Leader Election for Replicated Services Using Application Scores

Diogo Becker, Flavio Junqueira, Marco Serafini

Leader Election for Replicated Services Using Application Scores

Diogo Becker[1], Flavio Junqueira[2], and Marco Serafini[2]

[1] Technische Universität Dresden
Computer Science Department
Dresden, Germany
diogo@se.inf.tu-dresden.de
[2] Yahoo! Research, Barcelona, Spain
{fpj,serafini}@yahoo-inc.com

Abstract. Replicated services often rely on a leader to order client requests and broadcast state updates. In this work, we present POLE, a leader election algorithm that select leaders using application-specific scores. This flexibility given to the application enables the algorithm to tailor leader election according to metrics that are relevant in practical settings and that have been overlooked by existing approaches. Recovery time and request latency are examples of such metrics. To evaluate POLE, we use ZooKeeper, an open-source replicated service used for coordinating Web-scale applications. Our evaluation over realistic wide-area settings shows that application scores can have a significant impact on performance, and that just optimizing the latency of consensus does not translate into lower latency for clients. An important conclusion from our results is that obtaining a general strategy that satisfies a wide range of requirements is difficult, which implies that configurability is indispensable for practical leader election.

Keywords: leader election, replicated services, fault tolerance, performance

1 Introduction

Leader election is a fundamental primitive often used in practical systems, such as ZooKeeper [12] and Chubby [4], which are stateful middleware services used for coordination tasks of Web-scale applications. Given the size and extent of such applications, it is critical to prevent faults from bringing them to a halt, so both services use replication for masking faults and rely upon a primary server, *i.e.*, a leader, to propose state updates and to disseminate them using an atomic broadcast protocol. Consequently, the ability to elect a leader for the replicated service is critical to ensure progress.

In replicated services like ZooKeeper and Chubby, the leader performs more work than other servers, since it processes more messages and generates state updates. Many existing leader election algorithms rely on the identifiers of servers

to select a leader [1,9,11]. While designing ZooKeeper, however, one initial requirement regarding leader election was the ability to elect the server with the longest history of state updates among a quorum of servers. Such a server only has to push missing state updates to follower servers instead of pulling missing updates first. Because servers present comparable performance in data center deployments, such a leader not only enables faster recovery, but also provides the same performance while broadcasting as any other server would provide.

Over time, however, we encountered settings in which ZooKeeper servers either were running on heterogeneous hardware or presented different connectivity to other servers and application clients. Deployments spanning multiple data centers are important examples of such settings, where applications often present disaster-tolerance requirements. In some deployments, most clients are in one data center and electing a leader in the data center where most client requests arrive minimizes the request latency observed by clients. For some applications, such an optimization can be even more important than optimizing recovery time, as previously done by ZooKeeper, or the time required to terminate consensus, as with existing leader election algorithms such as the latency-aware algorithm of Santos *et al.* [17].

Although a number of leader election algorithms exist in the literature, there is no algorithm to our knowledge that can be easily adapted to specific constraints. This observation led us to reason about how to elect servers with properties other than the history length, and to use a generic *score* computed at runtime to classify servers according to *application-specific* properties.

In this work we propose an algorithm that takes a generic score as input to order servers during election, and we call it POLE (Performance-Oriented Leader Election). POLE is *configurable*, since it provides to an application the ability of selecting the desired properties of a leader server. To enable configurability, we implement application-specific scoring functions in an oracle module external to POLE. Before starting leader election, the local POLE module of a server queries its oracle to assign itself a score. Oracles compute scores using information either collected during regular operation or explicitly measured for estimating performance. Designing application-specific score oracles is simpler than implementing a new leader election that suits the application needs. Simplicity also comes from having each server assigning a score only to itself. After querying the oracle, servers share their scores, encapsulated in election messages, and try to elect the server with the highest score. POLE simply broadcasts scores and does not require reaching agreement on the ordering of servers before starting leader election, as required in the work of Sampaio and Brasileiro [16].

We show the flexibility and simplicity of our approach by implementing score oracles that optimize important metrics arising from real-world applications and were not considered by existing leader election algorithms: *mean request latency*, the mean time required for clients to complete a request; *worst-case request latency*, a metric often used to specific application requirements to the ZooKeeper service; and *recovery time*, the time it takes for a new leader to start operating again after the failure of the previous leader.

For evaluation, we implemented a prototype using the ZooKeeper code base[1] and emulated wide-area systems to investigate the performance of different application requirements. The results show that our request latency metrics can in some cases significantly diverge from consensus latency, which is the metric optimized by leader election algorithms such as the one of Santos *et al.* [17]. Depending on the particular setting, selecting an appropriate scoring function can elect leaders that provide up to 50% lower request latencies than arbitrarily elected leaders, while recovery-oriented oracles can elect leaders that provide minimal recovery times. Our results show however that obtaining a *general* strategy that fulfills multiple requirements is difficult.

The following list summarizes our contributions in this paper:

- We present POLE, the first leader election algorithm that is explicitly designed to enable application-specific performance configurability;
- We propose novel leader election oracles optimizing request latency and recovery time, metrics that cannot be directly optimized using the leader election algorithms proposed in the literature;
- We evaluate these oracles under a set of emulated wide-area settings and discuss trade-offs that operators deploying POLE might encounter.

Roadmap. The remainder of this work is structured as follows. Section 2 presents common application metrics and propose scores that approximate these metrics. Section 3 presents the algorithm and oracles using our scores. POLE and the oracles are evaluated in Section 4. We discuss further extensions in Section 5 and related work in Section 6. Finally, we conclude in Section 7.

2 Application Scores

In this section, we present several application-specific scoring functions. A scoring function $\theta(p)$ is a function that maps identifiers of servers (*id*s) to values called *scores*. We use the term θ score instead of scoring function $\theta(p)$ whenever the *id* of server p is clear from the context. Our scoring functions are based on two important metrics motivated by requirements that real-world applications impose on coordination services: recovery time and latency of (write) requests as perceived by clients. In particular, we focus on wide-area network (WAN) deployments, spanning multiple data centers. Such deployments are typical for applications with disaster-tolerance requirements. POLE, however, supports other scoring functions not explored in this work such as pre-defined preference lists. Table 1 summarizes our scoring functions.

2.1 Background: Replicated Coordination Services

A coordination service, such as Chubby and ZooKeeper, enables clients to interact through a shared data tree of simple small files; for instance, ZooKeeper

[1] `http://zookeeper.apache.org`

Table 1: Scores for server p

Score	Symbol	Description
maximum *zxid*	θ_z	length of the local history stored by p
consensus latency	θ_c	consensus latency if p becomes leader
request rate	θ_r	rate of client requests sent to p
mean request latency	θ_l	mean request latency if p becomes leader
worst-case request latency	θ_w	worst-case request latency if p becomes leader

by default does not allow files (znodes) larger than 1 MB. This data structure is replicated across servers of an ensemble to ensure availability and durability. To access the service, clients initiate sessions and manipulate these files through a file-system-like API. Although simple, this API is powerful enough to create complex synchronization mechanisms.

Coordination services are designed to be a consistent, reliable component of larger distributed applications. We consequently assume real deployments are properly provisioned (the service rarely saturates). Different from Chubby, ZooKeeper clients can connect to any server and this server processes locally read requests, thus avoiding the cost of running atomic broadcast. Deployments of coordination services consist typically of ensembles of 3 to 7 servers, depending on the application requirements. In deployments of these services, process and link failures are typically infrequent.

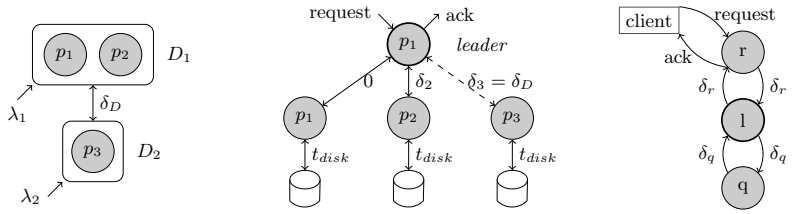
2.2 Request Latency

In wide-area settings, links often have different latencies, and the request latency perceived by the client application can be adversely affected by slow links. Consider the example with an ensemble of 3 servers deployed in two data centers D_1 and D_2 as depicted in Fig. 1a. Let δ_D be the latency of the links between any server in D_1 and any server in D_2. In a wide-area setting, data centers are geographically separated, so the link latency inside a data center, *e.g.*, between p_1 and p_2, is typically much lower than between data centers, *e.g.*, δ_D.

In leader-based atomic broadcasts protocols, the leader orders client requests, and in systems like ZooKeeper and Chubby, they also process them generating state updates that the leader broadcasts to followers (a leader server is also a follower). Following ZooKeeper terminology, each of these state updates is a *transaction*. When a server is elected leader, *e.g.*, server p_1, the links can be represented as in Fig. 1b, where δ_i denotes the latency of the link between server p_1 and server p_i. The leader can only acknowledge a request to the client after a *quorum* of servers accepted the respective transaction, by logging it to stable storage and replying. In this work, we consider a quorum to be a majority of servers in the set N of all servers of the system, ensuring that every two quorums have a common server.

The elapsed time between a broadcast by p_1 and the receipt of a quorum of replies is the *consensus latency* of p_1.[2] Let server l be a leader (or candidate)

[2] For our purposes, atomic broadcast and consensus are equivalent terms.



(a) 3 servers in 2 data centers, high link latency δ_D

(b) Links between leader and followers with latencies

(c) Total request latency

Fig. 1: Two data centers, leader-follower links, and latency of a request

and q be the server with *slowest* link to l in the quorum of servers with the *fastest* links to l, *i.e.*, q is the server with the $(\lfloor |N|/2 \rfloor + 1)^{\text{th}}$ slowest link to l. In the example, $l = p_1$, $q = p_2$, and $\delta_q = \delta_2$. The score θ_c of a server l is its consensus latency assuming it is the leader, *i.e.*, $\theta_c(l) = 2 \times \delta_q + t_{disk}$, where t_{disk} is the time it takes a server to store the transaction in stable storage. Modern disks have a write cache that improves significantly the performance of writes to disk, and enables short latency values. With such a buffer on, which is a typical choice in production, and persisting updates to disk, ZooKeeper is able to process operation in a few milliseconds including local area network latency. For our purposes, we can consequently ignore the disk latency. Additionally, we assume in the following discussion that latency of a link is roughly the same in both directions and relatively constant despite of spikes.

Electing the server with the fastest links to a quorum minimizes consensus latency, but not necessarily request latency. With ZooKeeper, as with other replicated services, the leader does not always directly receive all client requests. A follower r also receives them on behalf of the leader and forwards them to the leader l, keeping a session to the client (see Fig. 1c). Once the leader performed consensus on the transaction, the leader sends an acknowledgment to the follower that forwarded the request, which in turn sends an acknowledgment to the client. The *request latency* is thus the sum of the hop from follower r to l and back plus the consensus latency of leader l. In the example, if only the consensus latency is considered, then servers p_1 or p_2 are the best leaders because they have a fast link to a quorum, *i.e.*, to themselves and to another server in the same data center. However, the request latency actually observed by the client also needs to include the additional communication step from the replica to the leader, so the "slow" server p_3 could be the best leader if most of the requests arrive on it. Note that we assume clients first connect to replicas in their data centers, so the latency client-replica is negligible compared to WAN latencies.

We now define scoring functions (scores) that target short request latency. Let λ_T be the total request rate of the system. The request rate arriving in data center D_i or on server p_i is represented by λ_i. If not clear from the context, we explicitly indicate to which one λ_i refers. The scoring function $\theta_r(p)$ is the *mean request rate* received by p from clients, *i.e.*, an approximation of λ_i.

The scoring function $\theta_c(p)$ represents the consensus latency provided by p if it were the leader. The scoring function $\theta_l(p)$ represents the *mean request latency* provided by server p if it were the leader, *i.e.*, the consensus latency score θ_c plus the round-trip time between p and each follower r weighted by the follower's θ_r:

$$\theta_l(p) = \theta_c(p) + \frac{1}{\lambda_T} \sum_{r \in N'} \theta_r(r) \times \delta_{r,p} \tag{1}$$

The scoring function $\theta_w(p)$ represents the *worst-case request latency* provided by server p if it were the leader, *i.e.*, the consensus latency score θ_c plus the round-trip time to the follower r with the greatest $\delta_{r,p}$:

$$\theta_l(p) = \theta_c(p) + 2 \times max(\{\delta_{r,p} | r \in N'\}) \tag{2}$$

In both equations $N' \subseteq N$, N' contains no server which is suspected to be crashed; and, if the server follows a leader l, then $l \notin N'$. The latter restriction is because the score should represent the performance of the candidate in the case the leader crashes. Section 3 explains how these scores are used in the election.

2.3 Recovery Time

The second practical metric we use to evaluate a leader is the recovery time, since it directly influences the down time of a service. For the purposes of this work, we define *recovery time* of a replicated service to be the time the new leader takes to learn from a quorum of followers the chosen transactions it has not seen, if any, and propagating the transactions to servers that are missing them. After a quorum is synchronized by persisting transactions to stable storage and applying to the state, the leader can start processing new transactions.

In ZooKeeper, a transaction encompasses an idempotent state update generated by executing the client's request and an increasing transaction identifier called *zxid* [12]. The total order enforced by ZooKeeper is consistent with the *zxid* order. As in any practical replicated service implementation [4,12], the leader performs multiple consensus instances in parallel. Because servers do not proceed in lock-step, and the leader only requires a quorum of servers to guarantee progress, at any point in time different followers may have accepted a different number of transactions due to a number of factors. If a server is powered on after crashing or after being powered off, then its state can be arbitrarily behind. Resource contention and traffic spikes might also cause a particular server to lag behind. As a consequence, the amount of transactions a new leader has to learn to synchronize with other servers on recovery can be arbitrarily large. The recovery time directly depends on this amount of data and on how fast the links between leader and followers are. During the time the service is recovering, no new request is accepted, and consequently, minimizing this amount of time is critical for availability.

Using the last accepted *zxid* directly in the election is a sensible approach. In fact, this is the scoring function ZooKeeper election currently implements; the last accepted *zxid* of a server is its θ_z score. By electing the server with the

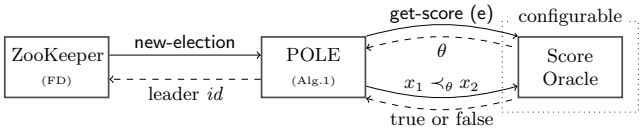


Fig. 2: Module interactions

highest θ_z, the leader does not have to copy missing transactions from other servers, thus making the time to recover negligible in local-area deployments, since it is essentially the time the leader takes to learn from a quorum of servers their last accepted *zxid*, but no transaction content. The server with the highest θ_z is called the *fattest* server.

3 The POLE Algorithm

POLE achieves configurability by allowing the operator to select on deployment a score oracle to order the processes [3] in an election round. A score oracle consists of a function get-score, which implements a scoring function $\theta(p)$ that assigns a score to the process; and a total order relation $\prec_\theta$ over the set of scores (a partial order relation is also sound with POLE, but the relations we use in this work are all total orders). For example, one possibility is to use the $\theta_z(p)$ scoring function, *i.e.*, to use *zxid*s as scores, and the total order over *zxid*s as the ordering relation. Figure 2 shows the interaction between the modules of our system.

Our POLE implementation in ZooKeeper breaks the leader election into two tasks: (unreliable) leader selection and (unreliable) failure detection. POLE, together with a score oracle, implements the first task, whereas ZooKeeper implements the second task. Both together implement an Ω leader election oracle [7]. When a ZooKeeper process detects that the current leader crashed, it invokes POLE. Next, POLE starts an election by invoking the oracle, which in turn assigns a θ score to the local process. POLE then shares the θ score with the other processes. In timely runs, where messages are not lost the process with the highest score, ordered with the $\prec_\theta$, becomes the new leader. POLE terminates by returning to ZooKeeper the *id* of the selected process.

3.1 Failure Detection

Failure detection is implemented in ZooKeeper because it is a byproduct of its operation. When a process is elected leader, it opens TCP connections to other processes. As long as a leader has open connections with a majority of processes, it remains able to commit the operations it proposes, and progress is guaranteed. This condition holds as long as the leader has timely links to a quorum, that is, it is f-connected [15]. If the leader process fails to open enough connections, or if too many connections are terminated due to faults or asynchrony, the process

[3] In this section, we use the term process instead of server to agree with the literature.

abandons its leadership by closing all its connections and executes POLE to perform a new election. Similarly, when a process notices that it has disconnected from the leader, it executes POLE. Our election algorithm only requires leader-follower failure detection information, so it never requires ZooKeeper to open more connections than those needed by the atomic broadcast protocol itself. Processes only accept a connection from the process they believe to be the leader. Hence, at any given time *at most* one leader has open connections with a *majority* of processes. Consequently, if two processes are leaders simultaneously, eventually one of them drops leadership.

3.2 Leader Selection Algorithm

We now describe in detail the leader selection algorithm. The pseudo-code is shown in Alg. 1. We assume messages can be delayed or lost, but not corrupted; processes can start at different points in time, can crash, and can recover.

The core idea of the algorithm is very simple. Once new-election is called by ZooKeeper, process i starts the election by requesting its current θ score from its oracle with get-score. It then invokes start-epoch and broadcasts a propose message $\langle P, epoch, \theta, i \rangle$. Epochs are used to identify election instances. Whenever process i receives a greater proposal tuple from some other process j, it overwrites its proposal and repeats the broadcast. Proposals are totally ordered with $\prec_{\mathrm{p}}$, since it encapsulates multiple ordering relations: first ordering by epoch, then by θ using $\prec_{\theta}$, and finally by id in case breaking ties is necessary. In a timely run, non-crashed processes eventually converge to the same leader. Note that get-score receives the current epoch as an argument, since some scoring functions depend on the epoch, such as the rotating scheme described below.

As in any practical implementation, the algorithm has to return a leader at some point. If process i received proposals from *all* processes, then process i can elect the highest proposal immediately. Due to delayed or lost messages and process crashes, process i might receive proposals only from a subset of processes. In general, leader-based broadcast protocols require at least a quorum to achieve progress. Therefore, if process i receives proposals from a quorum, it starts election-timer (Line 32), which terminates the election upon timeout. Whenever a proposal is delivered, the procedure start-timer? is called to check these conditions. Upon timeout (Line 50), leader is set to the id of the process with the highest proposal, and new-election returns (Line 26) to ZooKeeper. The returned leader might, however, be a crashed process if it crashes after broadcasting its proposal. In such cases, the failure detection implemented in ZooKeeper will eventually restart the election by calling new-election.

Because processes can be started at different points in time, and can be temporarily disconnected from each other, a process j can try to start an election when other processes already follow a leader. When process i believes to know who the leader is, it replies proposals from process j with a vote (V) message containing the value of leader (Line 35). If process j receives a quorum of vote messages (Line 54) indicating the same leader in the same epoch, then process j starts also believing in its leadership and sets its leader and epoch variables, thus

Algorithm 1: Selecting a leader in process i

```
1  import get-score, ≺_θ;
2  export new-election;
3  constant T;
4  timer election-timer, retry-timer;
5  init
6  |  leader ← ⊥;
7  |  epoch ← 0;
8  |  proposal-set ← vote-set ← ∅;
9  |  stop election-timer;
10 procedure start-epoch
11 |  leader ← ⊥;
12 |  proposal-set ← vote-set ← ∅;
13 |  send ⟨P, epoch, proposal⟩ to all;
14 function new-election
15 |  t ← T;
16 |  epoch ← epoch+1;
17 |  proposal ← (get-score(epoch), i);
18 |  invoke start-epoch;
19 |  while leader = ⊥ do
20 |  |  start retry-timer;
21 |  |  while leader = ⊥
22 |  |  |  ∧ retry-timer < t do nop;
23 |  |  if leader = ⊥ then
24 |  |  |  send ⟨P, epoch, proposal⟩ to all;
25 |  |  t ← t × 2;
26 |  return leader;
27 procedure start-timer? argument j
28 |  proposal-set ← proposal-set ∪ {j};
29 |  if |proposal-set| = |N| then
30 |  |  start election-timer with T + 1;
31 |  else if |proposal-set| > ⌊|N|/2⌋ then
32 |  |  start election-timer;
33 upon receive ⟨P, e, p⟩ from j do
34 |  if leader ≠ ⊥ then
35 |  |  send ⟨V, epoch, leader⟩ to j;
36 |  |  return;
37 |  if p ≺_p proposal then
38 |  |  send ⟨P, epoch, proposal⟩ to j;
39 |  else if e = epoch then
40 |  |  if proposal ≺_p p then
41 |  |  |  proposal ← p;
42 |  |  |  send ⟨P, epoch, proposal⟩ to all;
43 |  else
44 |  |  epoch ← e;
45 |  |  proposal ← (get-score(epoch), i);
46 |  |  if proposal ≺_p p then
47 |  |  |  proposal ← p;
48 |  |  invoke start-epoch;
49 |  invoke start-timer? with j;
50 upon election-timer > T do
51 |  (θ, l) ← proposal;
52 |  leader ← l;
53 |  stop election-timer;
54 upon receive ⟨V, e, l⟩ from j do
55 |  vote-set ← vote-set ∪ (e, l, j);
56 |  if ∃S ⊆ vote-set :
57 |  |  ∀ (e1, l1, j1), (e2, l2, j2) ∈ S :
58 |  |  e1 = e2 ∧ l1 = l2
59 |  |  ∧ |S| > ⌊|N|/2⌋
60 |  |  ∧ (∃ (e4, l4, j4) ∈ S : j4 = l1) then
61 |  |  epoch ← e1;
62 |  |  leader ← l1;
63 |  |  stop election-timer;
64 |  else
65 |  |  invoke start-timer? with j;
```

guaranteeing leader stability [1]. Line 60 checks if the leader replied with a vote message as well, what is important to avoid reelecting a crashed leader.

Whenever the election does not converge, *e.g.*, too many messages were lost and no quorum sent proposals to process i, a second timeout (retry-timer) is triggered and process i repeats its broadcasts (Line 24), exponentially backing off on each retry. Note that the timeliness implied by waiting for a quorum is not the minimal to enable atomic broadcasts [2,15]. Nevertheless, we have not yet observed the need for weaker timeliness assumptions.

3.3 Oracles

An oracle is the implementation of a scoring function and an ordering relation over set of scores. We implemented one oracle for each scoring function in Sec. 2. With history oracle, POLE elects the process with the highest θ_z. With request oracle, it elects the process that receives the largest volume of requests. With consensus oracle, it elects the process with the shortest consensus latency. With latency oracle, the process with the lowest mean request latency. And with worst-

case oracle, it elects the process with the lowest worst-case request latency. For comparison purposes, we also implemented an application-*unaware* oracle which represents a rotating leader selection based on the process identifier, a typical approach of existing protocols. The leader of the current epoch (also called round's leader) assigns itself a score of 1, while all other processes assign themselves a score of 0. Note that differently from traditional rotating elections, our oracle uses the same message pattern as any POLE election (broadcasts).

Different scores require different information to be acquired and transmitted between the processes. The θ_z and θ_r scores are already built into ZooKeeper and can be easily accessed by an oracle by querying the right components. To compute them, no information has to be transmitted between processes. In contrast, θ_l and θ_w require communication. The election module in ZooKeeper builds a clique graph encompassing all processes. These connections are encapsulated in a component called ConnectionManager, which implements links between the processes exclusively for the election. This component can be used by any oracle to transmit oracle messages if needed.

To compute θ_c, θ_l, and θ_w, each process i keeps a vector RTT_i with the mean round-trip times between i and all other processes. It sends "ping-pong" heartbeats to all nodes periodically via the ConnectionManager. Once a process crashes, it does not respond to heartbeats and is removed from RTT_i. The heartbeat period can be configured on a per deployment basis. Fairly long periods such as 100ms to 1s are sufficient in most cases (we use the default value of 1s). To calculate θ_c or θ_w, let $\overline{RTT}_i$ be RTT_i sorted in ascending order of values. The score $\theta_c(i)$ is the $(\lfloor |N|/2 \rfloor + 1)^{\text{th}}$ value of $\overline{RTT}_i$, and $\theta_w(i)$ is $\theta_c(i)$ plus the highest value in $\overline{RTT}_i$ (see Eq. 2). Note that we exclude the current leader from RTT_i because the scores are needed once the leader becomes faulty.

Finally, to compute θ_l, each process i needs to additionally keep a vector F_i with the value of $\theta_r(j)$ for all $j \in \Pi$. The sum of all elements in F_i approximates λ_T. The values of the scoring function $\theta_r(j)$ are transmitted in the heartbeat messages via the ConnectionManager. The final value is calculated by weighting RTT_i with the frequencies in F_i as in Eq. 1.

4 Experimental Evaluation

In this section, we evaluate the oracles described in Sec. 3.3. For readability, we often say an oracle provides low request latency instead of saying the server elected with this oracle provides low request latency.

4.1 Experimental Setup and Methodology

We performed all of our experiments on a cluster of 10 workstations. Each of them has 2 quad-core 2.0 GHz Xeon processors, 8 GB of RAM, a Gigabit Ethernet interface, and a 7200 RPM disk attached via SATA2. The running operating system is Debian GNU/Linux 5.0 with kernel version 2.6.26.

Table 2: End-to-end latency examples

source	target	mean rtt (ms)	sd (ms)
slac	caltech	9.88	0.10
tud	cern	20.75	0.11
slac	fnal	53.26	0.14
caltech	fnal	77.06	0.26
slac	cern	172.47	0.69

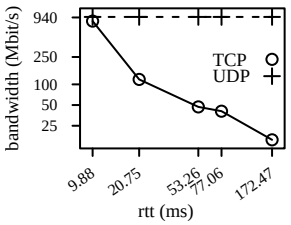


Fig. 3: Achievable bandwidth

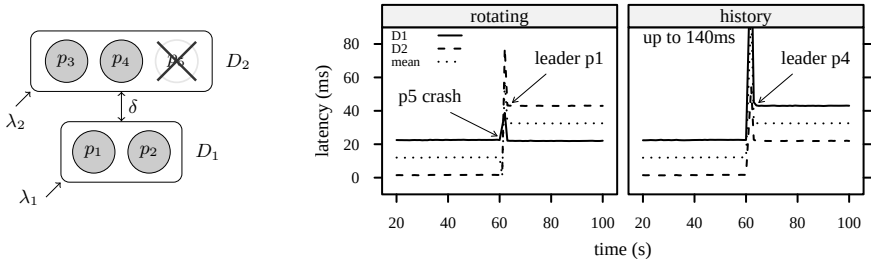
Implementation and Workload Generator. POLE and the oracles were implemented using ZooKeeper version 3.3.0. Our clients were implemented in Python using the asynchronous ZooKeeper API. They only perform *write* requests because, as explained previously, only these requests exercise the underlying consensus protocol. We use request sizes of 1 kB as in the work of Hunt [12].

We always start either a client or a server (*i.e.*, ZooKeeper server) per machine. We assume clients connect to the server with which they have the lowest round-trip time. Thus, we ignore the hop between client and server as in Fig. 1c.

Wide-area Setting Emulation. ZooKeeper ensembles are typically deployed in controlled environments. Consequently, instead of using testbeds such as PlanetLab,[4] we emulate a set of specific wide-area deployments using Netem, a network emulator available in the standard Linux kernel.[5] The deployments we define below use real round-trip times shown in Table 2, where the names refer to the following end-points: California Institute of Technology (`caltech.edu`), European Organization for Nuclear Research (`cern.ch`), Fermilab (`fnal.gov`), Stanford Linear Accelerator Center (`slac.stanford.edu`), and Technische Universität Dresden (tud, `tu-dresden.de`). All the round-trip values are aggregations of one month of data (November 2010) taken from the PingER project [6] except the values from `tud` to `cern`, which were performed by the authors (900 ping samples on the 2nd March 2011).

We use a Pareto distribution with mean given by the mean round-trip times in the table, and jitter of 2% of the mean. We use 2% jitter instead of the standard deviation to simplify the experiment setup. Figure 3 shows the achievable bandwidth using TCP and UDP for the given round-trip times measured in our cluster with Iperf.[7] Not shown in the figure is the bandwidth with round-trip time set to 0, which is about 940 Mbits/s as well. Note that, when the round-trip time is set to 0, the actual round-trip time is about 100 μs. In our experiments, we use round-trip times up to 77.06 ms, as higher round-trip times restrict the bandwidth excessively.

[4] `http://www.planet-lab.org`
[5] `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`
[6] `http://www-iepm.slac.stanford.edu/pinger`
[7] `http://iperf.sourceforge.net`



(a) 2 data centers topology (b) Request latency per data center

Fig. 4: Throughput and request latency with leader crash in two data centers

The communication between ZooKeeper servers uses TCP; consequently, we use only delay and jitter as emulation parameters because in TCP packet loss and reordering are perceived as jitter assuming the connections do not drop. The bandwidth of links can be limited by the emulator as well. Independent of emulator limits, note that TCP inherently limits the achievable bandwidth (see Fig. 3). To simplify our setup, we rely only on the protocol-limited bandwidth. Although TCP can use more bandwidth with non-standard optimizations such as Jumbo frames, we focus on typical installations of both hardware and software.

Experiments. In the following, each experiment represents an emulated WAN deployment in two or three data centers. Two data centers (Fig. 4a) enables clients close to a remote data center to perform *read* requests with lower latency. Three data centers (Fig. 5) allows one data center to crash without affecting the availability of the service given that there is no data center with a quorum of servers. We assign λ and δ parameters to these topologies, where λ_i is the mean request rate (req/s) arriving at data center D_i (distributed uniformly at random across the servers in D_i); and δ is the link latency between any two servers in two different data centers. We use the mean round-trip times in Table 2 to set δ, where $2 \cdot \delta = rtt$. Whenever appropriate, instead of giving δ parameters, we let the data centers be the end-points from Table 2, *e.g.*, slac, caltech, tud.

The experiments are executed for 120 seconds and always have as initial leader p_5. The settings were selected such that the leader p_5 is always initially the best leader. After 60 seconds, p_5 is killed. Measurements showing lines with no dots are single runs over time. Dots and bars represent mean aggregations over 5 runs. Latency aggregations refer to the steady state starting 10 seconds after the new leader is elected, *i.e.*, the last 50 seconds of a run. Recovery time aggregations refer to the interval from the time the new leader recognizes itself as such, up to when it processes the first request.

4.2 Request Latency and Random Request Distributions

We initially evaluate our oracles considering request latency when the request distribution arriving on each data center is arbitrary or unpredictable. In par-

ticular, we evaluate oracles that are unaware of request distribution: history, rotating, worst-case, and consensus oracles. We divide our results in two data centers and three data center deployments.

Two Data Centers. We show that in deployments of two data centers, when the atomic broadcast has to cross the slow link (δ in Fig. 4a), any oracle provides the same mean request latency after the crash of the initial leader p_5. Let D_1 and D_2 be respectively tud and cern, *i.e.*, $2 \cdot \delta = 20.75$ ms. Because we evaluate oracles using latency, we use a low request rate (λ_T) of 1000 req/s (8 Mbits/s). A low request rate does not overload the system, and consequently correct servers are synchronized when the leader crashes. The recovery time in such cases is roughly the same independent of which server is elected as leader. Requests arrive randomly on both data centers: $\lambda_1 = \lambda_2 = \frac{\lambda_T}{2} = 500$ req/s.

Figure 4b shows the request latency on each data center before and after the crash of p_5 for two oracles: rotating and history. Because servers have the same θ_z (they are synchronized), history elects the server with highest *id*, *i.e.*, server p_4. The rotating oracle elects the server with the next *id* modulo the number of servers, *i.e.*, server p_1. Server p_4 is in the same data center as the crashed leader, while p_3 in another. Note that the form of the spike at 60 s depends rather on the run than on the oracle. With both oracles, the mean request latency over all requests is initially around 10 ms and after the crash around 30 ms. This is so because there is a complete quorum in the data center of the leader p_5 before the crash, but not after. The new leader – despite of its location – has to send every transaction over the slow link to form a quorum. Therefore, *any* oracle elects an equally fast leader, and all servers can be considered to be the fastest.

Three Data Centers. We show that in three data center deployments (1) history fails to elect the fastest leader, and (2) worst-case can elect faster leaders than consensus with respect to mean request latency. To that end, we assign to the data centers D_1, D_2, and D_3 in Fig. 5 the end-points slac, caltech, and fnal in different combinations (see Table 3). These end-points have the following round-trip times: 9.88 ms for slac-caltech, 53.26 ms for slac-fnal, and 77.06 ms for caltech-fnal. Let $\lambda_1 = \lambda_2 = \lambda_3 = \frac{\lambda_T}{3} = 333.33$ req/s.

Figure 6 shows the results for the deployments described in Table 3. Each group of bars corresponds to experiments with different leaders (indicated below the figure) which are reported together with the data center in which they are

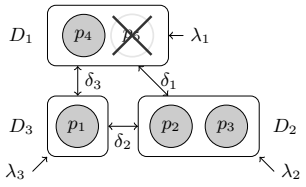


Fig. 5: Topology of 3 data centers

Table 3: End-to-end latency examples

Deployment	D_1	D_2	D_3
1	caltech	slac	fnal
2	slac	caltech	fnal
3	slac	fnal	caltech

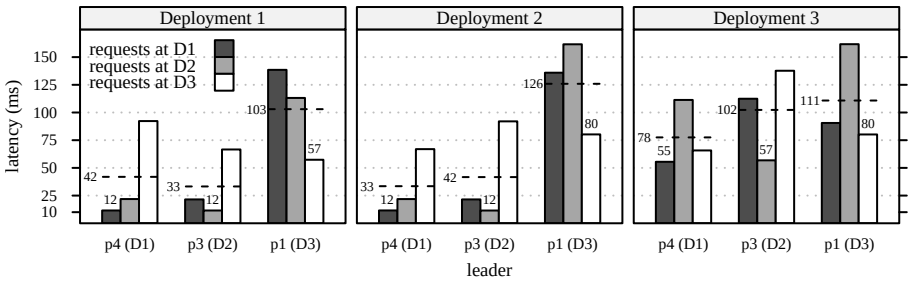


Fig. 6: Mean request latency per data center for different new leaders

located. Leaders provide different request latencies for requests arriving on the different data centers. For example, in Deployment 1, when server p_4 is elected, requests arriving on D_1, D_2, and D_3 are complete in roughly 12 ms, 22 ms, and 92 ms, respectively. The numbers over the bars represent the consensus latency of the leader (approximated by θ_c, see Sec. 2.2), while the dashed lines represent the mean request latency (approximated by θ_l), both rounded for readability. Leader p_4 in Deployment 1 has the consensus latency and mean request latency of about 12 ms and 42 ms, respectively.

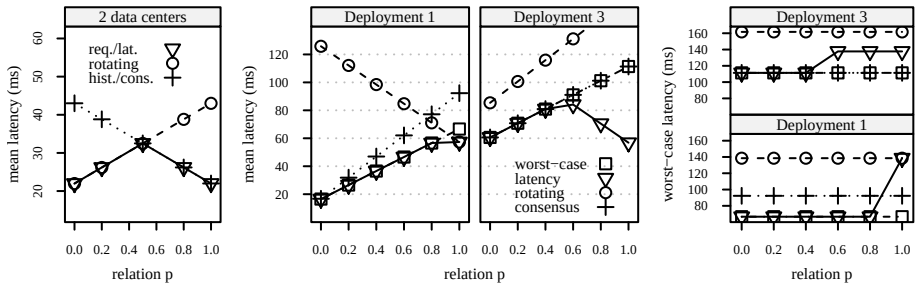
Due to how the server *id*s are assigned, history elects server p_4 in all deployments, while rotating elects p_1. By exchanging the location of p_1 and p_4, history and rotating behave in opposite ways. Therefore, no guarantees can be given on the performance of servers elected with them.

The worst-case oracle elects p_3 in Deployment 1 and p_4 in Deployment 2, minimizing both worst-case (highest bar of each leader) and mean request latency (dashed line). In contrast, consensus might elect either p_3 or p_4 depending on fluctuations of the consensus latency. In these deployments, worst-case might provide 20% lower mean request latency than consensus (33 versus 42 ms).

4.3 Request Latency and Uneven Request Distributions

We evaluate the request latency we obtain with our oracles when the request distribution arriving on each data center is uneven. In particular, we compare the performance of oracles that are aware of the request distribution (request and latency) with the remaining ones: history, rotating, worst-case, and consensus. We divide again our results in two data centers and three data center deployments.

Two Data Centers. We show that in two data centers request and latency oracles succeed to elect the *fastest* leader, while the remaining oracles do not. Figure 7a shows the same settings of the previous experiment in two data centers except that $\lambda_1 = (1 - p) \cdot \lambda_T$ and $\lambda_2 = p \cdot \lambda_T$. The y-axis represents the mean request latency after the crash with values p. Because after the crash all servers have the same θ_z and θ_c, history and consensus elect p_4. Similarly, worst-case (not depicted) calculates roughly the same θ_w for all servers (about 43 ms in Fig. 4b),



(a) Two data centers (b) Mean, 3 data centers (c) Worst-case, 3 dc

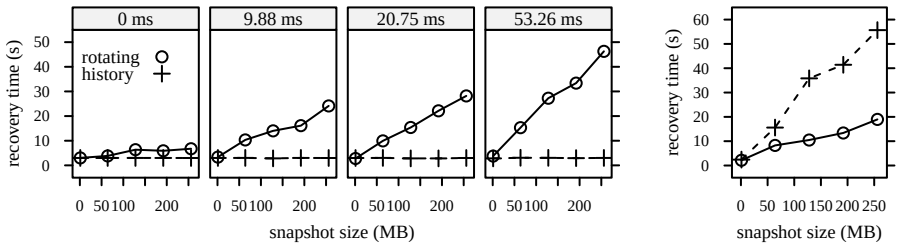
Fig. 7: Mean and worst-case request latency for different relations p

and, therefore, elects the server with highest *id*, *i.e.*, p_4. The rotating oracle elects server p_1. In contrast, request and latency oracles elect the fastest leader for different relations p. In this example, the latency with them is half of the latency provided with the other oracles when requests arrive only on one data center. To understand the reason, suppose $p = 0$. If p_1 is the leader, all requests arriving on D_1 have to travel only once over slow link. If p_4 is the leader, all requests are first sent to the leader, then atomic broadcast is performed, and finally a confirmation is sent back (see Fig. 1c). As long as we are only concerned with mean request latency, and requests are unevenly distributed among the data centers, request and latency elect the *fastest* leader in two data center deployments.

Three Data Centers. In three data center deployments, the request oracle does not elect the server with the shortest mean request latency. We give a simple counter example. Consider the experiments in Fig. 6. Assume all requests arrive at D_3, *i.e.*, $\lambda_3 = \lambda_T$, and $\lambda_1 = \lambda_2 = 0$. In Deployment 1, the fastest leader is server p_1, while in Deployment 2, server p_4. The request oracle elects however in both cases server p_1 because its θ_r is the highest.

We next discuss some insights on Deployments 1 and 3 from above except that the request distributions are now uneven. In Deployment 1, $\lambda_1 = \frac{1-p}{2}\lambda_T$, $\lambda_2 = \frac{1-p}{2}\lambda_T$, and $\lambda_3 = p \cdot \lambda_T$. And, in Deployment 3, $\lambda_1 = \frac{1-p}{2}\lambda_T$, $\lambda_2 = p \cdot \lambda_T$, and $\lambda_3 = \frac{1-p}{2}\lambda_T$. Figures 7b and 7c respectively show the mean request latency and the worst-case request latency. We can observe that:

1. Oracles that are unaware of request distribution elect arbitrarily slow leaders. Consider Deployment 1 with $p = 0$ (Fig. 7b). The rotating oracle provides mean request latencies more than 7 times higher than latency (125.8 ms versus 16.5 ms). Consider Deployment 3 with $p = 1$. The consensus oracle has mean request latency of 111.3 ms, while the latency 57.4 ms.
2. The consensus oracle also fails to elect the server with the shortest worst-case request latency (Fig. 7c). In Deployment 1, the consensus oracle provides worst-case request latency about 20 ms higher than the worst-case oracle.



(a) 2 data centers varying δ_D (b) 3 data centers

Fig. 8: Recovery time varying snapshot size using history and rotating oracles

3. worst-case provides 10 ms worse mean latency than latency (66.6 ms versus 57.4 ms) in Deployment 1, and more than 50 ms worse in Deployment 3.
4. Finally, latency does not provide minimal worst-case request latency when p increases in both deployments, being from 30 to 60 ms worse than worst-case.

4.4 Recovery Time

Electing a leader that provides the minimal *recovery time* in a LAN deployment is the default election oracle in ZooKeeper. The history oracle elects the *fattest* leader to avoid transferring state from followers to the leader (see Sec. 2.3). We start experimenting whether the history oracle provides the same property in WAN deployments and then we consider using the latency oracle for that end.

History Oracle. We show that the *fattest* leader provides the minimal recovery time with two data centers, but not necessarily with three. Figure 8a shows the recovery time for 4 deployments with 2 data centers (Fig. 4a) and different round-trip times ($2 \cdot \delta$). We force p_1 to be outdated when the leader crashes, so that, becoming leader or follower, server p_1 has to first receive a snapshot of the current state from another server. To outdate a server, we kill the server and restart it when the leader is about to crash. The size of snapshots is application dependent; we used from about 0 up to 250 MB. The history oracle elects server p_4, and the rotating oracle elects server p_1. A round-trip time of 0 ms is equivalent to a LAN deployment, where all servers are in the same data center. When the outdated server p_1 is elected, the recovery time directly depends on the size of the snapshot to be transferred. By partitioning the servers in two data centers, the recovery time when electing p_1 is "amplified" with the increasing round-trip time, while is minimal when electing p_4.

The history oracle does not provides the minimal recovery time in 3 data centers. Figure 8b shows such deployment of the 3 data centers (Fig. 5) with the following parameters: $2 \cdot \delta_1 = 77.06$ ms, $2 \cdot \delta_2 = 20.75$ ms, and $2 \cdot \delta_3 = 9.88$ ms. We force both servers in data center D_2 to be outdated with respect to the leader. After leader p_5 crashes, servers p_1 and p_4 have the same θ_z and need to bring at least another server in sync to start processing requests. Because the latency between D_1 and D_2 is shorter compared to the one between D_3 and D_2, server p_1 presents a shorter recovery compared to p_4.

Latency Oracle. It is easy to see that oracles such as latency cannot minimize the recovery time because it is guided by the request distribution. Consider the previous experiment with 3 data centers. Assume $\lambda_1 = \lambda_T$, and $\lambda_2 = \lambda_3 = 0$. The *fastest* is server p_4, while server p_1 provides minimal recovery time.

5 Extensions

We now briefly discuss some aspects not addressed in the paper: combination of scoring functions, how to perform the first election, and how to cope with workload changes and score miscalculations.

Co-optimizing Multiple Metrics. In our experiments, we considered oracles optimizing a single metric. Co-optimizing several oracles is not always trivial. For example, we have shown that worst-case cannot optimize the mean request latency, while latency cannot optimize the recovery time. However, a simple way to achieve co-optimization is the following. The application defines a priority order $\theta_1, \ldots, \theta_n$ among the scores of interest. Each server corresponds to a vector of scores in which we can easily identify an ordering function. To enable the comparison of vectors, it is possible to define equivalence ranges for some scores, such that all servers whose score is in the same class are considered equally good.

Consider the example where an application wants to elect a leader that optimizes the mean latency and, as a secondary goal, that minimizes recovery time. The application can combine $\theta_z(p)$ and a modified $\theta'_l(p)$, which truncates $\theta_l(p)$ in latency classes: 10 ms, 20 ms, etc. An oracle implementing such a combination of scoring functions can elect the server in the lowest latency class $\theta'_l(p)$ and with the highest $\theta_z(p)$, *i.e.*, the fattest among the fastest leaders.

Initial Election and Resignation. All experiments we performed already started with a leader, which later on crashed. For the first election, the get-score function simply blocks until enough information is collected to generate a score.

An application may also want to include *resignations*, which happen when the actual leader is not the actual best leader. There is a number of reasons for this to happen. For example, when a crashed server recovers and rejoins the ensemble, it might be a better leader than the actual leader is. Another reason can be that some score, as for example the request distribution, may change over time making the actual leader a worse leader than some follower.

POLE can also easily be extended to consider resignations as follows: The leader periodically sends messages to other servers querying their actual θ scores. Based on some threshold, the leader decides if some servers has a better score than itself, and resigns its leadership. In contrast to "after-crash" scores, as we have presented, "before-resignation" scores have to include the actual leader in RTT_i and F_i vectors because it is supposed to be alive during the next election.

6 Related Work

Consensus is the primitive underlying the atomic broadcast protocol [5]. To solve consensus, it is necessary and sufficient to elect a leader [5,7]. The problem of

leader election has been broadly investigated from many viewpoints, from minimizing the synchrony and reliability requirements imposed on links [2,15], to optimizing the Quality of Service of failure detection [8], to electing the leader that can minimize the latency of solving consensus [17]. The ZooKeeper atomic broadcast protocol (Zab) implements a variant of the atomic broadcast primitive called *primary order atomic broadcast* [13]. Similar to many consensus implementations, Zab uses a leader to suggest a total order of transactions.

Most leader election algorithms are, however, oblivious to the application needs and elect a leader based on its *id*, *e.g.*, the highest *id* in the alive-set of each process [9], or the next *id* when incrementing a counter [1,5]. In fact, leader election has been proposed as a *service* for multiple applications at the same time [9,18]. In this work, we have shown that the *service* approach is not sufficient when applications use different criteria to select a leader.

Selecting leaders based on performance has been previously proposed by Singh and Kurose [19,20]. They do not present the distributed algorithm to spread votes, but focus on different voting schemes taken from social sciences to elect the best leader when some votes might be wrong (*e.g.*, indicate a bad leader). Furthermore, they assume that processes that vote do not crash, and that messages cannot be lost.

Santos *et al.* [17] presented a rotating leader election that finds an optimal leader after a series of intermediate elections. In systems like ZooKeeper, this solution can be very costly because each reelection incurs additional recovery costs, even when servers have the same history, *e.g.*, snapshot reading, connection establishment, and the first phase of the consensus protocol. Furthermore, their approach focuses exclusively on *consensus latency*. As we have shown, it is often more important to minimize other metrics such as the *request latency*.

Sampaio and Brasileiro [16] propose a configurable solution for *id*-based elections by using a process-ordering oracle. This additional component changes the *a priori* order of processes at run-time based on application metrics (but the authors only evaluate *consensus latency*). Nevertheless, their solution is not of great practical interest because it requires the processes to use each consensus instance to agree on the *a priori* process ordering of the next consensus instance. Paxos [6,14] and ZooKeeper's atomic broadcast [13], however, promote the simultaneous execution of instances of consensus to achieve high performance in replicated services; Instead of electing leaders based on an previously agreed-upon order, POLE enables processes to locally assign scores to themselves and share their scores. Computing scores accurately is therefore critical for selecting an appropriate leader, and score deviations introduced, for example, due to sampling may lead to different processes being selected. Such deviations, however, do not preclude liveness.

The idea of broadcasting *ids* and electing the process with the highest one is the core of the Bully algorithm, which was proposed by Garcia-Molina [11] in the context of synchronous systems. Bully and its partially-synchronous counterpart, the Invitation algorithm [11], do not enable configurability and target generic master-worker applications. In contrast, POLE has been specifically designed

for replicated systems, and it provides mechanisms such as electing a process only once there is a quorum of processes running. Using scores as opposed to identifiers is also a key difference.

The separation of concerns between failure detection and consensus has been proposed previously by Chandra and Toueg [5]. While being conceptually fundamental, this strict separation of concerns does not necessarily result in more efficient implementations. Using application-level failure detection information, we can avoid sending failure detection messages when a stable leader exists [10]. Note that although some scoring function might require additional messages to calculate their θ scores, that is not the concern of the election algorithm itself.

Bakr and Keidar study the running time of a "communication round" of distributed algorithms using TCP over the Internet [3]. The chain replica-leader-quorum in this work relates to their *secondary leader* communication pattern.

7 Conclusion

In replicated services, leaders have a great impact on metrics of interest to client applications, such as recovery time and mean request latency. In this work, we have presented POLE, a general and flexible leader election algorithm for practical replicated systems. POLE uses an application-defined scoring function to assign scores to servers when selecting a leader. A variant of POLE that uses *zxid*s as scores has been deployed as part of ZooKeeper, and in this work we have proposed additional scoring functions that satisfy different application requirements. Our goal, however, was not to explore exhaustively the space of oracles, but instead to argue for the importance of providing such flexibility to applications deployed in heterogeneous settings, such as the ones encompassing multiple data centers.

Our experimental results illustrate trade-offs that designers face when designing oracles for POLE. When client requests are unevenly distributed, the consensus oracle described in the literature performs poorly compared to mean *and* worst-case request latencies. There are a few available options when deploying POLE. For minimizing the worst-case request latency, the worst-case oracle can be used. A simple request oracle can minimize the mean request latency for settings with two data centers. With three data centers, a more complex oracle such as the latency oracle is however necessary. Leaders elected with these oracles can be at least 50% faster compared to arbitrarily elected ones. In contrast, although history is able to minimize recovery time in two data centers, none of the presented oracles can minimize it in three data centers. The design of a generic recovery-time oracle is still an open problem; in general exploring further the space of oracles is subject of future work.

Acknowledgments. This work has been partially supported by the SRT-15 project (ICT-257843), funded by the European Commission. We also would like to thank Benjamin Reed and Christof Fetzer for useful discussions on this work.

References

1. Aguilera, M., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable Leader Election. In: Welch, J. (ed.) Distributed Computing, Lecture Notes in Computer Science, vol. 2180, pp. 108–122. Springer Berlin / Heidelberg (2001)
2. Aguilera, M., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing Omega in systems with weak reliability and synchrony assumptions. Distributed Computing 21, 285–314 (2008)
3. Bakr, O., Keidar, I.: Evaluating the running time of a communication round over the Internet. In: PODC '02: Proceedings of the 21st annual symposium on Principles of distributed computing. pp. 243–252. ACM, New York, NY, USA (2002)
4. Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th symposium on Operating systems design and implementation. pp. 335–350. OSDI '06, USENIX Association, Berkeley, CA, USA (2006)
5. Chandra, T., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM 43(2), 225–267 (March 1996)
6. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: Proceedings of the 26th annual ACM symposium on Principles of distributed computing. pp. 398–407. PODC '07, ACM, New York, NY, USA (2007)
7. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM 43(4), 685–722 (1996)
8. Chen, W., Toueg, S., Aguilera, M.: On the Quality of Service of Failure Detectors. IEEE Transactions on Computers 51, 561–580 (2002)
9. Fetzer, C., Cristian, F.: A Highly Available Local Leader Election Service. IEEE Trans. Softw. Eng. 25, 603–618 (September 1999)
10. Fetzer, C., Raynal, M., Tronel, F.: An adaptive failure detection protocol. In: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing. pp. 146–. PRDC '01, IEEE Computer Society, Washington, DC, USA (2001)
11. Garcia-Molina, H.: Elections in a Distributed Computing System. IEEE Trans. Comput. 31(1), 48–59 (1982)
12. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: ZooKeeper: Wait-free coordination for Internet-scale systems. In: Proceedings of the 2010 USENIX annual technical conference. USENIXATC'10, USENIX Association, Berkeley, CA, USA (2010)
13. Junqueira, F., Reed, B., Serafini, M.: Zab: High-performance broadcast for primary-backup systems. In: Proc. of the IEEE Int'l Conf. on Dependable Systems and Networks (DSN-DCCS) (2011)
14. Lamport, L.: Paxos made simple. ACM SIGACT News 32(4), 18–25 (2001)
15. Malkhi, D., Oprea, F., Zhou, L.: Ω Meets Paxos: Leader Election and Stability without Eventual Timely Links. In: Fraigniaud, P. (ed.) Distributed Computing, LNCS, vol. 3724, pp. 199–213. Springer Berlin / Heidelberg (2005)
16. Sampaio, L., Brasileiro, F.: Adaptive Indulgent Consensus. Dependable Systems and Networks, International Conference on 0, 422–431 (2005)
17. Santos, N., Hutle, M., Schiper, A.: Latency-aware leader election. In: Proceedings of the 2009 ACM symposium on Applied Computing. pp. 1056–1061. SAC '09, ACM, New York, NY, USA (2009)
18. Schiper, N., Toueg, S.: A Robust and Lightweight Stable Leader Election Service for Dynamic Systems. Tech. Rep. 2008/01, University of Lugano (Mar 2008)
19. Singh, S., Kurose, J.: Electing leaders based upon performance: the delay model. In: 11th Int'l Conference on Distributed Computing Systems. pp. 464 –471 (1991)
20. Singh, S., Kurose, J.: Electing "Good" Leaders. Journal of Parallel and Distributed Computing 21(2), 184 – 201 (1994)