

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. Similar impossibility results have been established for atomic broadcast and group membership, and should be as such relevant for implementations of a replicated service. However, the practical impact of these impossibility results is unclear. For instance, do they set limits to the robustness of a replicated server exposed to extremely high loads?

The paper tries to answer this question by describing an experiment conducted in a LAN. It consists of client processes that send requests to a replicated server (three replicas) using an atomic broadcast primitive. The experiment has parameters that allow us to control the load on the hosts and on the network and 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 val-

ues (1 ms). The result was surprising to us, as we expected that our atomic broadcast algorithm would stop delivering messages at such small timeout values. So, by trying to illustrate the practical impact of impossibility results, we discovered that we had implemented a very robust replicated service.

1 Introduction

A major problem inherent to distributed systems is their potential vulnerability to failures. Indeed, whenever a node crashes, the availability of the whole system may be compromised. However, the distributed nature of those systems provides the means to increase their reliability: distribution allows the introduction of redundancy in order to make the overall system more reliable than its individual parts. Redundancy is usually achieved by the replication of components or services. Although replication is an intuitive and readily understood concept, its implementation is difficult. Replicating a service in a distributed system requires that the states of all replicas of the service are kept consistent, which can be ensured by a specific replication protocol [Sch90, BMST93]. A replication protocol is typically implemented using group communication primitives, e.g. atomic broadcast [HT93].

However, Fischer, Lynch and Paterson have proven

*Research supported by a grant from the CSEM Swiss Center for Electronics and Microtechnology, Inc., Neuchâtel. A shorter version of this paper appeared in *Proc. of the 20th IEEE International Symposium on Reliable Distributed Systems (SRDS-20)*, October 2001.

an impossibility result for consensus in the asynchronous system model [FLP85], a result commonly known as the FLP impossibility result¹. The impossibility result also applies to atomic broadcast [CT96]. The impossibility of group membership — another problem related to replication — in asynchronous systems was also established [CHTCB96]. Formally, these impossibility results set a limit on the level of robustness that a replicated service can achieve. Practitioners, however, disregard these impossibility results, i.e., they consider them of no practical relevance. The reason is that real systems usually exhibit some level of synchrony, i.e., they are not exactly asynchronous. Consequently, the implications of the impossibility result to real systems are difficult to see, and these theoretical results are largely ignored in practice.

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. It is clear that if we use very slow failure detection, say with a timeout of one minute, our system could be extremely robust, as false failure suspicions would be avoided with a high probability. However, the behavior of our system in the case of a crash would be disastrous: with

¹An asynchronous system — which models a system with unpredictable CPU and channel loads — is a system in which there is no assumption neither on message communication delays nor on relative speeds of processes.

a timeout of one minute, it would take in the order of 1 minute to detect the crash, which means that the response time could be extremely bad (if the crash affects a process which has an important role at the moment of the crash). In order to avoid such robust, but badly performing systems, we were decreasing the failure detection timeout values. 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 (i.e., it always works). Thus, by challenging our implementation with high loads and small failure detection timeout values, we discovered that we had implemented a replicated service which is extremely robust in a LAN.

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 introduces the algorithms used in the experiment. Section 4 describes the environment and some features of the implementation. Section 5 explains how we tested the robustness of the replicated server. Section 6 describes in detail the results obtained in our experiments. Section 7 discusses these results. Finally, Section 8 concludes the paper.

2 Related work

To the best of our knowledge, no paper in the literature refers to practical experiences in which the implementation of replication is exposed to extremely high loads. Nevertheless, there are two papers [CF99, CTA00] which implicitly suggest ways of implementing extremely robust replicated servers, and support their arguments by performance evaluation results.

[CF99] introduces the timed asynchronous system model for distributed algorithms. This model, extended with what the authors call progress assumptions, allows them to solve consensus. Therefore, if the timed asynchronous model matches reality, one can build ex-

tremely robust replicated servers using algorithms developed for this model. The authors support the assumptions of their model by an extensive set of measurements performed in a LAN. They validate the assumptions of their core model even under high load. However, the progress assumptions (which make the model powerful enough to solve consensus) are validated only under moderate load (1/4 of the network capacity).

[CTA00] presents a failure detector based on heartbeats and proves that this algorithm is optimal (in terms of the quality of service measures defined in the paper and in the class of heartbeat failure detectors), for any kind of distribution for the message delays. The paper also gives an adaptive version of the algorithm that approximates the optimum even if the distribution of message delays is not known in advance or changes over time. The results are supported by analytical computations and a simulation study. Indeed, such a failure detector would be ideally suited to implement a replicated server which is extremely robust in a variety of environments and for a variety of loads. However, the paper assumes that message delays are *independent* random variables. This assumption is far from being true for two subsequent messages if the network is under high load (as we saw in our experiment when logging messages): if a message suffers a high delay, usually the next message suffers a comparably high delay as well.

3 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*) [Sch90, Sch93, Pol94]. 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 [CT96]. 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 $\diamond S$ and a majority of correct processes [CT96].²

Consensus. For consensus, we use the algorithm proposed by Mostéfaoui and Raynal [MR99] which is expressed in a very concise manner, and which improves the early consensus algorithm [Sch97]. Similarly to many other consensus algorithms, 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 relies on a failure detection mechanism implemented using heartbeat messages (Figure 1): each process periodically sends a heartbeat message to all other processes. Failure detection is parameterized with a timeout value T

²The atomic broadcast algorithm only has these restrictions because it uses the consensus algorithm.

and a heartbeat period T_h . Process p starts suspecting process q if it has not received any message from q (heartbeat or application message) for a period longer than T . Process p stops suspecting process q upon reception of any message from q (heartbeat or application message). The reception of any message from q resets the timer for the timeout T .

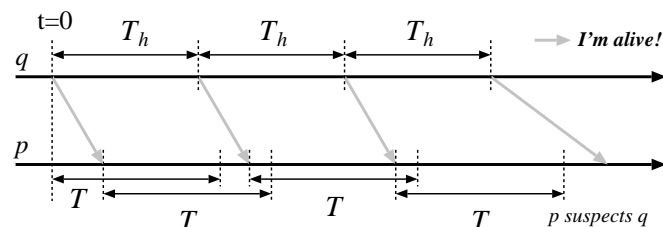


Figure 1. Heartbeat failure detection.

4 Environment and implementation issues

4.1 Environment

The experiment described in the previous section 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, as three is the minimum number of replicas for which the atomic broadcast algorithm used tolerates process crashes. Each server replica ran on a different host, while the remaining 12 hosts were used for the clients (there was more than one client per host). We had to use a lot of hosts, otherwise the client hosts turned out to be a bottleneck and thus we could not generate a sufficiently high rate of client requests. The algorithms were implemented in Java (Sun's JDK 1.3.0) on top of the Neko framework [UDS01].

4.2 Communication protocols

We have three types of messages with different delivery requirements in our system: (1) heartbeat messages, (2) client requests, as well as (3) messages between server replicas and replies to clients. We use

the UDP protocol for transmitting heartbeat messages, for the loss of a heartbeat message is not critical. The other messages need reliable transmission, therefore the straightforward choice is the TCP protocol (nevertheless, we chose UDP rather than TCP for client requests, for reasons discussed in the next paragraph). However, TCP has problems with extreme overload situations. In such situations, two hosts can be partitioned from each other for a long time, and TCP connections break, for a lot of retransmissions fail in a row. The number of times TCP tries to retransmit a packet is given by the parameter `tcp_retries2` of the Linux TCP implementation. We solved the problem by setting the parameter from the default value (15) to a very high value, for all the hosts involved.³

We could not use TCP for transporting client requests. To understand why, recall that the goal of our experiment was to investigate the behavior of our system under arbitrarily high loads. Therefore we had to avoid that flow control decreases the load on the system, i.e., the volume of client requests.⁴ In the context of our experiment (many clients on one host), the implication is that we cannot use a single TCP connection per host for the client requests: in this case, TCP's congestion control mechanism makes sure that the network never gets overloaded. We cannot use one TCP stream per client, either, for the servers would have to handle a huge number of simultaneous connections, more than the operating system allows. We have the same problem if we use one TCP connection for each request (and send the reply on another connection). The remaining choice for transporting client requests is UDP.⁵ This way, the network can be arbitrarily loaded with client requests.

³We could also write a protocol which re-establishes the connection and ensures that messages are delivered exactly once. But this amounts to re-implementing a major part of TCP's features.

⁴Otherwise, we would have constructed a controlled environment which includes the replicated service, *as well as its clients*. The high load scenarios we are interested in would not occur at all in such an environment.

⁵Retransmitting lost or dropped client requests is the responsibility of the client.

4.3 Flow control in the application

Flow control is an essential mechanism in distributed systems: it ensures that components do not receive more work than they can handle. Any non-trivial system needs flow control, but a lot of systems can rely on flow control offered by TCP. This was not sufficient for our system. We explain the reasons below and then present our flow control mechanism (implemented within the application layer).

Systems that rely on TCP use a send primitive that blocks whenever TCP's sending buffer fills up (e.g., because the receiver is slow). Blocking sends only work well for client-server interactions; they constitute a poor way of synchronizing more complex distributed systems. In our case, blocking sends led to deadlocks: under high load, all server replicas got blocked in their send operations and could not receive messages to resolve the deadlock. Using threads dedicated to receiving messages does not solve the problem. No deadlocks appear, but — depending on the exact implementation — the number of threads or the size of message queues continues growing until system resources are exhausted.

We solved the problem by adding an *outgoing message queue* in the application layer. Send operations which are non-blocking deposit messages in that queue, and a dedicated thread empties the queue and performs the TCP send operation. Without flow control, the outgoing message queues can still grow indefinitely and cause “out of memory” errors. Our (stop-and-go) flow control mechanism acts whenever the size of the outgoing message queue is above a threshold. When this happens, we disable the generation of (most) outgoing messages as follows:

- We disable the generation of heartbeat messages (but the timeout for suspecting processes does not change!). and
- We suspend the thread on the servers that receives client requests. This stops the generation of new outgoing messages. As a consequence, the UDP protocol which delivers client requests

will start dropping requests, and this eventually slows down clients, for they have to retransmit the dropped requests.

5 How robust is our system?

The correctness of a distributed algorithm has two aspects: *safety* (“nothing bad ever happens”) and *liveness* (“good things must 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 [CT96] 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.

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 [CT96]. The robustness of the algorithm is a major issue even if no crash occurs.⁶

5.1 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 , the rate of requests coming from the clients, i.e., the

⁶Note also that the FLP impossibility does not stem from the fact that crashes *do occur*, but from the fact that crashes *may happen* in an unanticipated manner at any point in the execution of the atomic broadcast algorithm, and that consequently, the algorithm has to be prepared for them.

number of requests per second.⁷ A large r generates a high load on the network and on the replicated server. The number of clients is sufficiently high to maintain any reasonable value of r , even if the server processes requests very slowly. In order to demonstrate that our system is robust, we have to show that our replicated server works for any setting of the application parameter 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 thus increase the time needed to solve consensus, and for the client the time to get the reply after sending the request. High timeout values increase the reaction time of the algorithm to process crashes.

As already mentioned in the introduction, 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.⁸ 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 .

5.2 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:

⁷Requests are generated by a Poisson process, thus we model independent requests. This is, however, not crucial to the experiment.

⁸Suppose that the coordinator of a round of the consensus algorithm crashes. At this moment, the other processes wait for either a message from the coordinator or that the failure detector starts suspecting the coordinator. With a timeout of one minute, the algorithm is blocked at this point for about one minute.

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 a certain number of rounds (R).

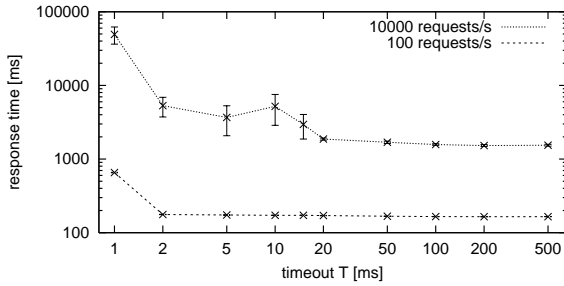
In every run of our experiment, one of these conditions is necessarily fulfilled. In case 1, we conclude that the algorithm was live in the current run: sending a request m using atomic broadcast eventually leads to the delivery of m , and thus to a reply to m . In case 2, the conclusion is that the algorithm was not live in the current run.

The values N and R should be chosen sufficiently high, to ensure that the behavior of the algorithm stabilizes. In the experiment, we used $N = 2500$ and $R = 1000$. $N = 2500$ was sufficient to ensure that each host participating in the experiment starts sending messages, and that at least 5 consensus are executed after the startup, for even the most unfavorable setting of the parameters r and T . $R = 1000$ was sufficient because we had no consensus that took 1000 rounds: the consensus algorithm was always live.

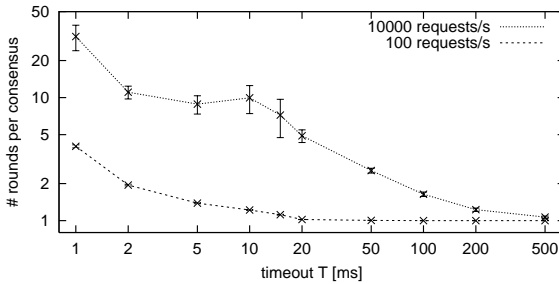
6 Results of our experiment

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). We present the detailed results of the experiment in this section and discuss those results in Section 7.

We performed measurements for a variety of client request rates. We only present two sets of results which are characteristic: one for 100 requests/s and one for 10 000 requests/s (Figures 2(a) and 2(b)). The rate of 100 requests/s is well below the capacity of the replicated server (which can process requests at a rate between 400 and 450 requests/s if heartbeats are sent infrequently). This rate corresponds to normal operation. Smaller rates give very similar results (with a different average response time). The other rate is 10 000



(a) Response time



(b) Number of rounds to reach consensus

Figure 2. Performance of the replicated server (three replicas) for an extreme and a moderate request rate r vs. the failure detection timeout T . Each point represents a mean value obtained from 100 independent experiments. The 90% confidence interval is shown.

requests/s. It is pointless to increase the request rate beyond this point because at this rate, the network is already saturated with requests.⁹ As for request rates between 100 requests/s and one for 10 000 requests/s, the observed behavior is in between the two extreme behaviors.

For both request rates and different timeout values, we measured two quantities: (1) the average response time (the time between the sending of a request and the reception of the corresponding reply, as seen by the client) and (2) the average number of rounds per consensus. The average response time is shown in Figure 2(a). The average number of rounds per consensus is shown in Figure 2(b). The characteristics of the “re-

⁹We measured that the clients can pass at most 7000 requests/s through the socket interface of their hosts, so the rate of requests which pass through the network cannot be higher than 7000 requests/s. Moreover, we are sure that our 12 client hosts are capable of achieving this rate: if we use just one single host with clients, it can send at a rate of 1450 requests/s.

sponse 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. In each curve, we can observe two kinds of behavior:

- for $r = 100/s$, the one behavior with a timeout value below $\bar{T} = 2$ ms, and the other behavior with a timeout value above \bar{T} .
- for $r = 10\,000/s$, the one behavior with a timeout value below $\bar{T} = 20$ ms, and the other behavior with a timeout value above \bar{T} .

At high timeouts ($T \geq \bar{T}$), the measured quantities are predictable and largely independent of the timeout: there is only a difference of 8% (moderate rate) and 30% (high rate) between the highest and the lowest response times. The average number of rounds is below 2 (except $r = 10\,000/s$ and $T = 20$ ms).

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, which were obtained from 100 experiments for each timeout and rate. 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. The distribution of the number of rounds is shown in Fig. 3, for the most extreme setting of parameters: $r = 10\,000/s$ and $T = 1$ ms (as Fig. 2(b) shows, this setting of parameters results in the highest number of rounds per consensus execution on the average).

Finally, note that we did not try to optimize the response times of the server (shown in Fig. 2(a)). Even under light load (5 requests/s), the average response time was 22.4 ms with a standard deviation of 5 ms (for a sample of 10 000 independent requests). Our goal was to evaluate the robustness of the server, and we could achieve this without optimizing the performance.

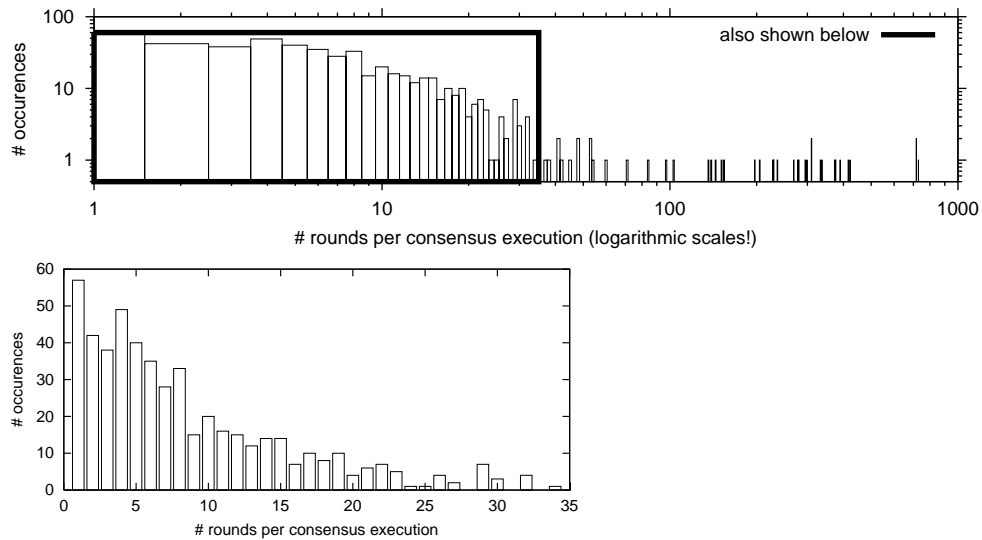


Figure 3. The distribution of the number of rounds per consensus execution, for $r = 10\,000/s$, $T = 1\text{ ms}$. 552 consensus executions are shown, coming from 100 independent experiments. The distribution of small round numbers is shown separately as well.

7 Discussion

The experiment showed that our replicated server is extremely robust. It worked under any conditions, even the most extreme ones: 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). In this section, we discuss why it proved to be so robust.

The replicated server is robust because the underlying atomic broadcast algorithm is robust. In turn, as the atomic broadcast algorithm uses a sequence of consensus algorithms to decide what messages it can deliver next (Section 3), the atomic broadcast algorithm is robust because the underlying consensus algorithm always terminates. This can be explained as follows.

Recall from Section 3 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 the consensus algorithm takes until it decides. The frequency of suspicions is

directly related to the failure detection timeout T , and indirectly to the load of the system, influenced by the client request rate r . We now examine how the system behaves as we decrease T .

If the T is high, consensus terminates in one round. This holds even if the system is loaded to the maximum extent, i.e., when client requests saturate the network ($r = 10\,000/s$). This is shown in Fig. 2(b), for $r = 100/s$ and $T \geq 20\text{ ms}$ and $r = 10\,000/s$ and $T \geq 500\text{ ms}$. The reason is that the coordinator is hardly ever suspected by the failure detector. This means that the coordinator can successfully send messages more often than at a frequency of $1/T$, as a failure detector stops suspecting a process whenever it receives a message from that process. This is not surprising: our Ethernet network strives to provide fair access to the transmission medium for each host on the network, and the result is that each host can successfully send a message every 500 ms.¹⁰

As we decrease T , suspicions get more and more frequent. With small timeout values, we expected that

¹⁰Actually, Ethernet is the LAN technology that is the least fair among all LAN technologies. FDDI, for example, guarantees a bound on the access time to the shared medium.

the coordinator of each round of the consensus algorithm would always be suspected, i.e., the consensus algorithm would forever proceed from one round to the next one without ever being able to decide. This is not the case: even for the smallest value for T and the highest possible load ($T = 1$ ms and $r = 10\,000$ ms in Fig. 2(b)) consensus executions take 30 rounds on the average, and the longest consensus execution we could find had 729 rounds. So, while consensus executions may take a large number of rounds and the number of rounds is rather unpredictable, each consensus execution terminates nevertheless. 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 4):
 - (a) Process q is the coordinator of round r , and process p is the coordinator of round $r + 1$. The third process is the slowest 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. 4).
 - (b) Process p sends m_1 to q , and *immediately after* it sends m_2 to q . Process q waits

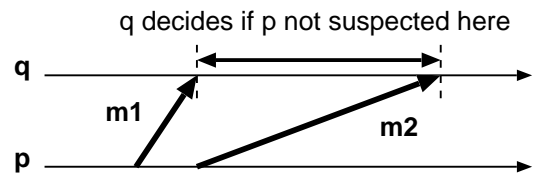


Figure 4. Consensus algorithm: q must suspect p before the reception of m_2 to prevent a decision in the current round. The late process is not shown.

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.

Our lowest setting for the failure detection timeout T was 1 ms. The question arises whether we could have lowered the timeout value and observe consensus executions that do not terminate. The answer is that we could have lowered T , and possibly observed consensus executions which do not terminate, but such small T values do not make sense in practice. The reason is the following. The motivation to decrease T is to speed up the reaction of the algorithm to crashes. The reaction time is difficult to define precisely, but it is certainly related to the performance of the replicated server from the client's point of view. Decreasing T further can improve the response time of the server by at most 1 ms in the case of a crash. This improvement is insignificant: recall from Section 6 that the best case (no crashes, no suspicions, light load) response time of

our server is 22.4 ms (with a standard deviation of 5 ms).

8 Future work

It would be interesting to perform a similar experiment in a Wide Area Network (WAN) setting as well. In such a setting, message delays vary a lot more than in a LAN: hosts can even be partitioned from each other. Thus the environment is closer to the asynchronous model in which the FLP impossibility result was proven, and an experiment might unveil its effect.

References

- [BMST93] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 8, pages 199–216. Addison-Wesley, second edition, 1993.
- [CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. on Parallel & Distributed Systems*, 10(6):642–657, June 1999.
- [CHTCB96] 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)*, pages 322–330, New York, USA, May 1996. ACM.
- [CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
- [CTA00] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *Proc. of the Int'l Conf. on Dependable Systems and Networks (DSN)*, pages 191–200, New York, USA, June 2000.
- [FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
- [HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 5, pages 97–146. Addison-Wesley, second edition, 1993.
- [MR99] 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)*, pages 49–63, Bratislava, Slovak Republic, September 1999.
- [Pol94] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6(3):289–316, May 1994.
- [Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
- [Sch93] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 7, pages 169–198. Addison-Wesley, second edition, 1993.
- [Sch97] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.
- [UDS01] 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)*, pages 503–511, Beppu City, Japan, 2001. <http://lsewww.epfl.ch/Publications/Byld/255.html>.