HOW TO DEAL WITH REPLICATION AND RECOVERY IN A DISTRIBUTED FILE SERVER

I. Arrieta-Salinas, J. R. Juárez-Rodríguez, J. E. Armendáriz-Iñigo and J. R. González de Mendívil
Departamento de Ingeniería Matemática e Informática, Universidad Pública de Navarra, 31006 Pamplona, Spain

Keywords: Replication, Crash-Recovery Model, Group Communication Systems, Virtual Synchrony, Distributed File Server, Testing.

Abstract: Data replication techniques are widely used for improving availability in software applications. Replicated systems have traditionally assumed the fail-stop model, which limits fault tolerance. For this reason, there is a strong motivation to adopt the crash-recovery model, in which replicas can dynamically leave and join the system. With the aim to point out some key issues that must be considered when dealing with replication and recovery, we have implemented a replicated file server that satisfies the crash-recovery model, making use of a Group Communication System. According to our experiments, the most interesting results are that the type of replication and the number of replicas must be carefully determined, specially in update intensive scenarios; and, the variable overhead imposed by the recovery protocol to the system. From the latter, it would be convenient to adjust the desired trade-off between recovery time and system throughput in terms of the service state size and the number of missed operations.

1 INTRODUCTION

Data replication is a well-known technique used for improving performance and enhancing fault tolerance in software applications. Two major classes of replication approaches are known in the literature, in terms of who can propagate updates: *active replication* (or state-machine) techniques (Schneider, 1993), in which any replica can propagate a received update request; and *passive replication* (or primary-backup) approaches (Budhiraja et al., 1993), where only the replica that acts as primary is in charge of receiving and propagating all updates, whereas the others act as backups. Replicated systems have traditionally assumed the *fail-stop* model (Schneider, 1984). Its main advantage resides in its simplicity, since replicas only fail when they crash, remaining forever in this state. Nevertheless, as replicas cannot connect to the system during normal operation, only the crash of a minority of replicas is tolerated. For this reason, there is a strong motivation to consider the *crash-recovery* model, in which replicas can dynamically leave and join the system. Despite being a desirable feature, this requires a recovery protocol, where joining replicas obtain the necessary changes to update their stale state. In this context, *Group Communication Systems* (GCS) (Chockler et al., 2001) greatly simplify the job of ensuring data consistency in the presence of failures. A GCS features a membership service that monitors the set of alive members and notifies membership changes by means of a view change, along with a communication service that allows group members to communicate among themselves.

The recovery of outdated replicas can be carried out in many ways. The simplest one would consist of a *total recovery*, by transferring the entire service state to the joining replica. This is mandatory for replicas that join the system for the first time, but it may also be adequate if most of the data have been updated since the replica failed. However, total recovery can be highly inefficient if the size of the service state is big or there have not been many updates since the joining replica went down. In such situations, it may be more convenient to perform a *partial recovery*, transferring only the changes occurred during the joining replica's absence. Partial recovery is possible thanks to *virtual synchrony* (Chockler et al., 2001); however, this property provided by the GCS expresses delivery guarantees that have nothing to do with processing. As a consequence, the real state at the joining replica may differ from the last state it is assumed it had before crashing, due to the fact that it may

Arrieta-Salinas I., R. Juárez-Rodríguez J., E. Armendáriz-Iñigo J. and R. Gonzalez de Mendivil J. (2009).
HOW TO DEAL WITH REPLICATION AND RECOVERY IN A DISTRIBUTED FILE SERVER.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 147-152
DOI: 10.5220/0002258001470152
Copyright © SciTePress

not have processed all delivered messages, causing the amnesia phenomenon (Cristian, 1991; de Juan-Marín, 2008). Thus, the joining replica will have to obtain two types of lost messages: *forgotten messages* that were delivered but not applied before failure, and *missed messages* that were delivered at the system during the disconnection period.

In this paper we present a replicated system that takes advantage of the properties provided by GCSs to support the crash-recovery model. As far as the type of replicated service is concerned, special attention has been paid to databases (Bernstein et al., 1987; Kemme et al., 2001). With the aim to study problems that may arise when the operation is not performed inside the boundaries of a transaction, we have focused on non-transactional services. In particular, we have implemented a replicated file server allowing clients to remotely execute basic operations over a structure of directories and files. Moreover, the file server manages a lock system to block files and temporarily prevent other clients from accessing them. We compare the performance of passive and active replication for the file server, depending on the workload and rate of reads and writes. This paper also assesses the overhead introduced by the recovery process, analyzing total and partial recovery in a variety of reconfiguration scenarios. We intend to determine the circumstances in which partial recovery performs better than total recovery, and discuss the advantages of a combination of both approaches.

The rest of the paper is organized as follows. Section 2 depicts the system model. Section 3 details the replication protocols we have used, whereas Section 4 includes our recovery alternatives. Section 5 presents the evaluation of our solutions for replication and recovery. Finally, conclusions end the paper.

2 SYSTEM MODEL

The implemented application consists of a replicated system supporting the crash-recovery model, which provides high availability for a file server. The system is partially synchronous, i.e. time bounds on message latency and processing speed exist, but they are unknown (Dwork et al., 1988).

The system model is shown in Figure 1. Replicas communicate among themselves using a GCS, which guarantees the properties of virtual synchrony. As for the group composition, we shall consider a *primary partition* service (Chockler et al., 2001). Each replica manages an instance of the replicated service (in this case, a file server). Replicas also run a replication protocol to ensure data consistency and a recovery protocol, which handles the dynamic incorporation of replicas. Each one is equipped with a persistent log.

When a client C_i wants to execute an operation at the replicated system, it must build a request req_{ij}, uniquely identified by the pair formed by the client identifier i and a local sequence number j that is incremented for each new request. C_i submits req_{ij} to one of the replicas using an asynchronous *quasi-reliable* point-to-point channel (Schiper, 2006) and waits for the corresponding result; hence, it will not be able to send other requests in the meantime. In order to cope with crashes of replicas, req_{ij} is periodically retransmitted to other replicas.

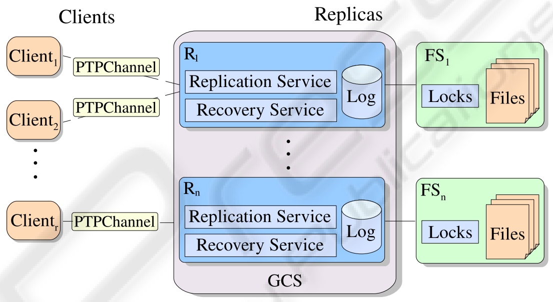

Figure 1: System model.

3 REPLICATION

Our replication protocols are based on the specifications given in (Bartoli, 1999), which provides the implementation outline for a passive replication protocol, along with the required modifications to transform it into an active replication protocol.

The algorithm for passive replication for a replica R_m supporting the fail-stop model is presented in Figure 2. R_m handles a local counter for update operations ($updateCnt$), as well as a list of pairs $\langle i, result \rangle$ denoted $lastUpd$, containing the result of the last update operation executed on behalf of each client C_i. During initialization, R_m applies the deterministic function *electPrimary()*, which ensures that all alive replicas agree on the same primary.

When R_m receives a read request, it directly executes it and sends the result back to the client (lines 7-9). On the contrary, if the request contains an update, backups forward it to the primary, whereas the primary sends an *Uniform Multicast* message (with FIFO order) by means of the GCS, containing the update request (lines 11-12). Uniform Multicast (Bartoli, 1999) enables each replica that applies an update to conclude that every other replica in the current view will eventually apply that update or crash, thus avoiding false updates. It is worth noting that read requests

```
1.  Initialization:
2.  $p$ := electPrimary() //Primary ID
3.  $updCnt$ := 0 //Counter for updates
4.  $lastUpd$ := $\emptyset$ // $\langle i, result \rangle$ tuples
5.
6.  a. Upon receiving (Request $\langle req_{ij} \rangle$) from PTPChannel
7.       ⋆ if (type($req_{ij}$) = read) then
8.          ◇ $result_{ij}$ := execute($req_{ij}$)
9.          ◇ send($result_{ij}$) to $C_i$
10.      ⋆ else // write operation
11.         ◇ if ($p = R_m$) then UFmulticast(Update $\langle req_{ij} \rangle$)
12.         ◇ else send(Request $\langle req_{ij} \rangle$) to $p$
13. b. Upon receiving (Update $\langle req_{ij} \rangle$) from the GCS
14.      ⋆ if ($sender(Update\langle req_{ij} \rangle) = p$) then
15.         ◇ $result_{ik}$ := lastUpd(i)
16.         ◇ if ($k < j$) then
17.            • $result_{ij}$ := execute($req_{ij}$)
18.            • $updCnt{+}{+}$
19.            • lastUpd(i) := $result_{ij}$
20.            • if (local($req_{ij}$)) then send($result_{ij}$) to $C_i$
21.         ◇ else if ($j = k$) then
22.            • if (local($req_{ij}$)) then send($result_{ik}$) to $C_i$
23. c. Upon receiving a vchg(V) from the GCS
24.      ⋆ $p$ := electPrimary()
```

Figure 2: Passive replication protocol at replica R_m.

are executed as soon as they are received for the sake of efficiency; thus, it is not ensured that a query will always reflect the latest system state.

Upon receiving an update request req_{ij} from the GCS, it is necessary to check that it was sent from the current primary (line 14), since the multicast primitive only guarantees FIFO order, so if there has been a change of primary, updates sent by the previous and the current primary may arrive to replicas in different order. Then, R_m checks whether req_{ij} is duplicated, by looking up at $lastUpd$ the last operation executed on behalf of client C_i (line 15). If req_{ij} is not duplicated, then R_m executes it, increments $updateCnt$, and if it is the one who received the request from C_i it sends the result (lines 16-20). In case the duplicate is the last request of C_i and R_m is the replica who received that request then it responds to C_i, because C_i might not have received the result (lines 21-23). Finally, upon receiving a view change, function *electPrimary()* is invoked, so as to choose a primary among surviving replicas (lines 23-24).

Transforming this protocol into an active one is pretty straightforward: all replicas act as if they were primary. Any replica that receives a request containing an update multicasts it, using *Uniform Total Order* (Bartoli, 1999) to guarantee that all replicas receive the same sequence of messages.

4 RECOVERY

Supporting the crash-recovery model does not only require discarding replicas that left the system as in the fail-stop model, but it also entails dealing with replica (re)connections. In the latter, upon a view change event, a recovery process must be performed to transfer the necessary information to the joining replica R_j, which will apply it to become up-to-date. In our model, during the recovery process all updated replicas continue processing incoming client requests.

The first step of the recovery process is to obtain the list of updated replicas and choose a recoverer among them. This can be done either by exchanging dedicated messages, as presented in (Bartoli, 1999), or by using the information about views (Kemme et al., 2001). Our model considers the latter option, as it does not require to collect multicast messages from all view members to know which replicas are updated. In our case, replicas keep a list of updated replicas during normal operation: when a view change reporting on the leaving of a replica is delivered, that replica is deleted from the list of updated replicas; when R_j finishes its recovery, it multicasts a message to inform on its successful recovery to all alive replicas, which will include it in the list. Upon starting the recovery process, R_j is delivered the list of updated replicas, chooses one of them to act as recoverer R_r and sends a recovery request to R_r. Upon receiving that request, R_r obtains the recovery information and sends it to R_j, not via the GCS but using a dedicated quasi-reliable point-to-point channel. If a timeout expires and R_j has not received the recovery information, it will choose another updated replica as recoverer. The transferred recovery information depends on the type of recovery. In the following we detail the types of recovery we have used in our system.

Total Recovery: R_r must send the service state (in our case, the whole structure of files and directories in the file server, along with the information regarding current locks), as well as the content of $lastUpd$.

Partial Recovery: In this case each replica must keep a persistent log to record information about applied updates. In our model we have not considered persistent delivery, as not all GCSs support it and its implementation is complex. If there is no persistent delivery, replicas must store recovery information during normal processing, that is, after processing an update operation the replica persistently stores the corresponding information, even if the current view is the initial view. This requires to introduce a new variable in the replication algorithm (Figure 2) denoting a persistent log (LOG), and include a new action after line 18 of Figure 2, in which $\langle updCnt, req_{ij}, result_{ij} \rangle$

is stored in the log. We assume that update operations are idempotent, so as to avoid inconsistencies during the recovery process. Before sending the recovery request, R_j restores its volatile state using the information from its local log. Then it sends a recovery request to R_r, containing the sequence number of the last applied update. R_r responds with the information related to updates with a higher sequence number than the one of the last update applied at R_j, thus including all forgotten and missed updates.

5 EVALUATION

Our testing configuration consists of eight computers connected in a 100 Mbps switched LAN, where each machine has an Intel Core 2 Duo processor running at 2.13 GHz, 2 GB of RAM and a 250 GB hard disk running Linux (version 2.6.22.13-0.3-bigsmp). The file server initially includes 200 binary files of 10 MB each. The persistent log for partial recovery is implemented with a local Postgresql 8.3.5 database. Each machine runs a Java Virtual Machine 1.6.0 executing the application code. Spread 4.0.0 has been used as GCS, whereas point-to-point communication has been implemented via TCP channels. In our experiments we compare the performance of the implementations of passive and active replication to find the influence of a number of parameters on the saturation point of the system. On the other hand, we assess the cost of the recovery process and compare total and partial recovery, considering the recovery time, the impact of recovery on the system's throughput and the distribution in time of the main steps of the recovery process.

5.1 Replication Experiments

We have evaluated the behavior of our replication protocols for the file server in a failure free environment, depending on the following parameters: number of replicas (from 2 to 8), replication strategy (active and passive), percentage of updates (20%, 50% and 80%), number of clients (1, 5, 10, 20, 30, 40, 50, 60, 80, 100, 125 and 150), number of operations per second submitted by each client (1, 2, 4, 10 and 20) and operation size (10, 25, 50 and 100 KB). A dedicated machine connected to the same network executes client instances. Each client chooses randomly one of the replicas and connects to it in order to send requests (read or write operations over a randomly selected file) at a given rate during the experiment. Each experiment lasts for 5 minutes.

Figure 3(a) shows the system performance obtained with 4 replicas while incrementing the number of clients, each one sending 10 requests per second. There is a proportional increase of the system throughput as the number of clients grows, until a saturation point is reached. As expected, system performance is inversely proportional to the operation size (due to the execution cost itself and network latency), and to the update rate (as read requests are locally processed, whereas updates must be propagated and sequentially applied at all replicas). In addition, active and passive replication have almost the same throughput levels when there is a low rate of updates, as reads are handled in the same way. In contrast, passive replication is more costly if there is a high rate of updates, since the primary acts as a bottleneck. We shall remark that, as the constraint of uniform delivery is responsible for the most part of multicast latency, the cost of update multicasts is the same in passive replication, where only FIFO order is needed, and active replication, which requires total order. In fact, Spread uses the same level of service for providing Uniform Multicast, regardless of the ordering guarantees (Stanton, 2009).

Figure 3(b) results from executing the same experiments as in Figure 3(a), but in this case with 8 replicas in the system. From the comparison between both figures we can conclude that an increase in the number of replicas improves performance when there is a low rate of updates, since read requests are handled locally and therefore having more replicas allows to execute more read requests. On the contrary, when there is a high rate of updates, performance does not improve, and it even becomes worse if the operation size is small, as the cost of Uniform Multicast increments with the number of replicas. However, if the operation size is big, the cost of Uniform Multicast is masked by the execution costs.

5.2 Recovery Experiments

In the following we present how the recovery experiments were run. The system is started with 4 replicas, and then one of them is forced to crash. The crashed replica is kept offline until the desired outdatedness is reached. At that moment, the crashed replica starts the recovery protocol. Figure 4 depicts the recovery time depending on the recovery type. In this case, no client requests are being issued during recovery. Total recovery has been tested with different initial sizes; therefore, the recovery process must transfer the initial data in addition to the outdated data. The results show that, in total recovery, the recovery time is proportional to the total amount of data to be transferred.

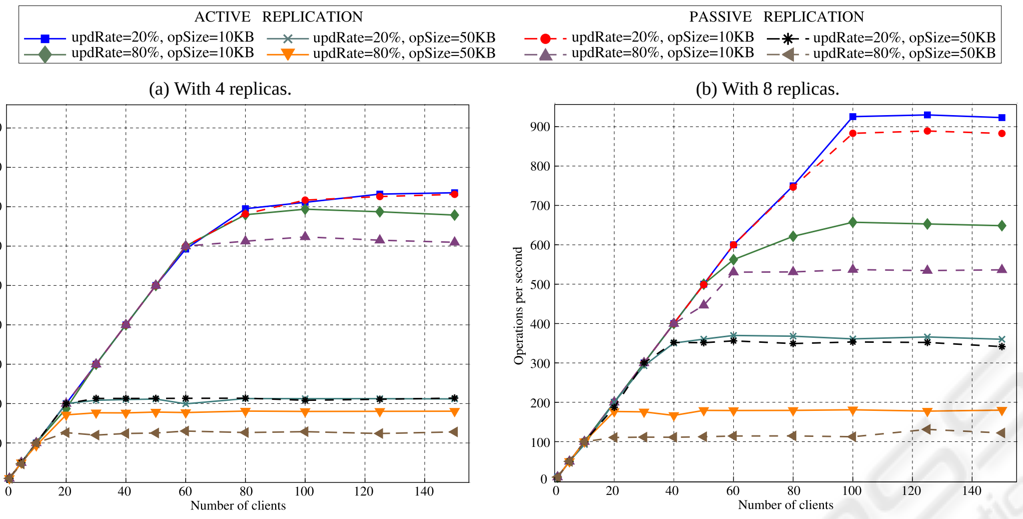

Figure 3: System throughput. Each client submits 10 requests per second.

On the other hand, in partial recovery the initial size has no effect on the recovery time, since only the outdated data have to be transferred. In this case, operation size has a relevant impact on the recovery time: if the operation size is small, a greater number of operations have to be applied, which takes more time than applying less operations of bigger size. We can infer that, when the total size of the service state is small, total recovery is more efficient, especially if the recovering replica has missed a lot of operations. On the contrary, if the service state is big in relation with the outdated data, partial recovery is more convenient.

We have performed the same recovery experiments as in Figure 4, but with clients sending requests at different rates during recovery, so as to evaluate the impact of attending client requests on the recovery process. Table 1 shows the recovery time for an outdatedness of 100, 250 and 500 MB with total and partial recovery. During recovery, there are 10 clients sending 10 requests per second each, with different operation sizes and update rates. In general, we can conclude that the recovery time is proportionally affected by the workload, as the recoverer has to process requests while retrieving the recovery information, and the network is also being used by the replication protocol. Furthermore, when there is a high update rate, the recovery process takes longer because the recovering replica must apply updates that were delivered during the previous steps of the recovery process, so as to catch up with the rest of the system. In the same way, the recovery process has an impact on the system's overall performance. In general, the average response time for client requests is incremented in a 60% during the recovery process.

Finally, we have measured the relative time to execute the four main steps of the recovery process: reading the recovery information at the recoverer and sending it to the recovering replica (*read*), obtaining the information from the network (*receive*), applying the information (*apply*) and processing updates received at the recovering replica during the previous steps (*catch up*). Figure 5 shows the percentage distribution in recovery time for each of the aforementioned steps. The interesting information conveyed by this figure is that, in total recovery, the recovering

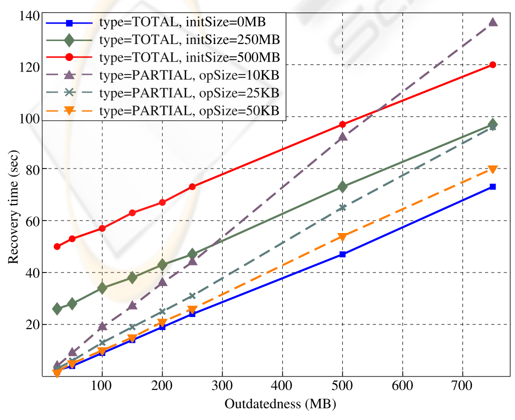

Figure 4: Recovery results (no clients during recovery).

Table 1: Recovery time (in seconds). There are 10 clients during recovery, each sending 10 requests per second.

Upd.	Op.	Total recovery			Partial recovery		
Rate	Size	100MB	250MB	500MB	100MB	250MB	500MB
20%	10KB	9	29	51	21	50	101
80%	10KB	10	30	53	25	70	129
20%	50KB	9	31	52	11	28	57
80%	50KB	12	34	58	17	32	71

replica spends most of the recovery process waiting for the recovery information, because the recoverer sends entire files (of 10 MB each) that need a considerable amount of time to be transmitted through the network, whereas writing each file on its local file server is a very fast task. In contrast, in partial recovery the major bottleneck is the *apply* task, as the recovery information consists in small parts of files, that are transmitted faster than the time needed by the recovering replica to write each piece of file.

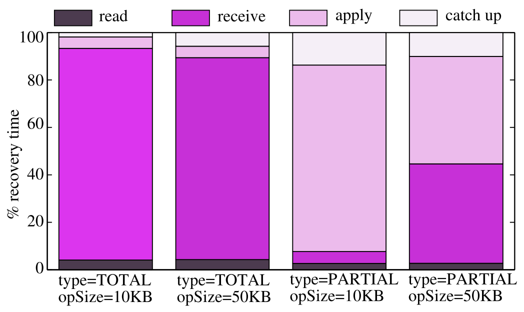

Figure 5: Percentage distribution in recovery time for each of the main recovery steps.

6 CONCLUSIONS

We have presented a replicated file server that satisfies the crash-recovery model by implementing some of the most representative replication and recovery techniques that make use of GCSs. When comparing our replication protocols, we have detected that in passive replication the primary acts as a bottleneck that limits system throughput, whereas in active replication the total order multicast defines the order of updates execution. The latency increase of this communication primitive is irrelevant since the cost of uniform delivery (needed by both protocols) is much greater. Moreover, the cost of uniform delivery depends on the number of replicas, so this parameter must be carefully chosen, specially if the workload is write intensive. On the other hand, one of the key aspects for an efficient fault tolerance is performing the recovery process as quickly as possible, while minimizing its impact on the service provided. Since total and partial recovery perform differently depending on the size of data and the number of missed operations, the recovery process could be improved by devising a combined solution, in which the recoverer would decide between total and partial recovery using a threshold based on the two aforementioned factors. Furthermore, it would be convenient to establish the desired trade-off between recovery time and system throughput, according to the necessary system requirements.

Finally, we shall point out that in our model request processing continues at updated replicas during recovery, which might be a problem in scenarios with high workload, as recovering replicas may not be fast enough to catch up with the rest of the system. It would be interesting to implement a solution to avoid this drawback without incurring unavailability periods, such as the one proposed in (Kemme et al., 2001), that divides the recovery process into rounds.

ACKNOWLEDGEMENTS

This work has been supported by the Spanish Government under research grant TIC2006-14738-C02-02.

REFERENCES

Bartoli, A. (1999). Reliable distributed programming in asynchronous distributed systems with group communication. Technical report, Università di Trieste, Italy.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison Wesley.

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. (1993). *Distributed Systems, 2nd Ed. Chapter 8: The primary-backup approach*. ACM/Addison-Wesley.

Chockler, G., Keidar, I., and Vitenberg, R. (2001). Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469.

Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78.

de Juan-Marín, R. (2008). *Crash Recovery with Partial Amnesia Failure Model Issues*. PhD thesis, Universidad Politécnica de Valencia, Spain.

Dwork, C., Lynch, N. A., and Stockmeyer, L. J. (1988). Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323.

Kemme, B., Bartoli, A., and Babaoğlu, Ö. (2001). Online reconfiguration in replicated databases based on group communication. In *DSN*, pages 117–130. IEEE-CS.

Schiper, A. (2006). Dynamic group communication. *Dist. Comp.*, 18(5):359–374.

Schneider, F. B. (1984). Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154.

Schneider, F. B. (1993). *Distributed Systems, 2nd Ed. Chapter 7: Replication Management Using the State-Machine Approach*. ACM/Addison-Wesley.

Stanton, J. R. (2009). The Spread communication toolkit. Accessible in URL: http://www.spread.org.