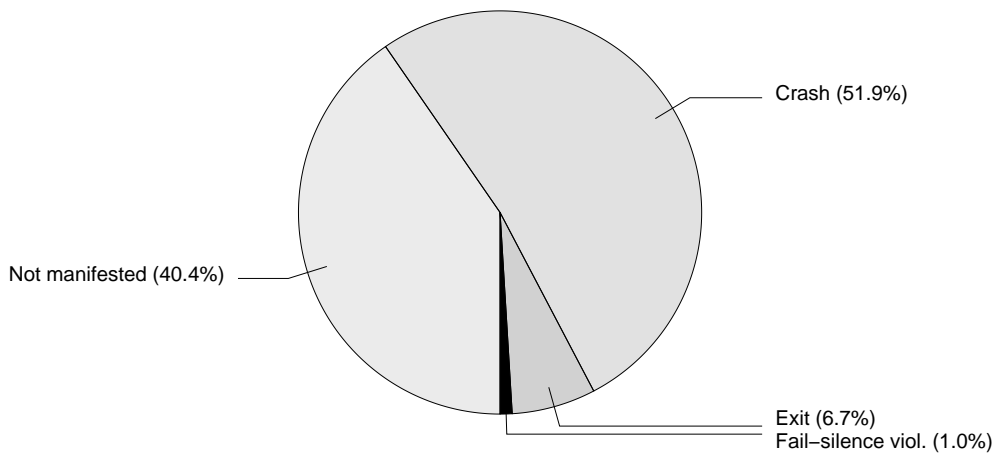


Figure 6.6 Middleware fault injection results for the watchdog module (activated faults).

Table 6.8 Middleware fault injection results for the watchdog module.

	Injected faults					
	Activated faults					FSV
	NM	CR	EX	FSV		
Cases	2000	104	42	54	7	1
%		5.2%	40.4%	51.9%	6.7%	1.0%

6.2.2.2 Fault injections to the application

Fault injections to the test module. The second campaign consisted of 4513 faults injected into the test module `test_mod`. The results are shown in Table 6.9 and Figure 6.7. Out of the 4513 injected faults, 3072 were activated (68.1%) and 493 manifested (34.2% of activated faults). These activation and manifestation rates are common for typical processes. The relatively high rate of fail-silence violations is due to the design of the test module. Due to the extensive output produced by the test module, faults that would otherwise not have been manifested produced an output different than the expected one, and are classified as fail-silence violations. The major result of this campaign is the low number of activated faults that resulted in fail-silence violations, database

Table 6.9 Middleware fault injection results for module `test_mod`.

	Injected faults								
	Cases	4513	Activated faults						
			NM	CR ^{†‡}	EX [†]	AD	HG	DCR [‡]	FSV
Cases	4513	3072	948	1242	4	309	3	2	566
% [§]		68.1%	30.9%	40.4%	0.1%	10.1%	0.1%	0.1%	18.4%

[†] All crashes and abnormal exits of the test module resulted in the watchdog restarting the module.

[‡] Including 2 cases with both DCR and CR. [§] Total of activated faults may exceed 100%.

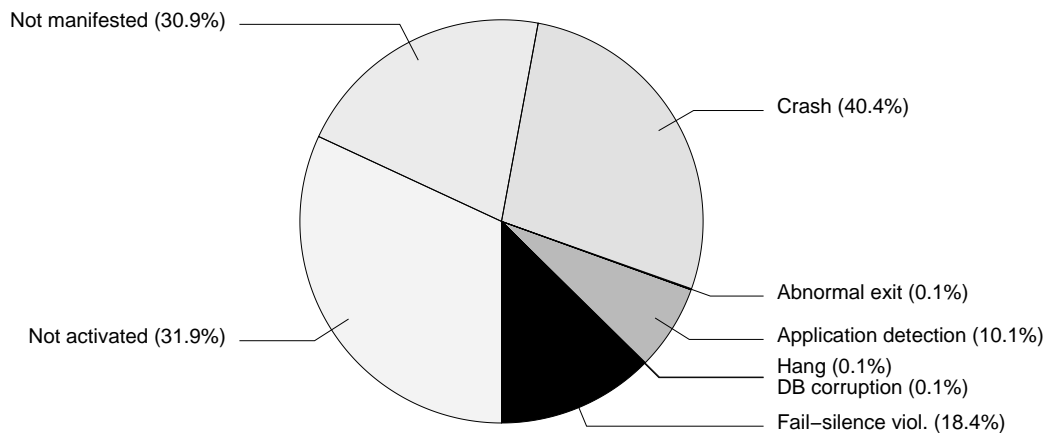


Figure 6.7 Middleware fault injection results for module `test_mod`.

corruptions, or hangs (18.6% of activated faults).⁸ These are the only manifested faults that cannot be easily recovered from by the middleware watchdog (unlike crashes and abnormal exits),⁹ or by procedures internal to the module that could be easily implemented (unlike application detected errors).

Fault injections to the database server. In the last campaign, 2000 faults were injected into the database server `dbserver`. Out of these 2000 injected faults, 609 were activated (30.5%) and 354 manifested (58.1% of activated faults). The results in Table 6.10 and Figure 6.8 show a high percentage of activated faults leading to database corruptions (17.2%). This number is much higher than observed in the previous two campaigns. Faults injected directly into the database server have a higher chance of corrupting the database. In contrast, a fault injected into an application process (such as the test module) must first propagate to the database server in order to have a chance of corrupting database contents. This scenario is much less probable. The results also show that, as for the test module, only a small fraction of activated faults led to fail-silence violations, database corruptions, or hangs (21.3% of activated faults).¹⁰

Table 6.10 Middleware fault injection results for the database server.

	Injected faults								
	Activated faults								FSV
	NM	CR [†]	EX	AD [‡]	HG	DCR ^{†‡}			
Cases	2000	609	255	279	27	23	1	105	24
% [§]		30.5%	41.9%	45.8%	4.4%	3.8%	0.2%	17.2%	3.9%

[†] Including 91 cases with both DCR and CR. [‡] Including 14 cases with both DCR and AD.

[§] Total of activated faults may exceed 100%.

⁸This number is derived by adding the number of faults that lead to a hang (3), to a database crash (2), or to a fail-silence violation (566), and dividing this sum (571) by the number of activated faults (3072).

⁹During the campaign, all crashes and abnormal exits of the test module were successfully recovered from by the watchdog restarting the module.

¹⁰This number is derived by adding the number of faults that lead to a hang (1), to a database crash (105), or to a fail-silence violation (24), and dividing this sum (130) by the number of activated faults (609).

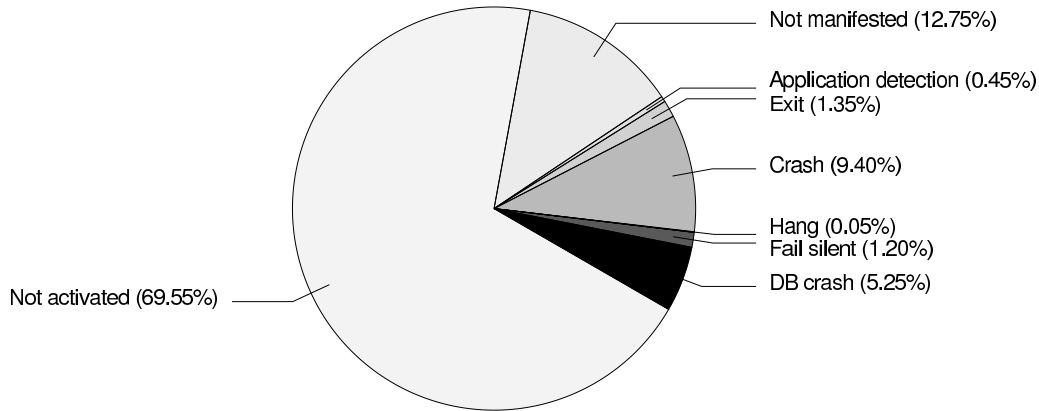


Figure 6.8 Middleware fault injection results for the database server.

Fault injections to the application. In a previous work, we injected 167 faults into two application¹¹ modules. During each run, a software workload emulator, running on a separate node of the network, was started. The results in Table 6.11 and Figure 6.9 show a 61.1% activation rate. This activation rate is artificially high due to a choice of injection points based on a profiling of the injected modules. The activation rate without using profiling was very low. Also, note that the database crash and hang rates are mainly due to the choice of injected functions. For instance, the mutex lock and unlock functions (`_db_mutex_lock()` and `_db_mutex_unlock()`) account for 3 out of 8 database crashes and 1 out of 2 hangs. During this experiment, fail-silence violations other than database corruptions were not monitored. However, even added together, hangs and database crashes account for only 9.8% of activated faults. Also, the version of the middleware and of the database server used by the application in these experiments was an older version and the database recovery was improved in newer versions.¹²

Table 6.11 Fault injection results for application modules.

	Injected faults						
	Activated faults						DCR
	NM	CR	AD	HG	DCR		
Cases	167	102	66	25	1	2	8
%		61.1%	64.7%	24.5%	1.0%	20.0%	7.8%

¹¹The application is using a different (though similar) version of the middleware.

¹²The fault injection to the watchdog, the test module, and the database server used that newer, improved version.

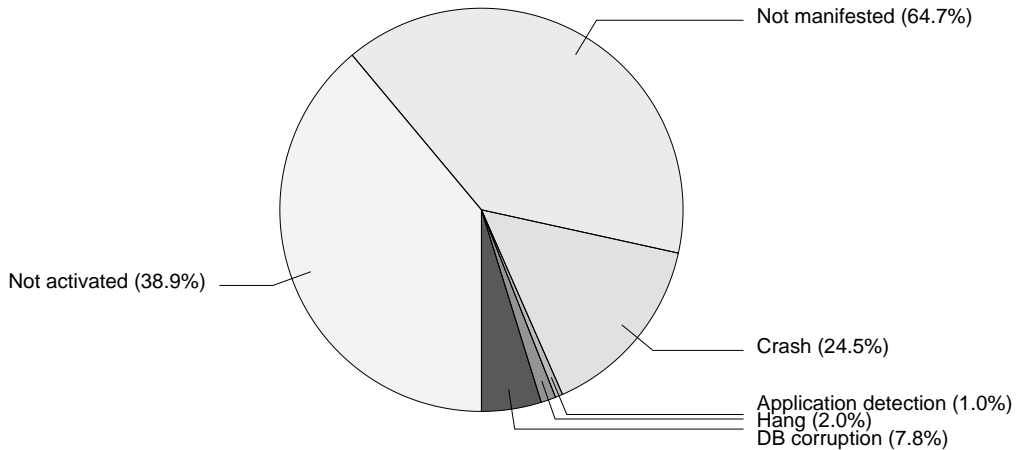


Figure 6.9 Fault injection results for application modules.

6.2.3 Fault injections to ARMORs

The objective of this work was to provide the node-failure and application crash detection and recovery elements within a hierarchical detection and recovery framework. The finer-grained detection and recovery of individual middleware-based modules were left to the middleware watchdog. The most relevant fault-injections to measure the effectiveness of the proposed solution in case of node-failure and application crash are SIGKILL signals. The results of Section 6.2.2 show that 98.4% of manifested faults in the middleware watchdog result either in a crash or an abnormal exit. Both of them can be modeled by SIGKILL.

6.2.3.1 Node failures

To measure the effectiveness of the ARMOR infrastructure, a campaign of 50 node failures was conducted. The node failures were simulated by killing (SIGKILL) the ARMOR daemon on the node running the middleware and the application. All node failures were successfully recovered. This result confirms the results given in [2] in similar experiments.

6.2.3.2 Embedded ARMORs failures

The effects of fault injections on standalone APP were discussed in Section 6.2.2. In this section we present the results of measures of fault injections on the middleware with ARMOR-based failover. In this campaign, 150 faults were injected into the watchdog, the test module, and the database

server. Each fault injection consisted of a SIGKILL signal. All crashes were detected and recovered from by the ARMORs.

6.2.4 Summary of the reliability analysis

The result of the reliability analysis of the hierarchical ARMOR-based error detection and recovery system can be summarized as follows: (1) the ARMOR infrastructure provided successful error detection and recovery for all (a) node failure, (b) ARMOR daemons failures, and (c) middleware watchdog and local intermediary process failures; (2) none of the faults injected into the local intermediary process or the remote server propagated to the application process; (3) all crashes of the remote server were successfully recovered from; and (4) only a small fraction of activated faults in the middleware modules (between 1 and 21.3%) was not recoverable.

CHAPTER 7

CONCLUSIONS

This thesis presents an ARMOR-based failover mechanism for off-the-shelf (or legacy) applications. This mechanism can be used to improve the reliability of applications without the need to modify to the application source code. The proposed solution permits a server application to be migrated from a failed node to a backup node transparently to the client and with access to an updated backup of the files stored on failed node. This mechanism is (1) based on the ARMOR framework that controls the other components, (2) uses the directory replication tool to maintain an updated copy of the files used by the application, and (3) uses an IP address migration tool to maintain connectivity between clients and the server application. The directory replication tool is a software tool that replicate on the backup node the files used by the application. The directory replication tool transfers to the backup node an initial copy of all files in selected directories and updates them each time they are modified by the application on the primary node. The directory replication tool uses the function call interposition technique [6]–[9]. The IP address migration tool allows the backup node to take over the IP address of the failed node. The IP address migration tool is based on ARP spoofing [3], [5]. The ARMOR framework [1], [2] provides self-checking, customizable reliability infrastructure for the two other components (the directory replication tool and the IP address migration tool) of this mechanism as well as for the application.

This mechanism was experimentally tested with a commercial application. The major findings are as follows: (1) all failures of elements of the proposed mechanism were successfully recovered, (2) the vast majority (79 to 99%) of failures in the application were successfully recovered, and (3) the performance overhead to the application, mainly due to file replication, is about (22%).

Future work. This mechanism could benefit from improvements to the directory replication tool and from a better integration with ARMORs. Possible improvements to the directory replication tool include: faster file and function call information transmissions, a finer-grained recovery procedure in case of remote server failure,¹ and a wider range of intercepted `glibc` functions. The remote server could also be integrated into the ARMOR infrastructure, and progress indicators could be added to both the local intermediary process and the remote server to detect hangs.

¹Now, all files need to be retransmitted in case of a remote server crash. The local intermediary process could maintain a list of files that were modified during the unavailability of the remote server.

REFERENCES

- [1] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. A. Whisnant, “Chameleon: A software infrastructure for adaptive fault tolerance,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 560–579, 1999.
- [2] K. A. Whisnant, “A process architecture and runtime environment for dependable distributed applications,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2003.
- [3] S. Horman, “Creating redundant Linux servers,” May 1998, [http://www.linux-ha.org/ failover/](http://www.linux-ha.org/failover/).
- [4] M. R. Barber, “Increased server availability and flexibility through failover capability,” in *Proceedings of the Eleventh Systems Administration Conference (LISA XI)*, USENIX, 1997, pp. 89–97.
- [5] Y. Volobuev, “Playing redir games with ARP and ICMP,” 1997, <http://www.rootshell.com>.
- [6] A. Baratloo, N. Singh, and T. Tsai, “Transparent run-time defense against stack smashing attacks,” in *Proceedings of the USENIX Annual Technical Conference*, 2000, pp. 257–262.
- [7] T. Tsai and N. Singh, “Libsafe 2.0: Detection of format string vulnerability exploits,” Avaya Labs, Murray Hill, NJ, Tech. Rep., Feb. 2001.
- [8] B. A. Kuperman and E. Spafford, “Generation of application level data via library interposition,” COAST Laboratory, West Lafayette, Indiana, Tech. Rep. CERIAS TR-1999-11, Oct. 1999, <https://www.cerias.purdue.edu/techreports-ssl/public/99-11.pdf>.
- [9] T. W. Curry, “Profiling and tracing dynamic library usage via interposition,” in *USENIX Summer*, 1994, pp. 267–278.

- [10] D. E. Comer, *Internetworking with TCP/IP: Principles, Protocols and Architecture*. 4th ed., Vol. 1, Upper Saddle River, NJ: Prentice-Hall, 2000.
- [11] D. C. Plummer, “RFC 826: Ethernet address resolution protocol or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware,” Nov. 1982, <ftp://ftp.internic.net/rfc/rfc826.txt>.
- [12] W. R. Stevens, *UNIX Network Programming. Networking APIs: Sockets and XTI*. 2nd ed., Vol. 1, Upper Saddle River, NJ: Prentice-Hall, 1998.
- [13] H. Pillay, “Setting up IP aliasing on a Linux machine mini-howto,” 2001, <http://www.tldp.org/HOWTO/mini/IP-Alias/>.
- [14] S. Loosemore, R. M. Stallman, R. McGrath, A. Oram, and U. Drepper, *The GNU C Library Reference Manual (for Version 2.1Beta)*. Cambridge, MA: Free Software Foundation, 1999, <http://www.gnu.org>.
- [15] R. L. Rivest, “RFC 1321: The MD5 message-digest algorithm,” Apr. 1992, <ftp://ftp.internic.net/rfc/rfc1321.txt>.
- [16] B. Baccala, “Preload libraries,” Nov. 2001, <http://www.freesoft.org/software/preload>.
- [17] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer, “NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors,” in *Proceedings of the 4th International Computer Performance and Dependability Symposium*, IEEE, Mar. 2000, pp. 91–100.

APPENDIX A

DIRECTORY REPLICATION TOOL MANUAL

This appendix explains how to use, from a user perspective, the directory replication tool described in Chapter 5.

A.1 Local Intermediary Process

The first step¹ is to run the local intermediary process. This is done by running, on the primary node, the command:

```
/usr/local/bin/lip
```

A.2 Remote Server

The next step is to run the remote server. This is done by running, on the backup node, the command:

```
/usr/local/bin/rs
```

A.3 Replication Library

Before executing the server application, on the primary node, the user needs to execute² (SH shell):

¹These three steps can be done in any order, however, the startup will be faster if you follow the order.

²Assuming that the replication library location is `/usr/local/lib/replib.so`.

```
export LD_PRELOAD=/usr/local/lib/replib.so
```

The user can also create the following script, where `/usr/local/bin/serverapp` is the name of the server application:

```
#!/bin/bash
export LD_PRELOAD=/usr/local/lib/replib.so
/usr/local/bin/serverapp
```

APPENDIX B

DIRECTORY REPLICATION TOOL ALGORITHMS

B.1 Replication Library

We give here the pseudocode for the write wrapper function.

```
1  ssize_t write(int fd, const void *buf, size_t count){
2      in_lib=true; /* To prevent recursive calls of the wrapper
3                   function. It is particularly important when
4                   writing to the log file. */
5
6      is_watched=is_file_fd_to_be_watched(fd);
7      if(is_watched and not in_lib){
8          open TCP connection to LIP;
9          send write_request to LIP;
10         wait for write_request_reply from LIP;
11         offset=lseek(fd, 0, SEEK_CUR); // Get current file position.
12     }
13
14     real_count=glibc_write(fd, buf, count);
15     bakerrno=errno; // Save the error number returned by write()
16
```

```

17  if(is_watched and not in_lib){
18      send write_arguments to LIP;
19      /* write arguments are:
20          - filename with path,
21          - offset (file position),
22          - realcount (number of bytes actualy writen). */
23
24      send write_data to LIP; /* The <count> bytes
25          contained in <buf>. */
26      close TCP connection to LIP;
27  }
28
29  errno=bakerrno;
30  return real_count;
31 }
32
33 is_file_fd_to_be_watched(int fd){
34     if(fd<0)
35         return false;
36
37     if(fstat(fd, &st)<0)
38         return false; // Unable to call fstat()
39
40     if(!S_ISREG(st.st_mode))
41         return false; // File is not a regular file
42         // according to fstat
43
44     if(readlink("/proc/self/fd/<fd>") does not
45         correspond to a file to be watched)

```

```

46     return false;
47
48     return true;
49 }
50
51 ssize_t glibc_write(int fd, const void *buf, size_t count){
52     pointer_to_glibc_write = dlsym(RTLD_NEXT, "write");
53     /* With handle RTLD_NEXT, dlsym returns a pointer to
54        the object named "write" from the libraries loaded after
55        the replication library, i.e., from glibc. */
56
57     return pointer_to_glibc_write(fd, buf, count);
58 }

```

B.2 Local Intermediary Process

Note that replication library is abbreviated as RL.

```

1  local_intermediary_process(arg){
2      listen for TCP connections from RL;
3      listen for TCP connections from RS;
4
5      send lip_restarted_msg to RS;
6
7      while(true){
8          wait for message;
9          if(wake-up call from RS){
10             transfer_files();
11         }
12         if(write_request from RL){

```

```

13     send write_request_reply to RL;
14     receive write_arguments from RL;
15     if(write_arguments not received from RL){
16         advertise_file();
17     }else{
18         receive write_data from RL;
19         if(write_data not received from RL){
20             advertise_file();
21         }else{
22             forward_func_call_to_RS();
23         }
24     }
25 }
26 if(open_request from RL){
27     send open_request_reply to RL;
28     receive open_arguments from RL;
29     if(open_arguments not received from RL){
30         advertise_file();
31     }else{
32         forward_func_call_to_RS();
33     }
34 }
35 }
36
37 forward_func_call_to_RS(){
38     send function_call_msg to RS;
39     wait for remote_ack from RS;
40     send function_call_data to RS;
41     wait for remote_ack from RS;

```

```

42 }
43
44 transfer_files(){
45     while(more files to transfer){
46         advertise_file();
47     }
48 }
49
50 advertise_file(){
51     compute MD5 checksum for file;
52     send advertise_msg to RS;
53     wait for message from RS;
54     if(request_msg){
55         send the file to RS;
56     }
57 }

```

B.3 Remote Server

```

1  remote_server(){
2      compute MD5 checksums for all watched files;
3      send wake-up call to LIP;
4      transfer_files();
5
6      while(true){
7          wait for remote_msg from LIP;
8          if(lip_restarted_msg){
9              compute MD5 checksums for all watched files;
10             send wake-up call to LIP;

```

```

11         transfer_files();
12     }
13     else if (function_call_msg){
14         send remote_ack to LIP;
15         receive function_call_data from LIP;
16         send remote_ack to LIP;
17         execute the function call on the backup file;
18     }
19 }
20 }
21
22 transfer_files(){
23     while(more files to transfer){
24         wait for advertise_msg from LIP;
25         compare MD5 checksums of advertised file and backup file;
26         if(MD5 checksums are equal){
27             send request_msg to LIP;
28             receive the file from LIP;
29         }else{
30             send skip_msg to the LIP;
31         }
32     }
33     delete any backup files that were not advertised by the LIP;
34 }

```

APPENDIX C

DIRECTORY REPLICATION TOOL LISTINGS

Due to the size of the code of the local intermediary process and the remote server (respectively 5151 and 2875 lines of code), the full source code is not included. Instead, their algorithms are provided in pseudocode in Appendix B. (The original source code is in C++.)

C.1 Replication Library

The replication library is composed of three C files, `config.h` (the configuration file), `replib.h` (the header file), and `replib.c` (the code file), and a Makefile to be used with `make` to build the library.

C.1.1 Source code of `config.h`

```
#ifndef DRT_CONFIG_H
#define DRT_CONFIG_H

/**
 * Directories to be watched
 */
#define WATCHED_DIR_PREFIX "/home/user/Test/"
#define WATCHED_DIR_PREFIX2 "/home/user/Test2/"
```

```

/**
 * This prefix is added to filenames on the remote node.
 * E.g., if REMOTE_PREFIX is "/tmp/RS", "/home/test/.bashrc" will
 * become "/tmp/RS/home/test/.bashrc".
 */
#define REMOTE_PREFIX "/tmp/DUPLIBREMOTE"

/**
 * Log file for the replication library (if logging activated
 */
#define LOG_FILE "/tmp/duplib.log"

/**
 * Log level for replication library
 */
#ifndef REPLIB_LOG_LEVEL
#define REPLIB_LOG_LEVEL 15
#endif
/* LOG_LEVEL */

/**
 * To disable logging:
 * #undef REPLIB_LOG_LEVEL
 */
#undef REPLIB_LOG_LEVEL

/**
 * Defines the port used for communication between the LIP and the
 * remote library.
 */

```

```

#define REPLIB_LIP_SERV_PORT 3105

/**
 * Defines the location of the LIP and the port number used for
 * file transferts.
 */
#define FT_SERV_NAME "ppc2.crhc.uiuc.edu"
#define FT_SERV_PORT 3177

/**
 * Defines the location of the RS and the port number used
 * communication of function call information between the LIP and
 * the RS.
 */
#define REMOTE_SERV_NAME "ppc.crhc.uiuc.edu"
#define REMOTE_SERV_PORT 5077

#endif /* DRT_CONFIG_H */

```

C.1.2 Source code of replib.h

```

#ifndef REPLIB_H
#define REPLIB_H

#define __USE_GNU
#include "config.h"
#include <limits.h>
#include <sys/types.h>
#include <stdio.h>

```

```

#define FUNCNAMELEN 12
#define MODE_LEN 3
struct fun_data_t {
    pid_t pid;
    char  function_ID[FUNCNAMELEN+1];
    int fd;
    pid_t child_pid;  /* for fork() */
    FILE* file;
    int flags;  /* for open() */
    int int_value;  /* For fputc() */
    mode_t mode;
    off_t pos;  /* for write() */
    size_t count;  /* for write() */
    ssize_t nmemb;
    ssize_t size;
    unsigned long f_pos; /* for "f" functions
                           (fwrite, fputs, etc.)
                           Using FILE* and not fd) */
    unsigned long request_id;
    char  f_mode[MODE_LEN+1];
    char  filename[PATH_MAX];  /* or old filename */
    char  newfilename[PATH_MAX];
    char  cwd[PATH_MAX];
};

struct write_request_reply_t {
    unsigned long request_id;
};

```

```
#endif /* REPLIB_H */
```

C.1.3 Source code of replib.c

```
/* The #define _GNU_SOURCE is to get asprintf and vasprintf */
#define _GNU_SOURCE
#include "replib.h"
#include <stdio.h>
#include <stdarg.h>
#include <dlfcn.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <errno.h>
#include <netdb.h>
#include <stdlib.h>
#include <time.h>
#include <sys/timeb.h>
#include <sys/stat.h>
#include <signal.h>
#include "config.h"

#define hton(a) (sizeof(a)==sizeof(long)?(htonl(a)):(htons(a)))

FILE *log_fd=NULL;

static int _no_log=0;
#define IN_LIB_LOG_LEVEL 2
```

```

#define LOG_FD log_fd

#ifdef REPLIB_LOG_LEVEL
#define REPLIB_LOG(level, format, args...) \
    if ((level <= REPLIB_LOG_LEVEL)\
        &&(in_lib<=IN_LIB_LOG_LEVEL)&&!_no_log){\
        REPLIB_LOG(LOG_FD, format, ## args);\
    }
#else
#define REPLIB_LOG(level, format, args...)
#endif

static int is_watched_file(int fd, char* buf, ssize_t size,
                           mode_t *mode);

extern char *canonicalize_file_name (const char *name);

#define O_CREAT          0100
/* from <bits/fnctl.h> Couldn't include fnctl.h directly
   because it defines open().*/

#define SEND_WRAP(function, args...) \
    if(_in_real_wrap==0){ \
        function(##args); \
    }

#define SEND_RQST_WRITE_WRAP(function, args...)\
    ((_in_real_wrap==0)?(function(##args)):(-1))

```

```

#define REAL_WRAP(function, args...) \
    _in_real_wrap++; \
    function(##args); \
    _in_real_wrap--; \
}

/**
 * Type definitions of intercepted functions
 */
typedef int (*open_t)(const char *filename, int oflag, ...);
typedef ssize_t (*write_t)(int fd, const void *buf, size_t count);
typedef int (*close_t)(int fd);

static int in_lib=0;
static int _in_real_wrap=0;

typedef char *(*getcwd_t)(char *buffer, size_t size);
static off_t getpos(int fd);
void replib_log(FILE *stream, const char *template, ...)
    __attribute__((format (printf, 2, 3)));
/* Tells compiler it uses a format string like printf */

static void *getglibcfn(const char *fn);

static int _in_replib=0;

void replib_log(FILE *stream, const char *template, ...){
    struct timeb t;

```

```

va_list ap;
int i;
char *s1=NULL,*s2=NULL;

ftime(&t);
i+=t.millitm/1000;
t.millitm=t.millitm%1000;
i=(int)t.time;
asprintf(&s1, "<%03d□%02d:%02d:%02d.%03d>□REPLIB(%d)□",
        i/(24*3600), (i/3600)%24, (i/60)%60, i%60,
        t.millitm , getpid());

va_start(ap, template);
vasprintf(&s2, template, ap);
va_end(ap);

if(s1&& s2){
    fprintf(stream, "%s%s", s1, s2);
}else{
    fprintf(stream, "Unable□to□produce□log□for□replib□for□pid□%d\n",
            getpid());
}

if(s1) free(s1);
if(s2) free(s2);
}

static ssize_t my_write(int fd, const void *buf, size_t count);
static ssize_t my_write(int fd, const void *buf, size_t count){
    static write_t real_write = NULL;

```

```

    if(!real_write)
        real_write = (write_t) getglibcfn("write");

    return real_write(fd, buf, count);
}

#define REPLIB_LIP_SERV_NAME "127.0.0.1"

/**
 * Low Level Functions
 * *****
 */
static int send_open(const char *filename, int flags,
                    mode_t mode, int fd)
{
    int sockfd;
    struct sockaddr_in servaddr;
    struct hostent *p_h;
    struct fun_data_t fun_data;

    if(!_in_replib)
        return 0;

    _in_replib++;

    bzero(&fun_data, sizeof(fun_data));
    strncpy(fun_data.function_ID, "open", FUNCNAMELEN);
    fun_data.pid=htonl(getpid());

```

```

getcwd(fun_data.cwd, sizeof(fun_data.cwd));

strncpy(fun_data.filename, filename, sizeof(fun_data.filename)-1);
fun_data.flags=htonl(flags);
fun_data.mode=htonl(mode);
fun_data.fd=htonl(fd);

sockfd=socket(AF_INET, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
p_h=gethostbyname(REPLIB_LIP_SERV_NAME);
if(p_h == NULL){
    REPLIB_LOG(1, "%s: unknown host. Unable to duplicate I/O.\n",
        REPLIB_LIP_SERV_NAME);
    _in_replib--;
    return -1;
}

bcopy((char*)p_h->h_addr, (char*)&servaddr.sin_addr,
    p_h->h_length);
servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(REPLIB_LIP_SERV_PORT);

if(connect(sockfd, (struct sockaddr*)&servaddr,
    sizeof(struct sockaddr_in))<0){
    REPLIB_LOG(1, "%s: unable to connect. Unable to duplicate I/O.\n",
        REPLIB_LIP_SERV_NAME);
    close(sockfd);
    _in_replib--;
}

```

```

    return -1;
}

REPLIB_LOG(1, "OPEN: sending %u bytes\n", sizeof(fun_data));
if(my_write(sockfd, &fun_data, sizeof(fun_data))<0){
    REPLIB_LOG(1, "%s: error writing on stream socket."
        "Unable to duplicate I/O.\n", REPLIB_LIP_SERV_NAME);
    close(sockfd);
    _in_replib--;
    return -1;
}

close(sockfd);
_in_replib--;
return 0;
}

static int send_request_write(int fd, const char *fnbuf,
    unsigned long request_id)
{
    int sockfd;
    struct sockaddr_in servaddr;
    struct hostent *p_h;
    struct fun_data_t fun_data;
    struct write_request_reply_t reply;

    if(!_in_replib)
        return -1;
}

```

```

_in_replib++;

REPLIB_LOG(2, "In_send_request_write\n");
bzero(&fun_data, sizeof(fun_data));
strncpy(fun_data.function_ID, "rqst_write", FUNCNAMELEN);
fun_data.pid=htonl(getpid());
getcwd(fun_data.cwd, sizeof(fun_data.cwd));
strncpy(fun_data.filename, fnbuf, sizeof(fun_data.filename)-1);
fun_data.request_id=htonl(request_id);
fun_data.fd=htonl(fd);

sockfd=socket(AF_INET, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
p_h=gethostbyname(REPLIB_LIP_SERV_NAME);
if(p_h == NULL){
    REPLIB_LOG(1, "%s: unknown host. Unable to duplicate I/O.\n",
        REPLIB_LIP_SERV_NAME);
    REPLIB_LOG(2, "Out_send_request_write\n");
    _in_replib--;
    return -1;
}

bcopy((char*)p_h->h_addr, (char*)&servaddr.sin_addr,
    p_h->h_length);
servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(REPLIB_LIP_SERV_PORT);

if(connect(sockfd, (struct sockaddr*)&servaddr,
    sizeof(struct sockaddr_in))<0){

```

```

REPLIB_LOG(1, "%s: unable to connect. "
            "Unable to duplicate I/O.\n",
            REPLIB_LIP_SERV_NAME);
close(sockfd);
REPLIB_LOG(2, "Out send_request_write\n");
_in_replib--;
return -1;
}

REPLIB_LOG(1, "RQST_WRITE: sending %u bytes (request_id=%lu)\n",
            sizeof(fun_data), request_id);
if(my_write(sockfd, &fun_data, sizeof(fun_data))<0){
    REPLIB_LOG(1, "%s: error writing on stream socket."
                "Unable to duplicate I/O.\n", REPLIB_LIP_SERV_NAME);
    close(sockfd);
    REPLIB_LOG(2, "Out send_request_write\n");
    _in_replib--;
    return -1;
}

REPLIB_LOG(1, "RQST_WRITE: waiting for authorization\n");
if(recv(sockfd, &reply, sizeof(reply), MSG_WAITALL)
    <sizeof(reply)){
    REPLIB_LOG(1, "%s: error reading on stream socket "
                "(waiting for write request reply). "
                "Unable to duplicate I/O.\n", REPLIB_LIP_SERV_NAME);
    close(sockfd);
    REPLIB_LOG(2, "Out send_request_write\n");
    _in_replib--;
}

```

```

    return -1;
}

if(ntohl(reply.request_id)!=request_id){
    REPLIB_LOG(1, "Bad request_id returned: %lu (expected=%lu)\n",
        ntohl(reply.request_id), request_id);
    close(sockfd);
    REPLIB_LOG(2, "Out send_request_write\n");
    _in_replib--;
    return -1;
}

REPLIB_LOG(1, "Received reepliy for request_id: %lu\n",
    request_id);
REPLIB_LOG(2, "Out send_request_write\n");
_in_replib--;
return sockfd;
}

static int send_write(int sockfd, int fd, const void *buf,
                    ssize_t count, off_t pos, char *fnbuf,
                    unsigned long request_id, mode_t mode)
{
    struct fun_data_t fun_data;

    if(!_in_replib||(sockfd<=0))
        return 0;

    _in_replib++;

```

```

REPLIB_LOG(2, "In_send_write\n");
bzero(&fun_data, sizeof(fun_data));
strncpy(fun_data.function_ID, "write", FUNCNAMELEN);
fun_data.pid=htonl(getpid());
getcwd(fun_data.cwd, sizeof(fun_data.cwd));
strncpy(fun_data.filename, fnbuf, sizeof(fun_data.filename)-1);

fun_data.fd=htonl(fd);
fun_data.count=htonl(count);
fun_data.pos=htonl(pos);
fun_data.request_id=htonl(request_id);
fun_data.mode=hton(mode);

REPLIB_LOG(1, "WRITE: sending %u bytes\n", sizeof(fun_data));
if(my_write(sockfd, &fun_data, sizeof(fun_data))<0){
    REPLIB_LOG(1, "%s: error writing on stream socket. "
        "Unable to duplicate I/O.\n", REPLIB_LIP_SERV_NAME);
    close(sockfd);
    REPLIB_LOG(2, "Out_send_write\n");
    _in_replib--;
    return -1;
}

REPLIB_LOG(1, "WRITE: sending %d bytes\n", count);
if(my_write(sockfd, buf, count)<0){
    REPLIB_LOG(1, "%s: error writing on stream socket. "
        "Unable to duplicate I/O.\n", REPLIB_LIP_SERV_NAME);
    close(sockfd);

```

```

    REPLIB_LOG(2, "Out_send_write\n");
    _in_replib--;
    return -1;
}

close(sockfd);
REPLIB_LOG(2, "Out_send_write\n");
_in_replib--;
return 0;
}

static off_t getpos(int fd)
{
    return lseek(fd, 0, SEEK_CUR);
}

/* Return 1 if file is a normal file, 0 otherwise */
/* buf needs to be at least of size PATH_MAX+1 */
static int is_watched_file(int fd, char* buf, ssize_t size,
                           mode_t *mode)
{
    char ln[PATH_MAX];

    struct stat st;
    int i;

    if(fd<0)
        return 0;

```

```

if (size < PATH_MAX + 1) {
    REPLIB_LOG(0, "ERROR: Size of buf in is_watched_file needs"
               " to be AT LEAST PATH_MAX + 1\n");
    return 0;
};

if (fstat(fd, &st) < 0) {
    REPLIB_LOG(4, "Unable to fstat fd %d\n", fd);
    return 0;
}

if (!S_ISREG(st.st_mode)) {
    /* Not a normal file */
    REPLIB_LOG(4, "File (fd=%d) is not a normal file\n", fd);
    return 0;
}

if (mode)
    *mode = st.st_mode;

if (snprintf(ln, sizeof(ln), "/proc/self/fd/%u", fd) <= 0) {
    REPLIB_LOG(4, "ERROR with snprintf()\n");
    return 0;
}

i = readlink(ln, buf, size - 1);
if ((i < 0) || (i >= size - 1)) {
    REPLIB_LOG(4, "Unable to readlink <%s>\n", ln);
    return 0;
}

```

```

}

buf[i]='\0';

REPLIB_LOG(0, "Operation on file <%s>\n", buf);

if(strncmp(buf, WATCHED_DIR_PREFIX, strlen(WATCHED_DIR_PREFIX))
    && strncmp(buf, WATCHED_DIR_PREFIX2,
        strlen(WATCHED_DIR_PREFIX2))){
    REPLIB_LOG(0, "file not watched\n");
    return 0;
}

REPLIB_LOG(0, "File watched\n");

return 1;
}

/**
 * getglibcfn(const char* fn) returns the glibc function named <fn>
 */
static void *getglibcfn(const char *fn)
{
    void *p_fn;

    in_lib++;

    p_fn=dlsym(RTLD_NEXT, fn);
    if(!p_fn) /* Unable to find function <fn> */

```

```

    {
        REPLIB_LOG(stderr, "dlsym(%s) error: %s\n", fn, dlerror());
        _exit(1);
    }
    in_lib--;
    return p_fn;
}

```

```
int open (const char *filename, int oflag, ...)
```

```

{
    static open_t real_open = NULL;
    int mode=0;
    int fd;
    char *can_fn=NULL;

    in_lib++;

    if(!real_open)
        real_open = (open_t) getglibcfn("open");

    if (oflag & O_CREAT)
    {
        va_list arg;
        va_start(arg, oflag);
        mode = va_arg(arg, int);
        va_end(arg);
    }

    _in_real_wrap++;
}

```

```

fd = real_open (filename, oflag, mode);
_in_real_wrap--;

if(filename){
    can_fn=canonicalize_file_name(filename);
    if(can_fn){
        REPLIB_LOG(4, "open: can_fn= <%s>\n", can_fn);
    }else{
        REPLIB_LOG(4, "open: ERROR: Unable to canonicalize filename "
            "<%s>\n", filename);
        REPLIB_LOG(4, "open: canonicalize filename: %s\n",
            strerror(errno));
    }
}

if(filename){
    REPLIB_LOG(4, "open ([%s/]%s, 0x%x, 0x%x)\t=%d\n",
        getcwd(NULL, 0), filename, oflag, mode, fd)
}else{
    REPLIB_LOG(4, "open [[%s/]filename=NULL] (0x%x, 0x%x)\t=%d\n",
        getcwd(NULL, 0), oflag, mode, fd)
}

if((fd>0)&&(can_fn))
{
    SEND_WRAP(send_open, can_fn, oflag, mode, fd);
}

if(can_fn)

```

```

        free(can_fn);

in_lib--;
return fd;
}

int __open (const char *filename, int oflag, ...)
{
    static open_t real___open = NULL;
    int mode=0;
    int fd;
    char *can_fn=NULL;

in_lib++;

if(!real___open)
    real___open = (open_t) getglibcfn("__open");

if (oflag & O_CREAT)
    {
        va_list arg;
        va_start(arg, oflag);
        mode = va_arg(arg, int);
        va_end(arg);
    }

_in_real_wrap++;
fd = real___open (filename, oflag, mode);
_in_real_wrap--;

```

```

if(filename){
    can_fn=canonicalize_file_name(filename);
    if(can_fn){
        REPLIB_LOG(4, "__open: can_fn= <%=s>\n", can_fn);
    }else{
        REPLIB_LOG(4, "__open: ERROR: Unable to canonicalize "
            "filename <%=s>\n", filename);
        REPLIB_LOG(4, "__open: canonicalize filename: %s\n",
            strerror(errno));
    }
}

if(filename){
    REPLIB_LOG(4, "__open([%s/]%s, 0x%x, 0x%x)\t=%d\n",
        getcwd(NULL, 0), filename, oflag, mode, fd)
}else{
    REPLIB_LOG(4, "__open_[[%s/]filename=NULL](0x%x, 0x%x)"
        "\t=%d\n", getcwd(NULL, 0), oflag, mode, fd)
}

if((fd>0)&&(can_fn))
{
    SEND_WRAP(send_open, can_fn, oflag, mode, fd);
}

if(can_fn)
    free(can_fn);

```

```

    in_lib--;
    return fd;
}

ssize_t write(int fd, const void *buf, size_t count)
{
    static int in_write = 0;
    static write_t real_write = NULL;
    ssize_t size;
    off_t offset;
    unsigned long request_id;
    char fnbuf[PATH_MAX+1];
    int sockfd;
    mode_t mode;
    int bakerrno=errno;

    in_lib++;
    if(!real_write)
        real_write = (write_t) getglibcfn("write");

    if(((!_in_replib)&&(in_write==0)&&is_watched_file(fd, fnbuf,
                                                    sizeof(fnbuf),
                                                    NULL))){

        in_write++;

        request_id=rand();

        REPLIB_LOG(4, "write(%d, void *buf, %d) request_id=%lu\n",
                  fd, count, request_id)

```

```

sockfd=SEND_RQST_WRITE_WRAP(send_request_write, fd, fnbuf,
                             request_id);

    in_write--;
}

offset=getpos(fd);
_in_real_wrap++;
errno=bakerrno;
size = real_write (fd, buf, count);
bakerrno=errno;
_in_real_wrap--;

if(!_in_replib){
    errno=bakerrno;
    return size;
}

if((in_write==0)
    &&is_watched_file(fd, fnbuf, sizeof(fnbuf), &mode)){
    in_write++;

    REPLIB_LOG(4, "write(%d, void*buf, %d)[@%lu]\t=%d\n",
                fd, count, offset, size)

    SEND_WRAP(send_write, sockfd, fd, buf, size, offset, fnbuf,
                request_id, mode);
}

```

```

    in_write--;
}

in_lib--;
errno=bakerrno;
return size;
}

/*****
 * Library constructor and destructor *
 *****/
typedef FILE>(*fopen_t) (const char *filename, const char *modes);
static void _replib_constructor() __attribute__((constructor));
static void _replib_destructor() __attribute__((destructor));
static void _replib_constructor(void)
{
    fopen_t fo;
    char process_name[PATH_MAX+1];
    int i;

    _in_replib++;

    /* Ignore SIGPIPEs */
    signal(SIGPIPE, SIG_IGN);

    fo=(fopen_t) getglibcfn("fopen");

    i=readlink("/proc/self/exe", process_name,
               sizeof(process_name)-1);

```

```

    if((i<0)||i>=sizeof(process_name)-1)){
        process_name[0]='\0';
    }else{
        process_name[i]='\0';
    }

    log_fd=stderr;

    ++_no_log;
    log_fd=fo(LOG_FILE, "a");
    --_no_log;
    --_in_replib;
    REPLIB_LOG(4, "Starting library constructor for process <%s>"
        "pid%d ppid%d\n", process_name, getpid(), getppid());
}

typedef int (*fclose_t)(FILE *stream);
static void _replib_destructor(void)
{
    fclose_t fc;
    REPLIB_LOG(4, "Exiting library destructor for pid%d ppid%d\n",
        getpid(), getppid());

    ++_in_replib;
    ++_no_log;
    fc=(fclose_t) getglibcfn("fclose");
    fc(log_fd);
    --_no_log;
    --_in_replib;
}

```

```
}
```

C.1.4 Source code of Makefile

```
LIBNAMECORE_____=_replib
LIBSUFFIX_____=_$so
LIBNAMESHORT_____=_$(LIBNAMECORE).$(LIBSUFFIX)
LIBNAME_____=_$(LIBNAMECORE).$(LIBNAMESUFFIX).4
CC_____=_gcc
C++_____=_g++
CCFLAGS_____=_-O2_-Wall_-fPIC_-g
LDFLAGS_____=_-shared_-Wl,-soname,$(LIBNAMESHORT)_-ldl

all_____::_____$(LIBNAME)

${LIBNAMECORE}.o_____::_____${LIBNAMECORE}.c_${LIBNAMECORE}.h
_____$(CC)_-c_-o_@_$(CCFLAGS)_$<

${LIBNAME}:_____${LIBNAMECORE}.o
_____$(CC)_-o_${LIBNAME}_$(LDFLAGS)_${LIBNAMECORE}.o
_____ln_-sf_${LIBNAME}_$(LIBNAMESHORT)

clean_____::
_____rm_-f_${LIBNAMECORE}.o
_____rm_-f_${LIBNAMESHORT}*
```