UC Irvine
ICS Technical Reports

Title
LDFS : a fault-tolerant local disk-based file system for mobile agents

Permalink
https://escholarship.org/uc/item/8954t67w

Authors
Gendelman, Eugene
Bic, Lubomir F.
Dillencourt, Michael B.

Publication Date
2001

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ICS

TECHNICAL REPORT

LDFS: A Fault-Tolerant Local Disk-Based File System for Mobile Agents

Tech Rpt. # 01-21

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Eugene Gendelman, Lubomir F. Bic, and Michael B. Dillencourt

Department of Information and Computer Science
University of California, Irvine, CA, USA
{egendelm, bic, dillenco}@ics.uci.edu tel:949 824-6306 fax: 949 824-4056

Information and Computer Science
University of California, Irvine

LDFS: A Fault-Tolerant Local Disk-Based File System for Mobile Agents

Eugene Gendelman, Lubomir F. Bic, and Michael B. Dillencourt

Department of Information and Computer Science
University of California, Irvine, CA, USA
{egendelm, bic, dillenco}@ics.uci.edu tel:949 824-6306 fax: 949 824-4056

Abstract

A local disk-based file system, LDFS, is an attractive way to speed up distributed applications. Local file access is much faster than accessing data on remote file servers through the network. LDFS is also scalable, as it does not rely on centralized file servers, and it exploits already existing resources (local disks) to provide storage. However, since individual workstations are less reliable and less available than file servers, LDFS must be made fault tolerant. We present an approach that integrates the LDFS with the distributed application. This is particularly suitable for mobile agent systems, because they can easily migrate to access remote files. LDFS avoids logging of individual file accesses, which are regenerated automatically from application messages. Our experiments show that the overhead of checkpointing with LDFS is generally smaller that with NFS, while access time to files decreases dramatically.

Keywords: fault tolerant computing, stable storage, mobile agent distributed systems

1. Introduction

The file system is an important part of any distributed environment. The performance of the file system sometimes defines the performance of the distributed application. The subject was extensively studied, and many systems were proposed [4, 5, 7].

Many factors affect the performance of the file system. Since the file system is usually located on a separate machine (or set of machines) [4, 8], all file accesses are sent through the network. In order to improve the availability of the data and to achieve fault tolerance, the files are being replicated [3, 6, 7, 8]. This introduces the need for protocols to preserve replica consistency for purposes of crash recovery, and concurrent file access.

RECEIVED

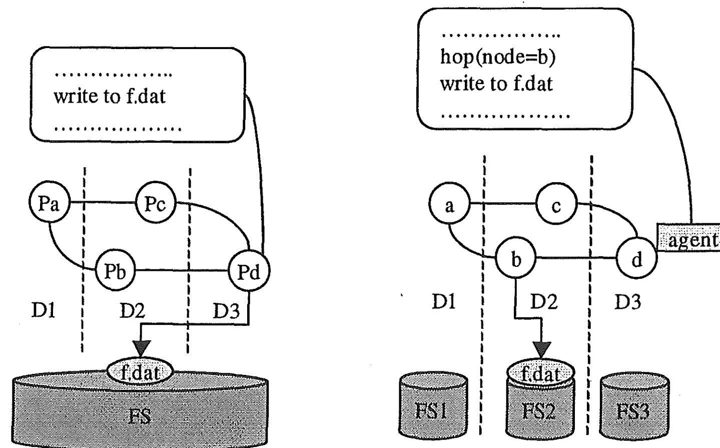
APR 15 2002

UCI LIBRARY

Aside from performance, there are two more characteristics of the file system that define its usability for distributed applications: fault tolerance [1, 7], since distributed applications may run for a long time, and its application-level extensibility, since in order to utilize the available resources, the file system should be easily portable and should not require any modification of the kernel [9].

In a message passing system, if several processes use the same file, a shared file system must be used. With mobile agent systems the use of a shared file system can be avoided. The files are logically attached to the logical nodes where they were open. In order for the agent to access the file, it has to be present at the logical node. In other words, instead of moving data to the code, we move code to the data. This concept is illustrated in Figure 1. As a result, there are no concurrent accesses to the file. Therefore, file can be stored on the local disk of the machine, which supports the logical node.

Such file access policy has several benefits compared to traditional file systems. Since the files are on the same machine with the calling process, the file operations are very fast. The system does not use dedicated file server, which makes it scalable. It utilizes the available resources, namely local disks. The machines, participating in the computation do not have to share a file system, which contributes to a better resource utilization.



a) Message passing system on the shared file system b) Mobile agent system on non-shared file system: the file f.dat is logically attached to the logical node **b**. Agent can access the file by first hopping to **b**.

Figure 1.

Several serverless file systems were proposed [1,2]. Comparing with those systems, LDFS is much simpler, as it does not have to take care of the concurrent file access and can always store files locally relative to the application process. As a result LDFS can use the file system interface provided on the node. The implementation of the LDFS in the mobile agent distributed system is described in [10].

The "weak spot" of such a file system is its lack of fault tolerance: in the case of a node

failure, all information stored on the local disk becomes inaccessible. This paper addresses this problem by providing the necessary mechanisms for fault tolerance. The paper is organized as follows: section 2 presents the design of the fault tolerant LDFS. The proposed algorithm is divided into two parts: replication and regeneration. Section 3 presents the approach to replication. In section we describe the regeneration techniques. The performance of LDFS is presented in section 5, and section 6 contains some final remarks.

2. Fault Tolerance in LDFS

Fault tolerance in a distributed computing system can be provided at two distinct levels: the *application* level and the *file system* level. Different approaches are generally used for each level. To make an application fault tolerant, *rollback-recovery* is the most common solution, which consists of three phases: taking periodic checkpoints, failure detection, and recovery. During the checkpointing stage each process captures its state and saves it to the stable storage. When failure occurs, the failure detection mechanism alerts the system. The recovery mechanism then recovers the system to a previously saved consistent system state [11] by loading files, saved during checkpointing. It is also possible to roll back only the failed process, instead of the entire system [12], by keeping track of the application messages sent after the checkpoint.

The fault tolerance of the file system is provided by data replication. This may be achieved using several different approaches, including *primary copy*, *majority consensus*, *weighted voting*, or *quorum consensus* [13]. In the case of a file server failure, another replica is used to replace the failed server. The main challenge is maintaining replica consistency. This requires the logging of all read/write requests sent between the file system and the application.

Figure 2(a) illustrates a typical approach. The rectangles represent application processes. The arrows are application messages. The circles are read/write requests to the file system. The line separating the application from the file system indicates that fault tolerance is provided at the two different levels independently, through different mechanisms: the application takes periodic checkpoints and logs the application messages, while the file system maintains file replicas and logs the read/write accesses to files. In other words, the correlation between the application messages and the file operations is not exploited; the application and the file system view each other as black boxes.

This could, at least in principle, be avoided if the application files were included in the state of the distributed application. Note that every application message triggers some number of file operations. This association between messages and sets of file operations is shown in Figure 2(b). Thus, when a machine is restarted from a checkpoint after a crash, the logged application

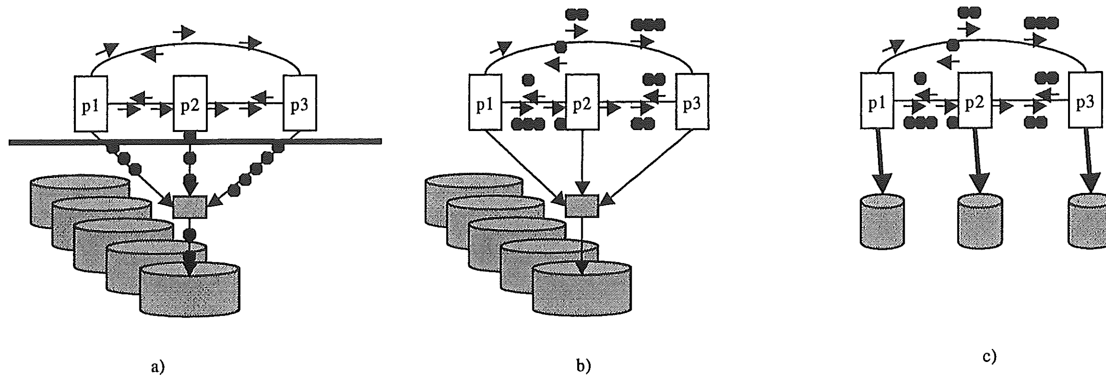


Figure 2. a) General file system. Messages are logged to provide fault tolerance of the distributed system. File accesses are logged for the file system recovery b) Shared file system integrated with the distributed system. Only messages are logged to provide both file system and distributed system fault tolerance c) Tightly coupled file system – distributed system.

messages are replayed. Each message then automatically regenerates the corresponding read/write request to the file system. In the case of a file server failure, a similar approach can be taken. The surviving replica communicates with the application and they both roll back to a previously saved state. Then the application messages are replayed and the read/write requests regenerated.

The main drawback of the approach described in the preceding paragraph is its complexity. If the application is distributed, then every node needs to figure out whether it should roll back to ensure system consistency. This decision is based on whether it performed file operations since the last checkpoint, and whether it communicated (directly or indirectly) with other processes that performed file operations. The resulting algorithm is very complex, and, if file accesses and interprocess communication are frequent, it is also quite inefficient.

The third approach—the one advocated in this paper—integrates application level and file system level fault tolerance into a single mechanism, which relies only on local disks as the storage medium. Figure 2(c) illustrates the basic organization of the local-disk based file system, LDFS. Similar to the approach in Figure 2(b), the read/write requests to the disks are not logged, but are regenerated from application messages. However, the use of local disks makes this approach much more efficient and the resulting protocols much simpler. The main reason is that every process and all files accessed from it are isolated on the same physical machine. Thus the failure of the application process and the failure of its local file system always happen together. Consequently, they are also recovered together, using a single protocol: The state of the failed

node is restored from a checkpoint, which also includes all its files. The logged application messages are then replayed, which brings the state of the files up to date automatically.

The LDFS approach also avoids the problem of the network partitioning at the file system level: a file can become inaccessible only if the node where the file resides is down. Since the file can be accessed from only that node, it can never be inaccessible.

To make the LDFS fault tolerant, all files (including checkpoint and application files) must be replicated. Otherwise, a node failure would make the information saved on its local disk unavailable, and the system would not be able to recover.

```
{ // main loop in daemon
  ........................
  if (time to checkpoint)
     checkpoint
     replicate
  if (file system failure)
     recover
     regenerate
  ........................
}
```

Figure 3. Additions to daemon

Every node in the system runs a daemon process, which interprets agents' behavior. Daemons are also responsible for system's fault tolerance, as shown in Figure 3. The additions to the basic rollback-recovery mechanism that make the LDFS fault tolerant are typed in bold; these are discussed in detail in the following sections.

3 Replication of Checkpoints

File replication techniques have been studied extensively [16-19]. However, because we integrate file system fault tolerance with application fault tolerance, we are concerned with file replication only at the time of checkpointing. Furthermore, replica consistency is preserved automatically, since the replicas change only at the time of checkpointing.

3.1 Including Application Files in Checkpoints

The state of the distributed system consists of the state of its processes, the state of the communication channels, and the state of the application files. Here we only discuss how the state of application files can be included in the checkpoint. This can be accomplished by replacing the existing commands for opening and closing the files with extended versions that maintain information about the open file in a data structure accessible by the checkpointing system. In Unix, this structure is created with the call to fopen(), and it consists of the file name, file operation mode, a file pointer, file pointer name (FILE * variable name), and the current position in the file. The first two entries are supplied as parameters to fopen(). The current file position is set during checkpointing using the ftell() system call. The current file pointer is obtained using the system function getFP(filePointerName). The application calls this function whenever the file pointer is

used. This guarantees that even after the checkpoint is started on another file system and the file is reopened, the application will be using the correct file pointer.

3.2 Ring Replication

All file systems that could potentially support system nodes are given a unique file system id (FSId). The FSId is determined by the location of the file system in the configuration file. The system configuration is described in more detail in [10].

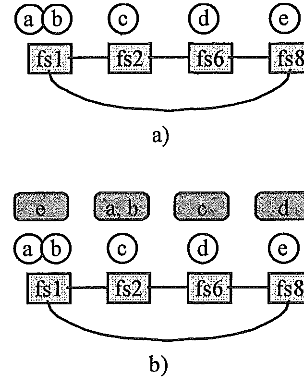


Figure 4. Checkpoint logging

File systems hosting active processes are arranged in a logical ring according to their FSId's; i.e., file system *i* is connected to its successor, *i*+1, and its predecessor, *i*-1, modulo the total number of file systems. We will refer to the successor of a file system as NextFS and to the predecessor as PrevFS. A ring with four file systems is illustrated in Figure 4(a). The rectangles represent file systems, and the circles above represent daemons running on machines with these file systems.

At each checkpoint, the daemon saves its state to its local file system. The daemon also sends a copy of its checkpoint file to one of the daemons residing on its neighbor NextFS. As a result, each checkpoint file has two copies: one on the local file system and the other on its neighbor's file system, as shown in Figure 4(b). The letters in the top rectangles represent daemon checkpoints logged on the file system. We will refer to the file system where the checkpoint copy is saved as LogToFS, and the file system whose checkpoints are copied locally as LoggedFS.

The application file replication depends on the file operation mode. Files open only for reading, have to be replicated only during the first checkpoint, or after the failure of one of the supporting file systems. In the case of write-only files, in which all writes append data to the end of the file, the increment written since the previous checkpoint is appended to the end of the replica file. In the case of read/write files, in which writes are allowed to modify entire file, the entire file is replicated at every checkpoint.

4 Recovery from Checkpoints

4.1 Reconnecting the Ring

Each daemon knows the IP address, the communication port, and the file system of every other daemon running in the system. Using this information, each daemon can determine its NextFS and PrevFS. At the beginning of each checkpointing phase each daemon computes its LogToFS, which equals to NextFS, and its LoggedFS, which equals to PrevFS.

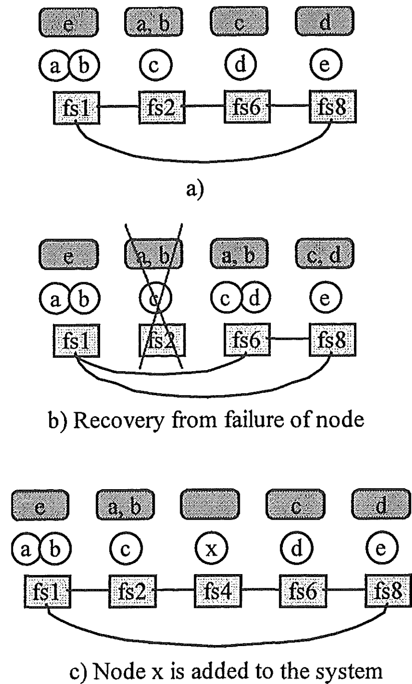


Figure 5. Failure

After a file system failure, each daemon checks, whether its PrevFS or NextFS have changed. If so, new PrevFS and NextFS are set using the current list of file systems. This reconnects the ring.

Each daemon also checks whether its LogToFS and LoggedFS have changed. If the node holding LogToFS failed then the daemon copies the LogToFS from the newly computed NextFS; if the node corresponding to LoggedFS failed, then the daemon copies the new LoggedFS from PrevFS. For example, after the failure of fs2 (Figure 5(b)), the LogToFS of daemons on fs1 becomes fs6, and LoggedFS of daemons on fs6 becomes fs1.

Node failure is the only situation when LogToFS and LoggedFS need to be changed; prevFS and a NextFS, on the other hand, also change on other occasions. For example, if a new daemon x on fs4 joins the system (Figure 5(c)), then nextFS for fs2 is fs4, prevFS for fs6 is fs4, but LogToFS for fs2 remains fs6, and LoggedFS for fs6 remains fs2.

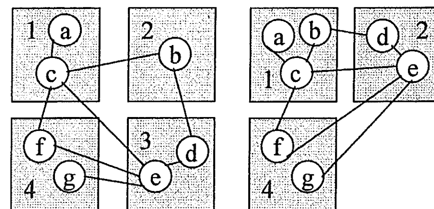


Figure 6. Redistribution of nodes after node 3 failure

4.2 Loading the State of the Failed Node

There are several ways the state of a failed node can be restored. In the simplest case, the failed processes are simply restarted on another machine. This was shown in Figure 5(b), where process c was restarted on the same node as d. In this case, each process is responsible for regenerating its own files that were lost together with the local file system.

In some mobile agent systems [14] the computation takes place in a logical, rather than a physical network. The computation is a collection self-migrating agents roaming the logical

network, where the logical nodes are mapped onto different physical nodes. Logical nodes can be migrated between machines at run time. This is used for load balancing, but it can also be used during recovery from failure. Instead of waiting for a failed machine to recover, its state can be distributed to other nodes by restoring all its logical nodes on other machines [15]. Figure 6 illustrates the failure of a node, 3. The logical nodes e and d of a failed node are restored on another machine and the logical network is remapped to balance the load.

Note that after the failure of some daemon, the correspondence between the live processes and checkpoint files is no longer one to one, as illustrated in figure 7. Figure 7(a) shows the initial configuration before the failure. Here, in the top rectangles, the letters outside of the brackets represent the primary checkpoint files of the local daemons. In the brackets we list replicas of checkpoints stored there by neighboring daemons. Figure 7(b) shows the situation after failure of fs2. Daemon c is loaded on fs6 and merged with d. As a result, there are now only four daemons running but there are five separate checkpoints that would have to be restored should another failure occurs before the next checkpoint is taken. After the checkpoint, the one-to-one correspondence is again restored, as shown in Figure 7(c).

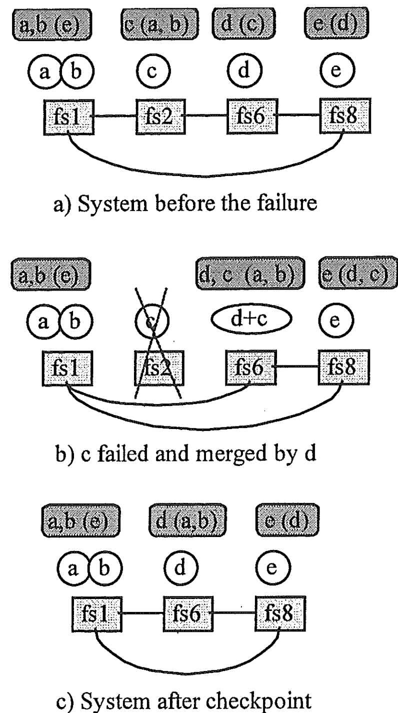


Figure 7. Logging with daemon merging

To handle multiple daemon failures that happen in a single checkpoint interval, active daemons keep track of the failed daemons. The list of the failed daemons and their primary file systems are kept on every machine. This list is cleared with the start of every new checkpoint.

We now present the regeneration algorithm for the systems that are able to merge the failed process with the running one, since this is the more complex case.

4.3 Regeneration Algorithm

In the case of a node failure, the failed replicas have to be regenerated [20, 21]. The proposed algorithm proceeds in three steps:

1. Detect file system failure
2. Determine which daemon should load the failed daemon state.

3. Regenerate all the replicas lost with the failed file system.

Step 1: Detecting file system failures

When a daemon failure occurs, live daemons first need to determine whether it is also a file system failure. We say that the file system failed if its contents become unreachable. Each daemon knows the IP address, the communication port, and the file system of every other daemon running in the system. This information allows each daemon to deduce whether the file system that supported the failed daemon is reachable or not: if the failed daemon was the last daemon on the file system, then the file system is unreachable and is considered as failed.

Step 2: Determining what daemon should load the failed daemon state

The failed daemon is loaded by the coordinator of the file system whose LoggedFS equals failed daemon's primary file system. The primary file system of the failed daemon is updated autonomously by all daemons to the file system where it is loaded.

Step 3: Regenerating replicas lost with the failed file system.

There are two types of files that are lost with the loss of the file system: the checkpoints associated with daemons currently running on the failed file system, and the replicas of all daemons whose LogToFS was the failed file system. Each daemon is responsible for restoring the missing copies of its files. For example, after the failure of fs2 in Figure 7(b), daemons a and b regenerate their checkpoints on fs6. The daemon that loaded the state of the failed daemon is responsible for regeneration of the lost files of the failed daemon. In Figure 7(b), d logs the checkpoint of c to fs8. Every daemon loads its state from its primary file system (PrimFS), except the daemons that were started after the last checkpoint: these don't load their state, but simply reinitialize the daemon.

4.4 An Illustrative Example

Table 1 illustrates how this algorithm works using an example. At the beginning, the system consists of 6 nodes (a-f) residing on five file systems (FS1 – FS5). When a new computer joins the system, the load balancer redistributes the load. At each step in the computation the changes to the system are emphasized in bold.

First, the six machines a-f join the system. *Step 1*: a system checkpoint is taken. The table shows which processes are run and which are saved on each file system. *Step 2*: d fails. FS4 becomes LogToFS for FS2. d is merged with e. Its checkpoint is logged to FS5. *Step 3*: a fails. Its checkpoint is merged with b. No logging is required. *Step 4*: d is restarted. No logging is required.

Step 5: c fails. c is merged with e and logged to FS5. Daemons on FS1 change their LogToFS to FS3. a and b are logged on FS3. *Step 6*: new checkpoint is taken.

#	Event	FS 1		FS 2		FS 3		FS 4		FS 5	
		Run	Log	Run	Log	Run	Log	Run	Log	Run	Log
0	Daemons a, b, c, d, e, and f are started										
1	CP 1	**a,b**	**a,b,f**	**c**	**c,a,b**	**d**	**c,d**	**e**	**d,e**	**f**	**f,d**
2	d fails	a,b	a,b,f	c	c,a,b	-	-	e	**c,d,e**	f	**f,d,e**
3	a fails	**b**	a,b,f	c	c,a,b	-	-	e	c,d,e	f	f,d,e
4	d is up	b	a,b,f	c	c,a,b	**d**	-	e	c,d,e	f	fde
5	c fails	b	a,b,f	-	-	d	**a,b**	e	c,d,e	f	**f,d,e,c**
6	CP 2	b	**b,f**	-	-	d	**b,d**	e	**d,e**	f	**e,f**

Table 1. Example of the system run.

4.5 Pseudocode

Figure 8 shows the pseudocode for the recovery algorithm. In case of failure, the daemon loads its own checkpoint (lines 3-4). Neighborhood coordinators load the checkpoints of the failed daemons (lines 6-8). The neighborhood coordinator is the daemon with the smallest IP number on the given file system. Each daemon can independently decide on the neighborhood coordinator using locally available system information. Note that the newly added, or previously failed daemon could be a neighborhood coordinator. Several times in this algorithm (lines 10 – 26) we determine whether some file system has failed. The LoggedFS and primFS of the failed daemons are updated at the last step of the algorithm to allow tolerating multiple daemon failures.

4.6 Correctness of the Recovery Mechanism

In this section we briefly sketch a correctness argument.

Definition 1

The system is *k-recoverable* (R^k) if the system can recover from any *k* node failures.

Definition 2

The replica is *reachable* iff it can be demanded by one of the running daemons during the recovery*. In our case the replica is reachable if it is saved on the PrimFS of the daemon whose state it represents, or if it is saved on the file system where LoggedFS equals daemon's PrimFS. When the file system fails, daemon's state is loaded by the coordinator of the LogToFS. The other daemons load their state from their primFS.

```
1    Add failed daemon(s) to the failed list.
2
3    If during this checkpoint interval I did not fail, and was not added to the system,
4    then load my checkpoint.
5
6    If I am a neighborhood coordinator,
7    then load checkpoints of failed daemons listed in the failed
8                       list with the same primary file system as self.
9
10   If my LogToFS failed,
11   then set my LogToFS to nextFS, and log all files I just loaded to the new LogToFS.
12
13   If my LoggedFS failed, and I am a neighborhood coordinator
14   then
15   {
16     Load checkpoints of the failed daemons, which had my LoggedFS as their
17                              primary file system.
18     Log these files to my LogToFS.
19   }
20
21   If there was a file system failure
22   then  Determine new primary file system for the failed daemons and update it in
23                              the failed list.
24
25   If my LoggedFS failed,
26   then set my LoggedFS to prevFS
```

Figure 8. Extension to recovery algorithm

<u>*Definition 3*</u>

The system is k-recoverable iff

- Rollback-Recovery mechanism works on the shared file system.
- After the failure recovery every data file has k+1 reachable replicas.

Correctness of the recovery scheme follows from the following two claims.

- The system is 1-recoverable after the first checkpoint.

* It is implied that there could be only one replica on every file system, since there could be only one file with a given name per file system.

- The system is 1-recoverable after the failure recovery.

1. **The system is 1-recoverable after the checkpoint** because the file systems are arranged in a ring and each daemon logs its files to the next file system in the ring. LoggedFS and LogToFS values are set. LoggedFS and LogToFS can only be changed at the next checkpoint, which would set them again, or after file system failure, which is covered in the second bullet.
2. **The system is 1-recoverable after failure recovery** because
   - The Rollback-Recovery mechanism works as was shown in [15]
   - The LogToFS-LoggedFS ring is reestablished (lines 10-11, 25-26 in Figure 8)
   - There are two reachable replicas of the state of the failed daemon; one is on the primFS and one is on the LogToFS (lines 16-18 in Figure 8).

5 Performance

Table 2 shows measurements of different file operations. All results are in microseconds. Depending on the operations performed on files, the local disk based system (LD in the table) is comparable or potentially much faster than NFS [4]. In particular, opening and closing a file is 41.2 times faster, and writing to the file is 14 times faster on LDBS.

Buffer Size (bytes)	Open-Close		Read		Write		Op-Read-Close		Op-Write-Close	
	NFS	LD	NFS	LD	NFS	LD	NFS	LD	NFS	LD
0	3.71	0.09	-	-	-	-	-	-	-	-
100	-	-	0.00	0.00	0.20	0.04	0.20	1.67	30.16	0.53
1,000	-	-	0.04	0.03	1.02	0.07	1.69	0.22	30.27	0.51
10,000	-	-	0.42	0.41	9.50	0.68	2.55	0.53	45.40	1.35
100,000	-	-	4.80	4.50	90.40	7.20	6.90	4.00	180.80	9.70

Table 2. Comparison of standard file operations on NFS and local disk (LD).

We also measured the checkpoint overhead induced by both saving the checkpoint to the NFS and to the local disk. The LDFS was integrated with the MESSENGERS mobile agent system [14]. In our experiment the logical nodes are connected in a logical ring, as shown in the Figure 9. There is one agent (called Messenger) that continuously travels around the ring. When it completes 1000 rounds, the application terminates. There are no other Messengers in the system. (Otherwise the size of the checkpoint would vary.)

The application was run without checkpointing, with checkpointing on NFS, and with checkpointing on LDFS. The checkpoints were taken every 30 seconds. The total execution times and the numbers of checkpoints taken were recorded. By subtracting the time the application took to complete without checkpoints from the execution time of the application with checkpoints, and dividing this by the number of checkpoints, we derived the time overhead of a single checkpoint. We measured the results for systems consisting of different numbers of daemons. The number of logical nodes per daemon was held constant, and the checkpoint size was always 1Mb. The results of the experiments are shown in table 3.

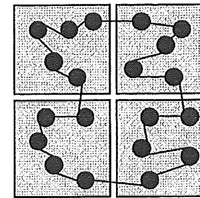


Figure 9. Logical network

We repeated this experiment on a set of machines connected through the Ethernet and a set of machines connected through a collision-free switch. The difference in checkpoint overhead was negligible.

# of daemons	Single checkpoint overhead (sec) taken on:	
	NFS	LDFS
2	0.636	1.000
4	1.000	1.214
6	1.500	1.300
8	2.250	1.383

Table 3. Checkpointing overhead for the checkpoint size 1Mb.

# of daemons	Single checkpoint overhead (sec) taken on:		
	NFS	LDFS on Ethernet	LDFS on switch
8	13.905	89.305	9.059

Table 4. Checkpointing overhead for the checkpoint size 10Mb.

6. Final Remarks

Using local disks as secondary storage for files can significantly speed up distributed applications, both in terms of the file accesses and when saving the processes' checkpoints. The integration of the local-disk based file system into a fault tolerant distributed system allows a simple and efficient implementation of the file system fault tolerance, which avoids logging of individual read/write accesses to files and eliminates the need for replica consistency protocols. The fault tolerance of the file system is provided automatically by the rollback-recovery protocol of the distributed application, and by replicating the checkpoint files on neighboring machines.

LDFS has been implemented as an application process and thus does not require any kernel modifications. The comparison of checkpointing time depends on many factors, including the size of the checkpoints, the speed, load and configuration of the underlying network, and a load on the file server. Our experiments show that LDFS is a better choice for applications with checkpoint sizes of 1Mb or less even on a slow network, where the distributed nature of LDFS is not fully utilized. On fast networks, LDFS is a better choice than NFS even when saving larger checkpoint files.

The current implementation of LDFS can tolerate a failure of one component file system at the time. This number could be increased by increasing the number of independent replicas. This could be accomplished by extending the simple ring structure into a structure with greater connectivity among nodes, such as a 2D or 3D mesh, or a group.

References

[1] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *In Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109--126, Copper Mountain Resort, Colorado, December 1995. ACM.

[2] T.Cortes and J.Labarta. PAFS: a new generation in cooperative caching. *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages I:267-274. CSREA Press, July 1998.

[3] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of the 13th Symposium on Operating System Principles*, pages 1--15, October 1991.

[4] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. *In Proc. of the Summer 1985 USENIX*, pages 119-130, June 1985

[5] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Trans. on Computer Systems*, 6(1), February 1988.

[6] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. *In Proc. of the 13th Symp. on Operating Systems Principles*, pages 226-238, October 1991.

[7] A. Birrel, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo Distributed File System. *Technical Report 111, Digital Equipment Corp. Systems Research Center*, September 1993.

[8] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Array of Inexpensive Disks (RAID). *In International Conference on Management of Data*, pages 109-116, June 1988.

[9] Nadine Peyrouze and Gilles Miller. FT-NFS: An Efficient Fault-Tolerant NFS Server Designed for Off-the-Shelf Workstations. *Symposium on Fault-Tolerant Computing*, pages 64-73, 1996.

[10] Eugene Gendelman, Lubomir F. Bic, Michael B. Dillencourt. Fast File Access for Fast Agents, *TR #00-39, 2000* http://www.ics.uci.edu/~egendelm/prof/publications.html

[11]Yong Deng and E. K. Park. Checkpointing and Rollback-Recovery Algorithms in Distributed Systems. *Journal of Systems and Software*, 25:59-71, 1994

[12]E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Trans. on Computers Special Issue On Fault Tolerant Computing*, 41(5):526-531, May 1992

[13] G. Popek, R. Guy, T. Page, and J. Heidemann. Replication in the Ficus Distributed File System. *In Proc. of the Workshop on the Management of Replicated Data*, pages 5--10, November 1990.

[14] C. Wicke. Implementation of an Autonomous Agents System. *Diploma thesis*, University Karlsruhe, 1998

[15] Eugene Gendelman, Lubomir F. Bic, Michael B. Dillenourt, An Application-Transparent, Platform- Independent Approach to Rollback-Recovery for Mobile-Agent Systems. *In proc. of 20th International Conference on Distributed Computing Systems (ICDCS 2000).* Taipei, Taiwan, April 2000.

[16] P.A. Alsberg and J.D. Day. *A principle for resilient sharing of distributed resources. In Proceedings of the Second International Conference on Software Engineering*, pages 627--644, October 1976

[17] Reed, D.P., and Svobodova, L.: SWALLOW: A distributed data storage system for a local network. In West, A., and Janson, P., ed. Local Networks for Computer Communications, *Proc. IFIP Working Group 6.4 International Workshop on Local Networks*. North-Holland, Amsterdam, 1981, pp.355-373.

[18] O. Wolfson, S. Jajodia, Y. Huang, An Adaptive Data Replication Algorithm. *ACM Transactions on Database Systems (TODS),* Vol. 22(4), pp. 255-314, June 1997

[19] R. Ladin, B. Liskov, L. Shrira, S. Ghemawat: Providing High Availability Using Lazy Replication. *ACM transactions on Computer Systems*, Vol. 10, No. 4, November 1992, Pages 360-391.

[20] C. Pu, J. D. Noe, and A. Proudfoot. Regeneration of Replicated Objects: A technique and its Eden implementation. *IEEE Transactions on Software Engineering*, 14(7):936-945, July 1988

[21] D. E. Long and J. L. Carroll. The Reliability of Regeneration-Based Replica Control Protocols. *In 9th IEEE Conference on Distributed Computing Systems*, pages 465-473, 1989.