FaTLease: Scalable Fault-Tolerant Lease Negotiation with Paxos

Felix Hupfeld*, Björn Kolbeck*, Jan Stender*, Mikael Högqvist*,
Toni Cortes†‡, Jonathan Martí†, Jesús Malo†

*Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
†Barcelona Supercomputing Center (BSC), c/ Jordi Girona, 1-3, Barcelona, Spain
‡Universitat Politècnica de Catalunya (UPC), c/ Jordi Girona, 31, Barcelona, Spain

ABSTRACT

A lease is a token which grants its owner exclusive access to a resource for a defined span of time. In order to be able to tolerate failures, leases need to be coordinated by distributed processes. We present FaTLease, an algorithm for fault-tolerant lease negotiation in distributed systems. It is built on the Paxos algorithm for distributed consensus, but avoids Paxos' main performance bottleneck of requiring persistent state. This property makes our algorithm particularly useful for applications that can not dispense any disk bandwidth. Our experiments show that FaTLease scales up to tens of thousands of concurrent leases and can negotiate thousands of leases per second in both LAN and WAN environments.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*

General Terms

Algorithms, Theory

1. INTRODUCTION

Many replication systems are based on a primary/secondary scheme where a designated primary replica acts as a sequencer of operations and is thereby responsible for the consistency of replicated data [5, 8, 15]. An example of this design is master-slave database replication where a statically-configured host assumes the task of enforcing a sequential order on operations.

The basic primary/secondary approach can be made fault-tolerant by adding a mechanism to hand over the sequencer role of the primary replica in case of a failure. Leases [6, 11] can be used for this task by designating their exclusive owner as the primary replica and by relying on their built-in timeout as a revocation mechanism in case of failures.

In general, a lease is issued to grant a designated host exclusive access to a resource for a limited period of time. This implies that the issuer must make sure that at any point in time, at most one valid lease exists for a resource. When leases are coordinated in a distributed manner, lease exclusiveness becomes a distributed consensus problem [4].

Paxos is an algorithm that solves distributed consensus while tolerating host failures, network partitions, and message loss. Paxos uses stable storage to ensure correctness after a host has recovered from a crash. Issuing a lease corresponds to a round of Paxos, which requires two writes to stable storage. This means that the maximum number of leases the system can concurrently handle is bound by the local disk bandwidth. In addition, the delay caused by the disk writes reduces the total response time of the system.

In this paper, we present the FaTLease algorithm, which tightly integrates leases with Paxos. By taking advantage of the timeouts that are associated with leases, it removes the necessity for stable storage [1] while maintaining the properties of Paxos.

We have implemented FaTLease as part of XtreemFS [7], a replicated object-based file system. XtreemFS employs a primary/secondary replication scheme using leases to elect one primary per file. This implies that file servers (object storage devices) must be able to handle large amounts of leases concurrently, since each open file is associated with one lease. Using regular Paxos would severely reduce the performance of read and write operations, as it requires two disk writes per lease negotiation.

Our paper is structured as follows. We first review related work (Sec. 2) and give an introduction to the Paxos algorithm (Sec. 3). In Section 4 we describe FaTLease, including its distributed consensus algorithm and the logic that extends the consensus algorithm to a distributed lease negotiation algorithm. In this context, we describe how correctness is ensured without the need for persistent state. In Section 5 we investigate how many leases our lease negotiation algorithm can handle in local and wide area networks and compare the results to regular Paxos. An additional set of experiments compares the disk bandwidth required by FaTLease and regular Paxos and demonstrates the effect a write-intensive application has on the performance of both algorithms.

HPDC'08, June 23–27, 2008, Boston, Massachusetts, USA.

2. RELATED WORK

The Paxos algorithm provides distributed consensus in unreliable environments. The algorithm is described in [9, 10] and thoroughly discussed in [1]. The general approach for adding a namespace to Paxos in order to run multiple instances of a distributed consensus has been sketched in [13] as Multipaxos. In [11] Lampson outlines how to implement highly-available services using Paxos and leases.

Results from [8] indicate that primary/secondary replication performs better than quorum-based approaches such as replicated state machines with Paxos. In [11], Lampson seconds this argument and suggests that Paxos should only be used to negotiate leases, while other update dissemination mechanisms should be used for actual data replication.

An alternative approach to elect a primary is to use a centralized lock service. In this approach, clients acquire locks from a central server that ensures exclusive access to a resource. Locks associated with timeouts are essentially equivalent to leases. In contrast, FATLEASE does not have a centralized server, but negotiates the lease among all participating hosts. Numerous lock services have been developed, here we present a selection of recent implementations that rely on Paxos.

Chubby [2, 3] is a fault-tolerant lock service. It has been designed to run on a small number of hosts, where each host holds a replica of a simple database. Any access to Chubby goes through a designated primary, which is elected with Paxos. The primary replicates all data modifications to the replicas also by using Paxos. Locks can be requested at the primary, which acts as a centralized lock service. Information on locks is stored in the replicated database so that another replica can take the role of the primary in case of failures. Basically, this means that each lock operation (acquire and release) requires a modification of the database, which results in a round of Paxos. To optimize throughput, multiple operations executed on the primary are replicated using a single round of Paxos. As stated by the authors, Chubby was designed as a low-volume file system with coarse-grained locks. Scalability is achieved by creating an arbitrary number of independent Chubby installations. In the Google File System [5], Chubby is used to elect primaries which ensure data consistency.

The Boxwood [12] file system implements a centralized locking service with master and slaves. It uses Paxos to replicate the state of the leases issued to clients. Similarly, the Frangipani file system [14] distributes the responsibility of the locks among all available lock servers. Paxos is used to replicate the locks and the partitioning information. This allows the other lock servers to take over the partition of a failed server.

3. PAXOS

This section gives a brief overview of the Paxos algorithm. We consider this section necessary, as FATLEASE is tightly interwoven with Paxos. Detailed descriptions of Paxos are given in [9, 10].

The Paxos algorithm allows multiple hosts to reach consensus on a single value. By relying on majority decisions, Paxos is able to tolerate message loss, network splits and host failures as long as responses from a majority of hosts are available. These fault tolerance properties combined with the need for only two communication round-trips to reach

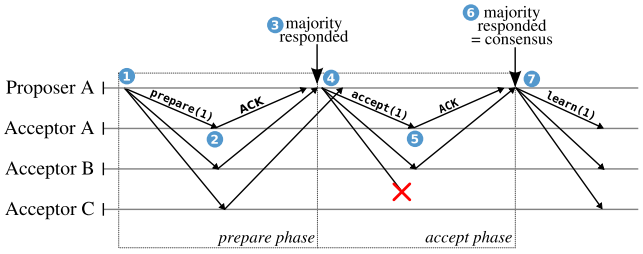


Figure 1: Example of a single round of Paxos with three acceptors (A,B,C). The two phases of Paxos are marked. The 'x' indicates message loss.

consensus have made Paxos a widely-adopted solution to the problem of distributed consensus in system development.

With Paxos, a host may adopt two roles: the *proposer* and the *acceptor* role. Proposers initiate the consensus process by sending proposals to acceptors. Each proposal consists of the proposed value itself and a globally unique ballot number that allows acceptors to chose one proposal over the other. Acceptors arbitrate concurrent proposals and decide on at most one of the proposals that may have been concurrently submitted. Subsequent reruns of the consensus process always result in the chosen value and can be used to learn about the result of the consensus.

For each submitted proposal, the consensus process consists of two phases of message exchange from the proposer to the acceptors (see Fig. 1). In the *prepare phase*, acceptors guarantee they will not accept any other proposal with a lower ballot number. In the *accept phase*, acceptors commit to one proposal and accept it. As soon as one proposal has been received and stored by a majority of the acceptors, the proposal is the consensus and can be disseminated to all participants.

To ensure correctness in face of crash-recovery of acceptors, Paxos requires that acceptors keep their state in stable storage. This implies that for each prepare and accept message, the acceptor has to update its state before responding. For a single round of Paxos, this leads to at least two writes to stable storage on each acceptor.

4. DISTRIBUTED LEASE NEGOTIATION

The goal of FATLEASE is to provide a method to issue leases in a distributed and fault-tolerant manner. In order to guarantee exclusive access to a resource, the algorithm must preserve the *lease invariant*. That is, all valid leases for a single resource must have the same lease owner.

A local instance of the algorithm runs on each host in the system. It provides operations to *acquire*, *renew* or *invalidate* a lease. At its core, the algorithm is based on a variant of Multipaxos, which offers *Paxos instances* as a means of reaching consensus on a particular lease (see Fig. 2), and which separates consecutive lease negotiation rounds. As part of the Multipaxos variant, each host runs a Paxos proposer and a Paxos acceptor. The lease negotiation logic guides Multipaxos in its creation of Paxos instances, so that the lease invariant is never violated. Lease negotiation operates in *cells*, which are namespaces that separate the lease negotiation process between resources.

This section is organized as follows. We first describe the system model, introduce basic definitions, and explain how

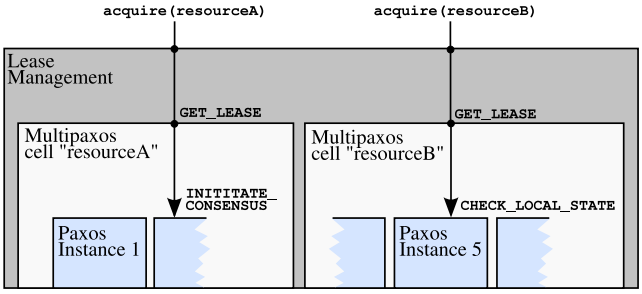


Figure 2: Lease management creates Paxos instances inside Multipaxos cells to negotiate single leases.

the FATLEASE algorithm acquires, renews, and invalidates leases. Then we discuss how the algorithm exploits an inherent notion of timeouts to avoid the persistent state that Paxos would normally require.

4.1 System Model and Definitions

We assume networks to be unreliable in the sense that messages can be lost, delayed or duplicated but message content is not altered. All system components adopt a crash-recovery failure model: a host may lose its non-persistent state when crashing, but can re-join the system afterwards.

We expect host clocks to be loosely synchronized, i.e. the difference between any two clocks does not exceed a certain maximum. Moreover, we assume that all components can be trusted, i.e. no byzantine behavior is considered.

Our protocol runs on a set of hosts $H = \{h_1, \ldots, h_n\}$. We formally define a lease λ as consisting of a lease owner $h \in H$ and an absolute time stamp $t \in \mathbb{N}$. $\lambda = (h,\ t)$ expresses that h holds the lease until the point in time t.

Each host $h_k \in H$ holds the state of the latest locally-known Paxos instance $p_k = (\lambda_{lrn},\ \lambda_{acc},\ b,\ i)$. $\lambda_{lrn},\ \lambda_{acc} \in (H \times \mathbb{N}) \cup \{\perp\}$ denote the lease eventually agreed on, and the last lease locally accepted, respectively, where $\perp$ expresses that no lease information is known. $b \in \mathbb{N}$ denotes the largest accepted ballot number, and $i \in \mathbb{N}$ denotes the number of the instance.

Furthermore, each host $h_k \in H$ is associated with a local clock c_k. $c_k(t)$ denotes the time returned by c_k at time t. We rely on the assumption that clocks are loosely synchronized, i.e. that an upper bound d_{max} exists, with $d_{max} \geq \max_{t,j,k} \{|c_j(t) - c_k(t)|\}$.

We call a lease $\lambda = (h_k,\ t)$ valid as long it has not yet timed out from the lease owner's point of view, i.e. $c_k(t_{now}) < t$ if t_{now} denotes the current point in time.

For all leases acquired by any host, we restrict the relative period of time in which the lease is valid to v. Our protocol requires that $v > d_{max}$.

4.2 Reaching Consensus on Leases

We formulate the lease negotiation problem as a distributed consensus problem for a lease λ. With the aim of reaching consensus on λ, a host starts the consensus process sending λ as a proposal to the acceptors of all hosts through its local proposer. The Paxos algorithm chooses at most one proposal as the consensus, which becomes the current valid lease.

Consecutive leases for the same resource are each negotiated in a separate Paxos instance that is identified by a unique number i. We implemented Multipaxos [13] to handle multiple Paxos instances. Multipaxos only separates the consensus of single instances, but does not further restrict them. In the context of lease negotiation, the lease negotiation logic that uses Multipaxos must control the creation of Paxos instances in such a way that the lease invariant is never violated. This means that the negotiation process for a new lease must not be started before the previous lease has timed out, i.e. the instance $i + 1$ may only be created when the instance i has timed out. In order to allow gap-free lease renewals, we will relax this requirement later so that two valid leases can exist when they have the same lease owner.

Paxos instances are automatically *started* as soon as an acceptor receives a Paxos message for that particular instance. Once consensus has been reached, the proposer will inform all acceptors of the outcome by sending them a *learn* message. Upon receiving this *learn* message, acceptors consider the instance to be *complete*. When the lease of an instance has timed out, the instance is considered to be *outdated* and the next instance can be started for a new lease.

In an unreliable environment, a host may miss single Paxos messages, which results in incomplete instances. A host may even completely miss the existence of instances. The former case is handled by re-running Paxos for that instance, the latter requires a mechanism for hosts to catch up to the current instance. We introduce a new message for this purpose: when a host that has been left behind sends messages in an instance that has already timed out, it will receive *outdated* responses from all acceptors that know of a newer instance. If a majority of acceptors respond, there is at least one acceptor that knows the current instance. The proposer simply takes the most recent instance number from the responses and then re-submits its proposition in the current instance.

We will now present the pseudo-code for the Multipaxos algorithm, with an extension that considers timed-out instances. The code is split into the proposer and the acceptor parts. The `INITIATE_CONSENSUS` procedure implements a proposer and is executed to acquire a lease. The event handlers `PREPARE`, `ACCEPT` and `LEARN` implement the acceptor role. We ommitted error handling code to enhance readability.

```
PROCEDURE INITIATE_CONSENSUS(λ, i)
  -- prepare phase (step 1, Fig. 1)
  -- generate a unique ballot number
  b ← GENERATE_UNIQUE_BALLOT()
  SEND PREPARE(b, i) TO H
  P ← RECEIVE FROM a majority of H
  IF ∃p ∈ P: p is OUTDATED THEN
    -- Multipaxos message to find current instance
    p_l ← (⊥, p.λ_acc, p.b, p.i)
    INITIATE_CONSENSUS(λ, p.i)
  ELSE IF ∃p ∈ P: p is NACK THEN
    -- another proposal had a higher ballot number
    wait some time
    INITIATE_CONSENSUS(λ, i)
  ELSE IF |{p ∈ P | p is ACK}| ≥ ⌊|H|/2 + 1⌋ THEN
    -- majority agreed (step 3, Fig. 1)
    P_acc ← {p ∈ P | p is ACK ∧ p.λ_acc ≠ ⊥}
    IF P_acc ≠ ∅ THEN
      -- an acceptor demands that some prior value is used
      P_max ← {p ∈ P_acc | p.b = max_{p∈P_acc} {p.b}}
      b ← p.b, p ∈ P_max
```

```
    λ ← p.λ_acc, p ∈ P_max
  END IF
  -- accept phase (step 4, Fig. 1)
  SEND ACCEPT(λ, i, b) TO H
  A ← RECEIVE FROM a majority of H
  IF |{a ∈ A | a is ACK}| ≥ ⌊|H|/2 + 1⌋ THEN
    -- majority accepted (step 6, Fig. 1)
    p_l ← (λ, λ, b, i)
    -- send outcome to all participants (step 7, Fig. 1)
    SEND LEARN(p_l) TO H
  ELSE
    -- restart with prepare phase
    wait some time
    INITIATE_CONSENSUS(λ, i)
  END IF
END IF
```

On receiving *prepare* and *accept* messages, acceptors initially compare their local instance number with the one sent by the proposer. If the proposer is not up-to-date, i.e. the proposer sent a smaller instance number than the greatest instance number known to the acceptor, an textitoutdated message is sent; if a proposer uses a newer instance than the one known to the acceptor, the acceptor will discard its old state and start a new instance. The latter situation can happen when acceptors miss messages or even full instances. In most cases, both proposer and acceptor know the same instance and plain Paxos can be executed.

To speed up the algorithm, we use *nack* messages in order to indicate that an acceptor knows of a more recent proposal. In plain Paxos as described in Sec. 3, acceptors simply fail to respond if they know a newer proposal, which may cause a communication timeout on the proposer side if not enough acceptors respond. Sending *nack* messages instead is an optimization, which does not affect the correctness of Paxos.

```
UPON PREPARE(b, i)
  -- step 2 in Fig. 1
  IF i < p_l.i ∧ p_l.λ_lrn = ⊥ THEN
    -- proposer has missed some instance(s)
    SEND OUTDATED(⊥, p_l.λ_acc, p_l.b, p_l.i)
  ELSE
    IF i > p_l.i THEN
      -- local acceptor has missed some instance(s)
      p_l ← (⊥, ⊥, b, i)
      SEND ACK(⊥, ⊥, b, i)
    ELSE IF b < p_l.b THEN
      -- acceptor has seen a newer proposal
      SEND NACK(⊥, ⊥, p_l.b, i)
    ELSE
      -- acceptor agrees to the proposal
      p_l.b ← b
      IF p_l.λ_acc ≠ ⊥ THEN
        -- acceptor forces proposer to use prior value
        SEND ACK(⊥, p_l.λ_acc, p_l.b, i)
      ELSE
        SEND ACK(⊥, ⊥, b, i)
      END IF
    END IF
  END IF

UPON ACCEPT(λ, i, b)
  -- step 5 in Fig. 1
  IF i < p_l.i THEN
    -- acceptor does not vote for values in outdated instances
    SEND NACK
  ELSE IF i > p_l.i THEN
    -- acceptor has missed instance and votes for value
    p_l ← (⊥, λ, b, i)
    SEND ACK
  ELSE IF b ≥ p_l.b THEN
    -- acceptor votes for proposal
    p_l.λ_acc ← λ
    SEND ACK
  ELSE
    -- acceptor has seen newer proposal and cannot accept
    SEND NACK
  END IF

UPON LEARN(λ, i)
  IF i > p_l.i THEN
    -- instance is unknown and needs to be created
    p_l ← (λ, ⊥, 0, i)
  ELSE IF i = p_l.i THEN
    -- instance is known:
    -- the consensus outcome (lease) is stored in the instance
    p_l.λ_lrn ← λ
  END IF
```

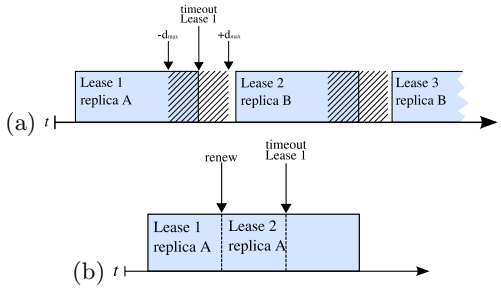


Figure 3: Lease timeouts and effect of loosely synchronized clocks and lease renewal. The safety period is marked with dashes.

4.3 Lease Operations

A crucial design goal of FATLEASE is to minimize the amount of messages that need to be exchanged in order to determine the current lease owner. A host therefore ought to be able to decide locally on the existence of a valid lease. We set the stage for this by disseminating Paxos *learn* messages that convey the latest lease which consensus has been reached on.

An issue that still remains is clock asynchrony. We cannot assume that the clocks of the participating hosts are closely synchronized. Since we have defined lease validity from the perspective of the lease owner's clock, we need to take care that no violation of the lease invariant can occur due to skews in the clocks of other hosts.

FATLEASE relies on the presence of loosely-synchronized host clocks. The algorithm assumes that some external protocol (like the Network Time Protocol) puts an upper limit on the clock drift between hosts. Incorporating a safety period d_{max} greater than the maximum clock drift guarantees that hosts can decide locally whether a lease has timed out, even when timeouts are interpreted based on local clocks (see Fig. 3a).

Acquiring Leases

Leases are acquired with the `GET_LEASE` procedure, which implements the lease logic by controlling the creation of new Paxos instances. The `GET_LEASE` procedure first queries the local Multipaxos acceptor with `CHECK_LOCAL_STATE` to find out what the locally known state of the lease for the resource is. Depending on the information returned, the procedure

might start the consensus process or continue with the information locally available.

Essentially, CHECK_LOCAL_STATE distinguishes between the cases that no lease is known, a valid lease is known, an outdated lease is known, and a potentially valid lease is known.

In the first case, no *learn* message has been received in the latest known instance, which causes GET_LEASE to propose its own lease in this instance. Which of the remaining three cases comes into play depends on the lease timeout, the local clock and the question whether the lease is held locally. In each of the cases, the current time on the local clock is compared to the lease timeout. If the current time is smaller than the lease timeout in case the local host is the lease owner, or the timeout decreased by d_{max} in case the local host is not the lease owner, the lease must be valid and can hence be immediately returned by GET_LEASE. Otherwise, if the current time is greater than the lease timeout and the lease timeout increased by d_{max}, respectively, the last known lease is outdated, and a proposal can be started in the following instance. In any other case, the lease is in the safety period $[t - d_{max}, t + d_{max}]$ and potentially valid, which means that the host has to wait until the lease has certainly timed out before proposing in a new instance.

In case the host has missed Paxos instances, invoking INITIATE_CONSENSUS will help it to catch up to the latest instance in the system. With each invocation, the result may be an *outdated* message, in case a newer instance exists that has already been completed (Fig. 1, *learn* message) and resulted in a valid lease. By repeating proposition attempts with increasing version numbers, the process will either end up in an instance with a valid lease and learn about it, or join an incomplete instance and participate in the arbitration of the next lease with its own proposal.

```
PROCEDURE GET_LEASE(): H × ℕ
  WHILE true DO
    λ ← CHECK_LOCAL_STATE()
    IF λ = none THEN
      -- the local information is outdated
      -- next instance is used
      INITIATE_CONSENSUS( (h_l, c_l(t_now) + v), p_l.i + 1)
    ELSE IF λ = locally_unknown THEN
      -- no information on current lease and instance
      -- can be deduced - the last known instance is used
      INITIATE_CONSENSUS( (h_l, c_l(t_now) + v), p_l.i)
    ELSE IF λ = wait THEN
      -- lease is in safety period
      -- wait before starting new instance
      wait some time
    ELSE
      -- lease information is available
      RETURN λ
    END IF
  DONE

PROCEDURE CHECK_LOCAL_STATE():
            (H × ℕ) ∪ {none, locally_unknown, wait}
  IF p_l.λ_lrn = ⊥ THEN
    -- no local information available
    RETURN locally_unknown
  ELSE IF p_l.λ_lrn.h = h_l THEN
    -- local host is primary
    IF c_l(t_now) ≤ p_l.λ_lrn.t THEN
      -- lease is still valid
      RETURN p_l.λ_lrn
    ELSE
      -- lease is outdated
      RETURN none
    END IF
  ELSE
    -- remote host is primary
    IF c_l(t_now) + d_max ≤ p_l.λ_lrn.t THEN
      -- lease is valid and not in safety period
      RETURN p_l.λ_lrn
    ELSE IF c_l(t_now) − d_max > p_l.λ_lrn.t THEN
      -- lease is outdated
      RETURN none
    ELSE
      -- lease is in safety period
      RETURN wait
    END IF
  END IF
```

Renewing Leases

With the algorithm as presented, a host that wants to extend its lease ownership beyond the expiration time of the current lease would have to try to acquire the next lease after the current lease has expired. By starting a proposal in instance n after the lease in instance $n-1$ has already become invalid, the lease owner would lose its ownership of the resource at least until consensus has been reached in n. During this time, other hosts could also compete for the lease in n and succeed.

In order to enable safe and gap-less lease renewal, we allow a lease owner to create a Paxos *renew instance* n already before its lease in $n-1$ has timed out (Fig. 3b). As this causes two valid leases with overlapping validity time spans to exist, it is necessary to ensure that the lease owner in n will be the same as in $n-1$, since otherwise, the lease invariant would be violated. With respect to this, we have to consider the possibility that hosts which do not own the lease and which have missed instance $n-1$ become aware of the existence of the renew instance n. Under these circumstances, it is necessary to keep such hosts from starting concurrent attempts to acquire the lease in the renew instance. If one of their attempts was successful, the lease invariant could not be preserved anymore.

We solved the problem by checking the previous instance before proposing in an incomplete instance. This implies that in addition to the current instance n (referred to as p_l in the pseudo code), hosts also have to store the previous instance $n-1$. The instance in which a new proposal is issued may now depend on the result in $n-1$. If n is incomplete, proposals may only be issued in this instance if the lease in $n-1$ is no longer valid. If the lease may still be valid or $n-1$ was missed, the proposal must be submitted in $n-1$ instead. Because this distinction of cases would have made the pseudo-code more complex and difficult to understand, we omitted the concept of renew instances from the presentation.

Invalidating Leases

When a host does not need a valid lease anymore, it should be able to invalidate it, so that another host can get a new lease without waiting for the current one to expire. The lease owner can invalidate a lease by sending an invalidation message that causes its receivers to proceed to the next Paxos instance immediately, without waiting for the current instance to time out. Receiving an invalidation message causes a host to set p_l to its successor instance, and therefore a new Paxos instance will be used on its next attempt

to acquire the lease.

```
PROCEDURE INVALIDATE()
  IF h_l = p_l.λ_lrn.h THEN
    -- local host is lease owner
    -- invalidate the local lease
    p_l ← (⊥, ⊥, 0, i + 1)
    -- disseminate invalidation messages
    SEND INV(p_l.i) TO H \ {h_l}
  END IF

UPON INV(i)
  IF i ≥ p_l.i THEN
    -- current or later lease is invalidated
    -- create a new lease in the next instance
    p_l ← (⊥, ⊥, 0, i + 1)
  END IF
```

4.4 Crash Recovery without Persistent State

The important aspect of our algorithm is its ability to avoid persistent state that Lamport's original Paxos requires [10]. In Paxos, a response to a prepare message implies a guarantee given by the acceptor that it will not respond to another proposal with a lower ballot number. This guarantee must be valid for the lifetime of the entire system. Therefore, acceptors must persistently store their internal state (ballot number and accepted value) in order to be able to recover from a crash. Acceptor state is changed when responding to either a prepare or an accept message, which involves two modifications in each round of Paxos. In FATLEASE, this refers to all state contained in p_l.

A trivial way to avoid persistent state would be to enforce crash-stop behavior on acceptors. If acceptors did not recover from crashes, the system would hang forever as soon as the majority of acceptors have crashed. However, we can take advantage of the fact that our consensus values are leases, i.e. have a limited lifetime. Since leases that have timed out are no longer of any use, consensus state becomes obsolete when the lease has expired. Acceptors can simply discard such state. Likewise, acceptors do not need to restore such state before rejoining the system after a crash. Since a lease will be regarded as invalid by all hosts after a period of at most $v + d_{max}$ has elapsed, waiting for this time before rejoining the system renders all local state obsolete. Therefore, hosts wait for $v + d_{max}$ before rejoining, rather than relying on persistent state.

With the basic algorithm as described in Sec. 4.2, discarding host state may violate the lease invariant. Since host clocks are not perfectly synchronized, hosts may dispose of obsolete state at different points in time. This may lead to a system configuration where a majority of hosts have already discarded their state, while a minority still knows the latest instance n (see Fig. 4a). Under these circumstances, the majority could successfully negotiate a new lease among each other in the initial instance 1. The minority might still know the formerly latest instance n and could therefore issue proposals in instance $n + 1$ (see Fig. 4b). Since FATLEASE as described in Sec. 4.2 will always accept proposals in higher instances than locally known, such proposals may be successful. This might lead to an illegal overlap in the validity time periods of instances 1 and $n + 1$.

We augment the algorithm such that an acceptor always checks its local state for a non-outdated instance with a lower instance number. If such an instance exists, the acceptor will reject the proposal and include the instance number of the valid lease in its response (see Fig. 4c). When a majority of acceptors answers with a *reject* message, the proposer can safely discard its local state and restart with the instance number received from the acceptors. Receiving reject messages from a majority indicates that this majority has already discarded its old state and restarted with instance 1. In the case where a majority has already accepted prepare messages in instance $n + 1$, the hosts will send *outdated* messages for proposals in any instance with a lower instance number (see Fig. 4d). The proposer will catch up with the current instance, as described in Sec. 4.2, `INITIATE_CONSENSUS`, lines 7-9. Both cases ensure that the lease invariant cannot be violated.

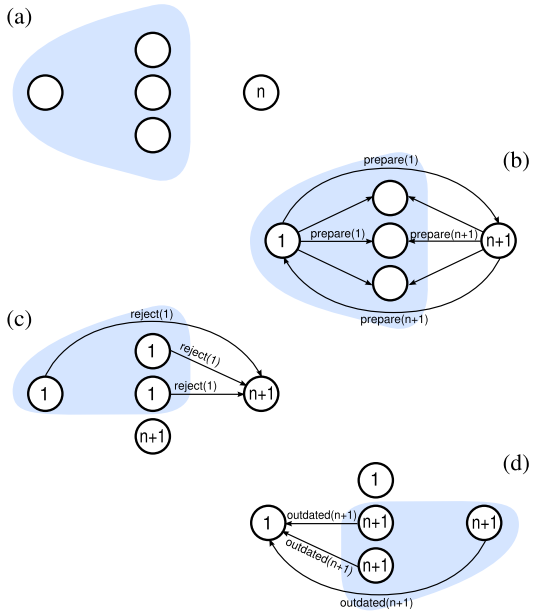


Figure 4: State is discarded asynchronously. This can lead to a majority of hosts (encircled) with empty state and a minority still having the old state (a). (b) illustrates concurrent proposals by hosts from both groups. In (c) instance 1 wins, in (d) instance $n + 1$ wins.

5. EVALUATION

We first demonstrate the scalability of the FATLEASE algorithm and show that it performs well even on wide area networks. In addition, we compare the performance of FATLEASE and regular Paxos.

We have implemented the lease negotiation algorithm as a single-threaded event-driven lease manager stage [16] as part of the XtreemFS file server [7]. In the file server, the FATLEASE algorithm is used to negotiate per-file leases for electing a temporary master that ensures data consistency for concurrent write operations. Lease negotiation messages are sent via UDP, since Paxos can tolerate message loss.

To allow for a fair comparison of FATLEASE and regular Paxos, we have used the same code base for both algorithms. The regular Paxos requires state to be written to hard disk for each prepare and accept. Before sending a response, the acceptor must ensure that writes are persistent by calling

`fsync`. To achieve a higher throughput for regular Paxos, we introduced an extra thread that writes multiple messages (if available) to disk before calling `fsync`.

5.1 Scalability: Number of Leases

The first experiment evaluates how FATLEASE performs under peak loads of concurrent lease requests. In this context, performance is the number of leases the system can handle per second.

More specifically, we measured the time from starting the first request until the last lease is acquired. This duration is divided by the number of lease requests, which gives us the throughput in terms of leases per second. We also measured the number of communication timeouts as they are expected to have an effect on the throughput. The number of failed leases, i.e. leases that cannot be acquired after seven retries, is used to estimate the system's maximum throughput.

We used three setups to point out different factors that limit the algorithm's throughput. First, a setup where all components are executed on a single machine is used to assess the maximum throughput of our implementation. Second, we estimate the effect of maximum network throughput by using two machines connected via a Gigabit Ethernet LAN. Finally, the effect of network link saturation is investigated in a wide area setup. All measurements were done with FATLEASE and with regular Paxos using stable storage.

Setup

The experiment ran on three machines, two of them (A and B) located at our site and one PlanetLab node located in Spain (C). Machines A and B are connected via a Gigabit Ethernet LAN and have 4 cores (Xeon with 2GHz and 3GHz) and 4 GB of memory each. Machine C has a single CPU (Pentium D 3.2GHz), 1 GB memory. The connection between A and C has a bandwidth of approx. 512kB/s per direction with a ping round-trip-time of 66ms.

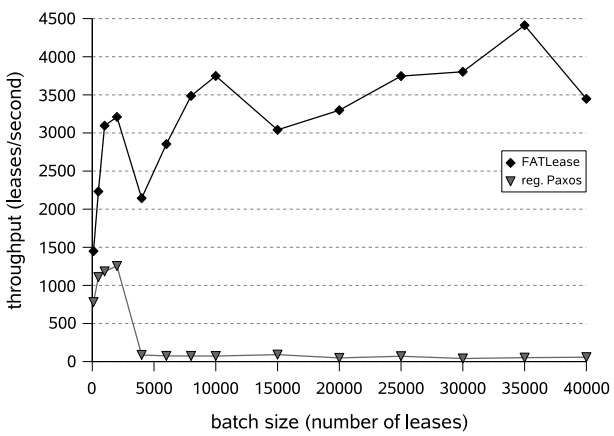
We used these machines in three different setups. In the *single machine setup* we executed one proposer and one acceptor on machine A. For the *LAN setup* we executed one proposer on machine A and the acceptor on machine B. In the *WAN setup* the proposer was executed on Machine A while the acceptor was running on the remote machine C.

A communication timeout is counted each time a proposer does not receive a response from a majority to a prepare or accept request within one second. For each lease the proposer initiates up to seven rounds before giving up (failure). To simulate request peaks, we submitted 100 to 40,000 lease requests in a single batch.

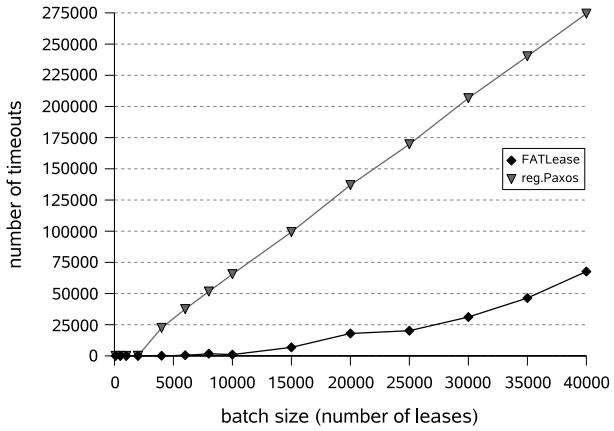
Results and Analysis

For each setup, we measured the throughput, the number of timeouts and the number of failed leases. Figure 5 shows the results for the single machine, Figure 6 for the LAN and Figure 7 the WAN setup. Unfortunately, the Linux VServer used on PlanetLab nodes does not pass the `fsync` operation to the underlying host system, therefore we were unable to give results for regular Paxos in the WAN setup.

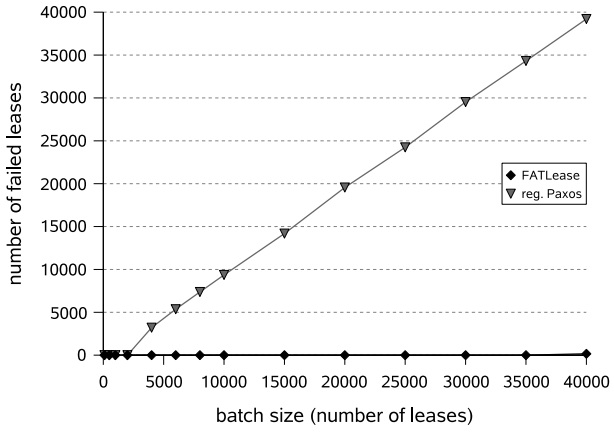
The single machine setup demonstrates the limits of our implementation. With standard hardware, a single lease negotiation thread is able to handle up to 35,000 lease requests without failures. With more leases, requests and incoming packets cannot be processed in time, which causes frequent timeouts (Fig. 5b).



a) throughput (leases per second)

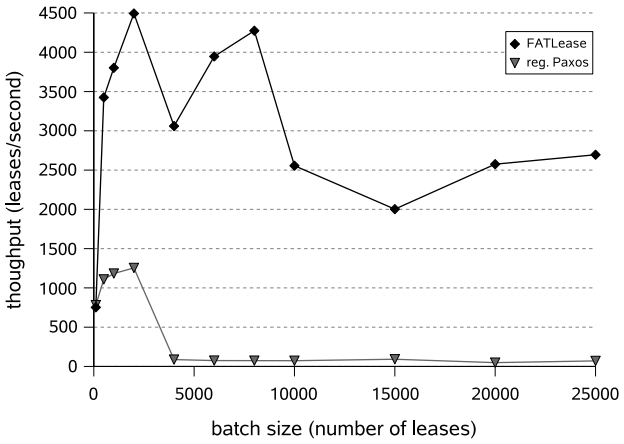


b) number of timeouts

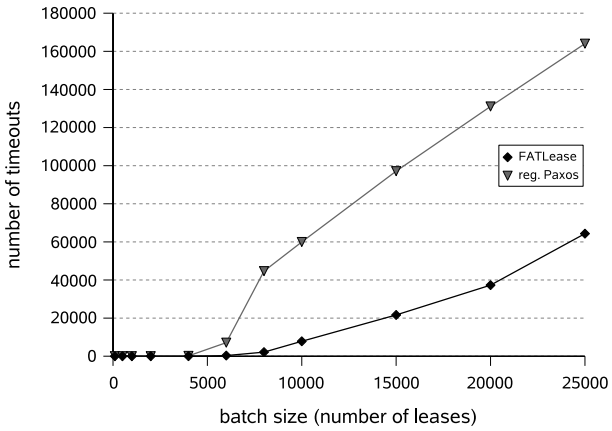


c) number of failed leases

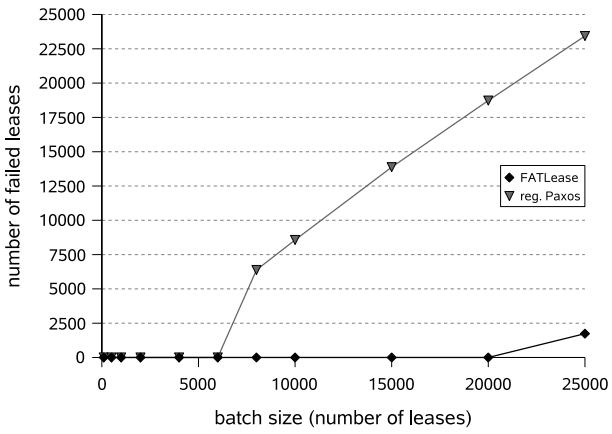
Figure 5: Concurrent lease negotiation in the single machine setup for an increasing number of leases.



a) throughput (leases per second)



b) number of timeouts



c) number of failed leases

Figure 6: Concurrent lease negotiation in the LAN setup for an increasing number of leases.

In the single machine and LAN setups, FATLease has a sharp drop in throughput for 4,000 leases. This is due to the fact that at 4,000 leases the first timeouts occur. This means that the proposer had to wait 1 second and then retransmit the request. So, these timeouts cause the proposer to take more time, which reduces the throughput. For regular Paxos, Fig. 5a and 6a show that the performance drops between 2,000 and 4,000 leases. This happens because disk bandwidth is not sufficient to ensure that all writes are synced within the one-second timeout. The number of timeouts and failed leases increases linearly with the batch size.

In the WAN setup, a maximum throughput is reached at a batch size of 500 leases. Here the outgoing bandwidth is approximately the same as the actual bandwidth of the network link (512kB/s). For a batch size of 1,000 leases timeouts occur which reduce the throughput in the same way as in the single machine and LAN setting.

With a throughput of more than 2,500 leases per second for 20,000 concurrent requests, we have demonstrated that FATLease is able to perform well even under heavy load on a local LAN. The WAN setup also demonstrates that our protocol is able to handle high lease volumes of up to 1,500 leases per second with low bandwidth (peak of accumulated bandwith is 750kB/s). In contrast to Paxos, FATLease is not limited by the disk bandwidth, which leads to a significantly higher throughput.

5.2 Scalability: Number of Hosts and Latency

In this experiment we investigate how network latency and the number of acceptors influence the duration of the lease negotiation with FATLease and regular Paxos. Since the number of acceptors would be equal to the number of secondary replicas in a replication system, this is an important measure to rule out the lease negotiation as a bottleneck.

Setup

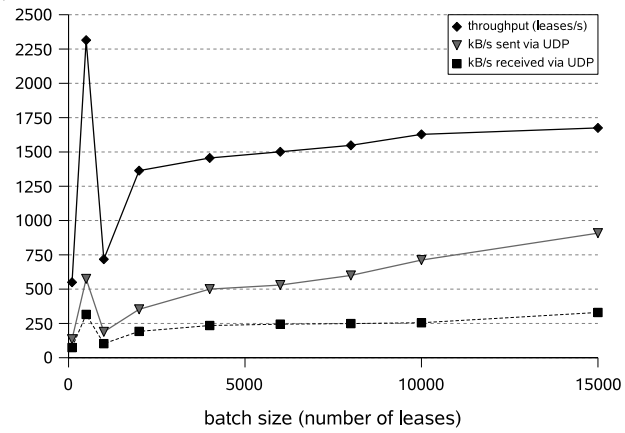
We used thirty machines, each running a single acceptor. A single proposer was run on machine B (see Sec. 5.1). All machines are connected through a Gigabit Ethernet LAN. To simulate additional network latencies between the proposer and acceptors, we delayed UDP packets. All acceptor machines have two dual core Xeon 2.66GHz processors and 8GB RAM.

We measured the duration for the negotiation of a single lease (from starting the request until receiving the response) for 1 to 30 remote acceptors. Two experiments were conducted, one with a round-trip time (RTT) of less than 1ms, and the other one with a simulated RTT of 55ms. For each data point we took the average of three measurements.

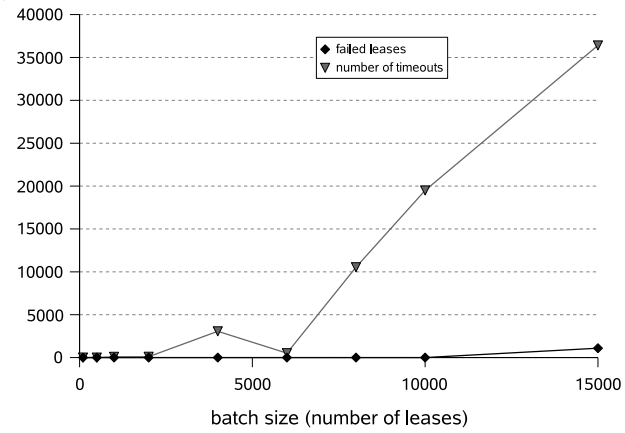
Results and Analysis

The duration is plotted in Figure 8, which shows that it stays constant even for increasing numbers of remote acceptors. This indicates that the lease negotiation is not a bottleneck when scaling the number of hosts.

It is clearly visible that the duration of the negotiation actually depends on the RTT. As consensus requires two rounds of communication, the network latency sets the lower bound for the duration. The additional latency of the two `fsync` operations required for regular Paxos is around 33ms (difference between the duration for FATLease and regular Paxos in Figure 8).



a) throughput (leases per second) and network bandwidth



b) number of failed leases and number of timeouts

Figure 7: Concurrent lease negotiation in the PlanetLab WAN setup.

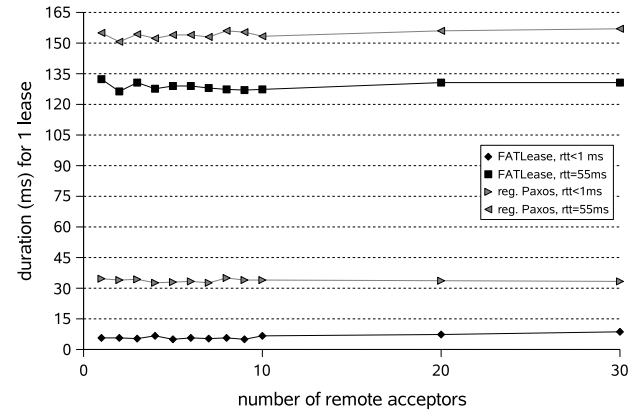


Figure 8: Scalability of lease negotiation depending on the number of hosts.

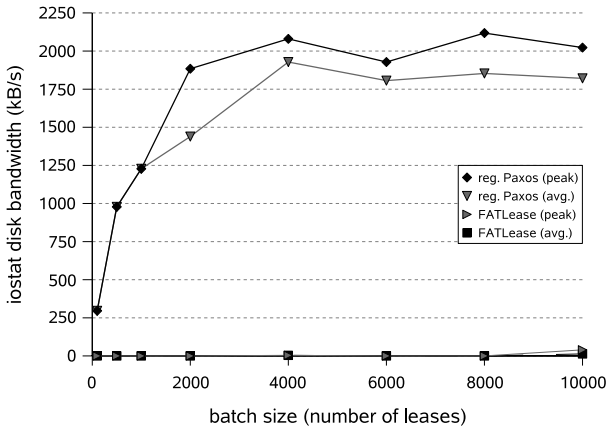


Figure 9: Disk bandwidth in kB/s reported by iostat

5.3 Effect of Disk Bandwidth

To demonstrate the effect regular Paxos has on the disk bandwidth, we conducted two further experiments. In the first experiment, we measured the disk bandwidth in the single machine setup for both FATLEASE and regular Paxos. In the second experiment, we show how an application that requires the full bandwidth for its operation affects the lease negotiation.

Setup

We used the same setup as in the single machine setup in Sec. 5.1. To measure the disk bandwidth we used `iostat` and took the peak and average values for the duration of the lease negotiation. For the second experiment, we used the IOzone file system benchmark to create heavy write load on the hard disk by executing a throughput test with a single writer of a 2 GB file (`iozone -t 1 -s2G`).

Results and Analysis

The disk bandwidth reported by `iostat` is plotted in Fig. 9. For regular Paxos, a maximum of 2,000 kB/s is reached for 4,000 leases. The fact that disk throughput stays nearly constant for larger batches indicates that the disk is congested. Consequently, the lease throughput decreases (see Fig. 5a and 6a). In contrast, FATLEASE shows a disk bandwidth which is nearly zero.

In the second experiment (Fig. 10), regular Paxos was not able to negotiate a single lease. Due to the high disk usage generated by IOzone, acceptors could not respond within the one-second response timeout interval. As expected, the massive disk usage had only little effect on FATLEASE. However, the throughput is lower than in the single machine setup in Sec. 5.1, which is due to the CPU usage of IOzone.

6. CONCLUSION

We presented FATLEASE, a fault-tolerant lease negotiation algorithm based on Paxos. FATLEASE uses distributed consensus to guarantee the exclusiveness of a lease, and profits from Paxos' ability to achieve consensus in presence of host and network failures. In the evaluation, we demonstrated that our algorithm achieves both high throughput and high concurrency even in wide area setups.

The scalability of FATLEASE makes it useful in scenarios

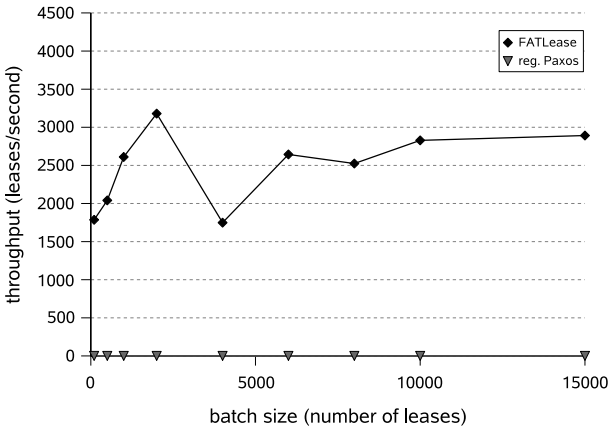


Figure 10: Influence of IOzone on throughput

where a large number of leases needs to be negotiated concurrently. In fact, FATLEASE has been implemented as part of the replication component of the distributed file system XtreemFS. There, FATLEASE is used to reach consensus on a primary server that is responsible for a particular replicated file.

FATLEASE improves Paxos-based lease negotiation by not having to resort to any persistent state. This allows us to scale beyond the limits that are set by the process-local stable storage. Especially applications that need the local storage resources for other tasks will benefit from this property, although the advent of flash-based storage might alleviate this difference. A further advantageous property of FATLEASE is the small amount of volatile state it maintains. Because we can exploit the timing constraints in deciding which Multipaxos instances to keep, we only need to track a maximum of two Multipaxos instances per lease.

Because FATLEASE is exploiting timing constraints, it has to make assumptions about the synchrony of clocks of the participating processes. We have factored in clock drift in the design of the algorithm so that it can tolerate a clock skew of d_{max}. However, this also means that faulting processes have to wait for $d_{max}+v$ until they are allowed to participate again in lease negotiation. We assume that if process clocks are synchronized by a time protocol, the bound of clock skew between processes will be of a size that does not dominate the startup delay of recovering processes. This also implies that a failed process will always lose its leases.

Acknowledgments

This work was supported by the EU IST program as part of the XtreemOS project (contract FP6-033576), and by the Spanish Ministry of Science and Technology under the TIN2004-07739-C02-01 grant. XtreemFS is a collaborative effort of the XtreemOS data management work package, and we thank all our partners for their valuable contributions. We also thank Monika Moser for her valuable comments and support.

7. REFERENCES

[1] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing paxos. *SIGACT News*, 34(1):47–67, 2003.

[2] M. Burrows. Chubby distributed lock service. In *Proceedings of the 7th Symposium on Operating System Design and Implementation, OSDI'06*, Seattle, WA, November 2006.

[3] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM Press.

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

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[6] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 202–210, New York, NY, USA, 1989. ACM Press.

[7] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. XtreemFS: a case for object-based storage in Grid data management. In *3rd VLDB Workshop on Data Management in Grids, co-located with VLDB 2007*, 2007.

[8] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, 2003.

[9] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

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

[11] B. W. Lampson. How to build a highly available system using consensus. In *WDAG '96: Proceedings of the 10th International Workshop on Distributed Algorithms*, pages 1–17, London, UK, 1996. Springer-Verlag.

[12] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, pages 105–120, 2004.

[13] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. *Theor. Comput. Sci.*, 243(1-2):35–91, 2000.

[14] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. *SIGOPS Oper. Syst. Rev.*, 31(5):224–237, 1997.

[15] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, pages 91–104, 2004.

[16] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.