Server Reintegration in a Replicated UNIX File System

Marcia Pasin   *Taisy Silva Weber*

Universidade Federal do Rio Grande do Sul
Instituto de Informtica
PPGC - Programa de Ps-Graduao em Computao
Caixa Postal 15064 CEP 91501-970 Brazil
{pasin, taisy}@inf.ufrgs.br

Abstract
A distributed system is a collection of nodes connected by a network, an ideal platform to provide high reliable computing due to the redundancy supplied by a great number of nodes. Node faults and network connection faults can be masked reconfiguring the system. However, sequential faults, that affect multiple nodes can decrease the performance of the system affecting the system reliability and availability. To avoid this, failed nodes should be reintegrated as soon as possible. This paper details the problem of reintegration of failed nodes in a replicated UNIX file system. We built a prototype with the recovery protocols required by the reintegration procedure.

Keywords: distributed systems, file replication, reintegration of failed server, UNIX systems.

1 Introduction

Distributed systems are composed of nodes connected by a high-speed network. At these nodes, we can found server processes that offer services to the other processes called clients. Servers provide the clients file service. The client sends to the server a request message asking for some service (e.g. read a block of a file). The server process does the work and returns the result or an error code to the client. The files can be replicated at different servers to provide a fault tolerant and to improve performance by supporting parallel reads on different replicas. It composes a distributed replicated system. If a node with a server process fails, there is another to replace it.

A problem rises when occur sequential faults in nodes with server processes. Sequential faults not only decrease the performance, but also affect the system availability. Failed servers must rejoin the system as soon as possible after repair to maintain the desired availability. Generally, a server reintegration is a manual task, but it leads to operation errors and unacceptable delays. Automatic techniques could provide fast and robust reintegration assuring the required system availability. The server reintegration:

a) maintains the full activity of the system correcting degradation generated by failures;

b) maintains the number of servers of the original system configuration and

c) increases the availability of information supplied to the clients, with the addition of a new server.

Reintegration doesn't just involve the recognition of new server in the network, but also the state update of the new server. That state depends on the environment and on the supplied service. The state update involves the file transference from an up-to-date server to the new server. This paper details the automatic reintegration of servers and discusses two update state protocols. The file replication approach can be used with primary site [1] or active replication [2] as will be shown. We are considering a replicated UNIX file system [3, 4].

The remaining of the paper is structured as follows: the section 2 introduces the file replication approaches used in distributed computing and masking failures; the section 3 describes the distributed replicated system model. The section 4 describes the up-to-date protocols. The section 5 describes the client service and the server reintegration concurrently. The section 6 shows the recovery protocols used by the reintegration procedure. The section 7 shows some results. Finally, the section 8 finishes the paper with our concluding remarks.

2 File replication approaches to masking failures

File replication disseminates file replicas among special nodes with server processes. If one replica is accidentally lost, there is other to replace it. The client processes can read and write this file replica accessing a server process. They use message passing protocols. If a client wants to read a file at a server, it sends to the server a message with the identification to the desired file. The server does the work and returns the result to the client using another message. The client does not know anything about file replicas: the replication is fully transparent. The client only knows the file name and asks the server about the file. It just makes the request to the desired file and waits for the result.

The servers that have the same file replicas compose a replication group. When a server in the replication group crashes, the other operational servers must ensure the integrity of the files continuing the service to the clients. A special approach is required to management the distributed replicas in the group and to provide failure transparency. It can be done by primary site [1] and active replication [2] protocols.

These approaches have been used in a variety of systems [3, 4 and 5]. The primary site approach uses a server called primary site and backup servers. The primary site manages the principal replica. A client only knows who is the primary site because the file replicas are transparent. When the primary receives a write message from the client, it disseminates to the backups the invocation before that it returns the response to the client. If the client sends to the primary a read message, it reads the required file and returns the client the response.

The active replication approach gives all replicas the same hole without the centralized control of the primary site. When the client sends to any server a write message, all replicas receive the message. Each replica does the work and returns the result or an error code to the client. The client waits until it receives the first result or the majority of identical results. If the operation is a read, it makes a similar operation.

Active replication and primary site approaches require non-crashed replicas receive the client message in the same order. They require a communication primitive that satisfies the proprieties [6] below:

a) order: all the file replicas must process different write messages from concurrent clients at the same order;

b) atomicity: all the file replicas must process a write message or no one can do it. If the write message can not be processed, the write operation is aborted.

However, the atomicity propriety can be relaxed to implement resiliency: if a replica crashes and does not process the write message, it will leave the replication group. The operation can be completed successfully even if some failures occur in the system. That is, the operation must be resilient to failures.

When a failure is detected, the failed server must be isolated, repaired and reintegrated into the group to not decrease the system reliability and availability. At this paper, only server failures are handled. Server failures are a lot more disruptive and disturb the system service.

There are two types of failures that need to be handled by a fault-tolerant system: node failures and communication failures. Node failures cause replica on that node to become inaccessible. The node stops its processing or crashes. A communication failure is a lot more disruptive and can lead to network partitioning. In this, nodes and links fail in a manner such that the remaining nodes are partitioned into sub-groups. Nodes in each sub-group can communicate with each other, but can not communicate with nodes of the other sub-groups.

3 The distributed replicated system

We assumed the system normal operation state (fig. 1) when the system is free of faults. When a faulty server is detected (for example by timeout), the failed server must be isolated, and its service is no more available. The system lost a server and needs mechanisms to recover the service, reconfiguring the replication group. The replication group could request the system administrator to replace or repair the lost server.

After system recovery and group reconfiguration, the system works degraded, because the fault tolerance capacity was reduced. The system lost a server. A robust system supports a determinate number of sequential faults before collapse. This number is associated with the number of file replicas in the replication group.

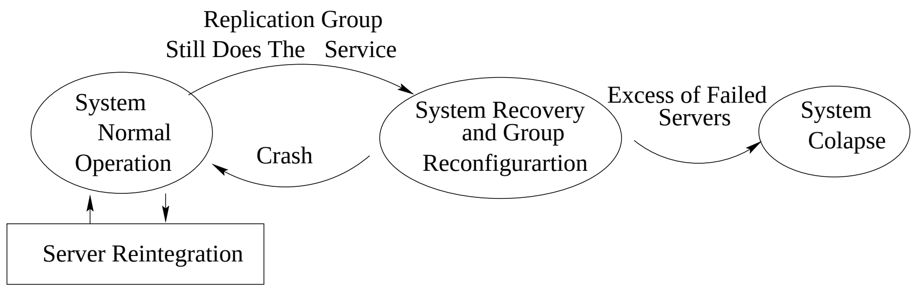

Fig. 1: Reliable replicated system procedures

The system must provide reintegration of failed servers to increase or maintain fault tolerance and performance. It can be done using server reintegration at the system normal operation state. The server reintegration begins when a new or repaired server restarts and returns to the network. Reintegration means server files recovery and server join into the replication group. The new server sends to the replication group messages to get the up-to-date file version and to recover its replicas. When the recovery terminates, the group is notified that the server is recovered and can participate as an active group member.

A server can be a member of more than one replication group, that means, a server can maintain file replicas of different replication groups. When such server must be reintegrated, the server must rejoin all the replication groups where it was a member. The reintegration in more than one group can be done concurrently.

4 Server reintegration and up-to-date protocols

Some replicated UNIX systems [3, 4, 5, 7] try to provide high reliability and availability maintaining the NFS (Network File System) [8] conventional semantic. However, systems as HA-NFS (High Availability NFS) [4] and the Harp [5] do not provide mechanisms to reintegrate repaired servers. In these systems, maintaining the number of servers in a replication group is a manual task, executed by system operator. Automatic reintegration is rarely provided or documented. An exception is the RNFS (Reliable NFS) [3].

In the RNFS, the automatic reintegration of failed server starts when the server is repaired and sends to the replication group a message requiring a join. The primary site in the group answers the invocation and starts the recovery protocol. The repaired server must collect information of the replication group to recover its files. When the recovery process terminates, the replication group must be notified. We developed two server recovery protocols for the RNFS: volume recovery differential recovery. These protocols are detailed at this paper.

In the volume recovery, the primary site starts a task that recursively spans the local image of the file volume under recovery. This volume contains all replication group files, which the new server asks to rejoin. All files and directories of this volume on the primary site are transferred to the new server. If the new server contains files of that replication volume that no longer exists on the primary site, these files are deleted. The result of this operation is a complete image transfer of the entire file volume. If the recovery process fails for any reason, it is stopped and the recovery will start again.

The differential recovery is more restrict than the volume recovery. When a server restarts after a failure, version numbers at all files will signalize at least the last completed operation on this files. During the recovery procedure, file version numbers are checked between the primary site and the new server in the group. Only files with different version numbers are copied. The wrapping of counters is handled by defining a maximal possible version number. This number also means an invalid version number. When file versions reach the maximum version number, version updating is stopped. During the next recovery round, all files having invalid version number will always be transferred, and the version numbers will return to zero on all connected copies.

The differential recovery is used when the failed server was just a short interval of time out of operation and the number of file updates in this period was not significant.

Recovery protocols can be used concurrently in the repaired server during reintegration to different replication groups (fig. 2). The server runs a process executing one of the protocols to each replication group that it wants to rejoin. To some groups could be interesting to transfer the complete replication volumes, to other to use the differential recovery protocol.

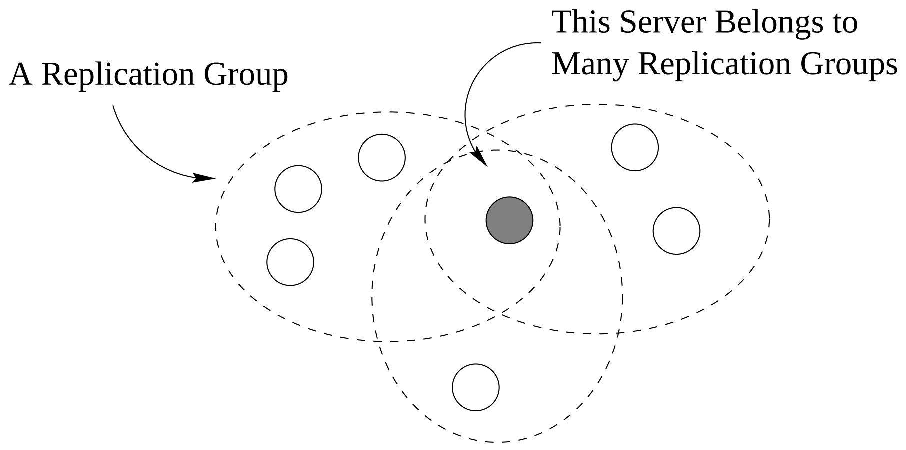

Fig. 2: Many replication groups can reintegrate a server concurrently

A performance function, that considers the number of files in the replication volume, the time elapsed from server failure to reintegration and the file update load of the group, decides the best strategy to each replication group.

5 Reintegration of server and the client service

While the replication group reintegrates the new server, the clients continue to ask services. In a high availability system, the replication group must be able to attend the client service and allow the reintegration of a server. The new server works like a client asking the replication group for many services.

The more straightforward solution to recovery a server, disables completely the service to the clients. It is not efficient because it decreases the system availability. A more efficient solution disables the write operations to the replication group files while the recovery does not terminate. This solution also decreases the system availability concerning update requests from the clients, but it continues to provide read service. The third solution locks write operations at the replication group and release the locks gradually. When the reintegration starts, all files of the group are locked to write operations. As files are restored in the new server, the locks are released. Thus, when a file is locked, it means that the recovery protocol did not yet transfer the file. When the lock is released, all replicas of that file can accept update requests in all servers of the group, including in the new server.

6 Recovery protocols

A prototype for the suggested recovery protocol was implemented using a toolkit called rpcgen [9], witch allows to build distributed application using RPC (Remote Procedure Call) [10].

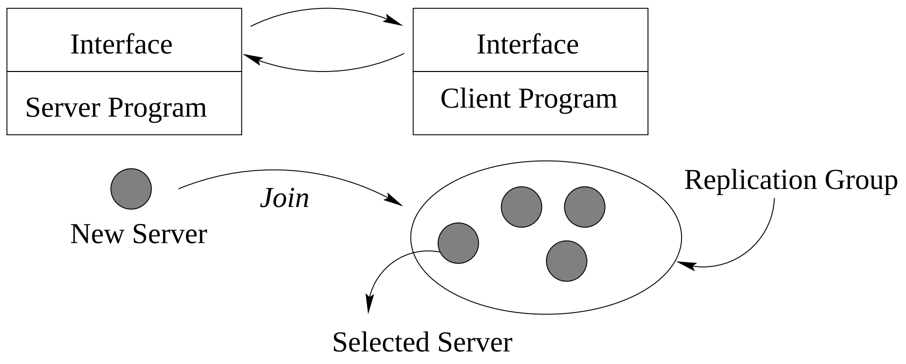

Fig. 3: Server recovery in a distributed system

First, we wrote an interface to be compiled by the rpcgen, a program with the server procedures to the new server and a program with client procedures to a selected server (fig. 3). The interface is used to do the communication between the server and client programs. A conventional C program language compiler generates the executable code and links them with a interface. The generated codes are executed in a UNIX environment, which can be SunOS or Linux. We assumed a distributed system with point-to-point reliable communication.

Our prototype implements the volume recover and the differential recovery. The volume recovery uses only the RPCs available at the conventional NFS. To implement the differential recovery protocol it was necessary to include a version file in each server (primary and backups) and a new RPC in the file system to manipulate this information.

The version file contains the file name and a version number associated to each file name. Before recovering a file in the new server, the file version number of this file at the primary site is checked. If the file version in the primary site is greater than the file version in the new server, then the file is replaced with the primary version. If both file versions are equal, then both files are assumed to be equal and the file at the new server do not need be replaced.

The prototype implements the volume recovery protocol and the differential recovery protocol. The volume recovery uses only the RPCs available at the conventional NFS (fig. 4).

```
volume_recovery (server.name, client.name) {
        fetch the file attributes - stat
        if file type is directory {
                create directory - MKDIR  RPC
                open directory - opendir
                read file of directory - readdir
                while exist files in directory {
                        volume_recovery (server.name, client.name)
                        read a file in directory - readdir
                }
                close directory - closedir
        } else {
                open file - open
                read data of file - read
                while exist data {
                        read data of file - read
                }
                close file - close
                write a file - RPC  CREATE
        }
}
```

Figure 4 - Volume recovery executed by client program (selected server)

We included a version file in both servers (new and selected) and a new RPC to manipulate this information to implement the differential recovery protocol (fig. 5).

```
differential_recovery (server.name, client.name) {
        fetch the file attributes - stat
        if file type is directory {
                create a directory - MKDIR  RPC
                open directory - opendir
                read a line of directory - readdir
                while exist files in directory {
                  differential_recovery (server.name, client.name)
                  read file of directory - readdir
                }
                close directory - closedir
        } else {
                return_version (client.name)
                return_version (server.name) - new RPC
                if server.version < client.version {
                        open file - open
                        read data of file - read
                        while exist data        {
                                read data of file - read
                        }
                        close file - close
                        write the file - RPC  WRITE
                }
        }
}
```

Figure 5 - Differential recovery executed by client program (selected server)

A group-synchronized clock could be used to avoid the additional RPC. It manipulates the file version number. In this solution, the time given by the global clock for each file update can be used as version number. This information can be found in the file inode [8]. Unfortunately, distributed systems with global synchronized clocks provided through clocks synchronization algorithms are not frequently found in real world.

The volume recovery and the differential recovery protocols proposed to an environment with primary site also can be used with active replication. An appropriate server must be selected in the active replication group. This server will be work like a primary site, recovering a failed backup server. In the active replication approach all servers in the replication group coordinate their activities to service the clients and no one acts as a primary during the normal operation and any one of them can act as a primary during the server reintegration.

7 Results

The recovery protocols use the RPCs of the conventional NFS, and a new RPC to return the file version using in the differential recovery protocol. We executed some tests with the prototype to verify our protocols. We are using a small data base. The table (fig. 6) built after the prototype execution shows that, to the same file system, the differential recovery protocol is more efficient than the volume recovery protocol.

Total File Number	Volume Recovery	Differential Recovery
no one	6.78 sec	3.86 sec
1/2 files	7.92 sec	6.90 sec
all files	7.20 sec	10.46 sec

Figure 6 - Volume recovery protocol versus differential recovery protocol

An exception occurs when the server needs to recover all files at the volume or most of them. In this case, the time computed by the differential recovery includes the volume recovery and all the comparisons needed to verify that the whole volume is out-of-date. The graphic values are omitted because we make a quality comparison. We are not interested in values because we are executing just a prototype to validate our algorithms (volume recovery and differential recovery).

8 Conclusions

A prototype was implemented to analyze the automatic reintegration of failed server in a UNIX replicated file system. The protocols use the RPCs of the conventional NFS, and a new RPC to return the file version using in the differential recovery protocol. If the system supports a group synchronization algorithm, a solution could be easily implemented to eliminate this additional RPC and overcome the disadvantage of incompatibility with the NFS. The problem of this solution is the overhead make by the clock synchronization algorithm.

The prototype shows that, to the same file system, the differential recovery protocol is more efficient than the volume recovery protocol, except when the server needs to recover all files or most of them. In this case, the time computed by the differential recovery protocol includes the volume recovery protocol and all the necessary comparisons to verify that the whole volume is out-of-date.

Now we are porting the recovery protocols to an environment with support to group communication primitives [11]. We intend implement the server reintegration and a replication approach to keep the consistent replicas to this environment using these primitives. The recovery protocols also can be used in a migration system [12] to do the state transference.

References

[1] Budhiraja, N.; Marzullo, K.; Schneider, F. B.; Toueg, S. The primary-backup approach. In: Mullender, Sape (Ed.). Distributed Systems. 2 ed. New York:

ACM Press, 1993. p.199-216.

[2] Schneider, F. B. Replication management using the state machine approach. In: Mullender, Sape (Ed.). Distributed Systems. 2. ed., NewYork: ACM Press, 1993. p. 169-198.

[3] Leboute, M.; Weber, T. S. A reliable distributed file system for UNIX based on NFS, In: Proceedings of the IFIP International Workshop on Dependable Computing and its Applications. DCIA 98, Johannesburg, South Africa, January 12-14, 1998. p. 158-168.

[4] Bhide, A.; Elnozahy, E. N.; Morgan, S. P. A highly available network file server. Proceedings of the USENIX, 1991. p.199-205.

[5] Liskov, Barbara et al. A Replicated UNIX File System. Operating Systems Review, New York, v.25, n.1, p.60-64, Jan. 1991.

[6] Guerraoui, Rachid, Schiper, Andr. Software-based replication for fault tolerance. Computer, p.68-74, Apr. 1997.

[7] Siegel, A.; Birman, K.; Marzullo, K. Deceit: a flexible distributed file system. Tech. Rep. 89-1042, Department of Computer Science, Cornell University, 1989.

[8] Sandberg, R.; Goldemberg, D.; Kleiman, S.; Walsh, D.; Lyon, B. Design and implementation of the Sun Network File System. Proceedings of the Summer USENIX Conference, 1985. p.119-130.

[9] Birrell, A. D.; Nelson, B. J. Implementing remote procedure calls. ACM Transactions on Computer Systems, No. 2, pp.39-59. 1984.

[10] Corbin, J. R. The art of distributed applications: programming techniques for remote procedure calls. Sun Technical Reference Library, Sun Microsystems, 1991. 321p.

[11] Hayden, Mark G. The Ensemble System. A dissertation presented to the Faculty of the Graduate School of Cornell University, Jan. 1998.

[12] Pasin, M.; Weber, T.S.; Jatene, B.; Geiss, L. Super-Amigos: um sistema tolerante a falhas para migrao de objetos em sistemas distribudos. Memorias CLEI, 1999. Asuncion, Paraguay.