Servers Reintegration in Disconnection-Resilient File Systems for Mobile Clients

Azzedine Boukerche Raed Al-Shaikh
PARADISE Research Laboratory
SITE, University of Ottawa
Email:{boukerch, rshaikh}@site.uottawa.ca

Abstract

Servers' reintegration is a mode of file system operation that allows file servers to synchronize their data after network partitions. The reintegration design supports the main objectives of the disconnection-resilient file systems, which is to provide high available and reliable storage for files, and guarantees that file operations are executed in spite of concurrency and failures. In this paper, we show that server reintegration is efficient and practical by describing its design and implementation in Paradise File System. Moreover, we report on its performance evaluation using a cluster of workstations. Our results indicate clearly that our design exhibits a significant degree of automation and conflict-free mobile file system.

1. Introduction

The last two decades have witnessed an increase in complexity and maturity of distributed file systems. Both well-established commercial and research systems have addressed a vast palette of users' needs in today's highly distributed environments. Those needs range from failure resiliency to mobility, to extended file sharing, and to dramatic scalability. However, even the most advanced of current systems fail to tackle the issue of continuous availability of every mobile client's data in today's increasingly wireless working environments. Henceforth, there is a need for a novel replication and a node reintegration technique, different than those used in existing distributed file systems aimed at mobility. More specifically, distributed file systems may experience different obstacles such as disconnection from the network resulting from denial of access to the other nodes, the limited network bandwidth, or serious conflicts when synchronizing back to the file system. Therefore, it is essential to come up with a file system solution that solves these problems and provides file system services during disconnections, manages node arrivals and departures, and works out data reintegration and conflict resolution on various file system objects.

In this paper, we explore the server reintegration mechanism that is used in Paradise Mobile File System (PFS) [1]. In section 2, we review the related work done on this field, and in section 3, we present a detailed description of our design. In section 4, we report the performance and tests results. The last section also states our conclusions.

2. Related Work

The two systems that are most closely related to our work are Coda [1,2,4] and InterMezzo [2,3]. They both support servers' reintegration by adopting replay logs to synchronize their data after disconnections and attempt to provide a conflict-free file system. For completeness, let us now present the related work in more detail.

2.1 Server integration in Coda File system

The Coda file system, a successor of AFS-2 that is developed at Carnegie Mellon University, is designed to react to any potential network failures [1]. It allows a user to continue working regardless of network failures as well as potential server disconnections. The namespace in Coda is mapped to individual file servers (Vice), at the granularity of sub-trees which is referred as *volumes*. At each client, a *cache manager (Venus)* dynamically obtains and caches volume mappings [1].

In order to achieve high availability, Coda caches the needed objects during connection and uses emulation operations to serve the disconnected clients. While disconnected, Venus services file system requests by relying solely on its cache's contents. It basically "emulates" the function of Vice and service the client's requests. When disconnection ends, *Venus* cache manager reintegrates all data and then switches back to server replication mode [2]. During emulation, Venus records sufficient logs to perform the update activity when it reintegrates to the file system. It maintains this information in a per-volume log, and each log entry contains a copy of the corresponding system call arguments as well as the version state of all objects referenced by the call [3].

The propagation of changes from client to server groups is accomplished in two steps [4]. In the first step, Venus obtains permanent *fids* for new objects and uses them to replace temporary *fids* in the replay log. In the second step, the replay log is shipped in parallel to the servers group, and executed independently at each member. Each server performs the replay within a single transaction, which is aborted if any error is detected.

The Coda replay algorithm consists of four phases. In phase one the log is prepared and all objects referenced in the log are locked. In phase two, each operation in the log is validated and then executed. The validation consists of conflict detection as well as integrity, protection, and disk space checks. Fore most logged operations, execution during replay is identical to execution in connected mode. Phase three, which is known as back-fetching [4], consists exclusively of performing these data transfers. The final phase is committing the transaction and releases all locks [4]. If reintegration succeeds, Venus frees the replay log and resets the priority of cached objects referenced by the log. However, f reintegration fails, Venus writes out the replay log to a local replay file, and all corresponding cache entries are then removed, so that subsequent references will cause refetch of the current contents at the servers group [3].

The check for conflicts in Coda relies on the fact that each replica of an object is tagged with a *storied* that uniquely identifies the last update to it [2]. During phase two of replay, a server compares the *storied* of every object mentioned in a log entry with the *storied* of its own replica of the object. If the comparison indicates equality for all objects, the operation is performed and the mutated objects are tagged with a new *storied* specified in the log entry. If a *storied* comparison fails, the action taken depends on the operation during being validated. In the case of a store of a file, the entire reintegration is aborted.

2.2 InterMezzo KML integration

InterMezzo is a filtering file system layer, which is placed between the virtual file system and a specific file system such as ext3, ReiserFS, JFS, or XFS [3]. It provides distributed file system functionality with a focus on high availability. It uses InterSync, which is a client-server system that synchronizes folders between a server system and its clients [2]. InterSync periodically pulls the server for changes and reintegrates those changes into the client file system. The changes are recorded on the server by the InterMezzo file system, which maintains a Kernel Modification Log (KML) [2] as the file system is modified. The modification log makes it possible to collect the changes in the server file system without scanning for differences during reintegration. InterSync synchronizes the file system by fetching the KML using the HTTP protocol. It then processes the records in the KML and when it comes across a file modification record, it fetches the file from the server again using the HTTP protocol. The KML file consists of records, each of which encodes a change to the file system. The records track in detail [5]: Objects were affected by the change, the identity and group membership of the process making the modifications the version of the object that was changed, the new attributes of affected objects and the record sequence number.

Typically a KML record is between 100-300 bytes in size, depending on the operation being performed and the length of the pathnames. Once the KML has been transferred from one system to another the process of reintegration can begin. The reintegration process goes through a few steps: First, InterMezzo unpacks the records in the KML segment, then it checks if the versions of the objects that are being modified match those given in the record. If they do not match, then this is an indication of a possible conflict. After that, InterMezzo makes the change to the file system and proceeds with the next record.

In addition to KML, InterMezzo maintains a secondary replication log called the synchronization modification log (SML) [3], which allows an empty

or heavily out of date client to synchronize in an efficient manner, as follows: a newly-connected client should do a *replicator status* call to find out the status of the KML. If it determines that it is older than the last KML truncation, i.e. the clients last received part of KML is older than the current KML logical offset, it first fetches the SML. Following the reintegration of the entire SML, the client must remove any files present in its local cache which were not referenced in the SML or updated on the client, because they are no longer on the server. Once the entire SML is reintegrated, the client fetches the next part of the KML. It integrates this with similarly relaxed conflict checking until it encounters the first record following the SML creation [3]. Once this segment of KML is reintegrated, the client is once again up to date and resumes activity as normal.

3. Servers Integration Design in Paradise File System (PFS)

As explored in Section-2, most available distributed files systems tailor their propagation algorithms to work for the client-side caches in order to guarantee data consistency. However, our goal in this paper is to propose an efficient algorithm that focuses on those file servers, which need to be synchronized with the centralized cache after disconnections. Before illustrating our algorithm, we present the PFS framework [2] in more detail:

The framework of PFS can be divided into three main stages: the *connected stage*, the *disconnected stage*, and *the re-joining stage*. Initially and while clients are connected, file system service is provided by the actual file servers. We define the connected stage as all file servers are viewable to clients and are able to answer their RPC requests. However, if one of the file server(s) does not respond to clients calls within a certain period of time, part of the file system is said to be disconnected and the system will switch to the disconnected stage. During this stage, the client will continue probing the disconnected server(s) on a regular basis. At the same time, part of the file system service is provided by the cache server, which is an independent file service as shown in figure-1. Finally, if both the file servers and the communication channel are back available, the file system will switch into the re-joining stage. In this phase, the communication link between the file system and the previously disconnected servers is re-established and file system services can be provided by these servers again. The propagation of the files, which were updated during the disconnection phase, is performed by the re-integrator module. During this phase, the file system propagates the updates made by the nodes during the disconnected stage back to the file server(s). Upon the successful termination of the re-joining process, the file server will switch back to the connected stage. Note that it is possible for the connection with the server to be lost again, bringing the client back into the disconnected phase.

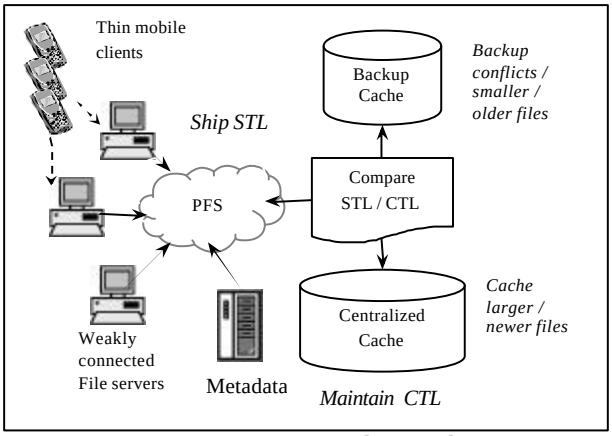

Figure-1: STL/CTL replaying rules

The backup cache is a secondary repository that saves file conflicts and is exported to the users, allowing them to view their backed-up files. It is the file owner's call to move these files from the backup cache to the actual file system.

3.1 PFS Logging and Reintegration

In PFS, reintegration is a transitory state through which the file servers reconnect back to the file system network after a disconnection. In this stage and as presented in section 3, the file servers reach a consistent state by synchronizing all modified data with the centralized cache and resolving all conflicts. This state is achieved by having two supporting phases, the logging stage and the reintegration stage.

3.1.1 The logging process

In PFS, the logging process starts once the file server looses connection with the metadata. To coordinate this action, the metadata periodically exchange heartbeat packets with the file servers to detect disconnections. When this event is triggered, both the cache and the file server will start the

logging process. In particular, the cache server will start maintaining a log file, which we refer to as Cache Transaction Log (CTL), *for each* file server that is disconnected from the network and will log all client accesses to the files in the centralized cache. On the other side, there will be other log files in each disconnected server, referred as Server Transaction Log (STL) that will log the actions done by the local users on the disconnected servers.

At the end of the disconnection stage, the system would be in an inconsistent state where objects on both sides, the file servers and the centralized cache, are modified and need to be synchronized. Both the STL and the CTL files consist of records that represent the changes to the file system during disconnection. The records track in details: The modified file name, the file size, the file owner and the MD5 signature of the file.

In our design, we minimize the network communication by shipping the STL logs and making the comparison locally on the cache server. This way, not only the communication is minimized, but the burden of replaying STL logs is shifted from the file servers to the centralized cache, which meets our goals of freeing the file servers as much as possible.

3.1.2 The Reintegration Process

The reintegration process is started the moment metadata triggers the heartbeat packets back. To demonstrate the reintegration process, consider PFS file servers *FSi and FSj* that are re-synchronizing with the cache server (*CS)*. At first, each file server will ship its STL file to the cache server, which will replay all the received STL files in parallel. In particular, *CS* will read each transaction for each reintegrating server, and identify which objects have changed during the disconnection stage. Then, *CS* communicates with *FSi* and issue one lock at a time on these objects. The locking step is important in order to prevent the local users on the file server from doing any modifications while the reintegration process. Once locks are successfully placed, the cache server will start the reintegration process.

Because reading STL logs are done in parallel, it could happen that the cache server locates two entries in two different STLs with the same file namespace, causing a conflict. In this case, the caches server has to decide which log entry to start with. The decision is made based on the following classification:

Let *t* be the last modified time of object *f*. $f \in FS1$ and $f \in FS2$ during the reintegration stage. In case of object *f* exists on both file servers, the caches server will resolve the conflict by locating the entry that has an older timestamp *t*. The file with an older timestamp entry will be moved to the backup cache. The reason is that we assume the newer objects will more probability that will be accessed in the near future, and therefore they should be available on primary cache (cache temporal locality [1]). However, f by coincidence both have the same timestamp, then the metadata will base its comparison on the sizes of these objects; the smaller size object is moved to the backup cache, while the larger object is copied to the centralized cache. The owner of the smaller object is notified by this operation.

4. Experiment Results and Performance Measurements

In this section, we describe the current state of implementation and evaluate the performance of our server integration mechanism. In our prototype, we used eight Linux machines running Coda file system, with PFS file system layer on top. Two of these machines are acting as servers (Vice), and the other four as file servers (Venus). All of the three machines are connected by 100Mb/s Ethernet. As to make PFS functional, the file system is also exported to the sixth machine, which has the Cache Replacement Algorithm (CRA) [2] code running on the file system and acting as the cache server as well. The primary cache size is 150MB and the backup cache is 100MB.

First, we ran a number of experiments to explore the behavior of the system. Typically an STL file with 10,000 records is 50-75KB in size. Figure-2 shows the relation between the STL size and the number of files conflicts. As shown in the figure, the larger STL file is, the more likelihood that an entry is repeated in the log, causing more object movements from the cache server to the backup repository. Repeated entries are generally resulted from updating a particular file more than once during a disconnection period. As we may notice in the figure, probability of conflicts tend to increase in a higher rate when STL logs exceed 10,000 entries.

Likewise we expect the relation between the number of STL logs and the likelihood of conflicts to be the same. That is, having more servers to

reintegrate (i.e., more STL logs to be shipped), would cause more conflicts to occur. Of course, we expect the process to consume more time because the cache will start switching from one STL log to another.

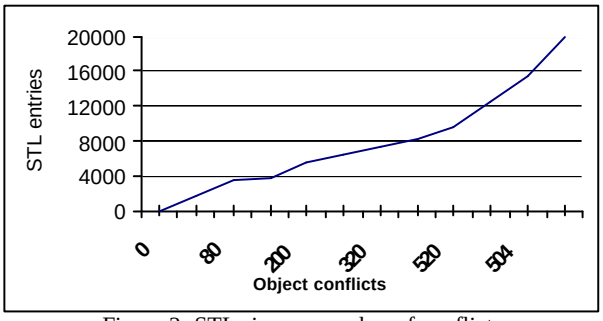

Figure-2: STL size vs. number of conflicts

The time of reintegration process is the period of which the metadata allocate the joining server to the time of integrating the last record in the STL and CTL files. Figure-3 shows how reintegration time is affected when more STL conflicts are introduced. Clearly, the metadata would need more processing time to resolve file conflicts and move the objects to the backup repository. Our simulation results shows that replaying a conflicting entry takes 21% more time than a conflict-free entry.

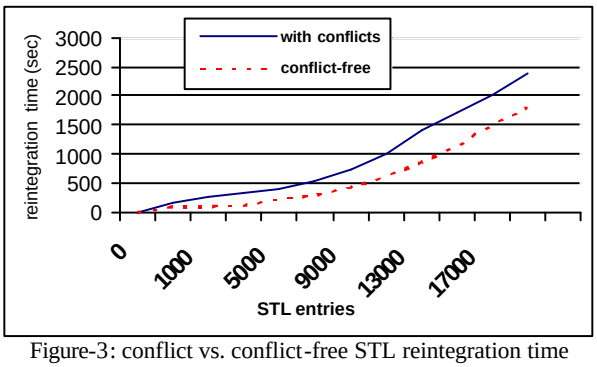

Figure-3: conflict vs. conflict-free STL reintegration time

In Figure 4, we show how reintegration time is affected by the number of reintegrating servers. The bottleneck for the integration process is determined by how many STL logs the cache server can process. In our experiment, the time taken to reintegrate all eight servers is almost linear, indicating that the cache server did not reach this bottleneck yet.

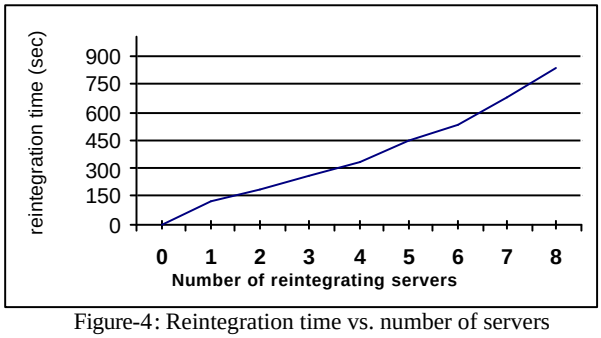

Figure-4: Reintegration time vs. number of servers

5. Conclusion and Future Work

In this paper, we have presented our servers reintegration design and implementation using Paradise File System. Our results clearly indicate that the design exhibits a significant degree of automation and supports the objectives of building a conflict-free mobile file system. We have also reported on its performance evaluation using a cluster of workstations. Our results indicates that efficient servers' reintegration is achievable in file systems, bearing in mind the right size of STL logs and the number of reintegrating servers.

Our future work includes incorporating the design into the VFS level; we expect that the reintegration time will be significantly reduced, allowing even longer STL replays to be feasible.

References

[1] Ebling, M.R., Mummert, L.B., Steere, D.C., Overcoming the Network Bottleneck in Mobile Computing, Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, Dec. 1994.

[2] Willick, D.L., Eager, D.L. and Bunt, R.B., Disk Cache Replacement Policies for Network Fileservers, Proc. 13th International Conference on Distributed Computing Systems, May 1993, 2-11

[3] Satyanarayanan, M., Scalable, Secure, and Highly Available Distributed File Access, IEEE Computer, Vol. 23, No. 5, May 1990.

[4] Braam, P. J., The Coda Distributed File System, Linux Journal, #50, June 1998.

[5] Lu Q., Satyanarayanan, M., Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions, Proceedings of the Fifth IEEE HotOS Topics Workshop, May 1995.

[6] Kistler, J.J., Satyanarayanan, M., Disconnected Operation in the Coda File System, ACM Transactions on Computer Systems, Vol. 10, No. 1, pp. 3-25, Feb. 1992.

[7] Peter J. Braam Philip A. Nelson., Removing Bottlenecks in Distributed Filesystems: Coda & InterMezzo as examples, Proceedings of Linux Expo 1999, May, 1999.

[8] Kistler, J.J., Disconnected Operation in a Distributed File System, School of Computer Science, Carnegie Mellon University, May 1993.

[9] Peter J. Braam. InterMezzo: File Synchronization with InterSync.

[10] Silvano M., Cache management algorithms for flexible file systems, ACM SIGMETRICS Performance Evaluation Review, December 1993.

[11] John C.S. Lui, Oldfield K.Y. So, T.S. Tam., NFS/M: An Open Platform Mobile File System, The 18th International Conference on Distributed Computing Systems (ICDCS'98), May, 1998.

[12] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout, Caching in the Sprite Network File System, ACM Trans. Computer Systems, Vol. 6(1), pp. 134-154, 1988.

[13] A. Boukerche, R. AlShaikh, Bo Marleau, “Disconnection-resilient Filesystem for Mobile Clients”, 2005

[14] R. AlShaikh, Highly available File System for Mobile Networks, Master thesis, Univ. of Ottawa. In preparation.