How Fast Can Eventual Synchrony Lead to Consensus?

Partha Dutta
EPFL, Switzerland

Rachid Guerraoui
EPFL, Switzerland

Leslie Lamport
Microsoft

9 March 2005

To appear in *Proceedings of the 2005 International Conference on Dependable Systems and Networks* (DSN 2005).

Abstract

It is well known that the consensus problem can be solved in a distributed system if, after some time T_S, no process fails and there is some upper bound δ on how long it takes to deliver a message. We know of no existing algorithm that guarantees consensus among N processes before time $T_S + O(N\delta)$. We show that consensus can be achieved by time $T_S + O(\delta)$.

Contents

1 Introduction

Unbounded message delays and continual process failures make it impossible to guarantee that a consensus algorithm will terminate [6]. Suppose there is some time T_S after which no process fails and messages are delivered within a bounded length of time. How soon after time T_S can an algorithm ensure that all nonfaulty processes have reached consensus? It is this question that we address.

We consider only omission (non-Byzantine) faults. We allow messages to be lost and processes to fail by stopping, but messages may not be corrupted and faulty processes may not perform incorrect actions. Byzantine faults are discussed in the conclusion. A failed process can restart at any time. Processes have timers that run at approximately the same rate after time T_S.

We say that the system is *stable* after time T_S, and we let δ seconds be the bound on message-delivery time after stability is reached. More precisely, we assume that when the system is stable, a nonfaulty process will receive and react to a message within δ seconds of when the message was sent. (Hence, δ includes the time needed to process the message after it is received.) We can therefore consider process actions to be instantaneous, processing time being counted as message-delivery time or, for actions generated by timeout, as part of the waiting time. We assume that a majority of the processes are nonfaulty at time T_S, and hence remain nonfaulty forever. (It obviously doesn't matter if processes fail after consensus has been reached, but we assume for simplicity that they never fail after time T_S.) We let N be the number of processes and number them from 0 through $N-1$.

Since consensus requires interprocess communication, and all messages sent before T_S might be lost, it must take at least $O(\delta)$ seconds after stability to reach consensus. (For presentation simplicity, we write $O(*)$ for both $O(*)$ and $\Omega(*)$.) This paper shows how this bound can be achieved. More precisely, we assume that all processes have (unsynchronized) local clocks that, after time T_S, have an error in their running rate of at most some known value $\rho \ll 1$. We describe an algorithm in which every process that is nonfaulty at time T_S decides by time $T_S + O(\delta)$. Every process that restarts after time T_S decides within $O(\delta)$ seconds after it has restarted. (Recall that we assume no process fails after time T_S.)

The processes do not know when time T_S has arrived; they have no way of knowing that the system has become stable. However, we do assume that the value of δ is known. Although consensus can be guaranteed even if δ is not known [5], the following informal argument suggests that an algorithm must know δ to achieve a time bound independent of the length of time that elapses before stability (T_S). A completely asynchronous, deterministic algorithm cannot solve consensus [6]. To ensure progress, an algorithm must use timeouts to keep from waiting forever for responses from failed processes. To ensure that consensus is reached after stability, the timeout intervals must be of length $O(\delta)$. If the value of δ is not known to the algorithm, then the only way to ensure progress seems to be to set the timeouts based on some guess γ of the value of δ, and to keep increasing γ until consensus is reached [5]. The time needed to reach consensus after stability is then $O(\gamma_S)$, where γ_S is the value of γ at time T_S. Since γ can grow without bound until stability is reached, consensus cannot be achieved within $O(\delta)$ seconds after stability. We therefore assume that δ is known.

In this paper, we measure the time required for all nonfaulty processes to decide in a consensus algorithm; the time required for the processes to terminate the algorithm might be higher. As processes never know if the system has reached stability, an algorithm cannot terminate until every process knows that all nonfaulty processes have reached a decision. Otherwise, processes might terminate before time T_S, and there could be a nonfaulty process that has not decided because it has been unable to communicate with any other process.

The problem of finding an algorithm that reaches consensus within $O(\delta)$ can be solved with some simple modifications to the Paxos consensus algorithm [3, 9, 10] if we assume that the bound on message-delivery time that holds after T_S also applies to messages sent before that time—in other words, every message sent before time T_S is either lost or delivered by time $T_S + \delta$. Furthermore, if we assume that on restart, a process knows that it has failed so that it can execute a special initialization procedure, then the problem becomes simpler. Without these assumptions however, the problem is hard because, even after time T_S, an algorithm must cope with obsolete messages either sent before T_S by failed processes or sent by newly-restarted processes. To our knowledge, all previous consensus algorithms require in the worst case at least $O(N\delta)$ seconds after stability to reach consensus.

Our primary solution is a round-based variant of the Paxos algorithm. In Section 2 we briefly recall the Paxos algorithm and then point out why any simple modification to Paxos does not decide within $O(\delta)$ seconds after T_S. In Section 3 we explain why typical round-based algorithms do not achieve the desired performance either. Section 4 presents our modified version of the Paxos consensus algorithm and its timing analysis. Section 5 briefly outlines another solution based on an algorithm of Pedone, Schiper, Urbán, and Cavin [14]. The concluding section briefly considers Byzantine failures.

2 Traditional Paxos

Before presenting our modified version, we describe the traditional Paxos consensus algorithm. We omit many details that, while crucial to its correctness, are irrelevant to our discussion.

The Paxos algorithm assumes a leader-election procedure whose correct operation is required only to ensure progress, not safety. Each process p maintains a natural number $mbal[p]$, called its *ballot number*, which it attaches as the field $m.mbal$ to every message m it sends. (The variable $mbal[p]$ was called $nextBal[p]$ in [9] and $Commit$ in [3].) The initial value of $mbal[p]$ doesn't matter; for later convenience we let it equal p. The process keeps $mbal[p]$ (and the rest of its state) in stable storage so it can restart after failure by simply resuming where it left off. Process p can execute the following actions:

Start Phase 1 At any time, if p believes itself to be the leader, then it can increase $mbal[p]$ to an arbitrary value congruent to p mod N and send a *phase 1a* message to every process (including itself).

Receive Phase 1a Message If p receives a *phase 1a* message m with $m.mbal > mbal[p]$ then it sets $mbal[p]$ to $m.mbal$ and sends a *phase 1b* message to process $m.mbal$ mod N.

Start Phase 2 If p receives a *phase 1b* message m with $m.mbal = mbal[p]$ from $\lceil N/2 \rceil$ different processes then it sends a *phase 2a* message to every process.

Receive Phase 2a Message If p receives a *phase 2a* message m with $m.mbal \geq mbal[p]$ then it sets $mbal[p]$ to $m.mbal$ and sends a *phase 2b* message to every process.

Decide If p receives *phase 2b* messages with the same *mbal* field from a majority of processes then it decides on a value.

Reject Message If p receives a *phase 1a* or *phase 2a* message m with $m.mbal < mbal[p]$, it sends a *rejected* message containing $mbal[p]$ to process $m.mbal$ mod N.

In a real implementation, once a process has decided, it would stop executing the algorithm and simply respond to every message by announcing the value it has decided upon. We ignore this optimization for now.

Suppose the leader-election procedure is guaranteed to choose a unique, nonfaulty leader within $O(\delta)$ seconds after the system is stable, and the leader spontaneously executes the *Start Phase 1* action every $O(\delta)$ seconds. The following plausible but incorrect argument shows that the Paxos algorithm then guarantees consensus within $O(\delta)$ seconds after stability. By time $T_S + O(\delta)$, a single process q believes itself to be the leader and executes the *Start Phase 1* action. If the value of $mbal[q]$ that q chooses is larger than the value of $mbal[p]$ for all other nonfaulty processes p, then consensus will be reached after all the *phase 1a*, *1b*, *2a*, and *2b* messages for ballot $mbal[q]$ are generated by and delivered to nonfaulty processes, which takes at most 4δ seconds. If $mbal[q]$ is less than $mbal[p]$ for some p, then q will receive a *rejected* message from p within 2δ seconds and can then execute the *Start Phase 1* action with a larger value of $mbal[q]$. In this case, consensus will be reached within 6δ seconds. Hence, consensus is reached within $O(\delta)$ seconds after T_S.

The argument that consensus is reached within 4δ or 6δ seconds after q executes *Start Phase 1* is fallacious. It assumes that no process receives a message m with $m.mbal$ greater than q's final choice of $mbal[q]$. However, there could be messages with higher $mbal$ fields that were sent by processes that have since failed, or by failed processes that just restarted. Receipt of such a message could prevent the algorithm from succeeding with the current value of $mbal[q]$, forcing q to choose a larger value. Since there could be as many as $\lceil N/2 \rceil - 1$ such failed processes, it could take $O(N\delta)$ seconds to reach consensus after q first executes the *Start Phase 1a* action.

We will present a version of the Paxos algorithm that does achieve consensus within $O(\delta)$ seconds of stability, without relying on any election algorithm. However, we first discuss round-based consensus algorithms.

3 Round-Based Algorithms

There are several consensus algorithms that work roughly as follows [2, 5]. Processes execute a sequence of rounds. A process executing round i ignores messages from lower-numbered rounds; if it receives a message from a higher-numbered round j, then it begins executing round j. If consensus is not reached in round i, then a timeout will cause some process to abort round i and begin round $i + 1$. (Although aborting might be attributed to the pronouncement of some oracle, such as a failure detector, the oracle's implementation issues the pronouncement when a timeout occurs.) To ensure that a round started after the system is stable succeeds in reaching consensus, a timeout interval of $O(\delta)$ must be used.

In these round-based algorithms, the round number plays the same role as the ballot number in the Paxos algorithm. The Paxos algorithm's problem of old messages with large ballot numbers can be avoided in round-based algorithms by not allowing a process spontaneously to enter round $i+1$ until it has learned that a majority of the processes have begun round i. This ensures that whenever a majority of the processes are nonfaulty, if a round i message has been sent, then there is a nonfaulty process executing round $i-1$ or higher. This implies that if i is the highest round being executed by some nonfaulty process when the system becomes stable, no old messages or restarted process can disrupt any round from $i+2$ on.

Eliminating the problem of obsolete messages does not ensure that round-based algorithms reach consensus within $O(\delta)$ seconds of stability. For round i to succeed, most of these algorithms require that a coordinator, generally process $i \bmod N$, be nonfaulty. Since there could be $\lceil N/2 \rceil - 1$ faulty processes, they could require $O(N)$ rounds to reach consensus, each round taking $O(\delta)$ seconds. There is a round-based consensus algorithm by Mostefaoui and Raynal [13] that relies on leader election, but considering that algorithm simply shifts our problem to that of electing a leader within $O(\delta)$ seconds of T_S, in the presence of obsolete messages and process restarts.

There is one round-based algorithm that does not rely on a coordinator or a leader—namely, the B-Consensus algorithm of Pedone et al. [14]. Section 5 outlines an approach to modifying that algorithm so it reaches consensus within $O(\delta)$ seconds of stability.

The number of rounds required to achieve consensus after stability is investigated in [4], which considers a round-based eventually synchronous model and looks at the number of rounds needed to achieve consensus after the first stable round is reached. However, the physical duration of a round is not specified in [4]. In fact, if processes might restart after T_S, then its algorithm does not achieve consensus within a constant number of rounds.

4 The Modified Paxos Algorithm

We now refine the Paxos algorithm to achieve consensus within $O(\delta)$ seconds after stability. Our new version has no explicit leader election. Instead, any process can perform a *Start Phase 1* action, under certain circumstances. The basic idea is to keep a process from choosing ballot numbers that are too large by emulating the way round-based algorithms avoid anomalously high round numbers.

Define the *session* of a ballot number b to be $\lfloor b/N \rfloor$ and define process p to be in session $\lfloor mbal[p]/N \rfloor$. Similarly, the session of a message m is the session of the ballot number *m.mbal*. We say that process p *enters session* s when its session number changes from some $t < s$ to s. We now modify the Paxos algorithm so a process does not enter session $s + 1$ until a majority of processes have entered session s. We also introduce timeouts and do away with the leader-election procedure, making leader election implicit in the Paxos algorithm itself. The use of timeouts makes the *Reject* action unnecessary.

Each process maintains a *session timer*. Whenever a process enters a new session, it resets its session timer so that if time T_S has arrived, then it will time out between 4δ and σ seconds later, for some $\sigma \geq 4\delta$ with $\sigma = O(\delta)$. This is possible because of our assumption that processes have timers with a known bound $\rho \ll \delta$ on their running rates after time T_S. Session timers are set initially to time out within σ seconds.

A process can execute the *Start Phase 1* action whenever (i) its session timer has timed out and (ii) it is either in session 0 or else has received a message with its current session from a majority of the processes. Condition (ii) means that, if $mbal[p] \geq N$, then p has received a message m with $\lfloor m.mbal/N \rfloor = \lfloor mbal[p]/N \rfloor$ from a majority of processes. When p performs the action, it chooses the new value of $mbal[p]$ to increase its session number by 1. In other words, it sets $mbal[p]$ to $(\lfloor mbal[p]/N \rfloor + 1)N + p$. Since this action increases p's session number, it also resets the session timer.

We make two additional changes to the algorithm. First, we have a process p send a *phase 1a* message to all other processes whenever it begins a new session. Second, we require that a process send a *phase 1a* message (with its current value of $mbal[p]$) if it has not sent a *phase 1a* or *2a* message within the past ϵ seconds, for an arbitrary positive $\epsilon = O(\delta)$. Since the Paxos algorithm works despite duplication of messages, it permits these extra *phase 1a* messages. (Any *phase 1a* message m is treated as if it were sent by process $m.mbal \bmod N$.)

Proof of Correctness

We now sketch a proof that every process that is nonfaulty at time T_S decides by time $T_S + O(\delta)$. Let $\mathcal{W}$ be the set of processes that are nonfaulty at time T_S, let s_0 be the maximum session number at time T_S of all processes in $\mathcal{W}$, let τ be the maximum of $2\delta + \epsilon$ and σ, and let $[T_a, T_b]$ be the time interval from T_a through T_b.

1. At any time after T_S, all messages sent before T_S and all failed processes have session number at most $s_0 + 1$.

Proof: A *Start Phase 1* action that advances a process session to s cannot be executed until a majority of processes are in session $s - 1$, and any majority of processes contains a process in $\mathcal{W}$.

2. If at time $T > T_S$ process p sends a *phase 1a* message with its session number s, then at some time in $[T, T+\tau]$ a process will enter a session t with $t > s$.

   Proof: Every process q in $\mathcal{W}$ receives p's *phase 1a* message by time $T + \delta$ and sends a *phase 1a* message with session number at least s, either then if q's session number is less than s or else ϵ seconds later. By $T + \epsilon + 2\delta$ process p will receive *phase 1a* messages from every process in $\mathcal{W}$ and will perform the *Start Phase 1* action when its session timer times out, if it has not already started a higher-numbered session.

3. At some time T_3 in $[T_S, T_S+\epsilon+\tau]$, a process enters session $s_3 \geq s_0+1$.

   Proof: Let p be a process in $\mathcal{W}$ with session number s_0 at time T_S. If no process in $\mathcal{W}$ enters a higher-numbered session by $T_S + \epsilon$, then p must send a *phase 1a* message by that time. The result then follows from step 2.

4. At some time T_4 in $[T_3, T_3 + \tau]$ some process executes *Start Phase 1* to enter session $s_4 \geq s_0 + 2$.

   Proof: By steps 2 and 3, some process enters a session numbered at least $s_0 + 2$ during $[T_3, T_3 + \tau]$. By step 1, it could only have done this by executing *Start Phase 1.*

5. At some time T_5 in $[T_4, T_4 + \tau]$, some process p_5 executes a *Start Phase 1* action to become the first process to enter session number $s_5 \geq s_0 + 3$.

   Proof: Steps 2 and 4 imply that some process p_5 enters a session $s_5 \geq s_0 + 3$ during $[T_4, T_4 + \tau]$, and step 1 implies that it can do so only by executing *Start Phase 1.*

6. At time $T_5 + \delta$

   (a) Every process in $\mathcal{W}$ is in session s_5.

   (b) No process is in a session higher than s_5.

   (c) If a nonfaulty process is in session s_5, then its session timer will not time out before time $T_5 + 4\delta$.

   (d) There is a process p_6 such that

i. p_6 entered session s_5 during $[T_5, T_5 + \delta]$ and set its current value of $mbal[p_6]$ by executing *Start Phase 1*.

ii. $mbal[p] \leq mbal[p_6]$ for all nonfaulty processes p.

Proof: By steps 1 and 5, no process can enter a higher session before first entering session s_5.

(a) Step 5 implies that by time $T_5 + \delta$, every process nonfaulty at time T_5 has entered session s_5, either by receiving a message from p_5 or another process that entered session s_5, or else by executing *Start Phase 1*.

(b, c) Every process in session s_5 entered during $[T_5, T_5 + \delta]$, so its session timer was reset during that interval and will not time out before $T_5 + 4\delta$. Hence, at time $T_5 + \delta$, no process is in a session higher than s_5.

(d) Let p_6 be the process p with the largest value of $mbal[p]$ at time $T_5 + \delta$. It could only have acquired that value by executing *Start Phase 1*.

7. By time $T_5 + 4\delta$, every process in $\mathcal{W}$ has sent a *phase 2 b* message m with $m.mbal = mbal[p_6]$.

   Proof: By steps 6 and 1, every process p that is nonfaulty at any time in $[T_5, T_5 + 4\delta]$ has $mbal[p] \leq mbal[p_6]$ throughout that period. Hence, by $T_5 + 2\delta$ every process in $\mathcal{W}$ receives p_6's *phase 1a* message; by $T_5 + 3\delta$ process p_6 receives *phase 1b* messages from every process in $\mathcal{W}$ and sends a *phase 2a* message; and by $T_5 + 4\delta$ every process in $\mathcal{W}$ receives the *phase 2a* message and sends a *phase 2b* message.

8. Every process in $\mathcal{W}$ decides on a value by time $T_5 + 5\delta$.

   Proof: By step 7 and the definition of the *Decide* action.

Adding things up, we see that every process nonfaulty at time T_S has decided by time $T_S + \epsilon + 3\tau + 5\delta$. With reasonably accurate timers, if processing time is negligible compared with message-delivery time, then we can take $\sigma \approx 4\delta$. By making $\epsilon \ll \delta$, so $\tau = \sigma$, we can make the decision time as early as about $T_S + 17\delta$. It seems likely that this bound could be improved by a more clever algorithm.

Process Restarts

We have just proved the result that every process that is nonfaulty at time T_S decides by time $T_S + O(\delta)$. We also have to show that every process p that restarts after time T_S decides within $O(\delta)$ seconds of when it is restarted. But this is a trivial consequence of the first result. The assumptions we have made about time T_S, and hence the first result, hold for all times $T'_S > T_S$. Substituting T'_S for T_S in the first result shows that, if process p restarts at time $T'_S > T_S$, then it decides by time $T'_S + O(\delta)$.

We can a derive better bound on how long it takes a process that restarts after T_S to decide. It can be seen from our proof that any process that restarts by time T_5 decides by $T_5 + 5\delta$. From T_5 on, a new session starts every τ seconds and delivers the requisite *phase 2b* message within 5δ seconds of its start.

As observed above, the decision time of a process that restarts after some processes have already decided can be reduced by having those processes periodically broadcast their decision.

Reducing Message Complexity

The message complexity of a consensus algorithm matters only when a system executes a sequence of separate instances of the algorithm. The operation of a well-designed system consists of long periods of stability, with timely communication and no failures, punctuated by occasional process or communication-network failures. We want to minimize the communication complexity during the stable periods and to prevent excessive message sending from delaying recovery from failures.

In ordinary Paxos, phase 1 is executed in advance for all instances of the algorithm, and all nonfaulty processes decide within 3 message delays when the system is stable. By setting ϵ large enough and using the appropriate acknowledgement messages, our modified version of Paxos can be made to have this same behavior in the stable case. In the same way, the modified algorithm can also be made to have the same behavior as normal Paxos in the event of process failure, as long as communication between nonfaulty processes is timely. Our modified algorithm will then send more messages than ordinary Paxos only in the event of communication failure.

Our algorithm's extra messages are the *phase 1a* messages it sends every ϵ seconds. We can have it send fewer *phase 1a* messages by increasing the value of ϵ, but this can increase how long it takes processes to decide after the system becomes stable. We can also add acknowledgements of receipt of *phase 1a* messages to other messages, so a process does not resend a *phase 1a* message to another process that has already received it. However, fast recovery from communication failure requires periodically sending messages to learn when communication has been restored. Frequent message sending is an unavoidable cost of fast recovery.

5 The Modified B-Consensus Algorithm

The B-Consensus algorithm of Pedone et al. [14] is a leaderless round-based algorithm using a message-delivery oracle. A round achieves consensus if more than $N/2$ processes are nonfaulty and all messages sent in that round are delivered by the oracle to all processes in the same order. We now sketch a method for modifying this algorithm to reach consensus within $O(\delta)$ seconds of stability.

We implement the message-delivery oracle as follows. All messages to be delivered by the oracle are broadcast to all processes and are timestamped with logical clocks [8]. This means that after a process receives a message m, all messages it sends have timestamps greater than that of m. The oracle delivers messages to a process in timestamp order, waiting 2δ seconds after the message is actually received by the process before delivering it.

We first show that when the system is stable, if there are no restarts, then the oracle delivers messages to all nonfaulty processes in the same order. A message m sent when the system is stable will be received by every nonfaulty process within δ seconds of when it was sent, after which every message sent has a higher timestamp. Therefore, having a process wait 2δ seconds before delivering m ensures that it has received every message with a timestamp lower than that of m that was sent after stability was reached. This implies that the oracle delivers the same set of messages to all processes in the same timestamp order.

With restarts, messages sent by a newly restarted process may be delivered in different orders to different processes. However, delivery order is significant only for messages received by a process in the current round. As with other round-based algorithm, the B-Consensus algorithm does not start round $i+1$ until a majority of processes have reached round i. Hence, if i is the highest round being executed by some nonfaulty process when the system becomes stable, round $i+2$ will not be disrupted by a message from a newly restarted process.

An analysis similar to the one for the modified Paxos algorithm shows that within $O(\delta)$ seconds of stability, the system begins a round that no obsolete message or restarting process can disrupt. That round succeeds within $O(\delta)$ seconds. The actual maximum delay is about the same as for the modified Paxos algorithm.

As described by Pedone et al., the B-Consensus algorithm requires that a process execute all previous rounds before entering a new round. A nonfaulty process could still be executing the first round when stability is reached. Processes therefore have to keep retransmitting their messages from all previous rounds to ensure that such a process is brought up to date within $O(\delta)$ seconds of stability. It is unreasonable to assume that such an arbitrarily large set of messages could be delivered within δ seconds. However, the algorithm is easily modified to allow a process to jump immediately to a later round when it receives a message for that round, without having to execute all previous rounds.

6 Concluding Remarks

Assuming that after time T_S no process fails, a majority of processes are nonfaulty, and every message is delivered within δ seconds of when it is sent, we have presented a version of the Paxos consensus algorithm that reaches agreement by time $T_S + O(\delta)$. We have also sketched a version of the B-Consensus algorithm of Pedone et al. that does the same. Finding such an algorithm is nontrivial because we make no assumption about messages sent before T_S and we allow failed processes to restart from where they left off. Although it must take $O(\delta)$ time after T_S, there probably exist algorithms that can reach consensus more quickly than these.

There are two natural ways in which we might extend our results—by allowing processes to fail after T_S, assuming a majority of processes remain nonfaulty, and by allowing Byzantine failures, assuming that more than 2/3 of the processes are nonfaulty. In both cases, it is impossible to reach agreement by time $T_S + O(\delta)$. Even a perfectly synchronous system with only crash failures requires $O(F)$ rounds, where F is the number of processes that actually fail [7]. With Byzantine failures, a malicious process may continue to be malicious after T_S. If there are M malicious processes, it must therefore take an asynchronous algorithm at least until time $T_S + O(M\delta)$ to reach agreement. We now briefly consider how this bound might be achieved.

Castro and Liskov have published a version of the Paxos algorithm that handles Byzantine faults and also solves the problem of anomalously high ballot numbers [1, 12]. However, their algorithm rotates through leaders until it finds a nonfaulty one, so it cannot ensure agreement before time

$T_S + O(N\delta)$. To reduce this to $T_S + O(M\delta)$, we would need some method of rotating through leaders that skips faulty but non-malicious processes. We do not know how to do this.

The third author has developed a version of Paxos that handles Byzantine faults without requiring a leader. Like the algorithm of Pedone et al., it reaches agreement if certain messages arrive at all processes in the same order. It should be possible to ensure that messages do arrive in the same order when the system is stable, even with M Byzantine faults, by taking $O(M\delta)$ seconds to send a message. However, the algorithm ensures progress only if more than 4/5 of the processes are nonfaulty. Lower-bound results suggest that this many nonfaulty processes are required by any Byzantine consensus algorithm that does not use a leader [11].

References

[1] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

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

[3] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the PAXOS algorithm. *Theoretical Comput. Sci.*, 243:35–91, 2000.

[4] P. Dutta, R. Guerraoui, and I. Keidar. The overhead of consensus failure recovery. IC Technical Report 200456, Ecole Polytechnique Fédérale de Lausanne (EPFL), June 2004.

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

[6] M. J. Fischer, N. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[7] M. J. Fischer and N. A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4):183–186, June 1981.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[9] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[10] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):18–25, Dec. 2001.

[11] L. Lamport. Lower bounds for asynchronous consensus. In A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 22–23. Springer, 2003.

[12] B. W. Lampson. The ABCDs of Paxos. `http://research.microsoft.com/lampson/65-ABCDPaxos/Abstract.html`.

[13] A. Mostéfaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, Mar. 2001.

[14] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proceedings of the 4th European Dependable Computing Conference (EDCC-4)*, volume 2485 of *Lecture Notes in Computer Science*, pages 44–61. Springer-Verlag, 2002.