Asynchronous Leasing

Partha Dutta   Romain Boichat   Rachid Guerraoui
Communication Systems Department
Swiss Federal Institute of Technology, CH-1015 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 time and was hence strongly tied to synchronous system models.

This paper introduces a new notion of lease devised for an asynchronous system model. We give the precise properties of our lease abstraction, we show how it can be implemented in an asynchronous system model, and we then illustrate its use by significantly improving the performance of a consensus-based atomic broadcast algorithm.

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. In practical transactional systems, locks are indeed leases: if they expire, the transaction is aborted. The notion of lease was also shown to be very powerful 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. Not surprisingly, lease implementations typically use time-outs and assume synchronised clocks. More precisely, 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]. The motivation of this work was to figure out whether some concept of lease could make sense in an asynchronous system model. Of course, the traditional concept of lease is impossible to implement in such a system model because process relative speeds and communication delays are unbounded. An interesting question follows: can we devise a *useful* notion of lease that is *implementable* in an asynchronous system model?

We show in this paper that the answer is *yes*. Roughly speaking, we introduce a concept of *asynchronous lease* where we replace the notion of time period with the notion of positive integer interval, i.e., we measure the term of a lease using a positive integer interval. A process (1) should not acquire a lease for an interval that overlaps with an interval for which a lease has already been granted, and (2) should 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 over a resource for a given interval of positive integers does not say much about any form of exclusive access to the resource at a given point in time. So what does an asynchronous lease buy us?

Not surprisingly, an asynchronous lease provides the adequate semantics when the mutual exclusion does not need to be ensured with respect to time, but with respect to intervals of positive integers. We illustrate the use of this notion by showing how it can significantly improve the performance of the atomic broadcast algorithm of [CT96]. The algorithm was devised as a sequence of consensus instances and, intuitively, we use our notion of lease 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 consensus communication step and directly impose its decision. In stable periods of the system (the most frequent periods in practice), we reduce the number of messages and communication steps needed to reach a final decision within each of these consensus instances. Interestingly, we apply our notion of lease to the atomic broadcast algorithm of [CT96] without actually changing the algorithm: we plug-in a specific *fast* consensus algorithm obtained by incorporating our notion of asynchronous lease into the consensus algorithm of [CT96].

The use of leasing in this manner is not new. Lampson suggested in [Lam96] the use of leases to improve the performance of the Paxos replication 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. Our concept of asynchronous lease could easily be adapted to the Paxos algorithm [Lam89], which is also devised as a sequence of consensus instances. In general however, it is not clear when exactly our notion of asynchronous lease can replace the traditional synchronous one. Studying the general applicability of asynchronous leases is however out of the scope of this paper.

In the context of this paper, we assume the crash-stop model of [CT96]. 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 an 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 an implementation of it and Section 4 illustrates its use. Section 5 concludes the paper with a practical observation.

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*

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

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 two following 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*, then

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

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

Let $LI_{s,e} = \{lease(s', e') \in LI \mid (s \leq s' \leq e) \vee (s \leq e' \leq e)\}$ where $s, e, s', e' \in \mathbb{N}$, then

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

To illustrate the semantics of a lease object, Figure 1 gives 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 indeterminism 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*.

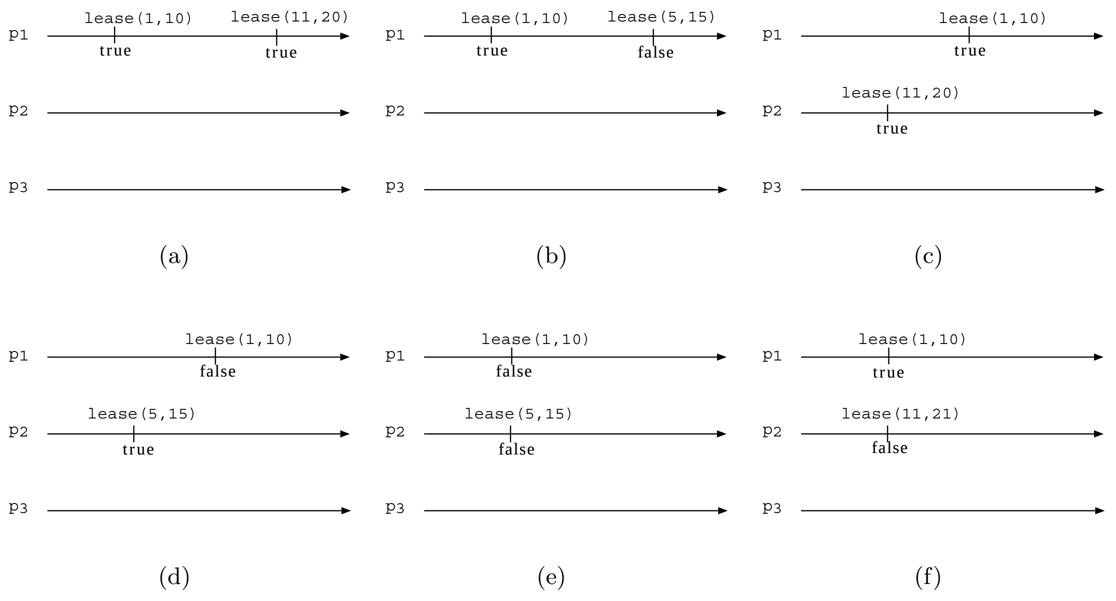

Figure 1: Lease examples

3 Lease Implementation

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 array *permission* which is initially empty. On invoking $lease(s, e)$, a process p sends a LEASE message m to all processes. On receiving this LEASE message m, a process q replies with an ACK_LEASE message if all values from $permission[s]$ to $permission[e]$ are $\perp$. Otherwise, q replies with a NACK_LEASE message.[2] 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 sends NACK_LEASE messages. If p (the process which invoked $lease(s, e)$) receives a majority of ACK_LEASE messages, the $lease(s, e)$ function returns *true*, otherwise the function returns $false$.[3]

```
1: for each process $p$:
2: **initialisation:** $permission[] \leftarrow \{\perp, \perp, \ldots\}$; $tmp \leftarrow true$
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** $(permission[i] = \perp))$; $permission[i] \leftarrow q$
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

Lemma 1. *(At most once leasing) No two leases have overlapping intervals.*

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 interval $[s', e']$, and (iii) the two intervals overlap.[4] When p or q acquire 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 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 $\perp$, k replies with a NACK_LEASE message to all subsequent (LEASE,*,*) messages whose interval contains x, e.g., m_2: a contradiction. □

Lemma 2. *(Mandatory leasing) 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 assume that there is no lease

[2]Note that p needs to consider the ACK_LEASE or NACK_LEASE messages corresponding to the current lease request. Therefore, the replies are tagged with the original LEASE message m to distinguish between the replies from two different lease invocations.

[3]This crash-stop implementation of *lease* (Figure 2) can be readily 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).

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

invocation with an interval that overlaps with $[s, e]$. Suppose 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 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. □

Lemma 3. *(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. □

Proposition 4. *The algorithm of Figure 2 implements a wait-free lease with a majority of correct processes.*

Proof: Immediate from lemmata 1, 2, and 3. □

4 Lease Application

This section describes an application of our asynchronous lease abstraction. We devise a new consensus algorithm, denoted *FastCT* which solves consensus in an asynchronous system augmented by a $\diamond S$ failure detector and the assumption of a majority of correct processes. FastCT is 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 for round 0 to impose directly its value. If the [CT96] atomic broadcast algorithm is based on our FastCT consensus algorithm (instead of the [CT96] consensus algorithm, i.e., uses our lease abstraction), then fewer communication steps are required. We first recall in this section the basics of the [CT96] consensus algorithm. We then explain the main differences between the [CT96] consensus algorithm and our FastCT algorithm. Finally, we give the correctness proofs of FastCT, and we discuss the performance of a FastCT atomic broadcast algorithm.

4.1 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.

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

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[6] 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 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.

4.2 FastCT Algorithm

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

4.2.1 Intuitive description

FastCT has two modes: (i) a FAST mode where a process tries to directly impose its value, and (ii) a NORMAL mode that roughly corresponds to the [CT96] consensus algorithm. A FAST mode consists of the first round of FastCT whose round number can be any integer between 0 and $n-1$. 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: the coordinator does not wait for ESTIMATE messages but directly sends a NEWESTIMATE message to all processes. Unlike the optimisation mentioned previously, p_1 is not necessarily the

[6]The algorithm also uses a reliable broadcast abstraction with two primitives R-Broadcast and R-Deliver [HT93].

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

coordinator of the FAST mode but any process can be coordinator. The NORMAL mode has the same steps as a round of the [CT96] consensus algorithm.

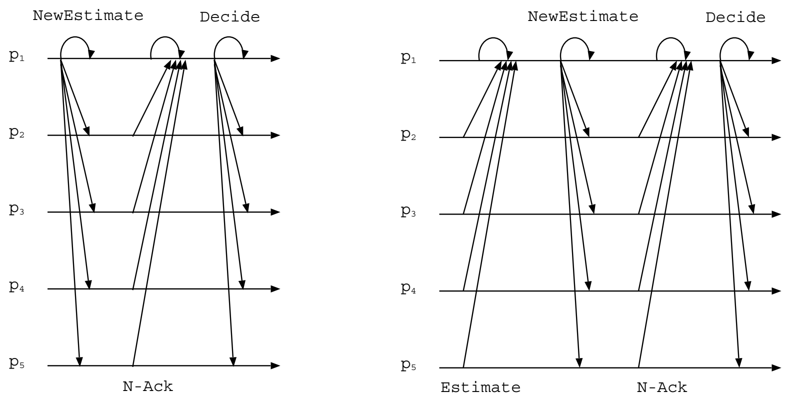

Figure 3: FAST and NORMAL mode

4.2.2 Detailed Description

Input parameters for FastCT. FastCT 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 FastCT 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.

The *Lease_status* variable. The variable *Lease_status* is a static array at a process p (line 2 of Figure 4), i.e, the variable is shared by all instances of FastCT at p. The variable maintains the lease status of p for all consensus instances. Recall that a lease is always requested on an interval (see Section 2). The value returned for the invocation of $lease(cn, cn + \lambda)$ is stored in *Lease_status* at all indices within $[cn, cn + \lambda]$. Therefore, subsequent consensus instance whose consensus numbers are within that interval can directly read from the array *Lease_status* and hence, do not need to invoke the *lease*() function, i.e., only one *lease*() invocation is made per λ instances of FastCT. Note that if $Lease\_status[cn] = \perp$ then p has not yet requested a lease for cn. If $Lease\_status[cn] = true$ then p has requested a lease for cn and has acquired the corresponding lease. Finally, if $Lease\_status[cn] = false$ then p has requested a lease for cn and it was refused.

Function *rotate*(). The function *rotate*() evaluates the round number and the corresponding coordinator at a process p. Remember that the FAST mode consists of only one round and has a round number between 0 and $n - 1$. The NORMAL mode consists of an unlimited number of rounds where the round number is greater than $n - 1$. For the FAST mode, i.e., the first round of the consensus, its round number is chosen such that eventually, at all correct processes the round number of the FAST mode is the same.[8] After the first round, the *rotate*() function forces

[8]I.e., all correct processes trust the same process as the coordinator of the FAST mode. To ensure the claim, we transform the failure detector $\diamond S$ to the failure detector Ω. Task $Transform$ maps the output of $\diamond S$ to Ω using the *Slander* algorithm given by [Chu98]. The $Min(count_p)$ in the function *rotate*() emulates the output of the failure detector Ω. Eventually, all processes output the same process in $Min(count_p)$.

p to jump to at least round number n. This jump, in fact, forces p to change from FAST mode to NORMAL mode.

Lease in FastCT. 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 here 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.

Decision. Unlike the [CT96] consensus algorithm, the round number of the first round in FastCT (i.e., FAST mode) is not pre-determined. It is dependent on the output of the 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. FastCT also uses the same scheme but this may block a process forever in FastCT. 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 this case does not violate termination. However, if p is correct then q eventually receives an ESTIMATE message (phase 1) from 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) q sends an ABORT message to p (line 46 of Figure 4), which terminates p's waiting in phase 3.

4.2.3 Correctness Proofs

We now prove the consensus properties for FastCT. Due to the similarity with the [CT96] consensus algorithm, the proofs only consider the cases where FastCT 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-rounds*. 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.*

```
for each process $p$
static $Lease\_status \leftarrow \{\perp, \perp, \ldots\}$; $trust_p \leftarrow p$; $count_p \leftarrow \{0, 0, \ldots\}$; start task $Transform$   {Executed once at $p$}
procedure propose($cn, v_p$)
  upon initialisation do
    $state_p \leftarrow undecided$; $estimate_p \leftarrow v_p$; $r_p \leftarrow 0$; $c_p \leftarrow 0$; $ts_p \leftarrow 0$; $fast_p \leftarrow false$; $firstc_p \leftarrow \perp$; $\lambda \leftarrow$ default 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,$q, r_p, estimate_q, ts_q$) | (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 (!$firstc_p$)] 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()
  upon initialisation: $next \leftarrow Min(count_p)$-1   {return the first index with the minimum element}
  $r \leftarrow next$; $next \leftarrow Max\{next + 1, n\}$; return $(r, r \bmod n + 1)$

task Transform
  repeat forever
    for ($q \in \diamond S.suspected$) do
      $count_p[q] \leftarrow count_p[q] + 1$
    send (FD, $count_p$) to all
  upon receiving (FD,$count_q$) from $q$ do
    $count_p \leftarrow Max(count_p, count_q)$   {Element-wise maximum of the two arrays}
```

Figure 4: FastCT algorithm in a crash-stop model

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.*[9] 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 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

[9]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.

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. 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. □

4.3 Fast Atomic Broadcast

Atomic broadcast is a useful primitive in distributed computing that ensures total order delivery of messages among processes. Intuitively, processes do not only agree on the set of messages they deliver, as with reliable broadcast, they also agree on the sequence of messages they 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 FastCT 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 *FastCT atomic broadcast.* To simplify 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 FastCT terminates in FAST mode.[10] We then compare the performance of our FastCT consensus algorithm (resp. FastCT

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

atomic broadcast) with the [CT96] consensus algorithm (resp. [CT96] atomic broadcast).

Stable period. We define a *stable period* as a period where (i) no process crashes or recovers, (ii) a process p is up, and (iii) every process that invokes $propose()$ trusts p as the coordinator of the FAST mode. If a FastCT instance is launched in a *stable period*, then 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 FastCT algorithm terminates in FAST mode.

Consensus performance. If our FastCT 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.[11] Thus, our FastCT algorithm is less efficient than the [CT96] consensus algorithm for stand-alone consensus.

Atomic broadcast performance. As our FastCT consensus algorithm exports the same primitive ($propose()$) and satisfies the same properties of the [CT96] consensus algorithm, we can use exactly the same atomic broadcast algorithm given in [CT96] that transforms consensus into atomic broadcast. We now give a simple sketch explanation of the following claim. *If there are always new messages being broadcast (and hence, new FastCT instances are launched), eventually all new FastCT instances terminate in* FAST *mode.* Since (i) eventually only correct processes are up, and (ii) task $Transform$ emulates the failure detector Ω, then $Min(count_p)$ 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 FastCT terminates in FAST mode.

Recall the discussion about the *Lease_status* variable from Section 4.2, it is sufficient to invoke $lease()$ once per λ number of FastCT instances.[12] In runs of atomic broadcast where there are a large number of consensus instances and a value of λ large enough, the number of communication steps required for acquiring the lease is insignificant. This can be further illustrated by the following example that compares the performance of FastCT and [CT96] atomic broadcast. We assume runs with a stable period, 100 consensus instances and different values of λ for the FastCT atomic broadcast implementation. 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 FastCT (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 FastCT 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 FastCT 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

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

[12]λ is the default lease interval in Figure 4.

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
FastCT 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 A Practical Remark

The algorithm of FastCT presented in 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 FastCT cannot be applied anymore until the end of this interval which is not efficient at all. Indeed, a prudent approach is to select a variable interval. An algorithm that selects λ should follow these simple observations.

- A priori, one cannot determine whether a process is correct or not. Thus, a lease should not be acquired for a infinite duration, i.e., there should be a finite maximum size for λ, e.g., λ_{max}.
- Eventually, all runs of atomic broadcast reach a stable period. Therefore, λ should eventually be equal to λ_{max}.
- A process should 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 $false$. 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.

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

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

[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.

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

[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 Equipement Corp, Palo Alto, September 1989. A revised version of the paper also appeared in TOCS vol.16 number 2.

[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.