Journal of Universal Computer Science, vol. 11, no. 5 (2005), 703-711
submitted: 30/11/04, accepted: 31/1/05, appeared: 28/5/05 © J.UCS

Replication: Understanding the Advantage of Atomic Broadcast over Quorum Systems

Richard Ekwall
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
nilsrichard.ekwall@epfl.ch

André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
andre.schiper@epfl.ch

Abstract: Quorum systems (introduced in the late seventies) and atomic broadcast (introduced later) are two techniques to manage replicated data. Despite of the fact that these two techniques are now well known, the advantage of atomic broadcast over quorum systems is not clearly understood. The paper explains exactly in what cases atomic broadcast is a better technique than quorum systems to handle replication.
Key Words: atomic broadcast, quorum systems, replication, isolation.
Category: C.2.4, D.4.5

1 Introduction

The requirement for highly reliable and available services has been continuously increasing in many domains for the last decade. Several approaches for designing fault-tolerant services exist. The focus in this paper is on software replication. Replication allows a number of replicas to crash without affecting the availability of the service.

Quorum systems was the first technique introduced to manage replication [14, 5]. Since this period, a lot of progress has been accomplished in the understanding of the problems related to replication. An important step has been the introduction of *group communication*, which defines a middleware layer that hides most of the hard problems related to replication [12]. The advent of group communication has temporarily led to a decrease of interest in quorum systems. However, there has been recently a renewed interest in quorum systems for Byzantine faults [11], an issue not addressed previously. Moreover, there are now here and there people disagreeing on the advantage of group communication over quorum systems for replication. The goal of the paper is to clarify this

Research supported by OFES under contract number 01.0537-1 as part of the IST REMUNE project (number 2001-65002).

issue, and point out precisely when and why group communication is a better solution.

The rest of the paper is organized as follows. Section 2 introduces our system model and three isolation degrees, a key issue to understand the respective scope of quorum systems and atomic broadcast. Section 3 discusses the absence of isolation requirements. Section 4 discusses the case when only read-write isolation is required. Section 5 discusses general isolation requirements. Finally, Section 6 concludes the paper.

2 Different isolation degrees

In the context of replication, one of the key issues is the semantics that has to be provided. We consider in the paper a finite set of processes, where each process issues a sequence of operations over a finite number of replicated data. Without restriction to generality, we consider that each operation is either a *read* or a *write*. A read operation reads one replicated data; a write operation writes one replicated data. The semantics defines the result of each of these read and write operations. One key aspect of the semantics is the *isolation* property, as defined in the context of database systems [15]. We distinguish the following degrees of isolation:

- *No isolation*: Any interleaving of operations is possible; only the semantics of each individual read or write operation is defined.
- *Read-write isolation*: In addition to the individual semantics of read and write operations, a read followed by a write on the same data are executed in isolation.
- *General isolation*: In addition to the individual semantics of read and write operations, any sequence of operations can be executed in isolation.

To illustrate the three cases, consider two processes p, q, two replicated data X, Y, and the following sequences of operations:[1]

- *Sequence of operations issued by p:* $r_p(X)$, $w_p(X)$, $r_p(Y)$, $w_p(Y)$
- *Sequence of operations issued by q:* $r_q(Y)$, $w_p(Y)$, $r_q(X)$, $w_q(X)$

With no isolation, any interleaving of operations of the two processes is possible.

We express isolation using [...] brackets. Here is the same same sequence of operations with read-write isolation (the consecutive read-write operations are executed in isolation):

[1] $r_p(X)$ (respt. $w_p(X)$) denotes a read (respt. a write) of data X by process p.

- *Sequence of operations issued by p:* $[r_p(X), w_p(X)\ ], [r_p(Y), w_p(Y)\ ]$
- *Sequence of operations issued by q:* $[\ r_q(X), w_q(X)\ ], [\ r_q(Y), w_p(Y)\ ]$

Finally, general isolation allows us to specify for example the following isolation requirement:

- *Sequence of operations issued by p:* $[r_p(X), w_p(X), r_p(Y), w_p(Y)\ ]$
- *Sequence of operations issued by q:* $[\ r_q(X), w_q(X), r_q(Y), w_p(Y)\ ]$

3 No isolation

Read-write operations with no isolation corresponds to the notion of *register* [10]. The strongest register semantics, called *atomic register*, ensure that the read and write operations behave as if each operation *op* issued by process p happened instantaneously at some time $t \in [\, op_{start}, op_{end}]$, where op_{start} is the time at which the *op* is issued by process p, and op_{end} is the time at which *op* has completed on p [10].

Atomic registers can be implemented in an asynchronous system (which is defined as a system in which there is no bound on the transmission delay of messages, nor on the relative speed of processes). Quorums are here well suited to implement atomic registers. As an example, consider the data X replicated on several servers X_i, where each server X_i manages (1) a copy of the data and (2) a version number. A quorum is defined as any subset of servers. Quorum systems distinguish *read* quorums and *write* quorums, which must satisfy the following properties [5]:

- Any read quorum has a non-empty intersection with any write quorum.
- Any two write quorums have a non-empty intersection.

Let n be the number of replicas. One standard way to satisfy these properties is the following [5]:

- A read quorum is any subset of servers of size $\lceil \frac{n+1}{2} \rceil$.
- A write quorum is also any subset of servers of size $\lceil \frac{n+1}{2} \rceil$.

The operation $w_p(X \leftarrow val)$ (write *val* to X) by p is performed as follows: (1) p reads the version number from a read quorum, (2) then the local variable vn is set to the highest version number read, and finally (3) the value *val* with version number $vn + 1$ is written to a write quorum.

The *read* operation $r_p(X)$ is slightly less intuitive: (1) the client reads the pair $(value, version)$ from a read quorum, (2) the read operation returns the value

val with the highest version number, and finally (3) the value *val* is written to a write quorum.[2]

The specificity of this solution can be summarized as follows: (1) data is sent back and forth between the servers X_i and the client process p, and (2) servers only send and receive data. We will come back to this point later.

4 "Read-write" isolation only

To show the limitations of the atomic register semantics, and the need for read-write isolation, consider the following sequence of operations, where process p wants to increment X, while process q wants to decrement X. If X is initially 0, then without read-write isolation, the following execution is possible:[3]

- $r_p(X \Rightarrow 0)$, $r_q(X \Rightarrow 0)$, $w_p(X \leftarrow 1)$, $w_q(X \leftarrow -1)$

This execution is clearly not desired (the final value of X must be 0). A correct execution requires that p and q execute the read-write sequence in mutual exclusion, i.e., in isolation. This can be expressed as follows, where E_{CS}/L_{CS} allows a process to enter/leave the critical section:

- Operations issued by p:[4] E_{CS}, $r_p(X \rightarrow u)$, $u \leftarrow u + 1$, $w_p(X \leftarrow u)$, L_{CS}
- Operations issued by q: E_{CS}, $r_q(X \rightarrow u)$, $u \leftarrow u - 1$, $w_q(X \leftarrow u)$, L_{CS}

4.1 Read-write isolation and the consensus problem

We first show that read-write isolation cannot be solved in an asynchronous system with crash failures. Then we discuss the implementation of read-write isolation (1) with quorum systems and (2) with atomic broadcast (a group communication primitive).

Consensus is a well known problem defined over a finite set of processes, in which each process has an initial value and all processes that do not crash have to agree on a common value that is the initial value of one of the processes [1]. Consensus is not solvable in an asynchronous system if processes may crash [4]. This impossibility also applies to read-write isolation; it follows directly from the fact that read-write isolation is powerful enough to solve consensus (see also [8]).

[2] Without (3), the atomic register semantics is not ensured. To see this, consider (a) $w_{p_1}(X \leftarrow w)$ by p_1 that starts at $t = 1$ and ends at $t = 6$, (b) a read operation $r_{p_2}(X)$ by p_2 that starts at $t = 2$, reads w, and terminates at $t = 3$, and (c) a read operation $r_{p_3}(X)$ by p_3 that starts at $t = 4$ and ends at $t = 5$. Without (3), p_3 could read an old value rather than w, which is required by the atomic register semantics.

[3] $r_p(X \Rightarrow v)$ denotes a read operation that returns the value v.

[4] $r_p(X \rightarrow u)$ denotes that the value returned by the read operation is stored into the local variable u.

To show this, consider consensus to be solved among n processes $p_1, \ldots, p_n$, with val_i the initial value of process p_i. Let the data be here a vector V of $n+1$ elements $V[0], \ldots, V[n]$. Initially, we assume $V[0] = 0$, and all other elements $V[j]$ undefined. Each process p_i executes Algorithm 1, where $V[0]$ is incremented and $V[V[0]]$ written inside a critical section (lines 2-5).

1: E_{CS}	{*Enter Critical Section*}
2: $r_{p_i}(V \rightarrow u)$	{*Read vector V into local vector u*}
3: $u[0] \leftarrow u[0] + 1$	
4: $u[u[0]] \leftarrow val_i$	
5: $w_{p_i}(V \leftarrow u)$	{*Write u to vector V*}
6: L_{CS}	{*Leave Critical Section*}
7: $r_{p_i}(V \rightarrow u)$	{*Read vector V into local vector u*}
8: decide $u[1]$	{*Consensus decision*}

Algorithm 1: Solving consensus with read-write isolation: code of process p_i

If at least one process p_i is correct, then $V[1]$ is written (with the initial value of one of the processes). Moreover, since all processes decide on the value $V[1]$, they all decide the same value, which is the initial value of one of the processes. So read-write isolation allows us to solve consensus, which shows the contradiction, i.e., read-write isolation cannot be implemented in an asynchronous system with process crashes.

4.2 Implementing read-write isolation with quorums

Since read-write isolation cannot be implemented in an asynchronous system with process crashes, we need additional assumptions. The quorum solution of Section 3 can be extended to provide read-write isolation if we can solve the mutual exclusion problem. Implementing mutual exclusion requires to handle the following situation:

- Process p executes E_{CS} and gets permission to enter the critical section.
- Process p crashes before leaving the critical section.

In this case, p will never release the critical section, i.e., the critical section must be released on behalf of p. This requires a crash detection mechanism that detects the crash of p if and only if p has crashed (the critical section must be released if and only if p has crashed). This corresponds to a *perfect failure detector* [1], which is a strong requirement. Note that in addition to a perfect failure detector, if the read/write quorums are defined as in Section 3, the solution also requires a majority of correct processes (to always ensure the existence of a read quorum and of a write quorum).

4.3 Implementing read-write isolation with atomic broadcast

We now describe a different solution to read-write isolation, which uses a group communication primitive, namely *atomic broadcast* (also called *total order broadcast*). Atomic broadcast allows to broadcast messages to a group of processes, while ensuring that messages are delivered by all members of the group in the same order. A formal definition can be found in [7, 3]. To show the implementation of read-write isolation with atomic broadcast, we model the execution of each process as follows:

1: E_{CS}	{*Enter Critical Section*}
2: $r_p(X \rightarrow u)$	{*Read X into local variable u*}
3: $u \leftarrow f(u)$	{*Update u*}
4: $w_p(X \leftarrow u)$	{*Write u to X*}
5: L_{CS}	{*Leave Critical Section*}

Algorithm 2: Model for read-write isolation: code of process p

Process p first reads X into a local variable u, then does some local computing expressed by the function $f(u)$, and finally writes the new value of u to X.

With atomic broadcast, denoted by $ABcast(\,)$, the above schema can be implemented as follows, using a technique called state machine approach [9, 13]. The technique distinguishes between (1) the code of process p (Algorithm 3) and (2) the code of a server X_i that manages a copy x_i of the data X (Algorithm 4).

1: $ABcast(f)$ to g_X	{g_X *is the group of servers* X_i}
2: wait to receive *done* from at least one server X_i	

Algorithm 3: Read-write isolation: code of process p

1: **loop**	
2: wait for the delivery of f sent by some process p	
3: $x_i \leftarrow f(x_i)$	{x_i *is the local copy of X managed by server* X_i}
4: send(*done*) to p	
5: **end loop**	

Algorithm 4: Read-write isolation: code of server X_i

Every server X_i receives the update functions f in the same order, and updates its copy x_i using the same update function. Moreover, each server x_i executes one update function before considering the next one. So Algorithms 3 and 4 correctly implement atomic registers with read-write isolation. Indeed,

the solution requires to solve atomic broadcast. Atomic broadcast is solvable in an asynchronous system augmented with the failure detector $\diamond\mathcal{S}$ among the group g_X,[5] and a majority of correct servers [1].

4.4 Discussion

If we compare the requirements of the quorum solution and of the atomic broadcast solution, we observe the following. The two solutions require a majority of correct processes; the quorum solution requires a perfect failure detector, while the atomic broadcast solution only requires the weaker failure detector $\diamond\mathcal{S}$ (see [1] for a comparison of failure detectors). To understand how much $\diamond\mathcal{S}$ is weaker than a perfect failure detector, note that $\diamond\mathcal{S}$ allows an *unbounded* number of false crash suspicions, while a perfect failure detector does not allow a *single* false suspicion.

What makes the difference? *In the quorum solution, the update function f is executed by the client process itself. In the atomic broadcast solution, the update function f is executed by the servers.* The former solution requires (1) mutual exclusion, and (2) to send data back and forth between the client and the servers. The atomic broadcast solution requires only to send the update function f to the servers. Executing f on the servers is a more clever solution than executing f on the client!

5 General isolation

We now discuss the implementation of general isolation. The quorum solution can trivially be extended to handle general isolation. Indeed, whether mutual exclusion protects two operations or more than two operations makes no difference.

Can the atomic broadcast solution be extended to handle general isolation? Yes, if specific conditions are met (which also means that the solution is not always applicable):

- when the update function can be defined statically, e.g., when the application can be implemented using stored procedures, and
- when the identity of the servers to which f must be sent is statically known.

The atomic broadcast solution may also require atomic broadcasts to multiple groups [6]. We now give two examples where these conditions are satisfied.

[5] $\diamond\mathcal{S}$ satisfies the following properties: (1) Eventually every process in g_X that crashes is permanently suspected by every correct process in g_X, and (2) there is a time after which some correct process in g_X is never suspected by any correct process in g_X [1].

Example 1: Consider two replicated data X and Y, representing two bank accounts. Assume a user that wants to withdraw an amount w from account X and deposit w on the account Y. This can be expressed by the following update function:

$$f \equiv (\ sub(X, w);\ add(Y, w)\ )$$

The user simply issues $ABcast(f)$ to $g_X \cup g_Y$, where g_X, respt. g_Y, are the group of replicas of X, respt. Y. Upon delivery of f a server X_i decrements its local copy x_i by w, and a server Y_i increments its local copy y_i by w.

Example 2: Let us modify slightly Example 1, such that the transfer of w from account X to account Y takes place if and only if $X \geq w$. This can be expressed as follows:

$$f \equiv (\ \textbf{if}\ X \geq w\ \textbf{then}\ \ sub(X, w);\ add(Y, w)\ \textbf{endif}\ )$$

This leads to the following problem: while a server x_i can evaluate the condition $X \geq w$, a server y_i cannot. Nevertheless, this case can still be implemented using atomic broadcast: each server x_i after the evaluation of the condition $X \geq w$, sends $true$ or $false$ to the servers in g_Y. A server in g_Y waits this message to know whether or not to execute the *add* operation.

In these two examples the set of data to be accessed is known statically. If this condition is not met, which is quite common in the case of database transactions, then the atomic broadcast solution cannot be used (since it cannot be known to which servers to send the update function). Note that the function could be sent to *all* servers, but this solution might be too costly or even impossible to implement.

6 Conclusion

There is a common misunderstanding of the advantage of group communication over quorum systems to manage replicated data. We have tried to clarify this issue by showing the basic difference between the two techniques: when isolation needs to be provided, *group communication consists in sending the update function to the data servers, while with quorum systems servers send the data to the clients where the update function is performed.* The first solution requires weaker extensions to the asynchronous system, and so has obvious advantages. We have also shown that the use of group communication is not restricted to read-write isolation. This contradicts the claim of Cheriton and Skeen in [2] in the context of the CATOCS controversy,[6] where they write that *CATOCS cannot ensure serializable ordering between operations that correspond to group of messages (...)*

[6] CATOCS = Causally and totally ordered communication support.

Locking is the standard solution.[7] As shown, this argument is not correct. Apart from this specific issue, we believe that the paper should allow in the future to clearly understand the merits of group communication over quorum systems to manage replication.

References

1. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.
2. D. R. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communications. In *14th ACM Symp. Operating Systems Principles*, pages 44–57, Dec 1993.
3. X. Défago, A. Schiper, and P. Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys*, 36(2):372–421, December 2004.
4. M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.
5. D.K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 150–159, December 1979.
6. R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous systems. *Theoretical Computer Science*, 254(1-2):297–316, January 2001.
7. V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.
8. M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM Symposium on Principles of Distributed Computing (PODC)*, pages 276–290, August 1988.
9. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, July 1978.
10. L. Lamport. On interprocess communications, i, ii. *Distributed Computing*, 1(2):77–101, October 1986.
11. D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11(4):203–213, 1998.
12. A. Schiper. Practical Impact of Group Communication Theory. In *Future Directions in Distributed Computing*, pages 1–10. Springer, LNCS 2584, 2003.
13. F. B. Schneider. Replication Management using the State-Machine Approach. In Sape Mullender, editor, *Distributed Systems*, pages 169–197. ACM Press, 1993.
14. R.H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Trans. on Database Systems*, 4(2):180–209, June 1979.
15. G. Weikum and G. Vossen. *Transactional Iinformation Systems*. Morgan Kaufmann, 2002.

[7] Note that atomic broadcast can be used for locking