Chasing the FLP Impossibility Result in a LAN or How Robust Can a Fault Tolerant Server Be?[*]

Péter Urbán[†] peter.urban@epfl.ch
Xavier Défago[‡] defago@jaist.ac.jp
André Schiper[†] andre.schiper@epfl.ch

[†]*École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland*

[‡]*Japan Advanced Institute of Science and Technology (JAIST), 1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan*

Abstract

Fault tolerance can be achieved in distributed systems by replication. However, Fischer, Lynch and Paterson have proven an impossibility result about consensus in the asynchronous system model, and similar impossibility results exist for atomic broadcast and group membership. We investigate, with the aid of an experiment conducted in a LAN, whether these impossibility results set limits to the robustness of a replicated server exposed to extremely high loads.

The experiment consists of client processes that send requests to a replicated server (three replicas) using an atomic broadcast primitive. It has parameters that allow us to control the load on the hosts and the network, as well as the timeout value used by our heartbeat failure detection mechanism. Our main observation is that the atomic broadcast algorithm never stops delivering messages, not even under arbitrarily high load and very small timeout values (1 ms). So, by trying to illustrate the practical impact of impossibility results, we discovered that we had implemented a very robust replicated service.

1. Introduction

Fault tolerance in distributed systems is often achieved by replicating components or services. Although replication is an intuitive and readily understood concept, its implementation is difficult. The states of all replicas have to be kept consistent, which can be ensured by a specific replication protocol [1, 2]. A replication protocol is typically implemented using group communication primitives, e.g. atomic broadcast [3].

However, Fischer, Lynch and Paterson have proven an impossibility result for consensus in the asynchronous system model [4] (FLP impossibility result). The impossibility result also applies to atomic broadcast [5]. A similar impossibility result has been established for group membership [6], another problem related to replication. These impossibility results set a limit on the level of robustness that a replicated service can achieve. However, these theoretical results are largely ignored in practice: practitioners disregard them as results irrelevant to real systems. The reason is that real systems exhibit some level of synchrony and consequently, the implications of the impossibility result to real systems are difficult to see.

On the other hand, no paper in the literature refers to practical experiences in which the implementation of replication is exposed to extremely high loads. How robust can a system be under these conditions? Do high loads actually prevent the system from making progress (as stated by the FLP impossibility result), and so limit the robustness of the system? How robust can a fault tolerant server be?

To answer these questions, we designed an experiment for a Local Area Network (LAN). It consists of client processes that send requests to a replicated server using an atomic broadcast primitive. The experiment has a parameter which specifies the load on the system (the rate of requests coming from the clients). The other parameter is the timeout used by our heartbeat failure detectors. The frequency of heartbeats is kept proportional to the timeout value: the smaller the timeout is, the faster the failure detection. Our intuition was that, as we decrease the timeout (and increase the frequency of heartbeats), the atomic broadcast algorithm would stop making progress at some point in its execution. Interestingly, our experiment showed that this was not the case: up to very small timeout values (i.e., 1 ms) and for arbitrarily high load conditions, the atomic broadcast algorithm never stops delivering messages. Thus, by challenging our implementation with high loads and small failure detection timeout values, we discovered that we had imple-

[*]Research supported by a grant from the CSEM Swiss Center for Electronics and Microtechnology, Inc., Neuchâtel.

mented a replicated service which is extremely robust in a LAN.

The paper is structured as follows. Section 2 introduces the algorithms used in the experiment, and Section 3 describes the environment. Section 4 explains how we tested the robustness of the replicated server. Section 5 presents results of the experiment, and we conclude with a discussion of these results in Section 6.

2. The experiment

Active replication. Our experiment consists of a replicated server and several clients. Each client repeatedly sends a request to the replicated server and waits for a reply. The server is replicated by means of *active replication* (also called *state machine approach*) [1]. In active replication, clients use atomic broadcast to send their requests to the replicas. Atomic broadcast ensures that all server replicas receive the client requests in the same order. Upon reception of a request, each server replica performs the same deterministic processing (in our case, writing a number to a file) and sends back a reply to the client. The client waits for the first reply, and ignores all further replies to the same request.

Atomic Broadcast. We use the Chandra-Toueg atomic broadcast algorithm [5]. The algorithm solves atomic broadcast by executing a sequence of consensus, where each consensus decides on a set of messages to be delivered. The atomic broadcast and the consensus algorithms are proven correct in the asynchronous system model with the failure detector $\diamond\mathcal{S}$ and a majority of correct processes [5].

Consensus. For consensus, we use the algorithm proposed by Mostéfaoui and Raynal [7] which improves the early consensus algorithm [8]. It is based on the rotating coordinator paradigm. Processes proceed in consecutive asynchronous rounds (*not* all processes are necessarily in the same round at a given time). In each round a predetermined process acts as the coordinator. The coordinator proposes a value for the decision. A round succeeds if a decision is taken in that round; if some process decides (and does not crash) it forces the other processes to decide, and thus the algorithm is guaranteed to terminate shortly. A round might fail when its coordinator crashes, or when its coordinator, while correct, is suspected by other processes. Consensus might terminate in a single round, i.e., the first round can already succeed. Some runs might require more rounds, though; in general, the more often the coordinator is suspected, the more rounds the algorithm will take to terminate.

Failure detection. The consensus algorithm is implemented on top of a failure detection mechanism with two parameters T_h and T. Each process sends heartbeat messages to all other processes with a period T_h. Process p suspects process q whenever it has not received any message from q (heartbeat or application message) for a period longer than T.

3. Environment and implementation issues

The experiment was run on a cluster of 15 PCs running Red Hat Linux 7.0 (kernel 2.2.16). The hosts have Pentium III 766 MHz processors and 128 MB of RAM, and are interconnected by a 100 Base-TX Ethernet. Three server replicas were used (such that the algorithms tolerate one process crash). Each server replica ran on a different host, while the remaining 12 hosts were used for the clients (more than one client per host). The algorithms were implemented in Java (Sun's JDK 1.3.0) on top of the Neko framework [9].

As our replicated server was supposed to work under extremely high loads, we had to be very careful about the choice of protocols and the flow control strategy to avoid distributed deadlocks and buffering problems. These issues are described in detail in the extended version of this paper [10].

4. How robust is our system?

The correctness of a distributed algorithm has two aspects: *safety* ("nothing bad ever happens") and *liveness* ("good things eventually happen"). We call an algorithm *robust* if it is both safe and live, even when exposed to extremely high loads. The atomic broadcast algorithm that we chose [5] is safe under any conditions. Therefore, robustness is related to liveness in our experiment: is our atomic broadcast always able to deliver messages? The goal of our experiment is to find an answer to this question. The experiment has parameters which influence the load conditions of the system. For various settings of these parameters, we ran the experiment and checked whether the atomic broadcast algorithm was live. This section discusses the parameters of the experiment, as well as the method used for verifying liveness.[1]

Parameters of the experiment. We classify the parameters of our experiment into two categories: (1) *application parameters*, over which the implementor of the server has no control, and (2) *system parameters*, over which the implementor of the server has full control.

An application parameter influences the load on the network and the hosts. Our application parameter is r [requests/s], the rate of requests coming from the clients; a large r generates a high load on the network and on the

[1]Note that we do not emulate process crashes in our experiment. This would primarily give information on the fault tolerance characteristics of the atomic broadcast algorithm, which are well understood [5]. The robustness of the algorithm is a major issue even if no crash occurs.

replicated server. In order to demonstrate that our system is robust, we have to show that our replicated server works for any setting of r.

Our system parameter is T, the timeout value for the failure detector. The time T_h between two consecutive heartbeat messages is set to $T/2$. Low timeout values yield frequent false suspicions, and high timeout values increase the reaction time of the algorithm to process crashes.

The robustness of our server can easily be increased by setting T very high, say to one minute. However, this would imply that the replicated server may block for a minute when a process crashes. We consider that such a behavior is unacceptable for a server replicated for high availability. For this reason, we explored how the replicated server behaves for small values of T.

Testing if the atomic broadcast algorithm can deliver messages. Given a setting of the parameters, how can we detect (1) if the atomic broadcast algorithm continues delivering messages forever or (2) if it will never deliver messages any more? The best that we can do is to detect conditions that allow us to conclude with some confidence that the behavior of the algorithm has stabilized. We use the following conditions to terminate a run of the experiment:

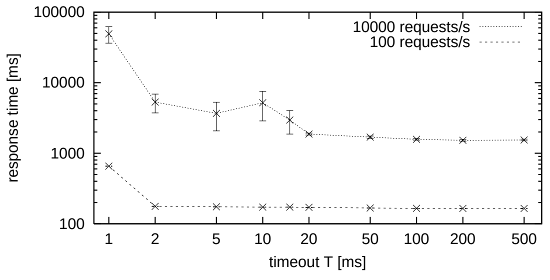
1. The clients have collected a certain number of replies (N) from the replicated server.
2. One instance of the consensus algorithm has not terminated after executing R rounds.

In every run of our experiment, one of these conditions is necessarily fulfilled. The values N and R were chosen sufficiently high to allow the system to stabilize (see [10] for details).

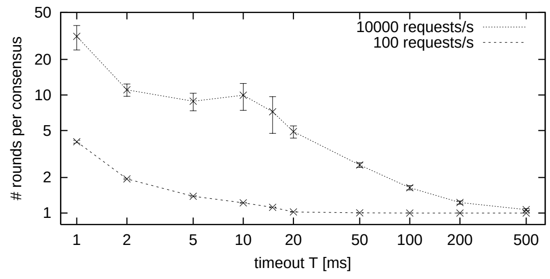
5. Results of our experiment

In spite of our expectations, we observed that the atomic broadcast algorithm works even under the most extreme conditions: a request rate that saturates the network (10 000 requests/s) and a very small timeout, approaching the resolution of the clock used (1 ms).

Due to lack of space, we can only present two representative sets of results, for two different request rates: 100 requests/s and $10\,000$ requests/s. 100 requests/s is a rate representing normal operation, well below the capacity of the server (420 requests/s). At $10\,000$ requests/s, the network is saturated with requests (as client hosts can send at most $7\,000$ requests/s). For these rates and different timeout values, we measured (1) the average response time and (2) the average number of rounds per consensus (Fig. 1). The characteristics of the "response time" curve and the "consensus rounds" curve are rather similar; this is not surprising, as the number of rounds per consensus execution largely determines the response time. At high timeouts, the quantities are predictable and independent of the timeout. At low timeouts, both the response times and the number of rounds increase as the timeout decreases. This is due to the more and more frequent failure suspicions. Both the response time and the number of rounds are highly unpredictable: this is shown by the large confidence intervals. We found that even at low timeouts, most consensus executions take few rounds, but a few instances of consensus take a lot of rounds and thus increase the average significantly (Fig. 2).



(a) Response time



(b) Number of rounds to reach consensus

Figure 1. Performance of the replicated server (three replicas) for an extreme and a moderate request rate r vs. the failure detection timeout T.[1]

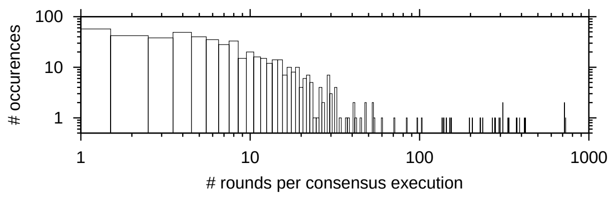


Figure 2. The distribution of the number of rounds per consensus execution, for $r = 10\,000/s$, $T = 1$ ms.

6. Discussion

The experiment shows that our replicated server is extremely robust (Section 5). The server is robust because

[1]Each point shows mean values from 100 experiments, with the 90% confidence interval.

the underlying atomic broadcast algorithm is robust, and in turn, the atomic broadcast algorithm is robust because the underlying consensus algorithm always terminates. This can be explained as follows.

Recall from Section 2 that processes proceed in rounds in the consensus algorithm. In each round, a predetermined process acts as the coordinator. A successful round is a round in which a decision is taken. A round might fail because its coordinator may be suspected by other processes. Therefore the more often suspicions occur, the more rounds a consensus algorithm takes until it decides. However, Figure 1(b) shows that even though consensus executions may take a large number of rounds (30 rounds on average even for the smallest values of T and the highest values of r) and the number of rounds is rather unpredictable, each consensus execution terminates nevertheless: the longest we observed had 729 rounds. By analyzing logs of messages produced during the experiment, we were able to understand the reasons for this. We present our arguments in three steps:

1. The consensus algorithm tries to decide repeatedly, in every round. Therefore, if the algorithm does not terminate, the failure of a round (i.e., the absence of decision in that round) must occur with high probability. We shall argue that this is not the case.
2. Out of our three processes, one is always late: it never participates actively in the algorithm. The reason is that the algorithm needs the cooperation of only two processes (this is why it tolerates one crash failure). Thus the process that finishes one consensus execution late is likely to finish all subsequent executions late.
3. The following scenario explains why unsuccessful rounds do not occur with high probability (Figure 3):

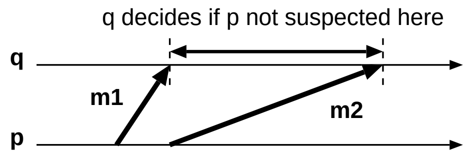


Figure 3. Consensus algorithm: scenario likely to lead to a decision. The late process is not shown.

(a) Process q is the coordinator of round r, and process p is the coordinator of round $r + 1$. The third process is the late one. This scenario likely repeats in every third round. The messages of the late process are late and do not influence the scenario (and are thus omitted in Fig. 3).

(b) Process p sends m_1 to q, and *immediately after* it sends m_2 to q. Process q waits for message m_1. The reception of m_1 is mandatory, i.e., q does not stop waiting for m_1 upon suspecting p.

(c) Upon the reception of m_1, process q waits (1) for message m_2 from p, or (2) until it suspects p. *If q receives m_2 before suspecting p, then q can decide.*

Application messages reset the timer of the heartbeat failure detector, hence the probability for q to suspect p before receiving m_2 is small. Consequently, in every third round (at least), the decision is likely to take place. Thus eventually, there is one round in which the coordinator decides, and forces the other processes to decide.

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys*, vol. 22, pp. 299–319, Dec. 1990.

[2] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed Systems* (S. Mullender, ed.), ACM Press Books, ch. 8, pp. 199–216, Addison-Wesley, second ed., 1993.

[3] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems* (S. Mullender, ed.), ACM Press Books, ch. 5, pp. 97–146, Addison-Wesley, second ed., 1993.

[4] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, pp. 374–382, Apr. 1985.

[5] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, pp. 225–267, Mar. 1996.

[6] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the impossibility of group membership," in *Proc. of the 15th Annual ACM Symp. on Principles of Distributed Computing (PODC'96)*, (New York, USA), pp. 322–330, ACM, May 1996.

[7] A. Mostéfaoui and M. Raynal, "Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach," in *Proc. of the 13th Int'l Symp. on Distributed Computing (DISC)*, (Bratislava, Slovak Republic), pp. 49–63, Sept. 1999.

[8] A. Schiper, "Early consensus in an asynchronous system with a weak failure detector," *Distributed Computing*, vol. 10, pp. 149–157, Apr. 1997.

[9] P. Urbán, X. Défago, and A. Schiper, "Neko: A single environment to simulate and prototype distributed algorithms," in *Proc. of the 15th Int'l Conf. on Information Networking (ICOIN-15)*, (Beppu City, Japan), pp. 503–511, 2001. http://lsewww.epfl.ch/Publications/ById/255.html.

[10] P. Urbán, X. Défago, and A. Schiper, "Chasing the FLP impossibility result in a LAN, or how robust can a fault tolerant server be?," Tech. Rep. DSC/2001/037, EPFL, Switzerland, 2001. http://lsewww.epfl.ch/Publications/ById/254.html.