Survivable Consensus Objects

(Extended Abstract)

Dahlia Malkhi Michael K. Reiter

AT&T Labs-Research, Florham Park, NJ, USA
{dalia,reiter}@research.att.com

Abstract

Reaching consensus among multiple processes in a distributed system is fundamental to coordinating distributed actions. In this paper we present a new approach to building survivable consensus objects *in a system consisting of a (possibly large) collection of persistent object servers and a transient population of clients. Our consensus object implementation requires minimal support from servers, but at the same time enables clients to reach coordinated decisions despite the arbitrary (Byzantine) failure of any number of clients and up to a threshold number of servers.*

1. Introduction

A consensus object is a shared object to which a client can propose a value and receive a value in return. The consensus object returns the same value to each client, and the returned value is one proposed by some client. Applications of consensus objects to achieving distributed coordination are numerous. For example, a consensus object can be used to implement distributed locking: Each client proposes its own identifier, and the consensus object returns the identifier of the client to which the lock is granted. If the consensus object supports "commit" or "abort" proposals, then it can be used to implement transactional atomic commitment.

In this paper we describe a protocol for implementing a *survivable* consensus object in a distributed system, i.e., a consensus object that retains its correctness despite the arbitrary corruption (Byzantine failure) of some number of components involved in its implementation. The system model for which we describe our consensus object implementation is one in which there is a (possibly large) set of persistent object servers and an unknown number of transient clients that interact with the servers asynchronously; see Figure 1. Clients communicate directly only with servers, performing operations on primitive objects (described later) implemented at the servers. Using these primitive objects, the clients *emulate* a consensus object with the specification described above. The result is a consensus object emulation that allows a client to obtain a consensus value in an *expected* low-degree-polynomial number of primitive object operations as a function of the number of clients (implementations that deterministically guarantee termination are known to be impossible to achieve [8]), regardless of how many other clients simultaneously engage the object. The consensus object is emulated with no server-to-server or client-to-client communication. Moreover, the consensus object retains its properties despite even the malicious behavior of any number of clients and a limited number of servers, and is thus survivable in an environment that may suffer Byzantine failures.

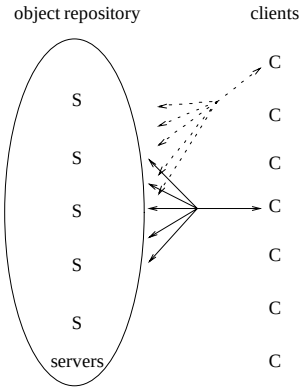

Figure 1. System model: Persistent servers support replicated objects

This system model is motivated by our larger ongoing effort to build a highly scalable, application-independent infrastructure of servers to support replicated objects, called Phalanx [15]. The goals of Phalanx impose a number of constraints that rendered existing consensus protocols inadequate. First, implementing a consensus object us-

ing an inter-server protocol, whose decision value is made available to clients, complicates server design and mandates server-to-server communication, which hurts scalability. Second, an inter-client consensus protocol would require the simultaneous participation of a high percentage of the client population to converge on a consensus value (see Section 2). A third alternative is a shared-memory approach, in which servers would support minimal, consensus-independent abstractions, e.g., reads and writes to shared variables [10], and the clients would emulate a consensus object simply by accessing those variables. This appears to be impossible, even using randomization, due to the following inherent difficulty: Using objects such as read/write registers, (Byzantine) faulty clients could configure the variables so that one correct client infers one consensus value, and then reconfigure the variables so that a later correct client infers a different value (and so that any traces left behind by the first correct client appear to have been made by a malicious one).

The consensus object implementation described here strikes a balance among these options. Servers support shared objects called *timed append-only arrays*. Clients then use these objects to emulate a consensus object. These objects enable clients to append values, but not to delete or modify previously appended values. In addition, each appended value is labeled with a logical time at which it was appended. Intuitively, these objects prevent the scenario described above because malicious clients cannot "undo" what they once did; they can only add to it. And, the timestamps partially capture the order in which different clients appended different values, which also cannot be reordered by malicious clients. At the same time, timed append-only arrays are simple enough to implement with no server-to-server communication and simple server-resident logic.

2. Related work

Consensus objects have traditionally been studied in two system models: The shared memory model and the message passing model. In each model, clients execute a distributed protocol to implement the consensus specification. In the shared memory model, clients communicate via shared memory locations. In the message passing model, clients communicate by exchanging messages over a network. An important distinction between the two is that in the shared memory model, consensus object implementations are possible in which each client can obtain the consensus value even if it is the only client that participates in the protocol [1]. In the message passing model, typically a threshold of (correct) processes need to simultaneously cooperate to achieve agreement.

As described in Section 1, our work mixes elements of both models: Clients communicate via shared objects as in the shared memory model, but these shared objects are emulated by exchanging messages with servers, a threshold of which are required to participate to emulate those shared objects. Other work has supported read/write shared-memory emulations in a message passing system, e.g., [12, 3], but these systems do not provide for fault tolerance. An emulation of atomic shared read/write registers tolerant of benign failures was provided in [2]. Compared with the above, our work differs in that (i) a consensus object is strictly stronger than any of the above object semantics; (ii) our protocols are survivable, i.e., resilient to Byzantine failures of any number of clients and a limited number of servers, and (iii) the underlying quorum techniques we employ for accessing servers [13, 14] scale to very large systems of servers.

Consensus cannot be implemented deterministically in a message-passing or shared-memory system, i.e., in a way that guarantees a unique consensus value and termination in a finite number of steps [8]. Numerous message-passing protocols have employed randomization to guarantee finite termination with probability one; see [4] for a survey. Aspnes and Herlihy [1] introduced a shared-memory randomized consensus protocol for a benign failure environment that uses read/write shared registers and terminates in expected time polynomial in the number of clients. Similarly, our consensus protocol is randomized and its expected converging time is polynomial in the number of clients (assuming a computationally bounded adversary). Of the previous works, our protocol most closely resembles [1], but differs significantly due to its tolerance to Byzantine faulty clients and servers, and due to its implementation in a message-passing (as opposed to shared memory) system.

3. System model and assumptions

Our system model consists of a group of *servers* and some number n of *clients*. Clients are denoted by $p_1, \ldots, p_n$, or just $p, q, \ldots$ when subscripts are unnecessary. Servers and clients need not be distinct. A *correct* client or server is one that obeys its functional specification. A *faulty* client or server, on the other hand, can deviate from its specification arbitrarily (Byzantine failures [11]), limited only by the assumptions stated below. Faulty clients and servers include those that fail benignly.

We assume that at most b servers fail, where b is a globally known constant, and that there is a *b-masking quorum system* $\mathcal{Q}$ known to all clients and servers [13]. That is, $\mathcal{Q}$ is a set of subsets (quorums) of servers, such that (i) for any $Q_1, Q_2 \in \mathcal{Q}$, $|Q_1 \cap Q_2| \geq 2b + 1$ and (ii) for any set B of servers where $|B| = b$, there is some $Q \in \mathcal{Q}$ such that $B \cap Q = \emptyset$. In our protocols, clients interact with servers by contacting a quorum of them. Intuitively, (i) enables clients to infer correct replies from the contacted quorum, (ii) ensures that a client can always contact a full quorum [13].

The mechanism by which clients communicate to servers is via a *quorum remote procedure call*. A client's invocation of Q-RPC(m), where m is a request, returns responses from a quorum of servers to the request m. To do this, Q-RPC(m) sends m to servers as necessary to collect responses from a quorum, and then returns these responses to the client. The Q-RPC module provides additional interfaces, e.g., that enable a caller to specify servers to avoid because those servers have been detected to be faulty (e.g., based on responses they returned to other Q-RPCs), or that enable a caller to issue a query to a partial quorum to complete a previous Q-RPC in which faulty servers returned useless values. For the purposes of this paper, however, we omit these interfaces from further discussion. Q-RPC can be implemented in an *asynchronous* system, i.e., without assuming any known bound on message transmission delays, and thus our protocols are suited for an asynchronous system. In our protocols, correct servers never send messages to other servers, and correct clients never send messages to other clients.

We assume the existence of trapdoor one-way functions [7], which are sufficient for constructing digital signature schemes (e.g., [16]). We assume that each correct server possesses a private key known only to itself with which it can *digitally sign* messages, and that any other client or server can verify the origin of a signed message but cannot forge the signature of any correct server. So, if a correct client or server attributes a signed message to a correct server u, then u sent it. Not all messages sent by servers will be signed; we will explicitly indicate that the message m is signed by u by denoting it $\langle m \rangle_u$. Aside from digital signatures, we will also make use of *function sharing* [6] primitives based on one-way functions. These techniques will be explained in the sections that use them.

4. Timed append-only arrays

The most basic function provided by the servers is the maintenance of a *timed append-only array* τ_j for each client p_j. A timed append-only array τ_j is a *single-writer multi-reader* object that allows p_j to *append* values to the array and any client to *read* values from the array. Informally, the object provides the following properties:

Append-Only: Values are appended to a timed append-only array in a sequential order.

Write-Once: Values appended to a timed append-only array are never modified or erased.

Timestamp: Each element in a timed append-only array is timestamped with a vector t satisfying the following:

> **Global-Timestamp**: $t[0]$ is a Lamport timestamp [9] that reflects the partial ordering of operations on all arrays, such that $t[0]$ is greater than the corresponding value in any array entry appended before this element was appended.
>
> **Client-Timestamp**: For each $0 < j \leq n$, $t[j]$ denotes, where not zero, that the $t[j]$'th append on τ_j was already completed when this element was appended.

A reader can access any element of the array. The reader obtains the vector timestamp along with the value of an array element, if written.

Intuitively, timed append-only arrays serve to implement non-malleable communication among Byzantine fail-prone clients: The first two properties guarantee that values are appended to each array in a sequential order that any reader can later observe. The timestamp properties enable clients to partially disambiguate the order in which appends were performed on different arrays.

4.1. Implementation

We begin by describing the implementation of timed append-only arrays. Let $\tau_1, \ldots, \tau_n$ denote the timed append-only arrays maintained by the servers. Each timed append-only array τ_j supports two kinds of operations: Client p_j can append some value v to τ_j by executing $\tau_j.\mathsf{append}(v)$, and any client can read the i-th value in τ_j by executing $\tau_j.\mathsf{read}(i)$. The $\tau_j.\mathsf{read}(i)$ operation is the simpler of the two, and so we discuss it first.

Each timed append-only array τ_j is represented in each server u by a sequence of addresses $\tau_{j,u}[1], \tau_{j,u}[2], \ldots$ that hold value/timestamp pairs. Each address is initially $\perp$. The protocol for a client to read the i-th element of τ_j is shown in Figure 2. The client executes a Q-RPC to obtain the value/timestamp pair in $\tau_{j,u}[i]$ from each server u in some quorum Q. More specifically, each server responds with a message of the form $\langle \tau_j\text{-}\mathsf{value} : i, \tau_{j,u}[i] \rangle_u$; note that this is digitally signed by u, so that it can be used in the append protocol below if necessary. The client obtains the result of the read by discarding any value/timestamp pair returned by b or fewer servers, and choosing the remaining (unique, as we show below) value/timestamp pair, say $\langle v, t \rangle$. The client also records the fact that it has read the i-th element of τ_j by setting the j-th element of a local array ts to be i (if larger than $ts[j]$), and retains the set $C_{j,v,t} = \{\langle \tau_j\text{-}\mathsf{value} : i, \langle v, t \rangle \rangle_{\hat{u}} : \hat{u} \in Q\}$ of at least $b + 1$ signed messages for $\langle v, t \rangle$.

The $\tau_j.\mathsf{append}$ operation is significantly more involved than the $\tau_j.\mathsf{read}$ operation; see Figure 3. Each server u maintains, in addition to array entries, a *timestamp generator* G_u and *echo inhibitors* $e_{k,u}, 1 \leq k \leq n$, all initially zero. The i-th $\tau_j.\mathsf{append}$ proceeds in three phases. In the first, p_j sends its timestamp vector ts to a quorum of servers. Recall the k-th element of ts ($1 \leq k \leq n$) indicates the

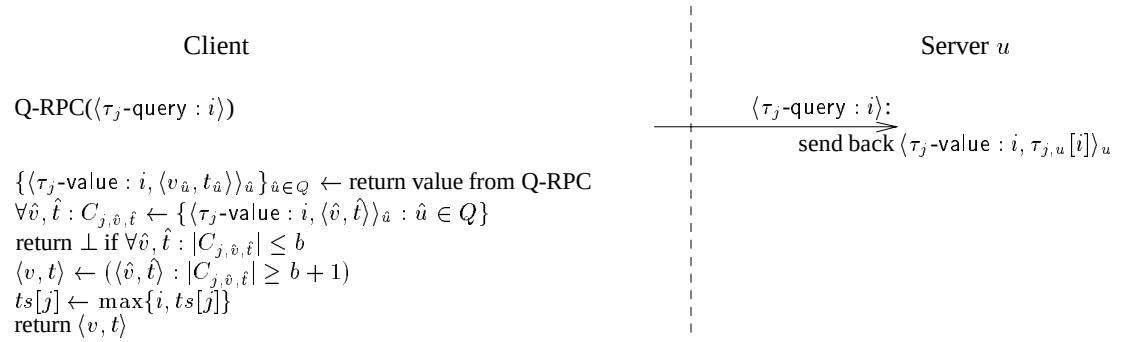

Figure 2. The operation τ_j.read(i)

highest index of τ_k that p_j has successfully read. In order to reply to p_j's request, each server u requires that for each $1 \leq k \leq n$, it holds a value in $\tau_{k,u}[ts[k]]$. Since this value may have been written to a quorum not containing u, p_j piggybacks $C_{k,\tau_k[ts[k]]}$ as needed on its request, which it collected from servers when it read $\tau_k[ts[k]]$, to enable u to fill in $\tau_{k,u}[ts[k]]$. The server u then verifies that for each $1 \leq k \leq n$, it holds a value in $\tau_{k,u}[ts[k]]$ and, if so, responds to p_j with its present value of G_u, digitally signed.

Upon the completion of this first Q-RPC, p_j now forwards the digitally signed replies back to the servers via a second Q-RPC. Based on these included messages, server u verifies that $i > e_{j,u}$ and then computes a vector timestamp t for this τ_j.append operation that agrees with p_j's ts array in positions $1 \ldots n$ and that has a zeroth element higher than any of the servers' timestamp generator values forwarded in the request. The server u "echoes" t together with the value v that p_j is appending, by digitally signing both values and sending them back to p_j. u then sets $e_{j,u} \leftarrow i$ so that it will never again echo a value for the i-th τ_j.append. Finally, after receiving echoes from a quorum of servers, p_j commits the append at a quorum of servers by forwarding these echo messages to them in a third Q-RPC. Upon receiving this third request, each server u assigns $\tau_{j,u}[i] \leftarrow \langle v, t\rangle$ and acknowledges. The purpose of the echoes is to ensure that no two correct servers write different values into $\tau_{j,u}[i]$; this is ensured because each server echoes only one i-th τ_j.append value.

4.2. Properties

In proving properties of this implementation, we need to introduce some additional notation and terminology. Note that reads by faulty clients are ignored in the following.

Definition 4.1 *A τ_j.read(i) operation by a correct client* begins *when the client initiates the corresponding τ_j.read protocol, and ends when the client returns from the τ_j.read protocol.*

Definition 4.2 *The i-th τ_j.append* begins *when some correct server receives $\langle \tau_j$-gettime : $i, ts\rangle$ from p_j, and it* ends *when $\tau_{j,u}[i] \neq \perp$ at each correct server u in some quorum.*

By this definition, once the i-th τ_j.append has begun, it is possible for it to end before the protocol in Figure 3 completes. In particular, if a client $p_{j'}$ reads the value of the i-th τ_j.append while that append is going on, and then performs a $\tau_{j'}$.append, its own $\tau_{j'}$.append could complete the i-th τ_j.append before the protocol for the i-th τ_j.append itself completes. This property is made precise in Lemma 4.3.

Definition 4.3 *Let e, e' be any two operations (other than reads by faulty clients). We say that e* happens before *e', denoted $e \prec e'$, iff e ends before e' begins. If $e' \not\prec e$, we denote it $e \preceq e'$.*

Note that $\prec$ forms an irreflexive partial order, and that if $e_1 \prec e_2 \preceq e_3 \prec e_4$, then $e_1 \prec e_4$. Moreover, for any two operations e_1, e_2 at a correct process, either $e_1 \prec e_2$ or $e_2 \prec e_1$. That is, the operations executed by a correct process are totally ordered. On the contrary, operations by a faulty client are not necessarily totally ordered by $\prec$. Nevertheless, if e_1 and e_2 are the k-th and k'-th τ_j.append operations, respectively, by a faulty process p_j (i.e., corresponding to $\langle \tau_j$-gettime : $k, ts\rangle$ and $\langle \tau_j$-gettime : $k', ts'\rangle$ messages from p_j), such that $k < k'$, then we will often use "$e_1 \prec e_2$" as a shorthand to denote this.

Lemma 4.1 (Write-Once) *Let $e_1 = \tau_j$.read(i) and $e_2 = \tau_j$.read(i) be two operations at correct clients. If e_1 returns $\langle v, t\rangle$, then e_2 returns either $\perp$ or $\langle v, t\rangle$.*

Lemma 4.2 (Append-Only) *Let e_1 be the i-th τ_j.append, and let $e_2 = \tau_j$.read(k), $1 \leq k \leq i$, be an operation by a correct process such that $e_1 \prec e_2$. Then e_2 does not return $\perp$.*

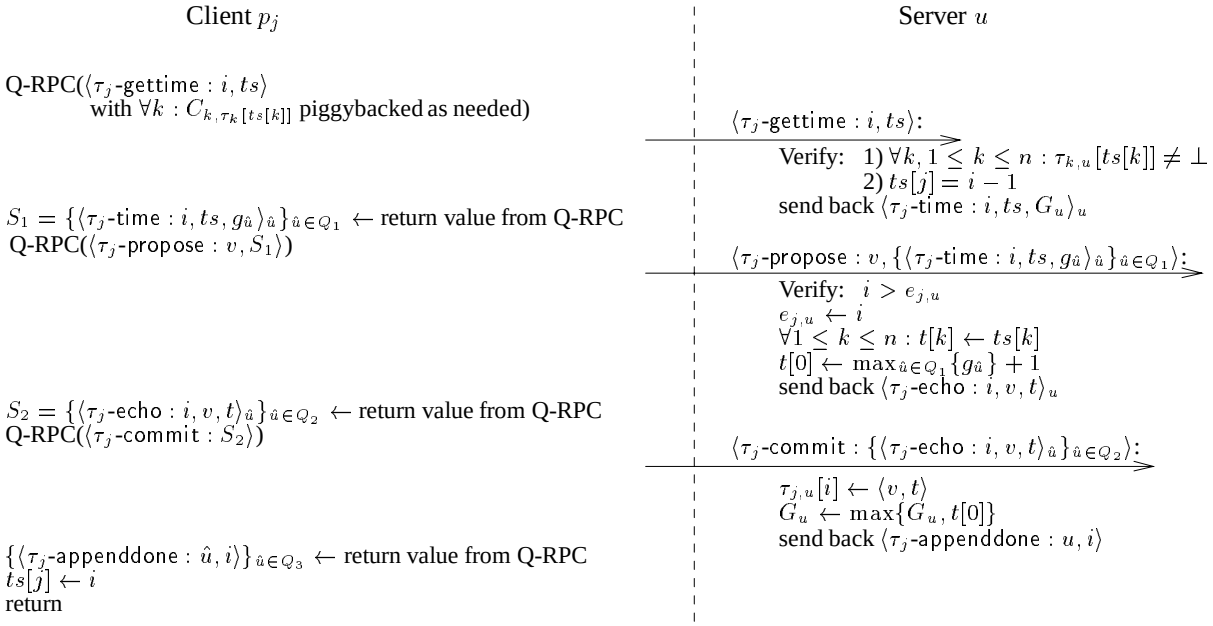

Figure 3. The i-th invocation of τ_j.append; v is the value appended

Henceforth, we will use the following notation: If e is an append operation of the form $e = \tau_j.\mathsf{append}(v)$, such that v is stored with timestamp t, we denote this timestamp by $T(e) = t$. The first lemma shows the main purpose of the non-zero entries of a vector timestamp.

Lemma 4.3 (Client-Timestamp) *Let e_1 be the i-th τ_j.append, and let $e_2 = \tau_{j'}.\mathsf{read}(k)$, where $1 \le j' \le n$ and $1 \le k \le T(e_1)[j']$, be a read operation by a correct process such that $e_1 \prec e_2$. Then e_2 does not return $\perp$.*

Lemma 4.4 (Global-Timestamp) *If $e_1 = \tau_j.\mathsf{append}(v)$, $e_2 = \tau_{j'}.\mathsf{append}(v')$, and $e_1 \prec e_2$, then $T(e_1)[0] < T(e_2)[0]$.*

Corollary 4.1 *Let e_1, $e_2 = \tau_j.\mathsf{append}(v)$, $e_3 = \tau_{j'}.\mathsf{append}(v')$, and e_4 be four events such that $e_1 \prec e_2$, $T(e_2)[0] \le T(e_3)[0]$, and $e_3 \prec e_4$; then $e_1 \prec e_4$.*

5. A consensus protocol

In this section, we describe a protocol by which clients can emulate a consensus object by performing a series of read and append operations on timed append-only arrays. Each client begins the protocol with an initial *preferred value* and ends the protocol by irrevocably *deciding* on a value. Intuitively, this decision value is the value "returned by" the consensus object. The protocol ensures the following two properties (due to space limitations, a proof of correctness is omitted).

Agreement: If any correct client decides v, then all correct clients decide v.

Validity: If any correct client decides v, then some client had v as its input value.

Our protocol employs a round structure and high-level strategy similar to [1], but otherwise differs significantly. In our protocol, each client executes a sequence of logical rounds until it reaches a decision. There is one timed append-only array per client for that client to communicate values to the system by appending them to its array. Since rounds at different clients proceed asynchronously, each client attaches its round number to each value it appends. In each round, a client starts by appending its currently preferred value, and then reads the latest values appended by all of the other processes to their arrays (a "global read"). Among these values, the ones with the highest round number are called the leaders' values. If the leaders agree (i.e., last appended the same values), it tries to adopt their value as its own preferred value and move to the next round (or decide); if the leaders disagree and it is a leader itself, it attempts to flip a (multivalued) coin, adopt the value of the coin as its preferred value, and then move to the next round.

Decision is possible for a leader when all the processes who disagree with its preferred value are at least two rounds behind. Intuitively, this protocol converges at the latest when a round starts with all the leaders preferring the same value. We argue in Section 7 that by properties of the coin flip, this occurs with some positive probability at each round.

Several points need to be refined in the description above. First, to adopt a new preferred value in a round (either the leaders' or by coin flip), a client twice performs a cycle of appending a value "announcing" its status and executing a new global read of all clients' latest values. If the status of the leaders' value hasn't changed during these two cycles (i.e., either the leaders still disagree, or the leaders still agree on the same value), then the client adopts the value it intended to. If, however, the client detects a change in the status of the leaders' value, then it starts the round over. The two append/read cycles guarantee that with two concurrently executing leaders, at least one will observe the other's value and act consistently.

Second, the process for performing the global read, i.e., reading all clients' last-appended values, involves reading the arrays of all clients up to the last filled slot, filtering out any invalid values appended by faulty clients (which will be defined precisely in Section 5.1), and returning a set of latest (valid) values from all arrays. Even though this is a compound operation, we will abuse notation and denote it by a single event Last, allowing it to be ordered as a single operation via $\prec$. In particular, if a Last operation e_1 starts before some append operation e_2 terminates, i.e., e_1 contains a primitive read operation e_1' such that $e_1' \preceq e_2$, then we say that $e_1 \preceq e_2$.

Third, we need to specify how to flip a random coin. For now, we denote this operation as subroutine Coin(). Meeting the two properties of consensus (Agreement and Validity) requires simply that Coin() return a value that was initially preferred by some process. In addition, the Coin() operation is important to the running time of the consensus protocol. This will be the topic of discussion in Section 7.

In terms of data structures, each client maintains several local variables: pref, which holds the client's present preferred value; r, which holds the client's current round number; $\langle r_j, v_j \rangle_{1 \le j \le n}$, which hold the latest (valid) round number/value pairs read from clients' arrays; leaderVals, which is the set of leaders' values; and leaderRound, which is the leaders' round number. And, as described above, there is a separate timed append-only array per client. At the beginning of its execution, each client appends $\langle 0, \text{pref} \rangle$ to its array. In the remainder of this paper, an operation τ_j.append by a client p_j is denoted simply by append, and is understood to apply to the array to which p_j is allowed to append.

The precise protocol executed by client p at round r is given in Figure 4. The Last subroutine, introduced above, implements a "global read" plus identification of the leaders' round and values. If the leaders agree, the LeadersAgree subroutine is invoked. This subroutine simply appends the leaders' value twice and if leader disagreement or a change of leader value is not observed in between these appends, it moves the client to the next round (or decides) with the leaders' value as its new preferred value. If leader disagreement is observed by Last, the LeadersDisagree subroutine is invoked. This routine appends $\perp$ twice and, if the client is a leader, adopts a new preferred value by flipping a coin, provided that leader agreement is not observed while this routine is executing.

```
(1)   private vars pref, r, ⟨r_j, v_j⟩_{1≤j≤n}, leaderVals, leaderRound

(2)   append(⟨r, pref⟩)
(3)   repeat
(4)       Last()
(5)       if (|leaderVals| > 1 ∨ ⊥ ∈ leaderVals)
(6)           LeadersDisagree()
(7)       else
(8)           LeadersAgree()
(9)   until round r + 1 enabled

(10)  subroutine LeadersAgree()
(11)      pref ← (v : leaderVals = {v})
(12)      append(⟨r, pref⟩)
(13)      Last()
(14)      if (leaderVals ≠ {pref})
(15)          return
(16)      append(⟨r, pref⟩)
(17)      Last()
(18)      if (leaderVals ≠ {pref})
(19)          return
(20)      if (r = leaderRound ∧ ∀j : (v_j ≠ pref ⇒ r_j ≤ leaderRound − 2))
(21)          decide(pref)
(22)      else
(23)          enable round r + 1

(24)  subroutine LeadersDisagree()
(25)      append(⟨r, ⊥⟩)
(26)      Last()
(27)      if (|leaderVals| = 1 ∧ ⊥ ∉ leaderVals)
(28)          return
(29)      append(⟨r, ⊥⟩)
(30)      Last()
(31)      if (|leaderVals| = 1 ∧ ⊥ ∉ leaderVals)
(32)          return
(33)      if (leaderRound = r)
(34)          pref ← Coin()
(35)      enable round r + 1

(36)  subroutine Last()
(37)      ∀j ∈ {1, ..., n}: i_max ← max{i : τ_j.read(i) is justified}
(38)                        ⟨⟨r_j, v_j⟩, t_j⟩ ← τ_j.read(i_max)
(39)      leaderRound ← max_{1≤j≤n}{r_j}
(40)      leaderVals ← {v_j : r_j = leaderRound}
```

Figure 4. Round r of the consensus protocol

5.1. Justified values

As indicated above, after each global read, the client uses the observed values to determine its next preferred value. In order to ensure correctness of our protocol, it is important that the plausibility of these observed values is verified before they are used; otherwise, a faulty client could append

arbitrary values in an effort to misguide future preferred values of other clients. We therefore introduce the notion of a *justified value*, which intuitively is an appended value that is consistent with the protocol and, in particular, with the values that the appender's preceding global read must have observed. After executing a global read, a client discards any unjustified values and forms its next preferred value based upon the justified values only.

Unfortunately, detecting justified values is not straightforward, because a reading client cannot accurately determine what values should have been observed by an appending client prior to its append. We will now make use of the Global-Timestamp property of our timed append-only arrays, to derive an estimation of the values that the appending client observed, as follows:

Definition 5.1 *Let e_p be an* append *operation executed by client p, and let e_q be a different* append *operation executed by client q. We say that e_p* definitely reflects *e_q if (i) $p = q$ and $e_q \prec e_p$ or (ii) $p \neq q$ and there exist* append *operations e'_p, e'_q executed by p and q, respectively, such that $e_q \prec e'_q$, $e'_p \prec e_p$, and $T(e'_q)[0] \leq T(e'_p)[0]$.*

We make use of Definition 5.1 as follows: Recall that in our consensus protocol, each process alternates global reads (i.e., invocations of Last) and append operations. Therefore, if p operates as prescribed by the protocol, then between e'_p and e_p as above, there must be a Last at p. By Corollary 4.1, this implies that p's Last before e_p observed the value appended by e_q. We now use this definition as a foundation for justification of appended values.

Definition 5.2 *Let O be a set of* append *operations generated in some execution of the system. A* serial execution *of O is a linear order on the elements of O that extends the per-client linear orders.*

Definition 5.3 *Let e be the i-th τ_j.*append*, with timestamp $T(e) = t$. The* justification set *for e is the set consisting of the k-th $\tau_{j'}$.*append *operations for all $1 \leq j' \leq n$ and all $1 \leq k \leq t[j']$.*

Definition 5.4 *Let e be the i-th τ_j.*append*. e is* justified *if its justification set O contains all operations that e definitely reflects and if there is a serial execution of O followed by e that is consistent with correct execution by p_j. A* justified value *is one appended in a justified* append.

We note that by the properties of timed append-only arrays, justification is well defined, i.e., it does not depend on the reader of the appended value. Furthermore, all correct clients' appends are justified, and if a client executes an unjustified append, then all of its future appends are also unjustified.

5.2. The Coin() operation

According to the protocol of Figure 4, when a leader repeatedly observes leader disagreement, it sets its preferred value to the output of a Coin() operation before moving to the next round. At a correct client, this Coin() operation (shown in Figure 5) returns a value taken from among the values that have been appended by all clients in the protocol. More precisely, the Coin() operation works by reading the value in the first element of each client's array (which, by definition, is justified if it is of the form $\langle 0, v' \rangle$ for some v'), and if this value exists, adding this value to a *view* of the values in the system. We say that the view so computed is *the view of the client in round r*. The Coin() operation returns an element of this view.

```
(1)  subroutine Coin()
(2)      view ← ∅
(3)      ∀j ∈ {1, ..., n} : ⟨v, t⟩ ← τ_j.read(1)
(4)                             if (⟨v, t⟩ ≠ ⊥ and is justified)
(5)                                 view ← view ∪ {v' : v = ⟨0, v'⟩}
(6)      k ← (flip() mod |view|) + 1
(7)      return k-th largest element of view
```

Figure 5. The Coin() operation

In Figure 5, the element returned from the view is selected by a flip operation that returns a nonnegative integer. Correctness of the protocol requires nothing more of flip, though flip is very important to the running time of the protocol, as we show in Section 7.

6. The flip protocol

Before analyzing the running time of our protocol, it is necessary to detail the implementation of the flip operation. This operation is implemented by a distributed protocol that returns the same value to every correct client that invokes it in round r, i.e., the flip value for round r is unique. In addition, the round-r flip value cannot be predicted by any client until some client completes the flip protocol for that round. Intuitively, in the flip protocol the servers generate a deterministic digital signature (such as an RSA signature [16]) on a string that includes the round number in which the flip protocol is invoked. By definition, digital signatures are unpredictable to those not knowing the key to generate them.

The signature generation process must ensure that faulty servers cannot compute flip values ahead of time. This is achieved by employing a *threshold signature scheme* to generate a signature. Informally, a (k, m)-threshold signature scheme is a method of generating a public key and m *shares* of the corresponding private key in such a way that for any message w, each share can be used to produce a *partial result* from w, where any k of these partial results can be

combined into the private key signature for w. Moreover, knowledge of k shares should be necessary to sign w, in the sense that without the private key it should be computationally infeasible to (i) create the signature for w without k partial results for w, (ii) compute a partial result for w without the corresponding share, or (iii) compute a share or the private key without k other shares. Our replication technique does not rely on any particular threshold signature scheme, provided that it is deterministic; the literature includes such schemes (e.g., [5, 6]).

We implement the flip protocol as follows. At service initialization time, a (k, m)-threshold signature scheme, with $k = b+1$ and m equal to the number of servers, is used to generate a public key and one share of the private key for each server. Each server's share is known only to itself; the corresponding public key is assumed to be available to all clients. The flip protocol for round r then proceeds simply as follows: the client executes a Q-RPC to obtain partial results from a quorum of servers for the "message" r, and combines them to form a valid signature for r. It returns this value, interpreted as a nonnegative integer.

It is worth reviewing several properties of the flip protocol that are necessary for the results of Section 7. First, due to the properties of a threshold signature scheme, the flip value for round r is known nowhere prior to some client completing the protocol for that round. Second, if we view the flip protocol as producing a result that is a sampling from the space of integers up to some large (i.e., much larger than |view|; see Section 5.2) bound, it is reasonable to assume flip samples uniformly at random from this space.[1] Third, because the flip protocol produces a digital signature for which all parties are assumed to have the verifying public key, any value claimed to be produced by the flip protocol for round r can be immediately verified. Fourth, because the threshold signature scheme is deterministic, the flip protocol returns the same value to any correct client that invokes it in round r. This does *not* imply that the Coin() operation returns the same value to correct processes that invoke it in round r, because each client may have a different view in round r (see Section 5.2). However, when all correct processes invoking the Coin() operation in round r have the same view, the Coin() operation will indeed return the same value everywhere.

7. Running time

One of the motivations guiding our design of a consensus object was to allow any single (correct) client to access our consensus object solo and obtain the consensus decision within a finite number of steps. In fact, in such a case, a solo client will obtain the consensus value within a small number of steps, specifically within four append and three Last operations. Even when multiple clients participate simultaneously, if a leader emerges quickly then every client may terminate after engaging in only a small number of protocol rounds and no coin-flips. We now proceed to describe the expected running time of our algorithm more generally.

Typically, one hopes that in the common case clients fail only benignly and do not exhibit malicious behavior. With the algorithm as described so far, we can prove that in this case, a client will complete the protocol in an expected $O(c^4 n)$ operations on timed append-only arrays (even in the face of up to the threshold b of Byzantine server failures), where $c \leq n$ is the actual number of clients that append values to their arrays before any correct process decides. The strategy used for proving this result is to show that in only c^2 rounds can Coin() operations return different values to different clients. Moreover, in each round r in which the Coin() operation returns the same value to all clients that invoke it, there is a constant probability that the value returned by the Coin() operation is the same as the first value appended in round r. When this happens, the algorithm will quickly terminate. The result is an expected $O(c^2)$ rounds in which each client executes $O(c^2 n)$ array operations, yielding a total of $O(c^4 n)$ operations.

The story is different for the worst-case running time for this algorithm in case of Byzantine client failures. In this case, the algorithm no longer terminates with probability one. The reason for this is twofold: First, there is nothing to prevent faulty clients from invoking the flip protocol for any round r far in advance, effectively rendering these flips predictable to faulty clients. By carefully controlling the scheduling of operations in the protocol, they can use this advance knowledge of flip results to prolong the protocol indefinitely. Second, even if flip values were withheld from clients for long enough, a faulty client might repeatedly use a different view in its Coin() operation than correct clients, thereby resulting in a different coin value than correct clients.

In order to prove termination in the general case, we are thus forced to make some modifications to the protocol. First, to prevent prematurely revealing flip values to faulty clients, we stipulate the following:

Stipulation 7.1 *A correct server does not respond to a client invoking the* flip *protocol for round r unless that client has executed* two *justified* append *operations in round r.*

Second, we force each client to explicitly append the value of view used in a Coin() operation, and the (verifiable) result of the flip operation, to detect a faulty client that attempts to report a different result from its Coin() operation:

[1] This appears to be a reasonable assumption for threshold signature schemes that generate RSA signatures. If this property is not realistic for a threshold signature scheme of choice, then passing the signature through a suitable cryptographic hash function (e.g., [17]) should adequately simulate the selection of a number uniformly at random from the space of all hash values.

Stipulation 7.2 *The Coin() operation returns, in addition to the selected value, the result of the* flip *operation and the value of* view *computed in the Coin() operation; the client* appends *this* flip *value and the* view *in the same* append *operation as the coin value (i.e., in its first* append *of the next round).*

Though seemingly minor additional stipulations, the first of these substantially increases server involvement in the protocol, in terms of the amount of protocol logic that must be server-resident and the message traffic sent to servers. This is due to the fact that each server is required to test for justification of append operations (which we have not required until now) prior to participating in a flip protocol. In order to make this test as efficient as possible for servers, each client can first forward copies of all previous commit and value messages (i.e., sets $C_{j,v,t}$ in Section 4) to each server that it contacts in the flip protocol (see Section 6), so that the server can update its local arrays and then restrict its attention to its own local arrays to determine justifiability of the client's append operations in that round.

With Stipulations 7.1 and 7.2 in place, we can prove the following:

Theorem 7.1 *Each correct client decides in expected* $O(c^4 n)$ *array operations.*

We emphasize that in practice, however, it may be desirable to omit Stipulations 7.1 and 7.2 and settle for a protocol that terminates with probability one in the case of benign-failures only. Though in theory the algorithm without these stipulations could be extended arbitrarily by faulty clients, in practice this would require substantial control over system scheduling by faulty clients.

8. Conclusion

A consensus object in a Byzantine failure-prone environment is a powerful abstraction, allowing individual clients to obtain a consensus value without waiting for other clients to invoke the object. We described an implementation of randomized consensus objects supported by a set of persistent servers, that can survive arbitrary failures of up to a threshold number of the servers and any number of clients accessing them. Due to the quorum-based replication techniques underpinning our implementation [13, 14], we expect that our protocol can scale to very large numbers of servers and clients. Several of the enabling mechanisms we have developed in our protocol are of general value in themselves: The timed append-only arrays can be used in other protocols to support non-malleable communication among clients when Byzantine failures are a concern; and the distributed coin-flipping technique of Section 6 can be useful in other randomized protocols.

References

[1] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of algorithms*, 11:441–461, 1990.

[2] H. Attiya, A. Bar-Noy and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.

[3] J. K. Bennet, J. B. Carter and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.

[4] B. Chor and C. Dwork. Randomization in Byzantine agreement. In *Advances in Computing Research, Randomness in Computation*, volume 5, JAI Press, edited by S. Micali, pages 443–497, 1989.

[5] Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures. In J. Feigenbaum, editor, *Advances in Cryptology—CRYPTO '91 Proceedings* (Lecture Notes in Computer Science 576), pages 457–469. Springer-Verlag, 1992.

[6] A. De Santis, Y. Desmedt, Y. Frankel and M. Yung. How to share a function securely. In *Proceedings of the 26th ACM Symposium on Theory of Computing*, pages 522–533, May 1994.

[7] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory* IT-22(6):644–654, November 1976.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 32(2):374–382, April 1985.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7):558–565, July 1978.

[10] L. Lamport. On interprocess communication (part II: algorithms). *Distributed Computing* 1:86–101, 1986.

[11] L. Lamport, R. Shostak and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4(3):382–401, July 1982.

[12] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[13] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing* 11(4), to appear. Preliminary version appears in *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 569–578, May 1997.

[14] D. Malkhi, M. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 249–257, August 1997.

[15] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.

[16] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21(2):120–126, February 1978.

[17] FIPS 180-1, Secure hash standard. Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, April 17, 1995.