Asynchronous Leasing*

Romain Boichat Partha Dutta Rachid Guerraoui
Distributed Programming Laboratory
Swiss Federal Institute of Technology in Lausanne

Abstract

Leasing is a very effective way to improve the performance of distributed algorithms without hampering their fault-tolerance. The notion of lease has traditionally been defined using a global notion of real time and was hence strongly tied to synchronous system models.

This paper introduces a notion of lease object devised for an asynchronous system model. We give precise properties of our lease abstraction, and show how it can be implemented in an asynchronous system model with a logical notion of time. We then illustrate its use by showing how it significantly improves the performance of a consensus-based atomic broadcast algorithm and we point out other possible lease applications.

1 Introduction

A *lease* is a contract that gives its holder specific rights over a resource for a limited period of time [GC89]. It can be viewed as a fault-tolerant *lock* [Lam96]: the lock is granted only for the term (time duration) of the lease and hence tolerates the failure of its holder. Leases are particularly useful in transactional systems. Locks can indeed be implemented as leases: if they expire, the transaction is aborted. The notion of lease is also useful in the context of caching: a lease grants to its holder control over writes to the cached resource (data) during the term of the lease.

The very notion of lease is strongly tied to a global time. Lease implementations typically use time-outs and assume synchronised clocks. Typically, if the maximum skew between the clocks of any two processes is ε and the lease of process p on a given resource expires at time t, then p knows that no other process will touch the resource before time $t - \varepsilon$ on p's clock [Lam96].

We introduce in this paper a concept of *asynchronous lease* where, roughly speaking, we replace the notion of real time with a notion of logical time (simply represented by positive integers). A process (1) cannot acquire a lease for an interval that overlaps with an interval for which a lease has already been granted, and (2) should be able to acquire a lease for any interval for which no lease has been requested. While it preserves the original intuition of the synchronous lease concept, and it can be implemented with a rather simple *quorum-based* algorithm in an asynchronous system model (as we show in this paper), our concept of asynchronous lease can obviously not be used in the same traditional manner. Indeed, holding a lease object over a resource for a given logical period does not say much about any form of exclusive access to the resource at a given point in physical time.

Nevertheless, we illustrate the use of this notion by showing how it can significantly improve the performance of asynchronous distributed algorithms. We consider here as an illustration the atomic broadcast algorithm of [CT96]. The algorithm was devised as a sequence of consensus instances and, intuitively, we use our notion of lease object to grant the holder of the lease the right to be the first coordinator for all consensus instances within the term of the lease. Basically, the first coordinator has the advantage that it can short-cut the first communication step in the consensus and

*This work is supported by the Swiss National Foundation (NSF)

directly impose its decision. In stable periods of the system (the most frequent periods in practice), we reduce by one third the number of messages and communication steps needed to reach a final decision. Interestingly, we apply our notion of lease to the atomic broadcast algorithm of [CT96] without actually changing the atomic broadcast algorithm: we plug-in a specific *fast* consensus algorithm obtained by incorporating our asynchronous lease object into the consensus algorithm of [CT96]. It is important to notice here that atomic broadcast is used only for illustration purposes. As we point out in the paper, one can easily apply the same idea for other forms of consensus repetitions, e.g., repeated non-blocking weak atomic commit and group membership [GS01].

It is also worthwhile noticing that Lampson suggested in [Lam96] the use of leases to improve the performance of the Paxos replication algorithm (actually an atomic broadcast algorithm) of Lamport [Lam89]. Our idea has the same flavour, except that Lampson considered the traditional notion of lease, i.e., relying on synchrony assumptions, whereas our notion is completely asynchronous.

In the context of this paper, we assume the crash-stop model of [CT96]. The distributed system consists of a set of n processes and each process is is denoted by a unique process identifier p, which is one of the first n positive integers. Processes fail by crashing and do not recover from a crash. A process which never fails is called a correct process. Processes communicate through *send* and *receive* primitives abstracting reliable communication channels [HT93]. As we point out in the paper, the ideas presented here could however easily be extended to a more practical crash-recovery system model [ACT00].

To sum up, the contribution of this paper consists in introducing a notion of lease that can be (1) implemented in a crash-stop asynchronous system and (2) shown to be useful in enhancing the performance of a distributed algorithm without hampering its fault-tolerance. Section 2 presents the specification of our lease abstraction, Section 3 gives its implementation and Section 4 illustrates its use. Section 5 concludes the paper with some general observations.

2 Lease Specification

In our context, a $lease$ (i.e., an asynchronous lease hereafter denoted $lease$) is a contract granting exclusive rights over a shared resource for a logical period. A logical period is a finite interval of positive integers. Leases grant exclusive rights over an interval; and for a given interval, a lease is acquired *at most once*. We do not require that leases are acquired for all intervals; there can be intervals for which no process acquires the lease.[1] We introduce some notations and terminology to precisely define the properties of the lease. Let $\mathbb{N}$ be the set of all positive integers, and let $s, e, s', e' \in \mathbb{N}$ with $s \leq e$ and $s' \leq e'$. Then, we define

Interval $[s, e]$**:** $\{i \in \mathbb{N} \mid s \leq i \leq e\}$.
Overlapping intervals: Two intervals $[s, e]$ and $[s', e']$ overlap if ($s \leq s' \leq e$) or ($s \leq e' \leq e$).
Lease(s,e): The function $lease(s, e)$ either returns *true* or *false*; $lease(s, e)$ is always invoked with $s \leq e$.
Leasing terminology: We say that a process p *acquires* the lease for the interval $[s, e]$ if p *invokes* $lease(s', e')$ (we say that the lease is requested) such that $[s, e] \subseteq [s', e']$ and $lease(s', e')$ returns *true*. We say that the lease is *refused* if $lease(s', e')$ returns *false*.

Lease properties. A lease is a shared object exporting the operation $lease()$ that is defined by the following two properties:

- **At most once leasing:** No two leases have overlapping intervals. More precisely,

Let LI be the set of all $lease()$ invocations (including the invocations which do not complete due to process crashes). Let L ($\subseteq LI$) be the set of all $lease()$ invocations which return *true*. We have

$$\forall\, l_1, l_2 \in L,\ (l_1.s \leq l_2.s \leq l_1.e) \vee (l_1.s \leq l_2.e \leq l_1.e) \ =\ \textit{false}$$

- **Mandatory leasing:** A lease cannot be refused if no lease has been requested for an overlapping interval. More precisely,

[1] Just like most real clock based lease schemes, there exist some intervals for which no process has the lease.

Let $LI_{s,e} = \{lease(s', e') \in LI \mid (s \leq s' \leq e) \vee (s \leq e' \leq e)\}$. We have

$$((LI_{s,e} = \{lease(s,e)\}) \wedge (lease(s,e) \text{ returns})) \Rightarrow lease(s,e) \in L$$

To illustrate the semantics of a lease object, Figure 1 depicts some runs where processes invoke the $lease()$ function. In Figure 1(a), the properties of the lease object are trivially satisfied. In Figure 1(b), the invocation of $lease(5, 15)$ returns *false* to ensure the *at most once leasing* property of the lease. This conveys the idea that, according to our semantics, even the very same process cannot acquire the same lease twice. Figure 1(c) depicts an example where both leases are acquired even if they are not requested with increasing order intervals. Figure 1(d) depicts a run where $lease(1, 10)$ must return *false* in order not to violate the *at most once leasing* property of the lease. In Figures 1(e) and 1(f), the lease invocations are concurrent. For Figure 1(e), both invocations return *false* and this example does not violate the *mandatory leasing* property. Note also that due to the non-determinism of the system, the following results could have been possible: $lease(1, 10)$ returns *false* and $lease(5, 15)$ returns *true*, or the opposite, $lease(1, 10)$ returns *true* and $lease(5, 15)$ returns *false*. Figure 1(f) depicts a run that violates the *mandatory leasing* property, i.e., both lease invocations should return *true*.

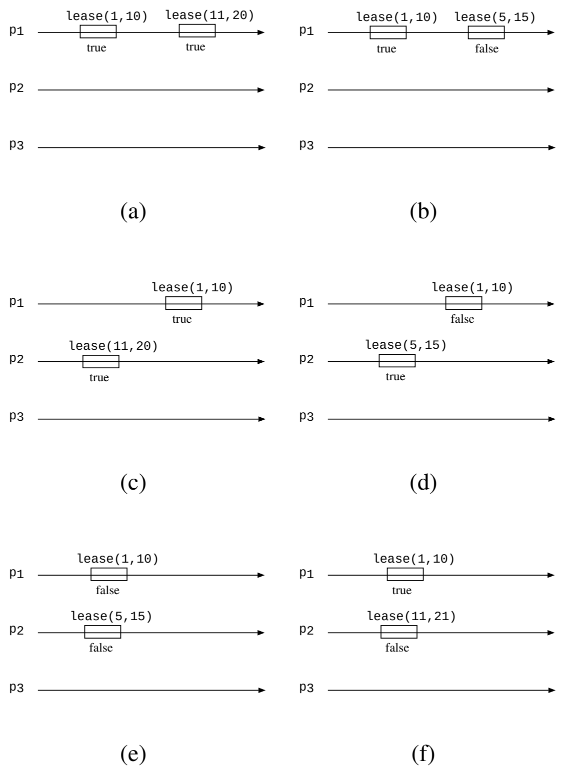

Figure 1. Lease examples

3 Lease Implementation

3.1 Lease algorithm

This section gives an implementation of the lease object in an asynchronous system model. Assuming a majority of correct processes, the algorithm of Figure 2 implements a wait-free [Her91] lease object. The algorithm works as follows. Each process has an infinite array $permission$, which is initially empty. On invoking $lease(s, e)$, a process p sends a LEASE message, say m, to all processes. On receiving m, a process q replies with (ACK_LEASE, m) message if all values from $permission[s]$ to $permission[e]$ are $\perp$. Otherwise, q replies with a (NACK_LEASE, m) message. Before replying with an ACK_LEASE or a NACK_LEASE message, q sets $permission[i]$ $(s \leq i \leq e)$ to the pid of the process requesting the lease. Thus, for all subsequent LEASE messages with an interval overlapping with $[s, e]$, q replies with a NACK_LEASE message. If p (the process which invoked $lease(s, e)$) receives a majority of ACK_LEASE messages, then the $lease(s, e)$ function returns *true*, otherwise the function returns *false*.[2]

3.2 Correctness proofs

Lemma 1. *The algorithm of Figure 2 satisfies the at most once leasing property: no two leases have overlapping interval.*

Proof. Suppose by contradiction that (i) a process p acquires the lease for the interval $[s, e]$, (ii) a process q acquires the lease for the inter-

[2]This crash-stop implementation of $lease$ (Figure 2) can be easily transformed to a crash-recovery model. The send and receive primitives are replaced by the s-send and s-receive primitives of a retransmission module [ACT00] to accommodate fair lossy channels, and before replying with an ACK_LEASE or a NACK_LEASE message, the $permission$ array is stored into stable storage (in order to recover from a crash).

1: for each process p:
2: **initialisation:** $permission[] \leftarrow \{\bot, \bot, \ldots\}$; $tmp \leftarrow true$
3: **procedure** lease (s,e)
4: send (m = (LEASE,s, e)) to all
5: **wait until** received (ACK_LEASE,m) **or** (NACK_LEASE,m) from $\left\lceil \frac{(n+1)}{2} \right\rceil$ different processes
6: **if** received only (ACK_LEASE,m) **then return**($true$) **else return**($false$)
7: **upon** receive $m' =$ (LEASE,s, e) from q **do**
8: $tmp \leftarrow true$
9: **for** $s \leq i \leq e$ **do**
10: $tmp \leftarrow$ (tmp **and** ($permission[i] = \bot$)); $permission[i] \leftarrow q$
11: **if** (tmp) **then** send (ACK_LEASE,m') to q **else** send (NACK_LEASE,m') to q

Figure 2. Wait-free lease implementation with reliable channels and a majority of correct processes

val $[s', e']$, and (iii) the two intervals overlap.[3] When p or q acquires a lease, both require to receive ACK_LEASE messages from a majority of processes. Since two majorities of processes always intersect, there must be a process k which sent an ACK_LEASE message in reply to m_1 = (LEASE, s, e), as well as to m_2 = (LEASE, s', e'). Let x be an integer within the intervals $[s, e]$ and $[s', e']$ (such an integer exists because the two intervals overlap). Without loss of generality, we can assume that process k received m_1 before m_2. Before sending an ACK_LEASE message in reply to m_1, $permission[x]$ at k is set to p (line 10). Since $permission[x]$ at k is no longer $\bot$, k replies with a NACK_LEASE message to all subsequent (LEASE,*,*) messages whose interval contains x, e.g., m_2: a contradiction. $\square$

Lemma 2. *The algorithm of Figure 2 satisfies the mandatory leasing property: a lease cannot be refused if no lease has been requested for an overlapping interval.*

Proof: Suppose that a process p invokes $l = lease(s, e)$ and that there is no lease invocation with an interval that overlaps with $[s, e]$. Assume by contradiction that l returns *false*. Therefore, by the algorithm of Figure 2, p has received a NACK_LEASE message from at least one process, e.g, process q. Since q sent a NACK_LEASE message, there exists an integer $i \in [s, e]$ such that $permission[i] \neq \bot$ at q. As $permission[i]$ is initially set to $\bot$ at all processes, process q must have then received another (LEASE, s', e') message such that $i \in [s', e']$. Thus, there is a lease invocation whose interval overlaps with $[s, e]$: a contradiction. $\square$

[3]Processes p and q may or may not be the same process.

Lemma 3. *The algorithm of Figure 2 is wait-free: after a process invokes $lease()$, either the process crashes or it eventually returns from the invocation.*

Proof. Suppose by contradiction that after invoking $lease()$, a process p does not crash and remains blocked forever in the $lease()$ method. Since p does not crash, p is a correct process. In the $lease()$ function, p can only block at line 5 waiting for ACK_LEASE or NACK_LEASE messages. Since p is correct and the communication channels are reliable, the LEASE message is received by all correct processes. By the properties of the reliable channels, the replies (ACK_LEASE or NACK_LEASE) from every correct process are eventually received by p. So, p is guaranteed to receive the replies from every correct process, i.e., a majority of correct processes. Thus, p eventually completes line 5 and the $lease()$ invocation returns: a contradiction. $\square$

Proposition 4. *The algorithm of Figure 2 implements a wait-free lease object.*

Proof: Immediate from lemmata 1, 2, and 3. $\square$

3.3 A more practical algorithm

For simplicity of presentation, our implementation of a wait-free lease (Figure 2) uses an infinite array to store the intervals for which lease requests were received ($permission[]$). In practice, the memory available to a process is finite, and therefore, the $permission$ array should be substituted by an interval tree [CLR90]. An interval tree provides efficient *insert*, *search* and *delete* operations for intervals and utilises memory proportionally to the number of nodes (intervals) present in the tree.[4] An interval tree T provides three operations for any interval $i = [s, e]$; (i) $IntervalInsert(T, i)$: adds i to T, (ii) $IntervalDelete(T, i)$: deletes i from T, and (iii) $IntervalSearch(T, i)$: if there is some interval x in T which overlaps with i, then the search returns x, otherwise, it returns $\perp$. We can substitute the $permission$ array by an interval tree T (line 2 of Figure 2), and lines 8, 9, and 10 are replaced by the following line:

$$tmp \leftarrow (\text{IntervalSearch}(T, [s, e]) = \perp);\ \text{IntervalInsert}(T, [s, e])$$

4 Lease Based Consensus

We devise a new consensus algorithm, denoted *LeaseCons* as a variant of the $\diamond S$-based consensus algorithm of [CT96] that integrates our notion of lease.[5] Roughly speaking, the lease helps generalise to every process the optimisation of the [CT96] consensus algorithm that allows process p_1[6] for round 0 to impose directly its value. If the [CT96] atomic broadcast algorithm is based on our LeaseCons consensus algorithm (instead of the [CT96] consensus algorithm), then fewer communication steps are required. We first present the basic building blocks of the consensus algorithms and then recall the basics of the [CT96] consensus algorithm. We then explain the main differences between the [CT96] consensus algorithm and our LeaseCons algorithm. Finally, we give the correctness proofs of LeaseCons. In Section 5 we describe an atomic broadcast algorithm based on LeaseCons and some other possible applications of LeaseCons.

4.1 Building blocks

The [CT96] consensus algorithm assumes a $\diamond S$ failure detector [CT96], which is a distributed oracle that outputs at every process a list $\diamond S.suspected$ such that, eventually, (i) at every correct process the list contains all faulty processes, and (ii) there is a correct process which is not in the $suspected$ list of any correct process. In addition the consensus algorithm assumes a majority of correct processes and a reliable broadcast module accessed by the primitives r-broadcast and r-deliver [HT93].

In addition to the above assumptions, our LeaseCons algorithm assume a Ω failure detector which outputs at each process a process $\Omega.trusted$ such that, eventually, the process that is output is the same correct process p_c, at all correct processes. [CHT96] has shown that the failure detectors Ω and $\diamond S$ to be equivalent: there is an algorithm that transforms Ω to $\diamond S$ and vice versa.

4.2 Consensus problem (reminder)

In the consensus problem, all processes are supposed to *propose* an initial value and eventually *decide* on the same final value, among one of the proposed values. Processes propose a value by invoking an operation *propose()* with their initial value as a parameter, and decide the value returned from that invocation. The processes must satisfy the following properties:

Validity: If a process decides v, then v was proposed by some process.

Integrity: Every process decides at most once.

Agreement: No two processes decide differently.

Termination: Every correct process eventually decides some value.

[CT96] Consensus algorithm. We present a brief description of the [CT96] consensus algorithm. The algorithm proceeds in rounds, with each round consisting of four phases. Each process starts from round 0, and executes successive rounds with increasing round numbers until it r-delivers a DECIDE message. The coordinator of a round, process c_p, is decided based on the rotating coordinator paradigm: $c_p = (r\ mod\ n)$+1 where r is the round number and n is the total number of processes.

In phase 1 of a round, every process sends an

[4] The size of the interval tree is finite as long as the number of lease invocations is finite.

[5] For subsequent discussion, we denote the later "[CT96] consensus algorithm".

[6] For simplicity, we denote the process with the lowest process id ($p = 1$), p_1.

ESTIMATE message to the coordinator. In phase 2, the coordinator gathers ESTIMATE messages from a majority of processes, selects the one with the highest timestamp[7] and sends it as a NEWESTIMATE message to all processes. In phase 3, a process either receives a NEWESTIMATE message from the coordinator, or the local failure detector suspects the coordinator. In turn, the process sends an ACK message (resp. a NACK message) to the coordinator if it receives a NEWESTIMATE message (resp. suspects the coordinator). In phase 4, if the coordinator receives a majority of ACK messages then the coordinator r-broadcasts a DECIDE message. All processes then move to the next round. A process executes one round after another until a DECIDE message is r-delivered, upon which it decides on that message and then terminates the algorithm.

The [CT96] consensus algorithm can be optimised for round 0 and process p_1. Instead of waiting for a majority of ESTIMATE messages, p_1 can directly impose its value by sending a NEWESTIMATE message to every process. This optimisation is possible since there is no lower round than 0 and it is useless to wait for ESTIMATE messages since no process was able to decide before. This modification significantly speeds up the algorithm by removing one communication step (send and receive of ESTIMATE messages). Note that the above optimisation works only if p_1 is up during round 0 and p_1 is not (incorrectly) suspected by any other process. If p_1 crashes, this optimisation cannot be applied to any future consensus. We use the notion of lease to generalise this optimisation for any process.

4.3 LeaseCons algorithm

Figure 4 presents our LeaseCons algorithm. We first describe intuitively the algorithm and then its data structure. We then delve into the differences between the [CT96] consensus algorithm and our LeaseCons algorithm, and finally give its correctness proofs.

[7]Informally, a timestamp is the round number in which the ESTIMATE was most recently updated.

4.3.1 Intuitive description

LeaseCons has two modes: (i) a FAST mode where the the coordinator tries to directly impose its value, and (ii) a NORMAL mode that roughly corresponds to the [CT96] consensus algorithm. The FAST mode consists of the first round of LeaseCons whose round number can be any integer between 0 and $n-1$ (depending on the output of the Ω failure detector). All subsequent rounds are executed in NORMAL mode. All processes on completion of the FAST mode, moves on to a round in NORMAL mode with round number n. Figure 3 summarises the differences between the FAST and NORMAL modes. The FAST mode has a different phase 2 (from that of [CT96] consensus): the coordinator does not wait for ESTIMATE messages but directly sends a NEWESTIMATE message to all processes. Unlike the optimisation of [CT96] consensus mentioned at the end of Section 4.2, p_1 is not necessarily the coordinator of the FAST mode but any process output by Ω can be coordinator. Rounds in the NORMAL mode have the same steps as a round of the [CT96] consensus algorithm.

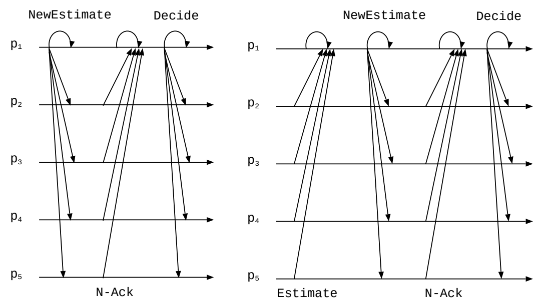

Figure 3. FAST and NORMAL mode

4.3.2 Detailed description

Input parameters for LeaseCons. LeaseCons consensus exports a function *propose()*. A process p invokes *propose()* with two parameters: cn and v_p. The parameter cn is the consensus number of the current consensus, and v_p is the value proposed by p for the consensus. As we discuss later LeaseCons in the context of atomic broadcast, we assume that each consensus instance is associated with a unique consensus number (cn). This consensus number is used while acquiring the lease. It is important to recall the difference between consensus number and round number: a consensus instance has a unique consensus number, whereas the consensus may proceeds into several rounds, each round with a different round number.

Function *rotate()*. The function *rotate()* evaluates the round number and the corresponding coordinator at a process. The first round number is chosen from $[0, n-1]$ such that the coordinator of the first round at each process p is the output of the local failure detector Ω (line 53). Selecting the round number on this criteria ensures that, eventually the coordinator (and the round number) of the first round is the same correct process at all correct processes. The first round is distinguished from other rounds by having a different phase 2 and we say that LeaseCons is in FAST mode.[8] All subsequent rounds have a round number greater than $n-1$ (line 55). Indeed, they have the same steps as a round in [CT96] consensus algorithm and we say that LeaseCons is in NORMAL mode.

Lease. Due to process crashes and unreliable failure detection, more than one process may trust itself to be the coordinator of the FAST mode. If more than one process tries to directly impose its value by sending a NEWESTIMATE message in FAST mode, there are runs which violate the agreement property of consensus. We use the lease to restrict the number of coordinators in FAST mode to at most one. Before starting as a coordinator of the FAST mode, a process needs to acquire the lease corresponding to the current consensus number. Even if more than one process tries to be the coordinator of the FAST mode, by the properties of the lease, at most one process succeeds in acquiring the lease and hence, at most one process sends a NEWESTIMATE message in FAST mode.

The $Lease\_status$ variable. The variable $Lease\_status$ is a static array at a process p (line 1 of Figure 4), i.e, the variable is shared by all instances of LeaseCons at p. The variable maintains the lease status of p for all consensus instances.[9] A process p tries to acquire lease for consensus number cn only if p wants to be the coordinator of the FAST mode of the consensus instance cn (i.e., the Ω failure detector module at p outputs p, line 53). Recall from Section 2 that a lease is always requested for an interval. Thus, to acquire the lease for consensus instance cn, p invokes $lease(cn, cn+\lambda)$ and stores the result (*true/false*) at every index of $Lease\_status$ within $[cn, cn+\lambda]$ (lines 16-18). Subsequently, for any consensus instance within $[cn, cn+\lambda]$, instead of invoking $lease()$ again, p can directly read the status of its lease acquisition from $Lease\_status$ (line 19).

Decision. Unlike the [CT96] consensus algorithm, the round number of the first round in LeaseCons (i.e., FAST mode) is not predetermined. It depends on the output of the Ω failure detector, and hence each process can have a different value of round number (and a different coordinator) in FAST mode. Recall that, in the [CT96] consensus algorithm, a process in round r sends and receives messages which are from round r. LeaseCons also uses the same scheme but this may block a process forever in LeaseCons. We show with the following two cases that introducing some extra messages avoids such deadlocks.

(1) Process p trusts itself as coordinator of the FAST mode (due to incorrect suspicion) and acquires the lease for the consensus, but only a minority of processes trust p as the coordinator of the FAST mode. Process p sends a NEWESTIMATE message and keeps on waiting in phase 4 for ACK/NACK messages from a majority of processes. Process p keeps on waiting since at most a minority of processes sends ACK/NACK messages to p. This case is avoided by sending a NACK message in reply to any NEWESTIMATE message sent in FAST mode if the receiving process does not trust the sending process as the coordinator of the FAST mode (line 48 of Figure 4).

(2) Process p trusts another process q as the coordinator of the FAST mode (due to incorrect suspicion), but process q does not trust itself to be the coordinator of the FAST mode (or fails to acquire the lease), and q is correct. In this case, p keeps on waiting for NEWESTIMATE messages from q and queries its failure detector (phase 3). It is possible that p never suspects q, since q is correct. So, p waits forever in FAST mode. If p is faulty, then

[8] Note that in FAST mode, phase 1 still occurs but it is not necessary since phase 2 does not wait for ESTIMATE messages.

[9] $Lease\_status$ can be more efficiently represented by an interval tree. (Section 3.3)

for each process p:
static $Lease\_status \leftarrow \{\perp, \perp, \ldots\}$ *{Executed once at p}*
procedure propose(cn, v_p)
upon initialisation **do**
$state_p \leftarrow undecided;\ estimate_p \leftarrow v_p;\ r_p \leftarrow \perp;\ c_p \leftarrow 0;\ ts_p \leftarrow 0;\ fast_p \leftarrow false$
$firstc_p \leftarrow \perp;\ \lambda \leftarrow$ default lease interval
while $state = undecided$ **do**
$(r_p, c_p) \leftarrow$ rotate()
if $(firstc_p = \perp)$ **then** $firstc_p \leftarrow c_p$ *{Remember the pid of coordinator of the* FAST *mode}*
Phase 1:
send (ESTIMATE,$p, r_p, estimate_p, ts_p$) to c_p
Phase 2:
if $(p = c_p)$ **then**
Phase 2F: *{Phase 2,* FAST *mode}*
if $(r_p < n)$ **then**
if $(Lease\_status[cn] = \perp)$ **then**
$tmp \leftarrow lease(cn, cn + \lambda)$
for $(cn \leq i \leq (cn + \lambda))$ **do**
$Lease\_status[i] \leftarrow tmp$
$fast_p \leftarrow Lease\_status[cn];\ Lease\_status[cn] \leftarrow false$
if $(fast_p)$ **then**
$estimate_p \leftarrow v_p$; send (NEWESTIMATE,$p, r_p, estimate_p$) to all
else
continue *{Go to* ***while*** $state = undecided$*}*
Phase 2N: *{Phase 2,* NORMAL *mode}*
else
wait until [for $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes q : received (ESTIMATE, $q, r_p, estimate_q, ts_q$) from q]
$msg_p[r_p] \leftarrow \{$(ESTIMATE,$q, r_p, estimate_q, ts_q$) $\mid$ (ESTIMATE,$q, r_p, estimate_q, ts_q$) p received from $q\}$
$t \leftarrow$ largest ts_q such that (ESTIMATE,$q, r_p, estimate_q, ts_q$) $\in msg_p[r_p]$
$estimate_p \leftarrow$ select one $estimate_q$ such that (ESTIMATE,$q, r_p, estimate_q, t$) $\in msg_p[r_p]$
send (NEWESTIMATE,$p, r_p, estimate_p$) to all
Phase 3:
wait until [(received(NEWESTIMATE,$c_p, r_p, estimate_{c_p}$) **or** (ABORT,c_p, r_p) from c_p) **or** $(c_p \in \Diamond S.suspected)$]
if [received (NEWESTIMATE,$c_p, r_p, estimate_{c_p}$) from c_p] **then**
$estimate_p \leftarrow estimate_{c_p};\ ts_p \leftarrow r_p$; send (ACK,$p, r_p$) to c_p
else
send (NACK,p, r_p) to c_p
Phase 4:
if $(p = c_p)$ **then**
wait until [for $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes q : received (ACK,q, r_p) or (NACK,q, r_p)]
if [for $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes q : received (ACK,q, r_p)] **then**
r-broadcast (DECIDE,$p, r_p, estimate_p$)
upon receive m from q **do**
if $(m.r_q < n)$ **then** *{Message from the* FAST *mode}*
wait until [phase 1 and phase 2 of the FAST mode are complete] *{Until* $fast_p$ *and* $firstc_p$ *are evaluated}*
if [$m =$ (ESTIMATE,$q, r_q, estimate_q, ts_q$) **and** $(!fast_p)$] **then**
send (ABORT,p, r_q) to q
if [$m =$ (NEWESTIMATE,$q, r_q, estimate_q, ts_q$) **and** $(firstc_p \neq q)$] **then**
send (NACK,p, r_q) to q
when r-deliver (DECIDE, $q, r_q, estimate_q$)
if $(state_p = undecided)$ **then** $decide(cn, estimate_q);\ state_p \leftarrow decided$

procedure rotate()
if $r_p \leftarrow \perp$ **then** *{If rotate is invoked for the first time}*
$r_p \leftarrow (\Omega.trusted) - 1$
else
$r_p \leftarrow Max\{r_p + 1, n\}$
return $(r, r\ mod\ n + 1)$

Figure 4. LeaseCons algorithm in a crash-stop model

this case does not violate termination. However, if p is correct then q eventually receives an ESTIMATE message (phase 1) from the FAST mode of p. If q does not send a NEWESTIMATE message in FAST mode (either because q does not trust itself to be coordinator or q does not get the lease), then q sends an ABORT message to p (line 46 of Figure 4) which terminates p's waiting in phase 3.

4.3.3 Correctness proofs

We now prove the consensus properties for LeaseCons. Due to the similarity with the [CT96] consensus algorithm, the proofs only consider the cases where LeaseCons is different from [CT96]. We introduce the notion of *interesting round* to simplify the description of the proof. A round is said to be *interesting* if a NEWESTIMATE message was sent in that round. If an *interesting round* is executed in FAST mode, then this round is denoted a *f-round*. All other *interesting rounds* are denoted *n-round*s. Note that, by the definition of FAST and NORMAL modes, a *f-round* has a round number less than n and executes phase 2F, whereas a *n-round* has a round number greater than or equal to n and executes phase 2N.

Lemma 5. *For an interesting round with round number k, if there exists another interesting round with round number l such that $l < k$ then k is a n-round.*

Proof. There are two cases to consider. First, round l is a *f-round* and assume p is the corresponding coordinator. Therefore, $fast_p$ is *true* at p in round l and *false* in all other rounds at all processes (at most once leasing property of lease). So k cannot be a *f-round*. However, k is an *interesting round*, hence k is a *n-round*. Second, round l is a *n-round*; thus, $l \geq n$, and $k > l \geq n$. So round k is a *n-round*. □

Lemma 6. *(Agreement) No two processes decide differently.*

Proof. If no process ever decides then the lemma is trivially true. If a process decides then it must r-deliver a DECIDE message. By the uniform integrity property of reliable broadcast, a coordinator has r-broadcast this message. By the algorithm of Figure 4, this coordinator has received a majority of ACK messages in phase 4. Let r be the smallest round number in which a majority of ACK messages were sent to the coordinator in phase 3. Let c denote the coordinator of round r, i.e., $c = (r \ mod \ n)$+1. Let $estimate_c$ be the estimate of process c, at the end of phase 2 in round r. By the algorithm of Figure 4, c must have sent a NEWESTIMATE message, and hence, round r is an *interesting round*.

We then claim that for all rounds $r' \geq r$, if r' is an *interesting round* then $estimate_{c_{r'}} = estimate_c$, where $c_{r'}$ is the coordinator of round r'. We prove this claim by induction on round numbers. The claim trivially holds for $r' = r$. Let k $(> r)$ be an *interesting round* and let us assume that the claim holds for all r' such that $r \leq r' < k$ (induction hypothesis). Let c_k be the coordinator of round k. Since r is an *interesting round* and $r < k$, from lemma 5, k is a *n-round*.[10] So, c_k must have received ESTIMATE messages from a majority of processes in phase 2N. Thus, there is a process p, such that (i) p sent an (ACK,p, r) message to c in phase 3 of round r, and (ii) (ESTIMATE,p, k,$estimate_p$, ts_p) belongs to $msg_{c_k}[k]$ in phase 2 of round k (line 27).

From the algorithm of Figure 4, $ts_p = r$ after round r (since p sent (ACK,p, r) message in round r). Since ts_p is non-decreasing, $ts_p \geq r$ in round k. Thus, if t is the highest timestamp in $msg_{c_k}[k]$, then $r \leq t < k$. Since there exists a process that has a timestamp equal to t, t is an *interesting round*. From the induction hypothesis and $r \leq t < k$, it follows that $estimate_{c_t} = estimate_c$. Since c_k adopts $estimate_{c_t}$ as its estimate, $estimate_{c_k} = estimate_c$. Hence, the claim is proved by induction. So, whenever a round is *interesting*, the estimate of the coordinator of that round at the end of phase 2 is $estimate_c$. From the algorithm of Figure 4, a DECISION message can only be sent in an *interesting round*. So, no process can decide differently from $estimate_c$. □

Lemma 7. *(Termination) Every correct process eventually decides some value.*

Proof. There are two possible cases. First, some correct process p decides, then p r-broadcasts a

[10]Note that proving k is a *n-round* is crucial. Otherwise, if k is an *f-round*, c_k would not have waited for ESTIMATE messages from other processes.

DECIDE message, and by the agreement property of reliable broadcast every correct process r-delivers that DECIDE message and decides. Second, no correct process decides. We claim that no correct process remains blocked forever at one of the **wait** statements. The proof is by contradiction. Let r be the smallest round number in which some correct process gets blocked forever at one of the wait statement. We only consider here FAST mode, i.e., the first round of the consensus and $r < n$. If $r \geq n$ then the case is similar to the [CT96] consensus algorithm. For phase 1, it is trivial since there are no wait statements. For phase 2, there is no blocking, since $r < n$, indeed a process executes phase 2F and there are no wait statements in phase 2F and the lease invocation is wait-free (see lemma 2). Therefore, the wait statement in line 44 is non-blocking since phase 1 and 2 of FAST mode are non-blocking. For phase 3, a process may possibly get blocked forever while waiting for NEWESTIMATE or ABORT messages. Let p be a correct process that waits for a NEWESTIMATE or an ABORT message from the coordinator k in round r, i.e., $r < n$ and $k = (r \, mod \, n)$+1. There are two cases to consider, (i) k crashes: due to the strong completeness property of the failure detector $\diamond S$, p suspects k and terminates the wait statement, and (ii) k is correct, there are again two sub-cases to consider:

1. The FAST mode at process k has a round number r' $(< n)$, different from r. Process k does not trust itself to be the coordinator of the first round ($k = (r \; mod \; n)$+1 $\neq$ $(r' \; mod \; n)$+1). Thus, k does not execute phase 2F and variable $fast_k$ at k is always *false*. Process k eventually receives an ESTIMATE message from p and sends an ABORT message in reply. Since both (p and k) are correct, p eventually receives this ABORT message and does not block at the wait statement.

2. The first round at process k has a round number r. Indeed, k trusts itself as a coordinator of the FAST mode and executes phase 2F. There are further two sub-cases:

2.1. Process k acquires the lease for this consensus. Process k sends a NEWESTIMATE message with round r to p. When p receives such message, it terminates the wait statement.

2.2. Process k fails to acquire the lease for this consensus. Variable $fast_k$ at k is *false*, therefore k sends an ABORT message in reply to the ESTIMATE message from p. Thus, p does not block at the wait statement.

For phase 4, assume that p is a correct process that waits for ACK or NACK messages from a majority of processes in phase 4 (of the first round). The correct processes that trust p as the coordinator of the FAST mode sends ACK or NACK messages to p in phase 3 of the FAST mode (since no correct process waits forever in phase 1-2). The correct processes for which p is not the coordinator in the first round ($firstc_p \neq p$) eventually receives a NEWESTIMATE message sent by p in phase 2. So, they reply with a NACK message to p (line 47-48). Thus, p receives ACK or NACK messages from every correct process, i.e., a majority of correct processes. Therefore, p eventually terminates the wait in phase 4. □

Proposition 8. *The algorithm of Figure 4 solves consensus in an asynchronous system augmented with* $\diamond S$*, and a majority of correct processes.*

Proof. The Ω failure detector can be reduced to $\diamond S$ failure detector in asynchronous systems [Chu98]. From the algorithm of Figure 4, it is clear that no process decides more than once (integrity). It is also clear that a coordinator receives only ESTIMATE messages that are proposed values (validity). The agreement and termination properties follow from lemmata 6 and 7. □

5 Lease Applications

5.1 LeaseCons atomic broadcast

Atomic broadcast is a useful primitive in distributed computing that ensures total order delivery of messages. It is defined by the two primitives a-broadcast and a-deliver. Intuitively, processes do not only agree on the set of messages they a-deliver, as with reliable broadcast, they also agree on the sequence of messages they a-deliver [HT93]. [CT96] gives an atomic broadcast algorithm based on consensus as an underlying building block. Basically, the processes execute a sequence of consensus instances, each instance being used to agree on a batch of messages: the processes use the same deterministic function to deliver messages within the same batch.

By plugging our LeaseCons consensus algorithm into the atomic broadcast algorithm of [CT96] (instead of the [CT96] consensus algorithm), we obtain a faster atomic broadcast algorithm, which we denote here *LeaseCons atomic broadcast*. To ease reading, the atomic broadcast based on the [CT96] consensus algorithm is denoted [CT96] atomic broadcast. We first define the notion of *stable period* and then describe the conditions under which LeaseCons terminates in FAST mode.[11] We then compare the performance of our LeaseCons consensus algorithm (resp. LeaseCons atomic broadcast) with the [CT96] consensus algorithm (resp. [CT96] atomic broadcast).

Stable period. We define a *stable period* as a period where (i) no processes crash, and (ii) the underlying failure detectors do not make any mistakes. Note that processes can crash before or after a stable period, but not during a stable period. In context of LeaseCons, a stable period implies that all participating processes start with the same coordinator (and hence, the same round number) in FAST mode. Furthermore, if the coordinator succeeds in acquiring the lease, then the LeaseCons algorithm terminates in FAST mode in a stable period.

Stand-alone performance comparison. If our LeaseCons algorithm terminates in FAST mode, the algorithm requires five communication steps: two steps for acquiring the lease, one step for sending and receiving the NEWESTIMATE message, one step for sending and receiving ACK messages, and one final step for the reliable broadcast. Whereas the [CT96] consensus algorithm requires four communication steps.[12] Thus, our LeaseCons algorithm is less efficient than the [CT96] consensus algorithm for stand-alone consensus.

LeaseCons atomic broadcast performance. The LeaseCons algorithm exports the same set of properties as the [CT96] consensus algorithm. Therefore, one can directly use the atomic broadcast algorithm of [CT96] to transform LeaseCons to LeaseCons atomic broadcast. Consider the following two observations regarding LeaseCons atomic broadcast.

- *If there are always new messages being atomic broadcast (and hence, new LeaseCons instances are launched), eventually all new LeaseCons instances terminate in* FAST *mode.* Since (i) eventually only correct processes are up, and (ii) the failure detector Ω eventually outputs the same correct process p_c at all correct processes. Process p_c is then trusted as the coordinator of the FAST mode by all correct processes. Therefore, p_c is the only process that requests new leases and succeeds in acquiring them. Process p_c sends a NEWESTIMATE message to all processes. Since all correct processes trust p_c, p_c eventually receives ACK messages from every correct process (a majority) and r-broadcasts a DECIDE message. Thus LeaseCons terminates in FAST mode.

- *A process does not invoke* $lease()$ *more than once per* λ *instances of LeaseCons.* Recall the discussion about the $Lease\_status$ variable from Section 4.3. λ is the interval size for which a process tries to acquire the lease. Assume that, for the LeaseCons instance numbered cn, coordinator of the FAST mode, c_p, invokes $lease()$. Either the invocation returns *true* or *false*, or c_p crashes. If c_p crashes then obviously it stops invoking $lease()$. If the invocation returns, then $Lease\_status[x]$ ($\forall x \in [cn, cn + \lambda]$) at c_p is assigned the value returned by $lease()$. By the LeaseCons algorithm, subsequently, c_p does not invoke $lease()$ till LeaseCons instance number $cn + \lambda$, as it already knows the $Lease\_status$ value for those consensus instances.

From the above observations, eventually only one in every λ instances of LeaseCons requires five communication steps ($lease()$ invoked in FAST mode), whereas the remaining instances require three communication steps ($lease()$ not invoked in FAST mode). Thus on average, each instance of LeaseCons requires $3 + \frac{2}{\lambda}$ communi-

[11] I.e., a DECIDE message is r-broadcast in FAST mode.

[12] Three steps for sending and receiving the following messages ESTIMATE, NEWESTIMATE, ACK/NACK, and one step for the r-broadcast of the DECIDE message.

cation steps. In practice, this observation holds whenever there is a long enough stable period.

This can be further illustrated by the following example (Figure 5) which shows the number of communication steps required by runs of LeaseCons atomic broadcast. We assume runs consisting of 100 instance of consensus in stable period and vary the value of λ in the LeaseCons implementation. For comparison purposes, we also provide the number of communication steps required by [CT96] atomic broadcast under similar conditions. Note that we do not count the number of communication steps required outside the consensus abstraction but used by atomic broadcast, e.g., the r-broadcast inside atomic broadcast, since these messages are equal in number for both cases. Figure 5 summarises the performance communication-step wise of the LeaseCons (resp. [CT96]) atomic broadcast implementation. The number of communication steps for [CT96] atomic broadcast is always 400 since 4 communication steps are required per consensus instance ($4 * 100 = 400$ communication steps). The number varies for LeaseCons atomic broadcast, since 3 communication steps are mandatory per consensus instance. The reminder comes from the lease invocation (2 communication steps). We divide the number of consensus instances (100) by λ to find out the number of lease invocations, e.g., for $\lambda = 10$, 10 invocations are mandatory ($100/\lambda = 10$). The total number of communication steps for LeaseCons atomic broadcast with $\lambda = 10$ is $(3 * 100) + (2 * 10) = 320$ communication steps. Note, in Figure 5, the dramatic decrease in the number of communication steps between $\lambda = 1$ and $\lambda = 10$. Note also that for $\lambda = 20$ and up, the gain in the number of communication steps is not important compared to the possible drawback in case the coordinator of the FAST mode crashes or the system becomes unstable.

λ	1	10	20	50	100
LeaseCons Atomic Broadcast	500	320	310	304	302
CT96 Atomic Broadcast	400	400	400	400	400

Figure 5. Impact of λ on the number of communication steps for 100 consensus instances in a stable period

5.2 Comparison of atomic broadcast performance

In this section we compare the performance of atomic broadcast [CT96] when different consensus algorithms are used as the underlying module. We impose the following restrictions for effective comparison:

1. We consider the following consensus algorithms used in the atomic broadcast: (i) [CT96] consensus algorithm with the simple optimisation for the first round (p_1 directly imposes its $estimate$), (ii) Early consensuses [Sch97], (iii) Fast Consensus [HR99], and (iv) LeaseCons with $\lambda = 10$.

2. We consider the runs in stable period only, which is the most frequent case. It is not clear how to compare the algorithms for the runs in which coordinators crash or the failure detectors make mistakes.

3. We do not count the communication steps which are not directly used by the consensus, but are required by the atomic broadcast, because those steps are same in number in every algorithm.

4. We consider the following failure patterns in a system composed of more than 4 processes, (i) $F0$: No process crashes before the stable period, [13] (ii) $F1$: Process p_1 crashes before the stable period starts, and (iii) $F2$: Processes p_1 and p_2 crash before the stable period starts. Recall that no process crashes during the stable period.

Figure 6 presents the number of communications steps required for runs of atomic broadcast with 100 consensus instances. The best-case performance of first three consensus algorithms ([CT96, Sch97, HR99]) are dependent on whether process p_1 is up or not. Thus, their performance is optimal in the failure-free case (failure pattern $F0$), and degrades when p_1 crashes (failure pattern $F1$ and $F2$). On the other hand, in a stable period, the performance of LeaseCons is independent of previously crashed processes. It is also important to notice that [CT96] and LeaseCons follow a centralised message exchange pattern (messages are received from, and sent to, the

[13] This case is similar to the notion of a nice run.

coordinator only), whereas [Sch97] and [HR99] follow a decentralised pattern. Hence, [CT96] and LeaseCons has $O(n)$ message complexity in a point-to-point network as compared to $O(n^2)$ for [Sch97] and [HR99].

	Failure Patterns		
	F0	F1	F2
CT96	300	400	400
Sch97	200	400	600
HR99	200	300	400
LeaseCons	320	320	320

Figure 6. Number of communication steps in atomic broadcast with 100 consensus instances

5.3 Alternative Applications

The notion of leasing in a distributed system was first introduced in the context of file caching [GC89]. Subsequently, leasing was always associated with the presence of a global clock. The primary objective of this paper was to specify and implement a leasing mechanism in a completely asynchronous setting, using logical time intervals instead of relying on a global clock. Furthermore, we illustrate an application of asynchronous lease in improving the performance of an atomic broadcast algorithm using a lease based consensus (LeaseCons). On a similar approach, LeaseCons can be applied to improve the performance of repeated non-blocking (weak) atomic commit for instance, that uses consensus as an underlying module [GS01]. It is important to notice that using LeaseCons (as a consensus module) to solve group membership does not always improve performance. In group memberships, consensus is invoked usually for view changes [GS01] which often implies unstable period, whereas LeaseCons is effective mostly in stable periods.

6 A Concluding Remark

To conclude, we would like to focus on the problem of determining a suitable lease interval, λ. Our implementation of LeaseCons (Figure 4) has a hard-coded lease interval, i.e., the interval does not change depending on the state of the system. Choosing a suitable lease interval λ is an optimisation problem and dependent on the network parameters. Selecting a large λ has the advantage of decreasing the number of lease invocations and thus increasing the performance of the atomic broadcast in stable periods. However, if a process crashes after acquiring a lease with a large interval, the optimisation of LeaseCons cannot be applied anymore until the end of that interval. Indeed, a prudent approach is to select a variable interval. An algorithm to determine λ may follow these simple heuristics:

- A priori, one cannot determine whether a process is correct or not. Thus, a lease should not be acquired for an infinite duration, i.e., there should be a finite maximum size for λ, λ_{max}.
- A process may start with $\lambda = \lambda_{max}/2$. To avoid a race condition when acquiring leases, a process should decrease its λ once a lease invocation returns *false* (a possible indication of unstable period). Subsequently, λ should be gradually increased once lease invocations start returning *true*. Note that an optimistic algorithm could start with $\lambda = \lambda_{max}$ if it is a known fact that the system is stable at initialisation.

References

[ACT00] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, May 2000.

[CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[Chu98] F. Chu. Reducing Ω to $\diamond$W. *Information Processing Letters*, 67(6):289–293, September 1998.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. McGraw-Hill Book Company, 1990.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[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.

[GC89] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[GS01] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, January 2001.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HR99] M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4):209–223, 1999.

[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.

[Lam89] L. Lamport. The part-time parliament. Technical Report 49, Systems Research Center, Digital Equipment Corp, Palo Alto, September 1989. A revised version of the paper also appeared in ACM Transaction on Computer Systems, 16(2):133-169, May 1998.

[Lam96] B. Lampson. How to build a highly available system using consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG-10)*, pages 1–15, Bologna, Italy, 1996.

[Sch97] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.