A Performance Evaluation of a Quorum-Based State-Machine Replication Algorithm For Computing Grids

Jean-Michel Busca[2] Marin Bertier[1] Fatima Belkouch[3] Pierre Sens[12] Luciana Arantes[1]
[1]LIP6, Université Paris 6 - CNRS, Paris, France
[2]INRIA, Rocquencourt, France
[3]Université Lille 2, Lille, France

Abstract

Quorum systems are well-known tools that improve the performance and the availability of distributed systems. In this paper we explore their use as a means to achieve low response time for network services that are replicated and accessed over computing grids. To that end, we propose both a quorum construction and a quorum-based state-machine replication algorithm that tolerates crash failures in a partially synchronous model. We show through the evaluation of a real implementation that although simple, this quorum construction and replication algorithm exhibit a response time 20% lower than that of a regular active replication algorithm in appropriate conditions.

1. Introduction

Replication is a key approach to improve network services availability and fault-tolerance. Following this approach, a mission-critical server may be instantiated in several *server replicas* coordinated by some replication protocol that preserves their mutual consistency. Server replicas are hosted on a set of nodes which are selected to exhibit independent failure probability, so that the overall availability of the service is actually improved.

However, this approach raises two important concerns regarding performances. First, replication induces significant cost overheads in terms of processing power and network bandwidth consumption: for instance, active replication protocols have all server replicas process every service request; besides, most replication protocols require additional messages on top of application-level requests and replies to preserve server replicas consistency. Second, replication can significantly increase update requests response time, since the processing of such requests generally requires to contact several, if not all server replicas. This issue is all the more important as in practice, reducing the probability of correlated failures often requires that server replicas be located in geographically diverse locations. As [3] showed it from a real case study, machines crashes are correlated within a given Internet segment, and network partitions are not rare, making it necessary to spread server replicas over several Internet segment that are well apart to actually increase overall availability and prevent the server from being totally disconnected from its clients.

Quorum systems are well-known tools that address the first issue. Informally, a quorum system is a collection of subsets of server replicas, every pair of which intersect. Thanks to the intersection property, each subset - namely a quorum - can act on behalf of the whole replicas group, which reduces server replicas load and decreases the number of messages needed. Similarly, overall availability is enhanced, since a single quorum is sufficient for the server to operate. These advantages were early recognized and formalized into quality metrics, which are used to compare various quorum constructions with one another, as described in [14].

Given these good properties, we may now ask whether quorum systems can also address our second concern about response time. More recent work has begun exploring this issue: Fu [6] first formalized the problem of minimizing quorum access time by introducing the notions of *mean-delay* and *max-delay* for quorum systems. Fu proposed polynomial-time algorithms to find max-delay optimal and mean-delay optimal quorum constructions for systems with specific topologies, such as trees and rings. Later, Tsuchiya et al. [19] extended Fu's work by proposing algorithms for finding max-delay optimal quorum constructions that apply to systems with arbitrary topologies. However, these works focus on quorum construction while in fact overall response time depends on the algorithm that is used as much as it depends on the quorum system itself. Also, to our knowledge, no study has so far compared the response time of a state-machine replication algorithm with that of its quorum-based counterpart.

In this paper, we explore the issue of improving response time of replicated service by using quorum systems, and we propose both a quorum construction and a quorum-based state-machine replication algorithm to that end. The quorum construction is especially suited for servers replicated over computing grids, as it takes advantage of the natural two-level hierarchy of the underlying network. The replication protocol derives from the standard quorum-based data replication algorithm, and operates in only two phases in failure-free cases to achieve low response time. Although very simple, this quorum construction and replication algorithm combined together prove to provide lower response time than a regular active replication algorithm.

The remainder of this paper is organized as follows. Section 2 provides background information and exposes the models we use at application, network and system levels. Section 3 presents the quorum construction that we designed to achieve low response time. Section 4 describes the replication algorithm we implemented to achieve quorum-based state machine replication. Section 5 presents performance measurements in various test configurations, compared to those of a standard active replication protocol. Section 6 concludes.

2. Background

2.1. Application Model

We consider a general client-server model. For the sake of availability and fault-tolerance, servers are replicated on several nodes, spread in geographically diverse locations. We assume that clients accessing a given server are uniformly distributed in space and direction with respect to server replicas.

We model a server as a deterministic state machine. Server workload is categorized into *write requests*, whose processing change the server state, and *read requests*, whose processing leaves the server state unchanged. We assume that the workload submitted to servers mainly consists of write requests.

2.2. Network model

Clients and server replicas are spread on a grid, which we view as a two-level hierarchy, as shown in Fig. 1: the lower level is formed by nodes grouped into LANs; the upper level consists of LANs interconnected through long haul networks.

This hierarchical view models two properties regarding communication delays. First, communication delays between two nodes only depend on the LANs that the nodes belong to. Second, local communication delays (within a LAN) are very short and rather constant, whereas distant communication delays (between two LANs) are long and subject to jitter.

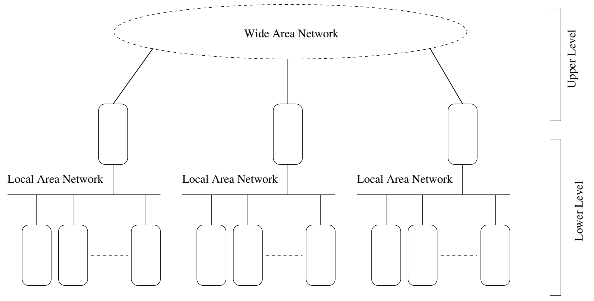

Figure 1. Two-level network hierarchy

2.3. System model

Failure model. We assume that server replicas and clients can experience crash-failures, that is, they may definitively stop operating at some random point in time. We assume that the network may delay or lose messages that clients and servers exchange, but FIFO order is preserved between any pair of peers.

Synchrony model. We consider the model of partial synchrony proposed in [5], which stipulates that, for every execution, there are bounds on process speeds and on message transmission times, although these bounds are not known and they only hold after some unknown time. This model allows us to implement an *eventually-perfect* failure detector [4], which in turn can be used to solve the consensus problem.

3. Quorum Construction

Given a server and the set of its replicas, our goal is to design a quorum construction that minimizes the server response time for every client, wherever it is located. Intuitively, the construction must be such that for every client, there exists a quorum that contains only the server replicas that are closest to the client. We first review existing quorum constructions and their relevance to our problem, and then we describe our solution and its rationale.

3.1. Common Structures

Quorum constructions can be divided into two categories. Majority (also named voting) quorum systems, whether simple as in [18] or weighted as in [7], were introduced first and make up the first category. In theses systems, each server replica is assigned a number of votes, and any set of replicas gathering a majority of the sum of votes is by construction a quorum. Although simple, voting quorum systems are attractive because they have the best availability. On the other hand, they also exhibit a large quorum size which impairs performance gains in terms of load and message complexity.

Then, [13] showed that not all quorum systems can be reduced to voting quorum systems, which led to the design of new quorum constructions. All of these constructions fall in the second category in that they rely on a virtual geometrical arrangement of server replicas to build intersecting sets. Thus, quorum systems can be built using finite projective planes [10], two-dimension grids [1] or trees [2], to cite a few of them. These quorum systems have a smaller quorum size than voting quorum systems, and some sophisticated constructions, such as crumbling walls [16], have optimal load and optimal availability with respect to their quorum size.

In addition, quorum constructions of both categories can be combined to form hierarchical quorum systems, as described in [8]. The goal of hierarchical constructions is to reduce quorum size while maintaining fair availability, or conversely, increase availability while maintaining fair quorum size.

We observe that constructing quorums of the first category only requires to gather a given number of server replicas (or a given number of votes in the most general case), whatever these replicas are. This makes it easy to use some additional criterion, such as communication delay, when selecting the server replicas that make up a quorum. By contrast, quorums of the second category are constructed by using a well-defined rule, which prevent server replicas from being freely chosen according to such additional criterion.

3.2. Solution

The solution we adopt is a two-level majority quorum system mapped on the hierarchy of the underlying network. Given a server replicated on several LANs and on several nodes in each LAN, a quorum is built by considering a (tie breaking) majority of the LANs the server is replicated on, and gathering a (tie breaking) majority of server replicas in each of these LANs. This construction ensures that any two quorums intersect at at least one node, while keeping the size of the intersection as small as possible.

Fig. 2 shows an example of such construction for a server replicated on three LANs. Server replicas are located on nodes 11 and 13 of LAN 1, on nodes 21, 22 and 23 of LAN 2, and on node 33 of LAN 3. The figure depicts two quorums: the first quorum spans LANs 1 and 2, and contains the server replicas hosted by nodes 11, 13, 21 and 22. The second quorum spans LANs 2 and 3, and contains the server replicas hosted by nodes 22, 23 and 33. The two quorums intersect at node 22.

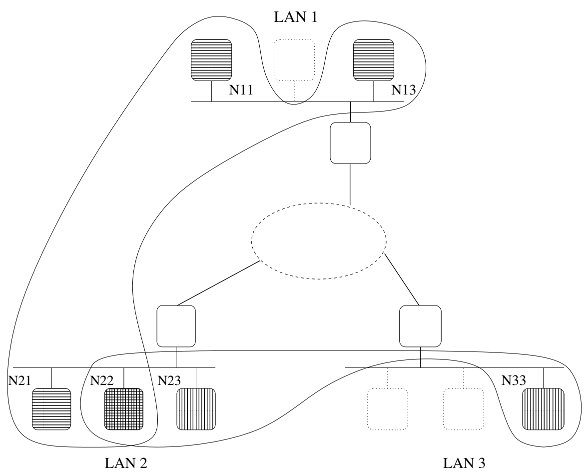

Figure 2. Two-levels majority quorum system

The reason for this construction is twofold. First, choosing the majority rule at the upper level of the hierarchy allows to select the LANs that are closest to any given client when constructing a quorum, thus ensuring optimal access time over all quorum constructions. Second, using a quorum construction to further split server replicas at the LAN level allows to build quorums with a smaller size. We again choose a majority construction at this level because in practice the number of server replicas per LAN does not exceed a few units, making it unnecessary to resort to complex quorum constructions with a small quorum size.

4. Replication Algorithm

The state-machine replication algorithm we propose combines the standard quorum-based data replication algorithm [11] and Nesterenko's quorum-based mutual exclusion algorithm [15]. We view the server state as a piece of data that clients update when submitting a request, and we use the quorum-based data replication algorithm to keep track of state updates accross requests. As client requests may be interleaved, we use the quorum-based mutual exclusion algorithm to atomically read and write back the server current state in some quorum and thus ensure *one-copy serializability*.

Data replication algorithm. Each data replica value is tagged with a version number denoting its freshness. In order to read data, a client queries some quorum to get a set of value/version number pairs, and retains as the current value of the data the one with the highest version number. In order to write data, a client performs two steps: first, it read the set of value/version number pairs from some quorum, and chooses a new version number greater than any of the read version numbers; then it updates all of the data replicas in the quorum to the new value and the new version number.

Mutual exclusion algorithm. Nesterenko's algorithm is a permission-based algorithm that derives from Maekawa's [10]. In order to obtain the lock, a client sends a *lock* message to some quorum, and then waits for all server replicas in the quorum to send back a *granted* message. In order to release the lock, the client simply sends a *release* message to the same quorum it obtained the lock from. To prevent dead-lock from occurring when concurrent lock requests are submitted, each request is timestamped, and the algorithm ensures through the use of *inquire* and *yield* messages that concurrent lock requests are granted in timestamp order.

Algorithms composition. We observe that both algorithms operate in two phases in the general case, and that these phases have matching timing. We can therefore merge the two algorithms by overloading the meaning of messages, instead of combining them sequentially. This results in a concise two-phase state-machine replication algorithm.

In the remaining of this section, we first present how quorums are selected before submitting a request, then we detail write and read requests processing in failure-free cases, and finally we examine the handling of failures.

4.1. Dynamic Quorum Selection

Quorum selection is performed on the client side and occurs every time a client sends a request to the server. The selection process is based on local data that the failure detector instance running on each client node dynamically computes and keeps up-to-date. These data consist in network delays between the client and each LAN hosting server replicas, and the estimated state (*up* or *down*) of each server replicas. Using this information, a quorum is contructed by selecting LANs that are closest to the client and by considering only those of the server replicas that are reported to be *up*. Within each LAN, server replicas are chosen at random to ensure load balancing. Ref. [9] shows that this approach is the most efficient in terms of response time and practical availability when using a replication protocol that locks and unlocks replicas.

4.2. Write Request Processing

Outline. Fig. 3 outlines the processing of a write request, which requires two phases and four messages: WRITE, STATE, APPLY, RESULT. For each message, the figure indicates in brackets its meaning regarding the data replication protocol and the mutual exclusion protocol, respectively.

The client first sends a WRITE message to each server replica in the quorum that it has selected. The WRITE message contains the server-level request to process, as well as a lock request, and the timestamp that uniquely identifies it. When server replicas receive the WRITE message, they process the embeded lock request according to the mutual exclusion algorithm. When a server replica eventually grants access to the client, it sends back a STATE message, and considers itself locked for this client. The STATE message contains the current state of the replica, along with its version number, as well as the grant reply bearing the lock request timestamp. When the client has collected all STATE messages from the quorum, it retains as the current server state the one bearing the highest version number.

The client then sends back an APPLY message to the quorum, which contains the current server state and its version number, along with a release request. When a server replica receive an APPLY message, it installs the specified state as the current server state and processes the request against that state. Then the server replica unlocks itself and sends back a RESULT message, which only contains the server-level reply. Finally, the server replica proceeds to the next pending WRITE request, if any.

Concurrent writes. Fig. 4 details the protocol operation in case of concurrent write requests sent by two clients, C1 and C2. C1 interacts with the quorum of three server replicas S1, S2 and S3, and C2 interacts with a second quorum, not depicted, which intersects with the first quorum on replica S2. While S2 is engaged in C1's write, it receives at time $t1$ a WRITE message from C2, which we assume bears a lower timestamp than C1's write. In order to avoid a potential dead-lock, S2 therefore tries to recall the permission it granted to C1 by sending it an INQUIRE message. When C1 receives the INQUIRE message, it is still in the first phase of its write, that is it has not collected all STATE messages yet. Therefore, C1 then sends back an YIELD message to S2, forgets about the STATE message S2 has just sent, and waits for S2 to send it again. When C2 completes its write, S2 sends to C1 a new STATE message, reflecting C2's write, and C1 proceeds as normal.

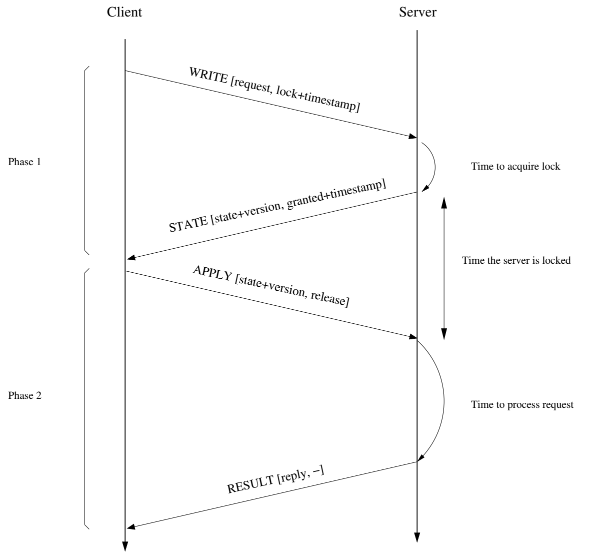

Figure 3. Replication protocol overview - Write request

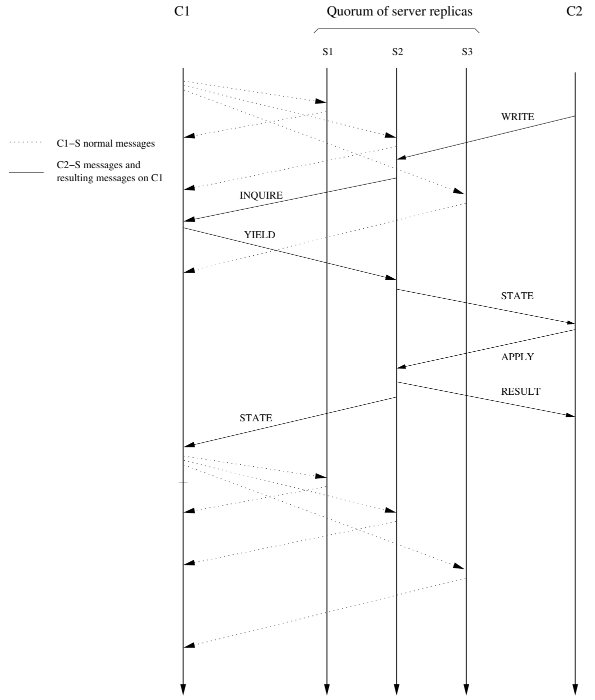

Figure 4. Client interacting with a server quorum - Concurrent writes

4.3. Read Request Processing

By definition, the processing of a read request does not change the server state. Consequently, it is not necessary to atomically read and write back a modified version of the state in some quorum, which eliminates the need of the locking mechanism used for write requests. The processing of read request thus only takes one phase and several read requests can execute concurrently with a write request, improving both read requests response time and overall request throughput.

4.4. Failures Handling

The liveness and safety of the replication algorithm in the presence of failures rely on the eventually-perfect failure detector that runs on every node. However, an important point to consider is that the failure detector may make mistakes and erroneously report a client or a server replica as having failed.

Server replicas failure. When a server replica fails during the processing of a write request, the liveness of the algorithm is ensured by the failure detector, which eventually reports the failure to the client. The client then cancel the current write by sending a specific CANCEL message to every server replicas in the quorum, and repeatedly retries the same write operation on successive quorums until the operation completes successfully. Assuming there exists at least one live quorum, sucessful completion is guaranteed to eventually occur because the number of possible quorums is finite, and, by definition, the failure detector will eventually stop reporting false failures.

Client failure. When a client fails during a write operation, some server replicas may remain locked waiting for the APPLY message that indicates which state to use for the write. The safety of the algorithm requires that these replicas do not unlock themselves until they acquire the most up-to-date server state. Thus, when the failure detector reports a client as having failed, server replicas start a recovery procedure in order to determine the newest state to apply. This procedure consists in executing the consensus algorithm described in [5] among all server replicas, each server replica proposing its current state. If the recovery procedure was launched because of a false detection of the client failure, the server replica replies with an error code to the APPLY message that they eventually receive, and the client restarts the current write operation.

5. Experimental Results

The quorum construction and the replication algorithm described in this paper have been implemented as part of the Dynamic Agent Replication eXtension (DARX) platform [12]. DARX is a framework for designing fault-tolerant

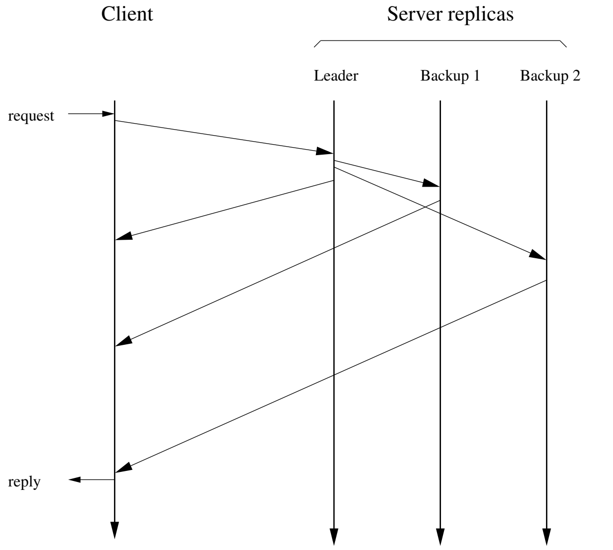

Figure 5. Active replication protocol

agent-based applications, which targets massive agent systems deployed over long haul networks. It readily provides application developpers with a regular active replication protocol, which we use as a reference point in our experiments. This protocol implements a leader/backup scheme in which the leader replica serializes incoming requests before forwarding them to backup replicas, as shown in Fig. 5.

5.1. Experiments Setting

Application. For the purpose of the performance tests, we use a dummy application, which consists of a single replicated server, accessed by several clients. The software is implemented in Java and executed using Sun's Java SDE, version 1.4.1, and RMI is used for communication support.

The server workload only consists of write requests. In order to generate heavy load conditions, each client repeatedly sends a request to the server without any intervening delay. The server replies to requests without delay, so that the measured reponse time is exactly the time induced by the replication protocol and network communication delays.

Network simulation. To be able to perform experiment on a grid, we use a network simulation environment running on the hardware platform shown in Fig. 6. In this architecture, each computer acts as a LAN of the grid, whereas each of the Java VMs running on a computer implements a node of the relevant LAN. This is achieved by means of the Ipnat and DummyNet [17] software installed on the router, which are used to partition the local network and to add message transmission delays.

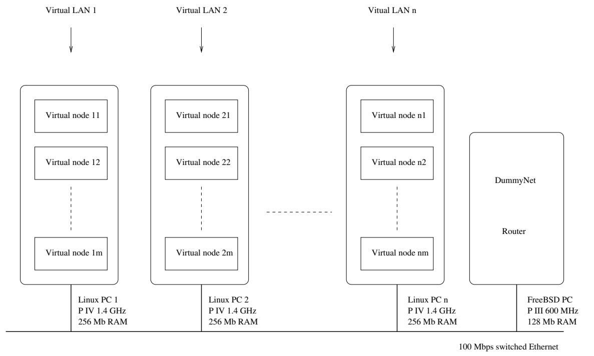

Figure 6. Test Platform

We emulate a fictitious configuration of five LANs, which are pair-wise connected through 100ms-delay communication links, except LANs 1 and 5, which must use one of the other LANs as an intermediate hop, resulting in a 200ms communication delay between them. Each LAN features six nodes, numbered *ij*, where *i* is the number of the relevant LAN and *j* is the number of the node within the LAN. Nodes x2 through x6 host server replicas, whereas nodes x1 host clients; when using the active protocol, the server leader replica is located on node 12.

Experiments protocol. To cover a wide range of situations, we evaluate the performance of the two replication protocols under light and heavy load, in various clients and server configurations, although always in failure-free cases.

A test run consists in every client sending from 100 to 300 write requests to the server, depending on the number of clients. Reported response times are measured on the client side, at the application level, and averaged over all clients of a given run, and then over at least three runs of the same type.

5.2. Results and Discussion

Light load. In the first part of the experiments, we measure reponse time under light load by having a single client query the server. The client is located on node 11, and the server is successively replicated over five LAN configurations, named LC1 through LC5. Configuration LC1 comprises only LAN 1, LC2 comprises LAN 1 and 2, LC3 comprises LAN 1, 2 and 3, and so forth. For each of these LAN configuration, we measure reponse time with 1, 3 and 5 server replicas per LAN, the total number of server replicas ranging from 1 to 25.

Fig. 7 plots the average response time under light load for the active and quorum protocols, for each of the server configurations. As expected, response time is driven by net-

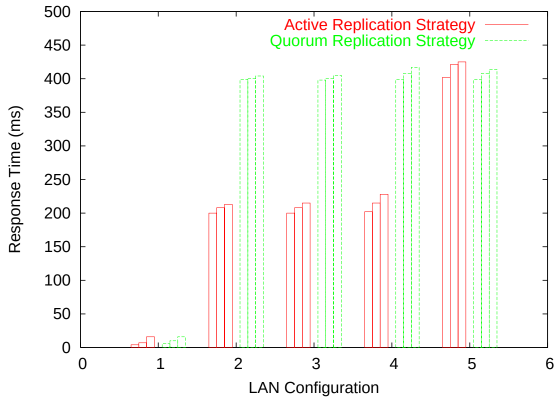

Figure 7. Average response time with 1 client for various server configurations

Figure 8. Average response time with 5 server replicas under increasing load

work delays when inter-LANs communication are involved, and it is only slightly affected by the number of server replicas per LAN. Fig. 7 also shows that for configurations LC2 through LC3, the response time of the quorum protocol is aproximately twice as high as that of the active protocol. This illustrates an important difference between the two protocols: on one hand, the active protocol involves all of the server replicas when processing a request, but this processing only requires one request-reply delay; on the other hand, the quorum protocol only involves a quorum of server replicas, but the interaction requires two request-reply delays. If we call *diameter* of a set of server replicas the maximum communication delay between any two elements of the set, this means that the diameter of the quorums must be at least twice as small as the diameter of the replication group for the quorum protocol to exhibit lower or equal response time than that of the active protocol. This condition is met in configuration LC5 - the diameter of the replication group is 200ms and the diameter of every quorum is 100ms - and we can see that the response times of the two protocols have similar values.

However we can see that network delays, although predominant, are not the only factor that influence response time, since we notice that the response time of the quorum protocol is actually slightly lower than that of the active protocol in LAN configuration LC5, for the same number of server replica per LAN. The reason for this difference, which is even more noticeable under heavy load conditions, is given below.

Heavy load. In the second part of the experiment, we measure response time under heavy load, when the server is replicated over LAN configuration LC5. Clients are uniformly distributed over nodes 11 through 51, and we create increasing load by increasing the number of clients per LAN, the total number of clients ranging from 5 to 25. We report two sets of figures: one with a single server replica per LAN, and the other with 5 server replicas per LAN.

Fig. 8 plots the average response time as a function of the number of (simulaneous) clients in server configuration LC5, with one server replica per LAN. As expected, the figure shows that the response time increases linearly with the number of clients for both protocols: on top of network delays, the response time now also reflects the time a request has to wait for preceding requests in the waiting queue. However, we notice that the response time of the quorum protocol is approximately 50% lower than that of the active protocol for the full range of clients number. As mentioned above, this result cannot be explained if we only consider network delays: it is actually the consequence of the load that each protocol places on server replicas. While the quorum protocol evenly distributes the load over all server replicas, the active protocol stresses the leader replica, which receives and dispatches all incoming requests and thus constitutes a bottleneck.

Fig. 9 reports the same experiment as Fig. 8, but this time with five server replicas per LAN. We can see that the response time of the active protocol is approximately the same than in the previous experiment. However, we notice that while the response time of the quorum protocol is still 20 to 25% lower than that of the active protocol, it has increased 50 to 75% compared to the previous experiment. The reason is that the number of conflicts between requests increases with the number of server replicas, and so does the number of INQUIRE and YIELD messages needed to resolve them.

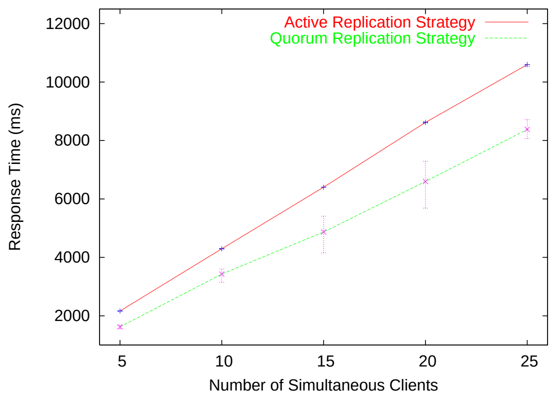

Figure 9. Average response time with 25 server replicas under increasing load

6. Conclusion

We have studied in this paper the use of quorum systems as a means to improve the response time of network services that are replicated and accessed on a wide scale. The quorum construction we have proposed is a two-level majority quorum system that maps onto the natural two-level hierarchy of grids' underlying network. While this construction features fair load and availability, its quorums have optimal access time over all possible quorum constructions. The state-machine replication algorithm we have designed merges a quorum-based mutual exclusion algorithm with a quorum-based replication algorithm and operates in only two phases in failure-free cases. By performing additionnal steps, this algorithm can recover from the crash of clients and servers in Chandra's partially synchronous model, which models the behaviour of wide-area networks.

We have developped a real implementation of this quorum construction and replication algorithm, and we have compared it to a regular active replication algorithm in various server and client configurations. We have identified the necessary conditions for our quorum-based replication algorithm to compete with the active replication algorithm, and we have shown that in such conditions, our algorithm outperforms by 20 to 50% the active replication algorithm.

References

[1] D. Agrawal and A. E. Abbadi. Exploiting logical structures in replicated databases. *Information Processing Letters*, 33:255–260, 1990.

[2] D. Agrawal and A. E. Abbadi. The generalized tree quorum protocol: An efficient approach for managing replicated data (corrigenda). *ACMTDS: ACM Transactions on Database Systems,*, 18, 1993.

[3] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *Symposium on Fault-Tolerant Computing*, pages 26–35, 1996.

[4] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proc. of Int'l Conf. on Dependable Systems and Networks*, Washington D.C., USA, June 2002.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[6] A. W.-C. Fu. Delay-optimal quorum consensus for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(1):59–69, 1997.

[7] D. K. Gifford. Weighted voting for replicated data. In *Proc. of 7th Symp. on Operating Systems Principles*, pages 150–162. ACM, december 1979.

[8] A. Kumar. Hierarchical quorums consensus: A new algorithm for managing replicated data. *IEEE Transactions for Computers*, 40(9):996–1004, 1991.

[9] M. Liu, D. Agrawal, and A. Abbadi. On the implementation of the quorum concensus protocol. In *Parallel and Distributed Computing Systems.*, 1995.

[10] M. Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, 1985.

[11] D. Malkhi. Quorum systems, 1999.

[12] O. Marin, P. Sens, J.-P. Briot, and Z. Guessoum. Towards adaptive fault-tolerance for distributed multi-agent systems. In *Proc. of European Research Seminar on Advances in Distributed Systems*, pages 195–201, May 2001.

[13] H. G. Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):481–860, 1985.

[14] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.

[15] M. Nesterenko and M. Mizuno. A quorum-based self-stabilizing distributed mutual exclusion algorithm. *Journal of Parallel and Distributed Computing*, 62(2):284–305, 2002.

[16] D. Peleg and A. Wool. Crumbling walls: A class of practical and efficient quorum systems. *Distributed Systems*, 10(2):120–129, 1997.

[17] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[18] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy database. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[19] T. Tsuchiya, M. Yamaguchi, and T. Kikuno. Minimizing the maximum delay for reaching consensus in quorum-based mutual exclusion schemes. *IEEE Transactions on Parallel and Distributed Systems*, 10(4):337–345, 1999.