Hybrid Distributed Consensus*

Roy Friedman[1], Gabriel Kliot[2], and Alex Kogan[3,**]

[1] Department of Computer Science, Technion, Haifa, Israel
[2] Microsoft Research, Redmond, WA
[3] Oracle Labs, Burlington, MA

Abstract. Inspired by the proliferation of cloud-based services, this paper studies consensus, one of the most fundamental distributed computing problems, in a hybrid model of computation. In this model, processes (or nodes) exchange information by passing messages or by accessing a reliable and highly-available register hosted in the cloud. The paper presents a formal definition of the model and problem, and studies performance tradeoffs related to using such a register. Specifically, it proves a lower bound on the number of register accesses in deterministic protocols, and gives a simple deterministic protocol that meets this bound when the register is *compare-and-swap* (CAS). In addition, two efficient protocols are presented; the first one is probabilistic and solves consensus with a single CAS register access in expectation, while the second one is deterministic and requires a single CAS register access when some favorable network conditions occur. A benefit of those protocols is that they can ensure both liveness and safety, and only their efficiency is affected by the probabilistic and timing assumptions.

Keywords: Consensus, cloud computing, message passing, lower bounds.

1 Introduction

Distributed consensus [19] is one of the most fundamental distributed computing problems. Over the last few decades, it was intensively explored in different computation models, including message passing and shared memory models. In this paper, we consider a novel hybrid model, where computing parties, or nodes, may exchange information by passing messages and/or by accessing a shared reliable and highly-available register. Our work is inspired by the proliferation of *cloud computing*, which may provide services that implement such a register.

Cloud computing is an emerging paradigm in which various services can be placed in a data center equipped with a management middleware that ensures the service's availability, fault-tolerance, and scalability in an almost transparent manner. Consequently, when designing contemporary distributed systems, it is tempting to resort to centralized architectures, in which the crux of the system is executed as a cloud service, which is

* This work is partially supported by the Israeli Science Foundation grant 1247/09 and by the Technion Hasso Plattner Research School.

** The work of this author on the paper was done while he was with the Department of Computer Science, Technion.

accessed in a client/server fashion by the participating nodes. As an example, the distributed consensus problem can be solved with a single cloud based *compare-and-swap* (CAS) register[1] as follows. Assume the range of decision values is D; let $\perp$ be a value not in D and initialize the CAS register to $\perp$. The consensus protocol is simply to have every node p_i invoke CAS with $\perp$ and its initial value v_i and then decide based on the value read from the CAS register [25]. It is easy to verify that this simple protocol solves consensus despite any number of benign failures among the (non-cloud) participants in a completely asynchronous environment (outside the cloud), while as we discuss later, such a CAS register can be easily implemented by existing cloud services.

The problem with this well known solution is that it requires each process (or node) participating in the consensus protocol to access the cloud hosted register at least once on each instantiation of the consensus protocol. This imposes a high load on the cloud servers, limiting their scalability, and may incur high monetary cost to the nodes themselves [30]. Hence, we are motivated to develop protocols that exploit a cloud hosted register to obtain simplicity and guaranteed fast termination while minimizing the number of accesses to the register.

Specifically, we make the following contributions: First, we present the formal model of hybrid distributed computing and a formal definition of efficient distributed consensus within this model. Second, we prove a lower bound on the number of cloud hosted register accesses required to solve consensus in the benign crash failure model. We show that whenever the number of potential failures is f (for $f < n$), any deterministic protocol that solves consensus in a hybrid asynchronous system requires invoking at least $f + 1$ cloud-based register operations. The proof itself relies on the proof of the famous FLP result [19]. However, unlike the FLP model, here we cannot immediately deduce the case of $f > k$ directly from $f = k$ (for any k). This is because whenever we increase f, we allow the protocol to use more CAS operations, giving it more power than it has with smaller values of f. Thus, we develop a novel inductive argument to show our result.

Third, we develop three efficient protocols for solving consensus despite benign failures. The first protocol always invokes exactly $f + 1$ CAS operations, thereby meeting the lower bound, as described above, for deterministic protocols in a hybrid asynchronous system. The second protocol utilizes an Ω-like oracle [2][2]. However, the reliance on the Ω-like oracle is only in order to ensure efficiency, so that only a single CAS operation will be invoked when certain network conditions are met. Termination and safety are always ensured, regardless of whether the oracle/black-box really provides its semantics or not. Our third protocol is probabilistic, and it ensures that the expected number of CAS invocations will be 1. Here again, termination and safety are always ensured, and randomization only affects the expected efficiency of the protocol.

Finally, in the Appendix, we show how to apply the hybrid approach to the non-blocking atomic commit problem [9]. In the full version of this paper, we also discuss how the approach can be extended to cover Byzantine failures [29].

[1] We assume the common semantics for the CAS register; see formal definition in Section 2.2.

[2] Later in the paper, we use the term *black-box* rather than oracle, to emphasize the fact that both the liveness and safety properties of the protocol are guaranteed even if the black-box fails to provide its semantics.

The rest of this paper is organized as follows: The model and basic definitions appear in Section 2. The main results appear in Section 3. We compare our work with related work in Section 4 and conclude with a discussion in Section 5.

2 Preliminaries

2.1 Basic Model and Assumptions

We assume an asynchronous distributed system as modeled in [7], but enhance it with a cloud hosted register. That is, a hybrid asynchronous system consists of a set of n *nodes* as well as a register R implemented by means of a cloud service. The register may support any set of operations that have a sequential specification [26], exposed through a well defined interface. The register is assumed to be highly available and fault-tolerant, meaning that any invocation of one of the operation on the register by any of the nodes is guaranteed to terminate with a response within a finite time[3].

The nodes themselves can be modeled as deterministic automata similar to what is done in [7], and their state therefore advances by taking *steps*. The communication between nodes is performed by sending and receiving messages over a *network*. Each step of a node includes receiving 0 or more messages, performing some computation, and then generating 0 or more messages to be sent to other nodes and/or to the cloud hosted register. The node that generates a message is called the *sender* and the node that receives the message is called a *receiver*. The receiver is always known to the sender and vice versa. The network is further assumed to be reliable, meaning that messages transmitted are eventually delivered once and only once, they are delivered without being altered and messages that are delivered were indeed sent by their sender. Note that these properties can be easily provided on top of weaker networks, e.g., by adding summaries to protect against data corruption and by an ACK or NACK based retransmission mechanism in *fair-lossy* networks [8].

Yet, the nodes and the network are assumed to be asynchronous in the sense that there is no bound on the time for performing a step of a node and the time between the sending of a message until its delivery at the receiver, also known as the *latency* of the message. Nodes can have access to a local clock, but the local clocks of different nodes are not synchronized.

External observers of the system may have access to a global time. Hence, this global time can only be useful for external analysis of events in the system. For convenience, we further assume that the range of the global time T is the set of natural numbers $\mathcal{N}$.

For our lower bounds, we further borrow the well known definitions of a *protocol execution*, a *protocol configuration*, an *execution prefix*, an *execution extension*, and *indistinguishable executions* from the textbook of [7]. Intuitively, we can assume a sequential *scheduler* that may schedule one node at a time in any order it wishes to. A protocol execution is the sequence of steps taken by each process whenever it is scheduled by the scheduler. Such a step depends on the node's code (the protocol) and its state

[3] We note that in reality even a highly reliable cloud service can become temporarily unavailable due to, e.g., network congestion. The liveness of hybrid consensus protocols discussed in this paper depends on the liveness of the cloud service implementing R.

at the beginning of the step. An execution prefix is a prefix of the sequence of operations composing an execution. A configuration for a given execution prefix is the collection of states of each node at the end of this prefix as well as the messages already sent but not received during this prefix. An initial configuration is the collection of initial states of each process. An execution extension is a possible sequence of steps that can be obtained from a given configuration based on the protocol and the scheduler. Finally, two executions (or execution prefixes) are indistinguishable if the processes participating in them receive exactly the same messages in each of their steps.

2.2 Benign Failures

In the benign crash failure model, up to f nodes may *fail* by crashing anytime during the execution of their protocol. A crashed node stops executing its steps and, in particular, stops sending messages. A node that fails is called *faulty*. Nodes that do not fail are called *non-faulty*, or *alive*.

2.3 Additional Services

CAS register: Our consensus protocols make use of a cloud hosted *compare-and-swap* (CAS) register object providing its usual semantics. That is, its interface includes a single method, whose signature is

```
object oldValue = compareAndSwap(object expectedValue,
                                 object newValue).
```

When invoked, this method atomically sets the value of the object to `newValue` if and only if its value at the time of invocation is equal to `expectedValue`. The method always returns the value of the object as it was just before its execution. The register is initialized with a special $\perp$ value, which cannot be a valid input of any node participating in the consensus protocol.

We focus on CAS since it is supported by existing cloud APIs, such as Windows Azure's REST API for accessing Azure Table and Blob Storage services (using the IF-MATCH header)[4] and Amazon Web Services Conditional Put for SimpleDB[5]. It can also be implemented by Yahoo's PNUTS `Test-and-Set-Write` operation [16]. Yet, the results for the benign failures model can be applied to any other object whose consensus number is ∞. Also, note that our lower bound does not assume any specific interface supported by the cloud hosted register. In particular, the lower bound holds for a register that supports any set of operations that have a sequential specification.

The $\tilde{\Omega}$ black-box: In one of our deterministic consensus protocols, we make use of a $\tilde{\Omega}$ black-box, which provides the following service. When invoked, it always returns the id of a single process that is presumed to be alive. Yet, whenever the system starts behaving in a synchronous way, then eventually all invocations of this black-box by

[4] `http://msdn.microsoft.com/en-us/library/dd179427.aspx`

[5] `http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/ConditionalPut.html`

all nodes return the id of the same process, which is also indeed alive. Clearly, any known implementation of an Ω failure detector in a system that eventually becomes synchronous, or in other words has a *Global Synchronization Time* (GST) [12, 18], can be used to implement a $\tilde{\Omega}$ black-box regardless of the timing assumptions. Examples of such implementations include, e.g., [2]. Thus, implementing $\tilde{\Omega}$ is out of scope.

Given an execution σ of a protocol using a specific implementation of a black-box *BX* of the class $\tilde{\Omega}$, if there exists a time t_1 such that from t_1 onward, every call by any node to *BX* returns the same node id and this returned node is non-faulty in σ, then we say that the *execution stabilization time*, denoted *EST*(σ,*BX*), is t_1. Otherwise, we define *EST*(σ,*BX*) to be ∞. Denote t_0 the global starting time of the protocol. Whenever *EST*(σ,*BX*) $\leq t_0$ we say that *BX* is *well behaved* in σ.

Random generator: In our randomized protocols, we assume that each node has access to a random numbers generator, which in the analysis is assumed to return truly uniformly distributed results. In practice, as the correctness of the protocols does not depend on this assumption, the assumption can be safely relaxed to any modern pseudo-random number generator that is common in modern computers with negligible impact on the actual performance of the protocols.

3 Hybrid Consensus with Benign Failures

3.1 Problem Statement

In the consensus problem, each node p_i starts with an initial value v_i from some range V. Each node p_i is required to compute a decision value. A protocol solving the consensus problem must satisfy the following properties:

Validity: The value decided on by each non-faulty node is one of the initial values.
Agreement: The decision values of all non-faulty nodes that decide are the same.
Termination: Eventually, every non-faulty node decides on some value.

As mentioned in the introduction, given that we assume the presence of a highly-available (CAS) register object, it is possible to solve consensus despite any number of nodes' crash failures by having each node access the register once with its initial value. Yet, this imposes scalability and economical problems as each instantiation of such a protocol involves a total of n CAS invocations. Consequently, we introduce the following definitions of *hybrid efficient consensus protocols.*

Definition 1 (Hybrid k-efficient execution). *Given an execution σ of a protocol for solving distributed consensus in which f nodes are faulty, σ is called* hybrid k-efficient *if the total number of register accesses in σ is at most k.*

Definition 2 (Hybrid k-efficient protocol). *A protocol for solving distributed consensus is called* hybrid k-efficient *if all its executions are hybrid k-efficient.*

Definition 3 (Hybrid efficient probabilistic protocol). *A randomized protocol for solving distributed consensus is called* hybrid efficient probabilistic *if the expected number of register accesses in an arbitrary execution of the protocol is* 1.

3.2 Lower Bounds

For simplicity, we show the proof for the binary case, i.e., when the only allowed values are 0 and 1. Extending it to a larger domain is trivial. Also, before stating and proving the lower bound, we repeat the known definition of *valency* [7,19], which plays a crucial role in the proof. Specifically, a configuration of a protocol solving consensus in called *bi-valent* if it has at least one execution extension in which the decision value is 0 and at least one execution extension in which the decision value if 1. A configuration is *uni-valent* if in all of its execution extensions nodes decide on the same value; if the value is 0, the configuration is called 0-valent and it is said to be 1-valent otherwise.

Theorem 1. *In a hybrid asynchronous system prone to f benign failures, there does not exist a hybrid f-efficient protocol.*

Before going into the proof's details, let us remark that, as mentioned in the introduction, unlike the standard asynchronous system model, here the fact that consensus cannot be solved in a hybrid f-efficient manner for $f = 1$ does not immediately imply that it holds for $f > 1$. This is because by the definition of a hybrid efficient protocol, increasing f also increases the power of the system by allowing the protocol to invoke more operations on the shared cloud hosted register. The proof below builds upon the FLP proof as it appears in the textbook of [7][6].

Proof. We prove the theorem by induction on f, the number of allowed failures and register accesses. As for the base of the induction, when $f = 1$, only a single register access is permitted. Clearly, when the register can only be accessed once, only the process that accessed the register knows the result of that access. (Note that in a trivial case when the register always returns the same value so that processes know the result of the access without actually accessing the register, the existence of a hybrid 1-efficient protocol would immediately contradict the FLP result [19]). Hence, this process can intuitively simulate the register access without anyone noticing.

More formally, if there exists a hybrid f-efficient protocol $\mathcal{P}$ with $f = 1$, then define a corresponding protocol $\mathcal{P}'$ in which whenever a process accesses the register in $\mathcal{P}$, then this same process would execute locally in $\mathcal{P}'$ the same computation as in the function supported by the register according to its sequential specification for a single invocation. By the assumption about $\mathcal{P}$, all its executions terminate and ensure the validity and agreement properties of consensus. Moreover, by construction, each execution σ of $\mathcal{P}$ has a corresponding execution σ' of $\mathcal{P}'$ that is indistinguishable from it. Hence, each such execution σ' also terminates and ensures validity and agreement. In other words, $\mathcal{P}'$ solves consensus in an asynchronous environment prone to failures with $f = 1$ (without accessing the cloud hosted register). This contradicts the famous FLP result [19].

Induction Step: Assume that the theorem holds for $f = k$, we will show that it holds for $f = k + 1$ as well. To that end, assume by contradiction that there exists a hybrid

[6] To be precise, in [7] there is only a proof for the read/write shared memory model. However, an earlier version [5] includes a complete proof for the message passing model that follows the same steps and terminology.

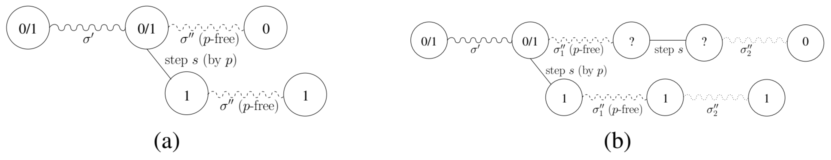

Fig. 1. Illustrations for the lower bound proof. Valences of configurations are indicated by values in boxes. Question marks specify configurations that might be either bi-valent or univalent.

f-efficient protocol $\mathcal{P}$ with $f = k + 1$. Hence, by the induction hypothesis, some of the executions of $\mathcal{P}$ must involve $k + 1$ register accesses (else, $\mathcal{P}$ solves consensus with only k operations on the register despite $f = k + 1$ failures).

We claim that $\mathcal{P}$ has at least one bi-valent initial configuration. The proof of this claim is exactly the same as the proof of the corresponding claim in the FLP result. Now, consider all executions of $\mathcal{P}$ that start from bi-valent initial configurations. Clearly, there is at least one such execution σ of $\mathcal{P}$ (that starts from a bi-valent configuration) in which at most k nodes have failed prior to the invocation of the $k + 1$ operation on the register. The existence of this execution can be easily proved by contradiction using a simple indistinguishability argument on the last failure prior to invoking the $k + 1$ operation on the register – this proof is eliminated for lack of space.

We further claim that the configuration immediately after invoking the $k+1$ operation on the register has to be uni-valent. Otherwise, we remain with a bi-valent configuration in an asynchronous system and no additional operations on the register can be invoked. Here we can apply the same arguments as in the FLP result showing that there has to be at least one such execution that is either infinite or violates agreement.

Thus, there has to be some step s taken by some process p in σ such that the configuration prior to s is bi-valent and the configuration after s is uni-valent. Also, s is either the $k + 1$ invocation of the register mentioned above, or a prior step in σ. Consider the prefix σ' of σ that ends just before s. Note that due to the determinism of $\mathcal{P}$ and the asynchrony of the system, for every (possibly empty) valid extension $\tilde{\sigma}$ of σ' that does not include any step of p, $\tilde{\sigma}s$ is also a valid extension of σ'.

Assume, w.l.o.g., that the configuration immediately after s is 1-valent. Since the configuration at the end of σ' is bi-valent, σ' has a valid extension σ'' such that $\sigma'\sigma''$ ends in a 0-valent configuration; denote by σ'' the shortest such extension. If σ'' does not include any step by p, then $\sigma'\sigma''s$ is a valid extension of σ'. Moreover, it is by definition 0-valent. Similarly, due to the fact that σ'' does not include any operation by p and the asynchrony of the system, $\sigma's\sigma''$ is also a valid extension of σ', yet is 1-valent. However, processes cannot distinguish between executions that extend $\sigma'\sigma''s$ and $\sigma's\sigma''$ (see illustration in Figure 1a). A contradiction.

On the other hand, if σ'' does include a step by p, then σ'' can be written as $\sigma''_1 s\sigma''_2$, where σ''_1 must include at least one operation and none of the operations in σ''_1 are by p while σ''_2 may include zero or more operations by any processes. Thus, we have that $\sigma'\sigma''_1 s\sigma''_2$ is 0-valent. However, since σ''_1 does not include any operation by p and due to the asynchrony of the system, $\sigma's\sigma''_1\sigma''_2$ is also a valid extension of σ', which is 1-valent. Yet, processes cannot distinguish between executions that extend $\sigma'\sigma''_1 s\sigma''_2$ and $\sigma's\sigma''_1\sigma''_2$ (see illustration in Figure 1b). A contradiction.

Algorithm 1. An $(f+1)$-efficient protocol - code for node i

```
1: undecided := true
2: if $i \leq (f+1)$ then
3:     $r :=$ CAS($\perp, v_i$)
4:     if $r == \perp$ then
5:         decide($v_i$)
6:     else
7:         decide($r$)
8:     end if
9: end if
10: decide($v$)
11: if undecided then
12:     undecided := false
13:     broadcast(DEC,$v$)
14:     return $v$
15: end if
16: upon receiving broadcast(DEC,$v$)
17:     decide($v$)
```

3.3 Upper Bounds

A Hybrid $(f+1)$-efficient Protocol. Our first protocol utilizes the CAS register implemented by cloud-based services as following. Each node having id smaller than or equal to $f+1$ invokes CAS. The first node to succeed, i.e., the first node to receive the special $\perp$ value as a response from CAS, decides on its value. Other nodes that invoke CAS and get some non-$\perp$ value decide on that value. The rest of the nodes simply wait until a *decide* message broadcasted by one of the deciders reaches them.

The pseudo-code of this simple protocol is given in Algorithm 1. In the following theorem we prove that our first protocol is hybrid $(f+1)$-efficient.

Theorem 2. *Protocol 1 is hybrid $(f+1)$-efficient.*

Proof. The safety properties of the protocol follow trivially from the properties of the CAS register and the fact that it is always invoked with the initial value of the process that calls it. As for termination, since at most f nodes can be faulty, in every execution there is at least one live process that will invoke CAS and therefore terminate. The failure efficiency of the protocol follows trivially from the code, where only the first $f+1$ nodes invoke CAS.

Notice that the code for handling DEC messages, and in particular Line 13 involve n (unreliable) broadcasts. These can be replaced by a more efficient reliable broadcast protocol, e.g., [27]. As this is a known trick, details are omitted for clarity and brevity.

An $\tilde{\Omega}$-Based Protocol. The problem with Protocol 1 is that in each execution of the protocol, the CAS register object is invoked $f+1$ times regardless of the actual number

Algorithm 2. $\tilde{\Omega}$-based protocol - code for node i

```
1:  for {$j = 1; j <= limit$ && $undecided$; j++} do
2:    if $p_i == \tilde{\Omega}$.get() or $j == limit$ then
3:      $r := \texttt{CAS}(\perp, v_i)$
4:      if $r == \perp$ then
5:        decide($v_i$)
6:      else
7:        decide($r$)
8:      end if
9:    end if
10:   wait $\Delta$ time
11: end for
    {The code for decide and for handling DEC messages is the same as in Algorithm 1}
```

of faulty nodes. It also requires knowing f up front. This brings the question of whether we can devise a protocol that will terminate with fewer CAS invocations (preferably, just one), at least when the environment behaves "favorably". We answer this affirmatively by presenting Protocol 2, which relies on an $\tilde{\Omega}$ black-box to limit the number of CAS invocations whenever it is well behaved.

The pseudo-code is given in Algorithm 2. There, a node repeatedly queries the $\tilde{\Omega}$ black-box and accesses the CAS register only when the black-box returns its id, or a threshold of iterations controlled by the configuration parameter *limit* has passed. Notice also that due to the properties of the CAS register object, safety and termination are always ensured with this protocol, regardless of the behavior of $\tilde{\Omega}$. The latter only impacts the performance of the protocol in terms of running time and the number of CAS invocations (or cloud accesses). In particular, if the failure detector is guaranteed to meet its specification (even at an arbitrary, unknown but finite, eventual time), then *limit* can be eliminated (or set to ∞).

Between iterations, each node waits for Δ time, which is the node's estimate for the time required for a computation step of the protocol including the expected latency of decision messages to propagate through the network. Clearly, a bad estimation of Δ does not hurt the correctness of the protocol, only its performance. Specifically, overestimating Δ may slightly increase the running time of the protocol as it delays polling the failure detector while underestimating Δ may result in redundant polling of the failure detector and potentially reaching *limit* iterations and invoking CAS redundantly simply due to a node that did not wait long enough for a decision message (DEC) to arrive.

The following lemma shows that in any execution in which the black-box is well behaved, only a single node needs to invoke the CAS register at the cloud.

Lemma 1. *Every execution of the protocol in which $\tilde{\Omega}$ is well behaved is hybrid* 1-*efficient.*

Algorithm 3. Randomized protocol - code for node i

```
for {$j = 1; j <=$ limit && undecided; j++} do
    if Random($n$) $==0$ or $j ==$ limit then
        $r :=$ CAS($\perp, v_i$)
        if $r == \perp$ then
            decide($v_i$)
        else
            decide($r$)
        end if
    end if
    wait $\Delta$ time
end for
{The code for decide and for handling DEC messages is the same as in Algorithm 1}
```

Proof. Consider an execution σ of the protocol. From the assumption that $\tilde{\Omega}$ is well behaved in σ, a single non-faulty process will evaluate the condition in Line 2 to `true` and therefore a single node will invoke CAS, decide, and transmit its decision to all others. Hence, all other alive nodes will decide and terminate without invoking CAS.

A Randomized Protocol. The randomized protocol is presented in Algorithm 3. It utilizes a random generator that accepts a parameter k and returns a uniformly selected integer value in the range $[0, \ldots, (k-1)]$. This protocol also relies on the configuration parameter called *limit*, which limits the maximal number of iterations that a node is willing to wait before invoking CAS deterministically. In each iteration, every node chooses to invoke CAS with probability $1/n$, as listed in Line 2. As before, between iterations, each node waits for Δ time. Here, underestimating Δ may result in redundant invocations of CAS simply due to a node that did not wait long enough for a decision message (DEC) to arrive.

Clearly, the worst case running time of the protocol is $\Delta \cdot$*limit* and in the worst case, there will be n invocations of CAS. For the following analysis, assume that Δ is correctly estimated. That is, once some node invokes CAS in a given iteration, then all nodes will decide in this iteration.

Denote by X the number of iterations required by the algorithm. When *limit* is set to ∞, the probability p that none of the nodes will invoke CAS in an arbitrary iteration is given by $(1-1/n)^n$. Obviously, the probability that at least one node will invoke CAS is $1-p$. Thus, the expected number of iterations until at least one node invokes CAS (and decides) is given by

$$E(X) = \sum_{j=1}^{\infty}[(1-p) \cdot j \cdot p^{j-1}] = \frac{1}{1-p} = \frac{1}{1-(1-\frac{1}{n})^n}.$$

Notice that when n is very large, $(1-\frac{1}{n})^n$ approaches e^{-1}, in which case the above expression becomes roughly 1.588. Moreover, the lower the value of *limit* is, the closer the expected number of iterations becomes to 1.

It is worth noting that when *limit* is set to ∞, the probability that the algorithm will require more than $a > 0$ rounds is given by $Pr(X > a) = \left[1 - \frac{1}{n}\right]^{n \cdot a}$. It follows then

$$Pr(X > \ln n) = \left[1 - \frac{1}{n}\right]^{n \cdot \ln n} < e^{-\ln n} = \frac{1}{n}.$$

In other words, with high probability, the algorithm requires $\ln n$ rounds or less.

We now calculate the expected number of CAS invocations. From the assumption on Δ, it is enough to consider the first iteration in which CAS was invoked (i.e., the last iteration of the protocol). Notice that the invocations of CAS in each individual iteration can be viewed as a set of n Bernoulli trials, each with probability $1/n$ of success. Hence, the expected number of invocations in each iteration is 1. As this is true for each iteration, it is also true for the iteration in which CAS was indeed invoked.

Note that safety and termination are always deterministically ensured for the probabilistic protocol we have presented. The randomization aspect only controls the expected running time and the number of CAS invocations. In other words, probability only affects performance, but not correctness. It is always possible to play with the parameter *limit* and the distribution of the random number generator in order to tradeoff faster termination vs. fewer CAS invocations.

4 Related Work

Since the famous FLP impossibility result for solving consensus was introduced [19], a plethora of papers on how to circumvent it have been published. Some follow the line of Chandra and Toueg as well as Lamport by enriching the environment with failure detector oracles [12, 13, 28], while others weaken the termination guarantees to being probabilistic, e.g., [6, 10, 11] to list a few.

In Disk-Paxos [20] and Byzantine Disk-Paxos [1], consensus is solved by relying of shared disks, such that each process can write to a certain block and read a certain fraction of other processes' blocks. These works also rely on a shared storage service. However, they only require read-write semantics from the shared storage, and hence must also rely on a leader oracle to ensure termination. The inspiration for these works are *storage area networks* (SAN).

Motivated by SAN and *active storage technology*, in Active Disk-Paxos [14], multiple fail prone *read-modify-write* registers are used to implement a *ranked-register* abstraction, which is then used to implement a Paxos style consensus protocol. The main benefit of this is that it enables solving consensus with an unbounded number of clients. In our work, we merely use the number of clients in the random protocol to ensure that with high probability only a single client will access the CAS object. As this number is not required for correctness, we can replace it with a rough estimate on the actual number of clients rather than an exact figure. Also, similarly to the original Disk-Paxos works, Active Disk-Paxos does not try to minimize the number of storage accesses.

Past work has investigated the minimal synchrony, and in particular the minimal number of synchronous links required to solve consensus [3,4,24]. The idea in this line of work is that synchrony is hard to ensure (whereas total lack of synchrony prevents deterministic solutions for consensus [19]). Thus, synchronous links are likely to be expensive or have lower bandwidth than asynchronous ones, which motivates investigating how to use them in the most parsimonious manner.

Wormholes is another approach for solving consensus, both Byzantine and benign, by relying on special secure, synchronous, and temper-proof channels [17]. As was shown in [17], wormholes can greatly reduce the complexity of solving consensus and improve the ratio of faulty processes required to solve the problem. There, too, wormholes are assumed to be expensive and offer lower bandwidth than the "standard" network. Consequently, wormholes should be used judiciously.

Both lines of research (minimal synchrony and wormholes) share the vision of adding some expensive service to enable and simplify solving consensus. Both also investigate how to use such a service wisely. In that sense, it is related to our work. However, in their case, the service is a special type of a communication link, whereas in our case it is a cloud hosted service.

Golab et al. [21] use the remote memory references (RMRs) metric to measure the performance of algorithms that solve consensus and other related problems in two asynchronous shared memory models. They consider blocking algorithms, and distinguish between local and remote memory accesses, where the latter traverse the processor-to-memory interconnect. They show that in this setting the consensus problem can be solved using only a constant number of RMRs, while the progress is guaranteed only when all active process are alive.

Guerraoui and Schiper discuss a related idea of a consensus service, which might be used by client processes to solve an agreement problem [23]. The consensus service is implemented by a set of server processes, which might be the same as or distinct from client processes. The paper concentrates on the generality of the suggested service, showing how it can be used to solve a series of agreement problems. Even though Guerraoui and Schiper consider communication costs of the described protocols, they do not distinguish between messages sent by clients and by servers, and thus do not strive to optimize the communication between clients and servers.

5 Discussion

In this paper we have focused on solving consensus in hybrid systems, where processes communicate by message passing and by accessing a shared highly-available register. In particular, inspired by the proliferation of cloud-based services, we have studied the problem of minimizing the number of register accesses for better scalability and cost reduction in cloud assisted implementations of the suggested hybrid model. The hybrid approach brings several benefits. First, in the case of benign failures, it enables solving consensus in an otherwise asynchronous environment with $f < n$ failures. The protocols are very simple and terminate quickly. Also, our randomized and $\tilde{\Omega}$-based deterministic protocol enable solving consensus with a single register access in expectation or in the "typical" case, respectively.[7] We have also shown a lower bound on the number of register accesses for deterministic protocols as well as a protocol that

[7] Let us reiterate that in these protocols, termination and safety are always ensured regardless of any synchrony assumptions and only the efficiency of the protocol depends on the behavior of the black-box or timeout setup.

satisfies this bound. A shortcoming of the $\tilde{\Omega}$-based and the randomized protocol is that when the black-box in not well behaved in the former or the timeout is not accurate in the latter, the protocol may require up to n accesses to the cloud hosted register. Limiting this number to $f+1$ in such cases is left for future work.

For the Byzantine case, we conjecture that the lower bound for deterministic protocols is $3f+1$ register accesses and $f < 3n$. In the full version of this paper, we show an algorithm that meets this presumed bound. We also show there a probabilistic protocol that terminates with just one cloud access by correct nodes in expectation while always ensuring termination and safety. In addition to proving the conjectured lower bound, an open problem related to Byzantine failures is the impact of the semantics of the register on the minimal number of register accesses in the deterministic case. That is, can stronger objects reduce the required number of register accesses?

Acknowledgments. We would like to thank Sergey Bykov and Alan Geller from Microsoft Research for their help and advice.

References

1. Abraham, I., Chockler, G.V., Keidar, I., Malkhi, D.: Byzantine disk paxos: optimal resilience with byzantine shared memory. In: Proc. of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 226–235 (2004)
2. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega with weak reliability and synchrony assumptions. In: Proc. of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 306–314 (2003)
3. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-efficient leader election and consensus with limited link synchrony. In: Proc. of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 328–337 (2004)
4. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Consensus with byzantine failures and little system synchrony. In: Proc. of the International IEEE Conference on Dependable Systems and Networks (DSN), pp. 147–155 (2006)
5. Attiya, H.: Lecture notes for course #236357: Distributed algorithms (spring 1993); Technical report, Department of Computer Science, The Technion (January 1994)
6. Attiya, H., Censor-Hillel, K.: Lower bounds for randomized consensus under a weak adversary. SIAM J. Comput. 39(8), 3885–3904 (2010)
7. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics, 2nd edn. John Wiley and Sons, Inc. (2004)
8. Basu, A., Charron-Bost, B., Toueg, S.: Simulating reliable links with unreliable links in the presence of process crashes. In: Babaoğlu, Ö., Marzullo, K. (eds.) WDAG 1996. LNCS, vol. 1151, pp. 105–122. Springer, Heidelberg (1996)
9. Bernstein, P., Hadzilacos, V., Goodman, H.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
10. Bracha, G.: An $o(\lg n)$ expected rounds randomized byzantine generals protocol. In: Proc. 17th Annual ACM Symposium on Theory of Computing (STOC), pp. 316–326 (1985)
11. Canetti, R., Rabin, T.: Fast asynchronous byzantine agreement with optimal resilience. In: Proc. 25th Annual ACM Symposium on Theory of Computing (STOC), pp. 42–51 (1993)

12. Chandra, T., Toueg, S.: Unreliable failure detectors for asynchronous systems. J. ACM 43(4), 685–722 (1996)
13. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43, 685–722 (1996)
14. Chockler, G., Malkhi, D.: Active disk paxos with infinitely many processes. In: Proc. of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 78–87 (2002)
15. Chu, F.: Reducing ω to $\diamond s$. Information Processing Letters 67(6), 298–293 (1998)
16. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. Proc. of VLDB Endowment 1, 1277–1288 (2008)
17. Correia, M., Neves, N.F., Lung, L.C., Veríssimo, P.: Low complexity byzantine-resilient consensus. Distributed Computing 17, 237–249 (2005)
18. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the Presence of Partial Synchrony. Journal of the ACM 35(2), 288–323 (1988)
19. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32, 374–382 (1985)
20. Gafni, E., Lamport, L.: Disk paxos. Distributed Computing 16, 1–20 (2003)
21. Golab, W., Hadzilacos, V., Hendler, D., Woelfel, P.: Constant-RMR implementations of CAS and other synchronization primitives using read and write operations. In: Proc. ACM Symposium on Principles of Distributed Computing, PODC (2007)
22. Guerraoui, R.: Non-Blocking Atomic Commit in Asynchronous Distributed Systems with Failure Detectors. Distributed Computing 15, 15–17 (2002)
23. Guerraoui, R., Schiper, A.: The generic consensus service. IEEE Transactions on Software Engineering 27(1), 29–41 (2001)
24. Hamouma, M., Mostefaoui, A., Trédan, G.: Byzantine consensus with few synchronous links. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 76–89. Springer, Heidelberg (2007)
25. Herlihy, M.: Wait-free synchronization. ACM Trans. Prog. Lang. Syst. 13(1), 124–149 (1991)
26. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Trans. on Programming Languages and Systems 12(3), 463–492 (1990)
27. Kaashoek, M.F., Tanenbaum, A.S., Hummel, S.F.: An efficient reliable broadcast protocol. ACM SIGOPS Operating Systems Review 23, 5–19 (1989)
28. Lamport, L.: The part-time parliament. IEEE Transactions on Computer Systems 16(2), 133–169 (1998)
29. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems 3(4), 382–401 (1982)
30. Wang, H., Jing, Q., Jiao, S., Chen, R., He, B., Qian, Z., Zhou, L.: Distributed systems meet economics: Pricing in the cloud. In: Proc. USENIX HotCloud (2010)

A Non-Blocking Atomic Commit

Non-blocking atomic commit (NBAC), originating in the area of databases [9], is a related problem to consensus, but with a twist. That is, in NBAC, each process starts with a 'yes' or 'no' value and the processes also need to decide on the same output value, which is either 'commit' or 'abort'. However, if at least one of the initial values is 'no', then the only allowed decision value is 'abort'. Further, if all initial values are 'yes', then the only allowed decision value is 'commit' unless at least one process has

Algorithm 4. A hybrid NBAC protocol - code for node i

```
1: send($v_i$) to everyone
2: wait until received $v_j$ from every node or $?P$ ==true or ($\exists j$, $v_j$ ==no)
3: if received $v_j$ from every node and ($\forall j$, $v_j$ ==yes) then
4:     decide(hybrid-efficient-consensus(commit))
5: else
6:     decide(hybrid-efficient-consensus(abort))
7: end if
```

failed, in which case it is also permissible to decide ‘abort’. As was shown in [22], when $n > 2f$, any solution to the NBAC problem requires a failure detector of the class $\Diamond S+?P$, where $?P$ is a failure detector that eventually returns ‘true’ if and only if at least one process has failed. We also remind the reader that the failure detector class $\Diamond S$ is equivalent to Ω [15]. Clearly, it is possible to define a hybrid k-efficient (or efficient probabilistic) protocol for solving NBAC in a similar manner to what has been done above for consensus (cf. Section 3.1).

As was shown in [22], NBAC can be easily solved using a reduction to consensus. For self-containment, this reduction is repeated in Figure 4, in which we replaced the invocation of consensus with an invocation to *hybrid-efficient-consensus*, which stands for any of the hybrid k-efficient or efficient probabilistic protocols mentioned above. In the listing in Figure 4, the initial vote of each node is denoted v_i. It is easy to verify that when invoked with a $?P$ failure detector in Line 2 and the consensus protocol of Lines 4 and 6 is hybrid k-efficient (or efficient probabilistic), then the overall NBAC protocol becomes hybrid k-efficient (or efficient probabilistic), solves NBAC for any $f < n$, and only relies on $?P$ for ensuring the safety and liveness properties of NBAC.

Notice that the use of a register does not eliminate the need for a $?P$ failure detector. This is because in NBAC, it may not be safe to decide ‘commit’ before hearing from every process or knowing with certainty that at least one of the processes has failed. Hence, the benefits of the hybrid approach for the NBAC problem is in increasing the resilience to $n-1$ failures while eliminating the need for the $\Diamond S$ (or Ω) failure detector.