An Architecture for Survivable Coordination in Large Distributed Systems

Dahlia Malkhi and Michael K. Reiter

Abstract—Coordination among processes in a distributed system can be rendered very complex in a large-scale system where messages may be delayed or lost and when processes may participate only transiently or behave arbitrarily, e.g., after suffering a security breach. In this paper, we propose a scalable architecture to support coordination in such extreme conditions. Our architecture consists of a collection of persistent data servers that implement simple shared data abstractions for clients, without trusting the clients or even the servers themselves. We show that, by interacting with these untrusted servers, clients can solve distributed consensus, a powerful and fundamental coordination primitive. Our architecture is very practical and we describe the implementation of its main components in a system called Fleet.

Index Terms—Distributed systems, scalability, survivability, quorums, Byzantine failures, consensus.

1 Introduction

In this paper, we propose a system architecture that supports efficient and scalable coordination among distributed clients. Our architecture strives to support coordination in perhaps the most difficult circumstances possible: We consider a widespread distributed system where message delays and losses are unpredictable and where clients may be transient, may need to take actions at different times (and may not be available at others), and may even exhibit arbitrary behavior, e.g., due to being corrupted by an attacker. To support coordination in this setting, we postulate a collection of persistent and generic object servers with which clients can interact (see Fig. 1), but that, like clients, may fail arbitrarily. We are developing a software system, called Fleet, that implements these servers and protocols for clients to use them to emulate shared data abstractions (e.g., shared variables and locks).

The properties that distinguish Fleet from other systems that provide shared data services are its *scalability* and its *survivability*: Fleet is designed to scale to hundreds of servers spread across a wide area and to be capable of serving thousands of clients at a time. Moreover, Fleet can survive the arbitrarily malicious corruption of up to a threshold of its servers and any number of clients while still providing useful services.

The Fleet architecture, and our coordination protocols using it, are motivated by large-scale distributed applications that have a need for shared state with intrinsic survivability and security requirements. These applications include, for example:

1. **Public key infrastructures**: Common to many proposals for public key infrastructures (PKIs) are on-line services that support critical functions. These functions may include certificate-generating services that create certificates (i.e., bindings associating attributes to public keys) as in [24], [35], revocation services that enable a client to promptly invalidate her certificate as in [12], [20], and directory services that enable a client to locate the most up-to-date certificate for a name or key, such as X.509 directories [15]. A PKI is a prime example of a system that may need both to survive hostile attempts to penetrate it due to the security requirements it embodies and to scale to worldwide proportions.
2. **Robust publishing and dissemination**: The Eternity service [2] is a proposed service that would enable a client to publish a document so that anyone can retrieve it, but so that nobody—even the author or an adversary with the means to mount a military strike against the service—could eliminate the document from existence or otherwise deny access to it. Such a service will inherently require massive replication over a wide area that can survive attempts to corrupt the data it holds. The Eternity service is one (ambitious) example of a broader class of robust publishing and dissemination services that Fleet is designed to support.
3. **National voting systems**: The AT&T Secure Systems Research Department (of which we were members in 1998) was tasked in 1998 with designing a fully electronic national election system for Costa Rica. Among the goals of this system was to ensure that each voter identifier can be used to cast a vote only once. Achieving this requires "locking" each voter identifier when it is used to cast a vote, thereby promptly precluding its use at any one of the more than

- D. Malkhi is with the School of Computer Science and Engineering, Hebrew University of Jerusalem, Israel. E-mail: dalia@cs.huji.ac.il.
- M.K. Reiter is with the Secure Systems Research Department, Bell Laboratories, Lucent Technologies, Murray Hill, NJ 07974.
- E-mail:reiter@research.bell-labs.com.

Manuscript accepted 1 Sept. 1999.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 110383.

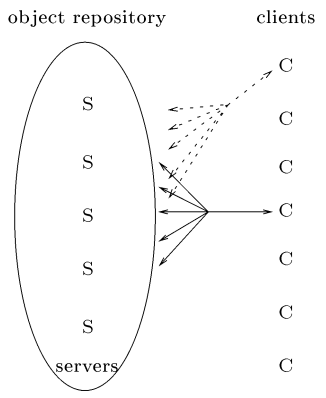


Fig. 1. Persistent servers support replicated objects.

1,000 other voting stations in the country. This locking is an example of distributed coordination that should survive tampering by parties trying to sway the election results.

Fleet is based on a class of techniques that we have recently developed for building survivable systems that must scale. At the foundation of Fleet are novel *quorum* constructions that enable clients to complete operations on shared data objects after interacting with only a typically small subset (quorum) of servers, with no centralized locking or management and with no server-to-server interaction. Quorums can be surprisingly small—e.g., comprised of only $O(\sqrt{n})$ out of a total of n servers—and, thus, client access protocols can efficiently scale to hundreds and possibly even thousands of servers. Quorum systems can be constructed to tolerate failures ranging from benign to fully arbitrary [27], [32], [30] and, for either type of failure, can provide either strict consistency guarantees or only probabilistic ones [31]. Fleet allows clients to dynamically tune the quorums they use for their application needs.

The goal of this paper is to give an overview of this architecture with an eye toward how it can be used to support client coordination. We describe various shared data elements emulated by servers that constitute powerful tools for client interaction in the most challenging settings. In particular, we present an access protocol by which clients can interact with these servers to emulate a shared *consensus* object. This 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.

In our consensus protocol, clients communicate directly only with servers, performing operations on primitive objects implemented at the servers. One of the contributions of this paper is the design of a new object type, called *timed append-only array* (TAOA), which servers emulate in order to facilitate consensus among clients. TAOAs are shared objects to which clients can append values and from which they can read. In an append operation, a client irreversibly writes a value at the next available slot in an array. The servers include a timestamp with an appended value which is returned in a read operation together with the value. These timestamps serve to capture the partial ordering of appends on all the arrays in the system. The value and timestamp are both impossible to modify or erase, even by the original writer. Intuitively, TAOAs support nonmalleable communication among the clients using shared objects in that, once a client appends a value to a TAOA, it is prevented from modifying it or from denying the relative ordering of appends captured by its timestamp. At the same time, TAOAs are simple enough to implement in our setting, involving a few message exchanges between clients and quorums of servers.

Using TAOAs, 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 [11]), regardless of how many other clients simultaneously engage the object. Moreover, the consensus object retains its correctness despite even the malicious behavior of any number of clients and a limited number of servers and is, thus, survivable in a Byzantine environment.

Our consensus object exemplifies many of the strengths of the Fleet architecture. It is implemented using data sharing primitives that are emulated by the servers using no server-to-server or client-to-client communication. Hence, for example, a client that proposes a value alone obtains the consensus value with only a few message exchanges between the client and a fraction (quorum) of the servers. Moreover, some of the mechanisms supporting the consensus protocol are of general value, e.g., TAOAs provide a form of nonmalleable communication among arbitrarily faulty clients and servers. A distributed coin-flipping technique used in our protocol can be useful in other randomized protocols as well.

The remainder of this paper is structured as follows: We provide a comparison to prior work in Section 2. We give an overview of the Fleet architecture in Section 3. We then focus on the design of TAOAs in our architecture in Section 4, which are employed in Section 5 to emulate a consensus object. Section 6 concludes.

2 Comparison to Prior Work

The foremost approach for building a survivable service today is *state machine replication* (SMR) [37], in which every (available) server receives, processes, and responds to every client request; some examples of systems implementing this approach are [38], [34], [13], [16], [6]. Because every server must reliably receive every request, this approach generally does not scale well. By employing our own advances to quorum systems, specifically those that mask Byzantine server failures, our approach allows each operation to complete at only a fraction of the servers. This reduces the load inflicted on each individual server and provides better scalability and availability than SMR.

More generally, Fleet serves as an interesting case study in data replication using quorum systems, which have seldom been deployed in practice despite over 20 years of quorum research (e.g., [14], [40]; see [25] for an overview of quorum systems). The main reason that quorums have not been utilized in practice is that most data replication systems require a client's read operation to access only a single replica (and, when possible, a local one) for best read performance, which in turn requires that write operations be delivered to all replicas. This approach has been taken, e.g., in group communication systems (see [33]), distributed transactional systems (e.g., Argus [22] and Thor [23]), and shared-memory emulation systems (e.g., Ivy [21] and Munin [5]). However, accessing only a single replica precludes any ability to mask arbitrary server failures and, therefore, is unsuitable for our survivability goals. Moreover, as recently discussed in [41], the communication overhead of accessing multiple servers has been steadily decreasing with improvements in network technology and, thus, in some cases, these costs may no longer outweigh the benefits of reduced overall server load that quorums can offer.

One of the primary focuses of this paper is the enhanced coordination services that our architecture can support, in particular, consensus objects. Many systems support only shared data with primitive read/write operations on it (e.g., Ivy [21], Munin [5] and Vault [13]). Others, primarily transactional systems (e.g., Thor [23] and [6]), allow compound operations on data, but these operations cannot be guaranteed to terminate in settings such as ours. All of our protocols are guaranteed to make progress so long as some quorum in the system remains available. Specifically, in our consensus protocol, which is the strongest coordination object we provide, we employ randomization to guarantee that operations terminate with probability one.

Our goals also render existing consensus protocols inadequate for coordination in our setting. 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 [3]. In the message passing model, typically, a threshold of (correct) processes need to simultaneously cooperate to achieve agreement.

The consensus implementation described here 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. This approach is more suited to our goals than executing a standard message-passing consensus protocol either among the servers (where the decision value is made available to clients) or the clients themselves. Indeed, implementing a consensus object using an inter-server protocol complicates server design and mandates server-to-server communication, which would hurt the scalability of Fleet. And, an interclient consensus protocol would require the simultaneous participation of a high percentage of the client population to converge on a consensus value.

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 [11]. Numerous message-passing protocols have employed randomization to guarantee finite termination with probability one; see [7] for a survey. Aspnes and Herlihy [3] 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 [3], 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 Fleet Overview

In this section, we give an overview of the Fleet architecture. Fleet is based on a design similar to a rudimentary prototype system called Phalanx that we developed to support an electronic voting application [28]. Fleet shares no code with Phalanx, however, and substantially generalizes and extends that system with additional capabilities, e.g., for intrusion detection [1] and dissemination of updates [26]. These additional capabilities will not be our focus here.

As discussed in Section 1, we presume a system consisting of possibly large numbers of *servers* and *clients* that need not necessarily be distinct. Our system admits the possibility that a client or server can *fail*, in which case, it may deviate from its specification arbitrarily (Byzantine failures), including collaborating with other faulty clients and servers. A client or server that is not faulty (i.e., conforms to its specification) is *correct*.

There is a parameter b that characterizes the fault tolerance of the system, in that we assume that at most b servers are simultaneously faulty (though any number of clients can fail). In a long-lived system, there might be many transient server failures over a long period of time that eventually exceed any resilience threshold b. Therefore, to tolerate more than a cumulative number of b failures over time, we assume that a server that is detected as faulty is eventually recovered. Informally, a server is "recovered" when its state becomes one that could have resulted from executing every update operation in which the server participated, as well as possibly some additional update operations. In practice, there are various ways to recover servers. For example, a benign failure can be immediately recovered if the server keeps its local copies of objects in nonvolatile storage. In some cases, a "real" Byzantine failure can be recovered in practice by resetting the server

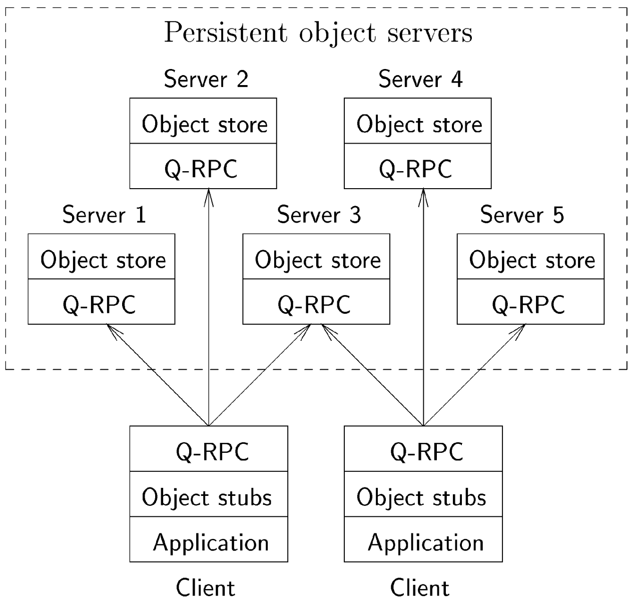


Fig. 2. Fleet system architecture.

to its initial state and waiting until it receives an update to each object it has in store locally (so its state is more recent than when the server failed).

3.1 Quorum Systems

As mentioned in Section 1, our protocols leverage the power of *quorum systems* to make operations as efficient as possible. A quorum system $\mathcal{Q}$ is a set of subsets of servers with the property that, for any $Q_1, Q_2 \in \mathcal{Q}$, $Q_1 \cap Q_2 \neq \emptyset$. Intuitively, this property can be used to ensure that consistency is preserved across multiple operations performed at different quorums. For example, supposing only benign failures for the moment, if a client reads a variable at a quorum of servers, then, because there is a server in that quorum that also received the last-written value of the variable, the client will be sure to obtain it. Of course, in our setting, simply requiring a nonempty intersection between two quorums may not suffice since all servers in that intersection may be "truly Byzantine" faulty. Therefore, the Fleet system makes use of several variations of quorum systems that are better suited to our environment.

One example of a quorum system, that is used in Fleet and that will be useful in subsequent sections as well, is a *b-masking quorum system* [27]. A b-masking quorum system $\mathcal{Q}$ is a set of subsets (quorums) of servers such that 1) for any $Q_1, Q_2 \in \mathcal{Q}$, $|Q_1 \cap Q_2| \geq 2b + 1$ and 2) 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, 1) enables clients to infer correct replies from the contacted quorum, and 2) ensures that a client can always contact a full quorum [27].

The architecture of Fleet is multilayered, as depicted in Fig. 2, the bottom-most layer being a client-server protocol called Q-RPC. Given a quorum system, 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 calling routine 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 calling routine to issue a query to a partial quorum to complete a previous Q-RPC in which faulty servers returned useless (e.g., syntactically incorrect) values. A companion paper [1] discusses some of the techniques we use in Fleet for mining server responses in order to detect failures. For the purposes of this paper, however, we omit these interfaces and techniques 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.

3.2 Data Objects

As depicted in Fig. 2, Fleet servers implement an object store that is accessed via Q-RPC by object stubs residing at clients. An application programmer using Fleet need not be aware of the underlying replication and is supplied with an emulation of a persistent, shared data store. Through the consistency mechanisms built into our protocols, the application has the illusion that every data item has a single copy. These shared data elements are a powerful tool for building higher level applications in the most challenging settings. In [28], we demonstrate one such application, namely an electronic voting system. Some of the shared data types supported by the Fleet architecture are listed below.

1. **Shared storage**: The most simple type of object that facilitates information sharing in Fleet is a shared variable. Shared variable abstractions support read and write operations by clients. For most classes of failure assumptions, we can guarantee *atomic* updates [18] and, under the most severe failure scenarios—variable readers, writers, and (a limited number) of servers are all potentially fault—then we provide only *safe* semantics [18]. The protocols for implementing these variable semantics are presented in [28]. As a practical matter, we note that shared variables can be used to store arbitrary contents, e.g., files and directories.
2. **At-most-one mutual exclusion**: In addition to the need for shared state, many applications need mechanisms to coordinate updates to that state or to otherwise enforce mutual exclusion for other operations. One type of exclusion Fleet provides is an at-most-one mutual exclusion object. This object supports a *contend* operation, which succeeds in at most one among all contending clients. If a client is alone in contending for mutual exclusion, it is guaranteed to succeed, though no client might succeed if multiple clients contend for it. This object is useful in some applications where one may need to achieve exclusive access to certain resources, but

contention for those resources may a priori be unlikely or indicative of foul play. In particular, one of the goals in an electronic election system may be to ensure that each voter identifier can be used to cast a vote only once. Achieving this may require enforcing mutual exclusion on attempts to cast a vote for each voter identifier, thereby precluding its use at multiple voting stations. Contention for a voter identifier indicates an attempt to use the voter identifier multiple times, in which case, all voters can be (and probably should be) delayed until this contention is resolved.

3. **Consensus objects**: A consensus object is a shared object to which a client can *propose* a value and receive a single value in return. The consensus object returns the same value to each client and the returned value is one proposed by some client. Consensus is useful for achieving distributed coordination in a variety of scenarios. For example, consensus objects may be used to provide an *exactly-one* mutual exclusion, i.e., an object guaranteeing that some client among contending ones succeeds in acquiring it. One of the main contributions of this paper is an algorithm for emulating a consensus object supporting potentially arbitrarily faulty clients. This is the most powerful coordination object that we have designed for the Fleet architecture, and is the topic of discussion in subsequent sections of this paper.

3.3 Scale

The scalability that we believe Fleet can achieve is based on two factors: efficient protocol design that enables Fleet to scale well as the number of clients grows and the use of quorum systems that enable Fleet to scale well as the number of servers grows. In this section, we briefly consider these two factors.

In our design, clients need never communicate with one another nor do servers communicate among themselves and, thus, interaction is limited to occur between clients and (quorums of) servers. We have designed the objects supported by servers, such as the shared variables of [28] and the objects described in Section 4, to ensure that servers process a small constant number of messages and perform a small constant number of computations (in particular, digital signatures and verifications) per client operation. Thus, growth in the number of clients should degrade performance of client operations no worse than linearly. And, with a proper choice of quorum system, this degradation can be far smaller than linear. Our emulation of a consensus object described in Section 5 does not scale quite as elegantly because the amount of work a client must do in this emulation is a function of the number of clients that have proposed values for the consensus object. However, we expect that, in the applications we envision, consensus objects will seldom be contended for by more than a handful of clients at any time.

Scalability with growth in the number of servers is primarily dictated by the quorum system used, as quorum systems exist with a wide array of properties [27], [30], [32], [31]. For example, there are Byzantine constructions that have quorum sizes as small as $O(\sqrt{bn})$. Moreover, probabilistic constructions exist with such quorum sizes that simultaneously have outstanding availability. These latter constructions admit a certain well-defined probability of inconsistency in any given operation.

The best quorum system to use can differ from protocol to protocol. Thus, we are constructing Fleet to be flexible as to the quorums it uses to maintain objects and to allow even switching between quorum systems dynamically at run time. In particular, different Fleet objects can be maintained simultaneously using different quorum systems.

For particular applications, it may be possible to further tune the quorum system used to enhance the performance of our protocols. For example, if it is known that variable reads far outnumber variable writes, then it should be possible to employ a quorum construction with distinguished read quorums and write quorums that optimize reads at the cost of more expensive writes.

4 Timed Append-Only Arrays

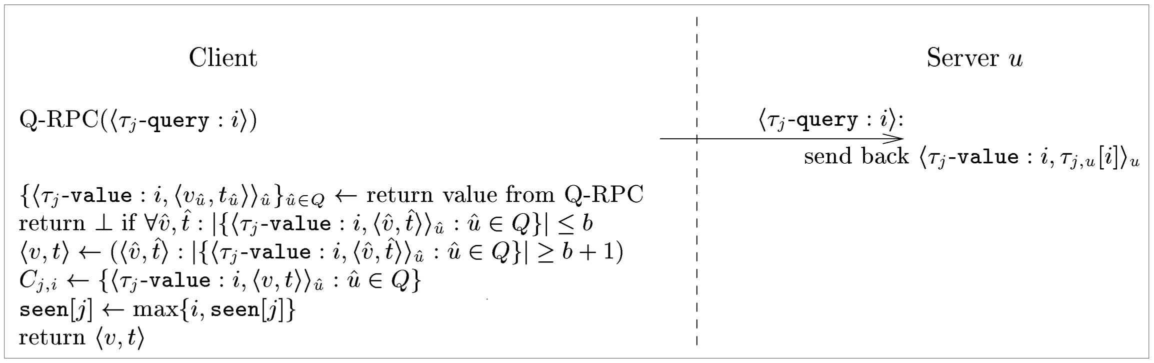
In this section, we detail an example data abstraction that is supported by our architecture, namely a *timed append-only array* (TAOA). TAOAs enable clients to append values, but not to delete or modify previously appended values. In addition, each appended value is labeled with a logical time at which it was appended. As will be shown in Section 5, timed append-only arrays are strong enough to enable (randomized) consensus among clients and, at the same time, are simple enough to implement in an asynchronous environment, with no server-to-server communication and simple server-resident logic. Intuitively, these objects implement nonmalleable communication among clients, because faulty 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.

We denote the clients by $p_1, \dots, p_n$ or just $p, q, \dots$ when subscripts are unnecessary. A TAOA 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 τ_j in a sequential order.
- **Write-Once**: Values appended to τ_j are never modified or erased.
- **Timestamp**: The ith element appended to τ_j is timestamped with a vector t that satisfies the following properties for each $1 \leq k \leq n$. First, if $t[k] > 0$, then the $t[k]$th append on τ_k completed before the ith append to τ_j completed. Second, if the ℓth append to τ_k completed before the $(i-1)$th append to τ_j began, then $t[k] \geq \ell - 1$.

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.

Our implementation of TAOAs requires the use of a b-masking quorum system as described in Section 3. And, like many of the Fleet protocols, our implementation of timed



```
Client                                                        Server $u$

Q-RPC($\langle \tau_j$-query : $i \rangle$)          ──────→   $\langle \tau_j$-query : $i \rangle$:
                                                                  send back $\langle \tau_j$-value : $i, \tau_{j,u}[i] \rangle_u$

$\{\langle \tau_j$-value : $i, \langle v_{\hat{u}}, t_{\hat{u}} \rangle \rangle_{\hat{u}}\}_{\hat{u} \in Q} \leftarrow$ return value from Q-RPC
return $\perp$ if $\forall \hat{v}, \hat{t} : |\{\langle \tau_j$-value : $i, \langle \hat{v}, \hat{t} \rangle \rangle_{\hat{u}} : \hat{u} \in Q\}| \le b$
$\langle v, t \rangle \leftarrow (\langle \hat{v}, \hat{t} \rangle : |\{\langle \tau_j$-value : $i, \langle \hat{v}, \hat{t} \rangle \rangle_{\hat{u}} : \hat{u} \in Q\}| \ge b + 1)$
$C_{j,i} \leftarrow \{\langle \tau_j$-value : $i, \langle v, t \rangle \rangle_{\hat{u}} : \hat{u} \in Q\}$
seen$[j] \leftarrow \max\{i$, seen$[j]\}$
return $\langle v, t \rangle$
```

Fig. 3. The operation τ_j.read(i).

append-only arrays requires the use of digital signature schemes (e.g., [36]). 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 $\langle m \rangle_u$.

4.1 Implementation

We begin by describing the implementation of TAOAs. Let $\tau_1, \ldots, \tau_n$ denote the TAOAs maintained by the servers. Each TAOA τ_j supports two kinds of operations: Client p_j can append some value v to τ_j by executing τ_j.append(v), and any client can read the ith value in τ_j by executing τ_j.read(i). The τ_j.read(i) operation is the simpler of the two and, so, we discuss it first.

Each TAOA 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 ith element of τ_j is shown in Fig. 3. 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 $\langle \tau_j$-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 ith element of τ_j by setting the jth element of a local array seen to be i (if larger than seen$[j]$) and retains the set $C_{j,i} = \{\langle \tau_j$-value : $i, \langle v, t \rangle \rangle_{\hat{u}} : \hat{u} \in Q\}$ of at least $b + 1$ signed messages for $\langle v, t \rangle$.

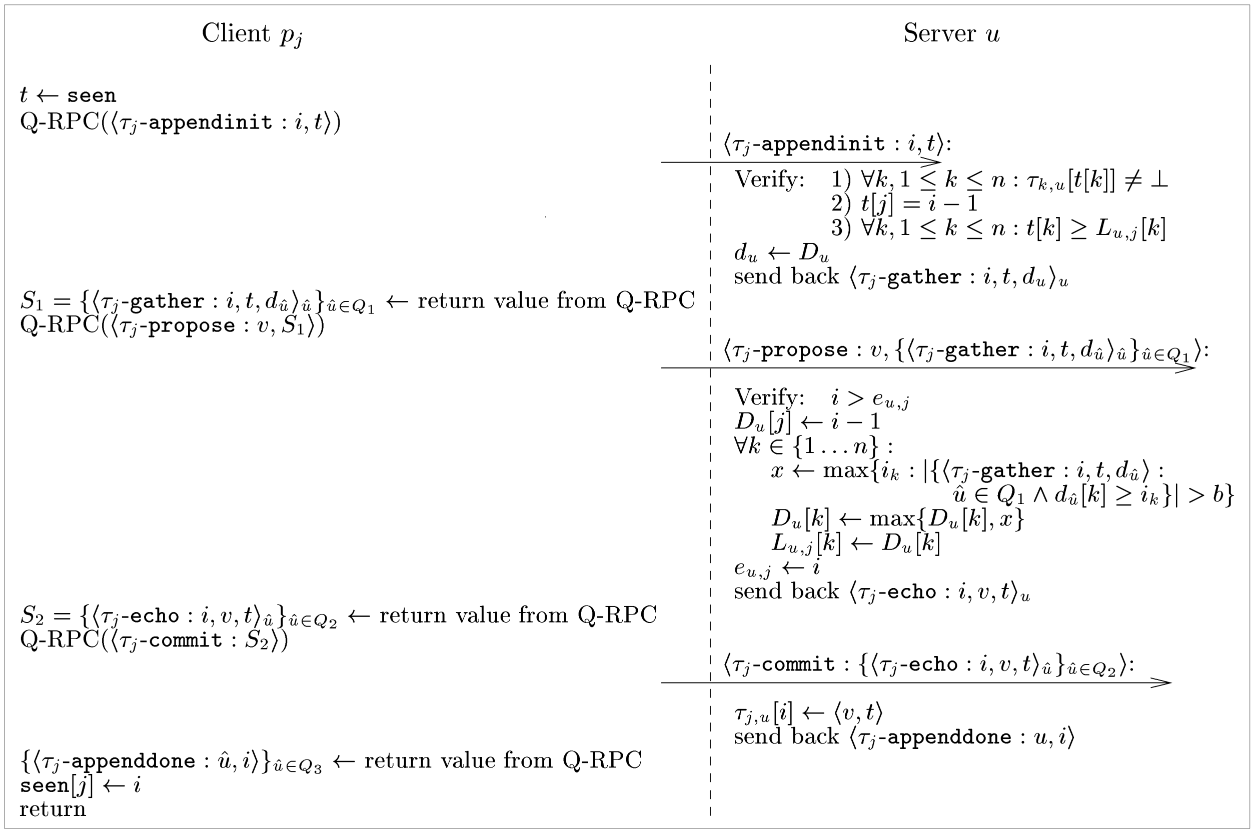
The τ_j.append operation is significantly more involved than the τ_j.read operation; see Fig. 4. Each server u maintains, in addition to array entries, a vector D_u and a per-client vector $L_{u,j}$, both of which are vectors of indices into TAOAs. Intuitively, $D_u[j]$ records the highest index i such that u knows that the ith append to τ_j has completed (a precise definition of "completed" is given in Section 4.2). $L_{u,j}[k]$ simply records the value of $D_u[k]$ when p_j executed its last append at u. Server u uses $L_{u,j}$ as a "lower-bound" on the timestamp of p_js next append, as detailed below. Each server u also maintains an echo-inhibitor $e_{u,k}$, $1 \le k \le n$, per array, initially zero, to guarantee that each server echoes only one ith τ_j.append value.

The ith τ_j.append proceeds in three Q-RPCs. In the first, p_j sends its timestamp vector $t =$ seen to a quorum of servers. Each server verifies several properties. First, each server u requires that, for each $1 \le k \le n$, it holds a value in $\tau_{k,u}[t[k]]$. Since this value may have been written to a quorum not containing u, p_j piggybacks $C_{k,t[k]}$ as needed on its request, which it collected from servers when it read $\tau_k[t[k]]$. (These piggybacked messages are not shown in Fig. 4.) Second, each server u verifies that t reflects any append that was already complete at the previous τ_j.append, that is, that $t[k] \ge L_{u,j}[k]$ for all k. This implies that p_j is expected to read all the arrays for all appended values that were completed by the time p_js previous append completed. Each server u then responds to the client with its current value of D_u, which indicates the appends that u knows to have completed. This completes the first Q-RPC.

In the second Q-RPC, p_j includes the responses to the first Q-RPC, along with the value v that it intends to append. Each server u updates its D_u vector to incorporate the additional knowledge it can glean from the forwarded responses about which appends have completed and, then, copies D_u into $L_{u,j}$; this new value of $L_{u,j}$ will be used in the $(i+1)$th τ_j.append to verify that p_j read from all arrays in the interim. The server u also verifies that $i > e_{u,j}$ and, if so, "echoes" v and t to p_j digitally signed. 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 ith τ_j.append value. Server u then sets $e_{u,j} \leftarrow i$ so that it will never again echo a value for the ith τ_j.append.

Upon the completion of this second Q-RPC, p_j now forwards the digitally signed replies back to the servers via a third Q-RPC. Upon receiving this request, each server u assigns $\tau_{j,u}[i] \leftarrow \langle v, t \rangle$ and acknowledges.

Note that, as mentioned above, between its ith and $(i+1)$th τ_j.append operations, p_j is expected to read all the arrays for all appended values that were completed by the time p_j completed its ith τ_j.append; otherwise, the $(i+1)$th τ_j.append will not complete. This is a technical condition



```
Client $p_j$                                              Server $u$

$t \leftarrow$ seen
Q-RPC($\langle\tau_j$-appendinit : $i, t\rangle$)
                                                          $\langle\tau_j$-appendinit : $i, t\rangle$:
                                                          Verify:  1) $\forall k, 1 \leq k \leq n : \tau_{k,u}[t[k]] \neq \perp$
                                                                   2) $t[j] = i - 1$
                                                                   3) $\forall k, 1 \leq k \leq n : t[k] \geq L_{u,j}[k]$
                                                          $d_u \leftarrow D_u$
                                                          send back $\langle\tau_j$-gather : $i, t, d_u\rangle_u$
$S_1 = \{\langle\tau_j$-gather : $i, t, d_{\hat{u}}\rangle_{\hat{u}}\}_{\hat{u} \in Q_1} \leftarrow$ return value from Q-RPC
Q-RPC($\langle\tau_j$-propose : $v, S_1\rangle$)
                                                          $\langle\tau_j$-propose : $v, \{\langle\tau_j$-gather : $i, t, d_{\hat{u}}\rangle_{\hat{u}}\}_{\hat{u} \in Q_1}\rangle$:
                                                          Verify:  $i > e_{u,j}$
                                                          $D_u[j] \leftarrow i - 1$
                                                          $\forall k \in \{1 \ldots n\}$ :
                                                              $x \leftarrow \max\{i_k : |\{\langle\tau_j$-gather : $i, t, d_{\hat{u}}\rangle$ :
                                                                           $\hat{u} \in Q_1 \wedge d_{\hat{u}}[k] \geq i_k\}| > b\}$
                                                              $D_u[k] \leftarrow \max\{D_u[k], x\}$
                                                              $L_{u,j}[k] \leftarrow D_u[k]$
                                                          $e_{u,j} \leftarrow i$
                                                          send back $\langle\tau_j$-echo : $i, v, t\rangle_u$
$S_2 = \{\langle\tau_j$-echo : $i, v, t\rangle_{\hat{u}}\}_{\hat{u} \in Q_2} \leftarrow$ return value from Q-RPC
Q-RPC($\langle\tau_j$-commit : $S_2\rangle$)
                                                          $\langle\tau_j$-commit : $\{\langle\tau_j$-echo : $i, v, t\rangle_{\hat{u}}\}_{\hat{u} \in Q_2}\rangle$:
                                                          $\tau_{j,u}[i] \leftarrow \langle v, t\rangle$
                                                          send back $\langle\tau_j$-appenddone : $u, i\rangle$
$\{\langle\tau_j$-appenddone : $\hat{u}, i\rangle\}_{\hat{u} \in Q_3} \leftarrow$ return value from Q-RPC
seen$[j] \leftarrow i$
return
```

Fig. 4. The ith invocation of τ_j.append; v is the value appended.

that is in place here due to the fact that, in Section 5, we will use TAOAs to implement consensus and will need this condition to prove correctness. Thus, it is not entirely accurate to present append and read operations as independent operations, since the completion of an append depends on the clients previous execution of read operations. Nevertheless, for tractability of presentation, it is easiest to present TAOAs in this form.

Moreover, these read operations can be replaced by one large "scan" operation that reads all arrays to their last appended elements, using only one exchange with a quorum of servers. This is a practical necessity when implementing TAOAs. However, for ease of the exposition, we describe each read as done separately.

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* completes *when the client returns from the τ_j.read protocol.*

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

Note that, in this definition, we had to capture the duration of appends by faulty clients, as well as correct ones. Therefore, the definition is driven by the effects of an operation on the (correct) servers in the system. Also, note that, by this definition, once the ith τ_j.append has begun, it is possible for it to complete before the protocol in Fig. 4 completes. In particular, if a client $p_{j'}$ reads the value of the ith τ_j.append while that append is going on and then performs a $\tau_{j'}$.append, its own $\tau_{j'}$.append could complete the ith τ_j.append before the protocol for the ith 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 $e \prec e'$, if e completes 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 kth and k'th τ_j.append operations, respectively, by a faulty process p_j (i.e., corresponding to $\langle\tau_j$-appendinit : $k, t\rangle$ and $\langle\tau_j$-appendinit : $k', t'\rangle$ messages from p_j) such that $k < k'$, then we will often use "$e_1 \prec e_2$" as a shorthand to denote this. If e is an append operation of the form $e = \tau_j$.append(v) such that v is stored with timestamp t, we denote this timestamp by $T(e) = t$.

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

Proof. A correct server u assigns $\tau_{j,u}[i] \leftarrow \langle \hat{v},\hat{t}\rangle$ in two different protocols: The ith $\tau_j.\mathtt{append}$ or any $e = \tau_{j'}.\mathtt{append}$ such that $T(e)[j] = i$ (due to the piggybacking of $C_{j,i}$). In the latter case, u sets $\tau_{j,u}[i] \leftarrow \langle \hat{v},\hat{t}\rangle$ only after receiving signed τ_j-value messages (i.e., $C_{j,i}$) that show that some correct server u' has already assigned $\tau_{j,u'}[i] \leftarrow \langle \hat{v},\hat{t}\rangle$. So, the first correct server u to assign $\tau_{j,u}[i] \leftarrow \langle \hat{v},\hat{t}\rangle$ must do so in the ith $\tau_j.\mathtt{append}$ protocol.

Suppose that e_1 returns $\langle v,t\rangle$. The first correct server to execute $\tau_{j,u}[i] \leftarrow \langle v,t\rangle$ did so (in the ith $\tau_j.\mathtt{append}$ protocol) after receiving a message $\langle \tau_j\text{-}\mathtt{commit} : \{\langle \tau_j\text{-}\mathtt{echo} : i,v,t\rangle_{\hat{u}}\}_{\hat{u}\in Q}\rangle$ for some quorum Q. Since any two quorums intersect in some correct server and since any correct server $\hat{u}$ sends at most one message of the form $\langle \tau_j\text{-}\mathtt{echo} : i,*,*\rangle_{\hat{u}}$[1] (due to the management of the echo counter $e_{\hat{u},j}$), no other correct server u' ever assigns $\tau_{j,u'}[i] \leftarrow \langle v',t'\rangle$ where $v' \neq v$ or $t' \neq t$. So, the only value that e_2 could return, other than $\perp$, is $\langle v,t\rangle$. □

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

Proof. The proof is by induction on i. Suppose the lemma holds for the ith $\tau_j.\mathtt{append}$ and let e_1 be the $(i+1)$th $\tau_j.\mathtt{append}$. By the definition of $\prec$, $e_1 \prec e_2$ implies that each correct server u in some quorum assigned $\tau_{j,u}[i+1] \leftarrow \langle v,t\rangle$ for some $\langle v,t\rangle$ before any correct server received $\langle \tau_j\text{-}\mathtt{query} : k\rangle$. There are two cases to consider:

- **Case 1**: If $k = i+1$, then, since quorums overlap in at least $b+1$ correct servers, at least $b+1$ servers return $\langle v,t\rangle$ to the client in their τ_j-value messages. Thus, the read returns $\langle v,t\rangle$.
- **Case 2**: Now, suppose that $1 \le k \le i$. Before any correct server u sends $\langle \tau_j\text{-}\mathtt{gather} : i+1,t,d_u\rangle_u$ (and, thus, before any correct server u' sends $\langle \tau_j\text{-}\mathtt{echo} : i+1,v,t\rangle_{u'}$), u verifies that $\tau_{j,u}[i] \neq \perp$ as shown in Fig. 4. So, since the first correct server u to assign $\tau_{j,u}[i+1] \leftarrow \langle v,t\rangle$ requires $\langle \tau_j\text{-}\mathtt{echo} : i+1,v,t\rangle_{\hat{u}}$ messages from a quorum of servers (see the proof of Lemma 4.1), the fact that the $(i+1)$th $\tau_j.\mathtt{append}$ completed implies that the ith $\tau_j.\mathtt{append}$ previously completed. If we let e_3 denote the ith $\tau_j.\mathtt{append}$, this means that $e_3 \prec e_2$. So, the result follows from the induction hypothesis. □

The next lemma shows an "upper-bound" of values that can be in $T(e)$. The subsequent lemma shows a "lower-bound" of values that must be in $T(e)$.

Lemma 4.3 (Timestamp-UpperBound). *Let* e_1 *be the ith* $\tau_j.\mathtt{append}$*, and let* $e_2 = \tau_{j'}.\mathtt{read}(k)$*, 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* $\perp$.

1. The symbol "$*$" denotes any value, i.e., a "wildcard."

Proof: For the first correct server u to assign $\tau_{j,u}[i] \leftarrow \langle v,t\rangle$ (i.e., $T(e_1) = t$), it must first receive a τ_j-commit message containing τ_j-echo messages from a quorum of servers. Recall that, for this to happen, every correct server $\hat{u}$ in some quorum verified that $\tau_{j',\hat{u}}[t[j']] \neq \perp$ before sending its τ_j-gather message. This implies that the $t[j']$th $\tau_{j'}.\mathtt{append}$ has completed by the time that e_1 completes. So, by Lemma 4.2, e_2 does not return $\perp$. □

Lemma 4.4 (Timestamp-LowerBound). *Let* e_1 *and* e_2 *be* $\tau_j.\mathtt{append}$ *events, and* e_3, e_4 *be* $\tau_{j'}.\mathtt{append}$ *events, such that* e_1 *is the* i_1*th* $\tau_j.\mathtt{append}$ *and* $e_1 \prec e_2 \prec e_3 \prec e_4$*. Then,* $T(e_4)[j] \ge i_1$.

Proof. Since $e_1 \prec e_2$ and e_1 is the i_1th $\tau_j.\mathtt{append}$, then e_2 is the i_2th $\tau_j.\mathtt{append}$, where $i_2 > i_1$. Hence, after e_2 completes, every correct server u in some quorum has $D_u[j] \ge i_2 - 1 \ge i_1$. Before e_3 can complete, $p_{j'}$ must obtain $\tau_{j'}$-gather messages from a quorum of servers which intersect e_2's quorum in at least $b+1$ correct servers. Therefore, when $p_{j'}$ forwards these messages to a quorum Q of servers, every correct server u in Q sets $D_u[j] \ge i_1$ and, thus, $L_{u,j'}[j] \ge i_1$. Then, every correct server $\hat{u}$ in the first quorum Q' accessed in e_4 verifies that $T(e_4)[j] \ge L_{\hat{u},j'}[j]$. Since at least one correct server (in fact, $b+1$ correct) is in $Q \cap Q'$, $p_{j'}$ must send $T(e_4)$ such that $T(e_4)[j] \ge i_1$ in order for e_4 to complete. □

In the case of Lemma 4.4, we say that e_4 *definitely reflects* e_1.

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 TAOAs. 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.

- **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 initial value.

Our protocol employs a round structure and high-level strategy similar to [3], but, otherwise, differs significantly. In our protocol, each client executes a sequence of logical rounds until it reaches a decision. There is one TAOA 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 5.5 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 5.5.

In terms of data structures, each client maintains several local variables: pref, which holds the clients present preferred value; r, which holds the clients 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 $\langle 0, \texttt{pref}\rangle$ 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 Fig. 5. 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 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.

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.

Testing for justification is made possible by the properties of our array timestamps. Recall that, in the write protocol, servers set a "lower bound" on timestamps, guaranteeing that every appended value has a timestamp vector that succeeds all definitely reflected values (see Lemma 4.4).

Definition 5.1. *Let e be the ith τ_j.append, with timestamp $T(e) = t$. The* justification set *for e is the set consisting of the kth $\tau_{j'}$.append operations for all $1 \le j' \le n$ and all $1 \le k \le t[j']$.*

Definition 5.2. *Let e be the ith τ_j.append. e is* justified *if all previous τ_j.appends are justified and if the value appended in e is consistent with correct execution by p_j assuming that p_j observed exactly the appends in its justification set. A* justified value *is one appended in a justified* append.

We note that, by the properties of TAOAs, 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 Fig. 5, 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 Fig. 6) 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.

In Fig. 6, 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,

```
private vars pref, $r$, $\langle r_j, v_j \rangle_{1 \le j \le n}$, leaderVals, leaderRound          ; pref set in round $r-1$

append($\langle r, \texttt{pref} \rangle$)
repeat                                                                                ; main loop
    Last()
    if ($|\texttt{leaderVals}| > 1 \vee \perp \in \texttt{leaderVals}$)
        LeadersDisagree()
    else
        LeadersAgree()
until round $r+1$ enabled

subroutine LeadersAgree()
    pref $\leftarrow (v : \texttt{leaderVals} = \{v\})$
    append($\langle r, \texttt{pref} \rangle$)                                          ; first announcement
    Last()
    if ($\texttt{leaderVals} \neq \{\texttt{pref}\}$)
        return
    append($\langle r, \texttt{pref} \rangle$)                                          ; second announcement
    Last()
    if ($\texttt{leaderVals} \neq \{\texttt{pref}\}$)
        return
    if ($r = \texttt{leaderRound} \wedge \forall j : (v_j \neq \texttt{pref} \Rightarrow r_j \le \texttt{leaderRound} - 2)$)   ; all who disagree are 2 rounds behind
        decide(pref)
    else
        enable round $r+1$

subroutine LeadersDisagree()
    append($\langle r, \perp \rangle$)                                                  ; first announcement
    Last()
    if ($|\texttt{leaderVals}| = 1 \wedge \perp \notin \texttt{leaderVals}$)
        return
    append($\langle r, \perp \rangle$)                                                  ; second announcement
    Last()
    if ($|\texttt{leaderVals}| = 1 \wedge \perp \notin \texttt{leaderVals}$)
        return
    if ($\texttt{leaderRound} = r$)
        pref $\leftarrow$ Coin()
    enable round $r+1$

subroutine Last()
    $\forall j \in \{1, \ldots, n\}$:
        $i_{\max} \leftarrow \max\{i : \tau_j.\texttt{read}(i) \text{ is justified}\}$
        $\langle \langle r_j, v_j \rangle, t_j \rangle \leftarrow \tau_j.\texttt{read}(i_{\max})$          ; read the last value in every array
    leaderRound $\leftarrow \max_{1 \le j \le n}\{r_j\}$
    leaderVals $\leftarrow \{v_j : r_j = \texttt{leaderRound}\}$
```

Fig. 5. Round r of the consensus protocol.

though `flip` is very important to the running time of the protocol, as we show in Section 5.5.

5.3 Correctness

In this section, we prove that the protocol above meets the Agreement and Validity properties given at the beginning of Section 5. We use the following notation: $\texttt{append}_p^r(x)$ is an execution of $\texttt{append}(x)$ by p in round r; $\texttt{append}_p^{r,i}(x)$ is the ith such execution; $\texttt{append}_p^{r,-i}(x)$ is the ith-to-last such execution (in particular, $\texttt{append}_p^{r,-1}(x)$ is the last); and $T(\texttt{append}_p^{r,i}(x))$ is the timestamp associated with $\texttt{append}_p^{r,i}(x)$ (and, similarly, for $T(\texttt{append}_p^{r,-i}(x))$). Let $\texttt{Last}_p^r$ denote an execution of `Last` by client p in round r; $\texttt{Last}_p^{r,i}$ is the ith such execution; and $\texttt{Last}_p^{r,-i}$ is the ith-to-last such execution (in particular, $\texttt{Last}_p^{r,-1}$ is the last).

Definition 5.3. *If* $\texttt{append}(\langle r, v \rangle)$ *is the first justified execution of* $\texttt{append}(\langle r, * \rangle)$ *by* p *in round* r*, then we say that* p initially prefers *v in round r.*

Definition 5.4. *In any execution, a client is* first to execute $\texttt{append}(\langle r, v \rangle)$ *if this* `append` *is justified and its justification set contains no justified operations of the form* $\texttt{append}(\langle r, v \rangle)$.

When a client is first to execute $\texttt{append}(\langle r, v \rangle)$ and also initially prefers v in round r, we say, in short, that a client is first to initially prefer v in round r.

Lemma 5.1. *If all initial preferences in round r are v, then no client (justifiably) executes* $\texttt{append}(\langle r', v' \rangle)$ *($v' \neq v$) or* $\texttt{append}(\langle r', \perp \rangle)$*, where $r' \geq r$.*

Proof. For a contradiction, let p be first to execute $\texttt{append}(\langle r', v' \rangle)$ ($v' \neq v$) or $\texttt{append}(\langle r', \perp \rangle)$, where $r' \geq r$. For this to be justified, there must be an `append` in its justification set in which a leader last appended something other than v. However, since this justification set must also contain p's own last `append` of v in round r or higher and since any other value so far appended by any other client in round r or higher is v (by assumption), all

```
subroutine Coin()
    view ← $\emptyset$
    $\forall j \in \{1, \ldots, n\} : \langle v, t \rangle \leftarrow \tau_j$.read(1)
                    if ($\langle v, t \rangle \neq \perp$ and is justified)
                        view ← view $\cup \{v' : v = \langle 0, v' \rangle\}$
    $k \leftarrow$ (flip() mod |view|) + 1
    return $k$-th largest element of view
```

Fig. 6. The Coin() operation.

leaders agree on v. Thus, v' and $\perp$ are not justified. □

Lemma 5.2. *If all initial preferences in round r are v, then each correct client that enables round $r+1$ decides v by the end of round $r+1$.*

Proof. For a contradiction, let p be the first correct client to not decide in round $r+1$. Since p is the first one, p is a leader when in round $r+1$ and, by Lemma 5.1, p initially prefers v in round $r+1$. Because all clients in round $r+1$ (justifiably) append only v (Lemma 5.1), p executes its LeadersAgree routine to completion. If p cannot decide at the end of this subroutine, then there must be a client that (justifiably) executes $\texttt{append}(\langle r', v' \rangle)$, $v' \neq v$, or $\texttt{append}(\langle r', \perp \rangle)$, where $r' \in \{r, r+1\}$. This contradicts Lemma 5.1. □

Lemma 5.3. *If a correct client decides v in round r, then, prior to deciding, all of its executions of* $\texttt{append}(\langle r, * \rangle)$ *were* $\texttt{append}(\langle r, v \rangle)$.

Proof. Let p be the correct client that decides v in round r. Since p decides, it must appear to be a leader right before deciding (line 20), and, hence, p observes no other clients in any round $r' > r$. Now, suppose for a contradiction that p executed $\texttt{append}(\langle r, \perp \rangle)$ or $\texttt{append}(\langle r, v' \rangle)$ at some point prior to deciding. If p had executed $\texttt{append}(\langle r, \perp \rangle)$ (in LeadersDisagree), then, for the rest of round r, p would have observed the leaders in disagreement because at least one leader, namely itself, last appended $\perp$; this would have caused p to move to round $r+1$, contradicting the assumption that p decides in round r. If p had executed $\texttt{append}(\langle r, v' \rangle)$ prior to deciding, then subsequently, either it observed leader disagreement and a contradiction results as in the previous case or it detected only leader agreement (on v'), which contradicts the assumption that p decides v in round r. □

Lemma 5.4. *If a correct client decides v in round r, then no client (justifiably) initially prefers $v' \neq v$ at round r.*

Proof. Suppose that p decides v at round r. By Lemma 5.3, it appended only v in round r. For a contradiction, let q be first to append $\langle r', v'' \rangle$ for any $r' \geq r$ and $v'' \neq v$. Consequently (by Lemma 5.1), q is also the first to initially prefer some $v' \neq v$ at round r, i.e., q's initial append in round r has a justification set that contains no operations of the form $\texttt{append}(\langle r', v'' \rangle)$, where $r' \geq r$ and $v'' \neq v$. We now look at the last two append operations performed by q in round $r-1$, namely $\texttt{append}_q^{r-1,-2}(\langle r-1, * \rangle)$ and $\texttt{append}_q^{r-1,-1}(\langle r-1, * \rangle)$. If the values appended in these appends were v, then q would have completed round $r-1$ in LeadersAgree with a preferred value of v and then initially preferred v in round r, contrary to our assumption. Thus, q's last two appends in round $r-1$ do not have value v. Since p decides v in round r, p must not read these two appends, i.e., $\texttt{Last}_p^{r,-1} \preceq \texttt{append}_q^{r-1,-2}(\langle r-1, * \rangle)$, and, therefore:

$$\begin{aligned}\texttt{append}_p^{r,-1}(\langle r, v \rangle) &\prec \texttt{Last}_p^{r,-1} \\ &\preceq \texttt{append}_q^{r-1,-2}(\langle r-1, * \rangle) \\ &\prec \texttt{append}_q^{r-1,-1}(\langle r-1, * \rangle).\end{aligned}$$

Thus, $\texttt{append}_p^{r,-1}(\langle r, v \rangle) \prec \texttt{append}_q^{r-1,-1}(\langle r-1, * \rangle)$. Moreover, because

$$\texttt{append}_q^{r-1,-1}(\langle r-1, * \rangle) \prec \texttt{append}_q^{r,1}(\langle r, v' \rangle)$$

and

$$\texttt{append}_p^{r,-2}(\langle r, v \rangle) \prec \texttt{append}_p^{r,-1}(\langle r, v \rangle),$$

we know that

$$\texttt{append}_q^{r,1}(\langle r, v' \rangle)$$

definitely reflects $\texttt{append}_p^{r,-2}(\langle r, v \rangle)$, which means that $\texttt{append}_q^{r,1}(\langle r, v' \rangle)$ is not justified. □

Theorem 5.5 (Agreement). *If any correct client p decides v, then all correct clients decide v.*

Proof. Let q decide v in the lowest decision round r. By Lemma 5.4, no client (justifiably) initially prefers any $v' \neq v$ in round r. By Lemma 5.3, no client decides $v' \neq v$ in round r and, by assumption, no client decides in any round lower than r. By Lemma 5.2, all correct clients decide by round $r+1$ and, by Lemma 5.1, they all decide v. □

Theorem 5.6 (Validity). *If any correct client decides v, then some client had v as its initial value.*

Proof. It suffices to argue that any $\texttt{append}(\langle r, v \rangle)$, $r > 0$, is justified only if some client executed $\texttt{append}(\langle 0, v \rangle)$ as its first operation. So, suppose that no client executes $\texttt{append}(\langle 0, v \rangle)$ as its first operation and that p is first to execute $\texttt{append}(\langle r, v \rangle)$ for some $r > 0$. Since p is first, it did not adopt v as its preferred value by adopting the leaders' value in LeadersAgree. So, it must adopt v as its preferred value as the result of executing a Coin() operation. However, a Coin() operation at a correct process is guaranteed to return a value that some process initially appended. So, $\texttt{append}(\langle r, v \rangle)$ is not justified, a contradiction. □

5.4 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 [36]) 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 1) create the signature for w without k partial results for w, 2) compute a partial result for w without the corresponding share, or 3) 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., [8], [9]).

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 5.5. 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.[2] 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.

2. 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., [39]) should adequately simulate the selection of a number uniformly at random from the space of all hash values.

5.5 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 $O(c^2)$ 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 further stipulations on the protocol. First, to prevent prematurely revealing `flip` values to faulty clients, we stipulate the following:

Stipulation 5.1. *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:

Stipulation 5.2. *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 previous `commit` and `value` messages (i.e., sets $C_{j,i}$ in Section 4) as needed to each server that it contacts in the `flip` protocol (see Section 5.4) 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.

In the remainder of this section, we analyze the (worst-case) expected running time of the protocol with these additional stipulations. However, we emphasize that, in practice, it may be desirable to omit Stipulations 5.1 and 5.2 and settle for a protocol whose termination is guaranteed (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.

Definition 5.7. *A process p* appends a coin value *in round $r+1$ if the value in its initial* `append` *in round $r+1$ is accompanied by a view and the (valid) result of a* `flip` *operation. The value is called p's* coin value for round *r and the view is called p's* `view` for round *r.*

Definition 5.8. *The coin operation for round r is* multivalent *if any two clients have different (justified)* `views` *for round r.*

Note that if the coin operation is not multivalent in round r, then any two coin values appended in round $r+1$ are the same. In this case, we refer to this unique value as simply the coin value for round r.

Lemma 5.5. *Let p be a client that appends coin values in rounds r_1+1 and r_2+1, $r_1 < r_2$, where v is an element of p's* `view` *in round r_1. If q appends coin values in rounds r_3+1 and r_4+1, $r_4 > r_3 > r_2$, then v is an element of q's* `view` *in round r_4.*

Proof. First, we know that q appends $\perp$ twice in round r_3, i.e., $\texttt{append}_q^{r_3,-2}(\langle r_3, \perp\rangle), \texttt{append}_q^{r_3,-1}(\langle r_3, \perp\rangle)$, and then appends some coin value in round r_3+1, $\texttt{append}_q^{r_3+1,1}(\langle r_3+1, *\rangle)$. For brevity, we shall denote these appends in the remainder of the proof without their arguments, i.e., $\texttt{append}_q^{r_3,-2}$, and so on. Likewise, q appends $\perp$ twice in round r_4 before appending a coin value in r_4+1 and p appends $\perp$ twice in rounds r_1, r_2, respectively, before appending coin values in r_1+1, r_2+1. We shall use the shorthand notation for all of these events.

Our goal is to show that $\texttt{append}_q^{r_4+1,1}$ definitely reflects $\texttt{append}_p^{r_1+1,1}$ and, hence, that v must be included in q's view for round r_4. To this end, assume for contradiction that $\texttt{append}_q^{r_3+1,1} \preceq \texttt{append}_p^{r_2,-2}$. Hence,

$$\begin{aligned}\texttt{append}_q^{r_3,-2} &\prec \texttt{append}_q^{r_3,-1}\\ &\prec \texttt{append}_q^{r_3+1,1}\\ &\preceq \texttt{append}_p^{r_2,-2}\\ &\prec \texttt{append}_p^{r_2,-1}\\ &\prec \texttt{append}_p^{r_2+1,1},\end{aligned}$$

and we have that $\texttt{append}_p^{r_2+1,1}$ definitely reflects $\texttt{append}_q^{r_3,-2}$, and hence cannot be a coin value, contrary to the assumption. Therefore, we must have $\texttt{append}_p^{r_2,-2} \prec \texttt{append}_q^{r_3+1,1}$. Since,

$$\texttt{append}_p^{r_1+1,1} \prec \texttt{append}_p^{r_2,-2} \prec \texttt{append}_q^{r_3+1,1} \prec \texttt{append}_q^{r_4+1,1}$$

we have that $\texttt{append}_q^{r_4+1,1}$ definitely reflects $\texttt{append}_p^{r_1+1,1}$. So, any value in p's `view` in round r_1 must be in q's view in round r_4 or else $\texttt{append}_q^{r_4+1,1}$ is not justified (by Stipulation 5.2). □

Lemma 5.6. *The total number of rounds with multivalent coin operations is at most $(c+1)c$.*

Proof. The proof uses a counting argument based on the property established in Lemma 5.5. Let v be any process' initial value. We consider two "phases" for v. In v's first phase, we count rounds where v appears in some process' view up to (and including) the first round in which v has been included in the same process' views twice. In v's second phase, we count rounds where v has appeared in the same process' views for two (or more) rounds, but was not included in some process' view for this round. Suppose that phase one includes k rounds, where $k \geq 2$, i.e., in the kth round that v appears in a view, it previously appeared in a view of the same process. Note that the $k-1$ processes that have included v in their views up to this point must do so forever. According to Lemma 5.5, in v's phase two there can be at most $c-k+1$ rounds. Thus, the number of rounds in v's phases one and two are: $k+(c-k+1)=c+1$. Since there are c initial values, this means that the number of rounds in all initial values' phases is $(c+1)c$. Since any multivalent round is included in some value's phase one or two, this means that there are at most $(c+1)c$ multivalent rounds. □

Lemma 5.7. *Let p be the first client to complete two (justified)* `append` *operations in round r and let p's initial preference in round r be v. If the coin operation for round r is not multivalent and has value v, then the only justified initial preference in round $r+1$ is v.*

Proof. Suppose, for a contradiction, that q is first to initially prefer any $v' \neq v$ in round $r+1$. Because the coin for round r is not multivalent and has value v, q's initial preference of v' in round $r+1$ could be justified only if q completed round r by executing $\texttt{append}_q^r(\langle r, v'\rangle)$ twice in `LeadersAgree`. Because p is the first client to complete two justified append operations in round r, we know that $\texttt{append}_p^{r,2}(\langle r,*\rangle) \prec \texttt{Last}_q^{r,2}$ and, so,

$$\texttt{append}_p^{r,2}(\langle r,*\rangle) \prec \texttt{Last}_q^{r,2} \prec \texttt{append}_q^{r,-1}(\langle r,v'\rangle)$$

So, $\texttt{append}_p^{r,2}(\langle r,*\rangle) \prec \texttt{append}_q^{r,-1}(\langle r,v'\rangle)$. Moreover, because $\texttt{append}_p^{r,1}(\langle r,v\rangle) \prec \texttt{append}_p^{r,2}(\langle r,*\rangle)$ and $\texttt{append}_q^{r,-1}(\langle r,v'\rangle) \prec \texttt{append}_q^{r+1,1}(\langle r+1,v'\rangle)$, it follows that $\texttt{append}_q^{r+1,1}(\langle r+1,v'\rangle)$ definitely reflects $\texttt{append}_p^{r,1}(\langle r,v\rangle)$ and thus is not justified. □

Lemma 5.8. *Each correct client completes the protocol in an expected $O(c^2)$ rounds.*

Proof. Consider any round $r > 1$. If the coin operation for round $r-1$ is not multivalent and (if executed anywhere) has a value equal to the initial preference of the first process to complete two appends in round $r-1$, then, by Lemma 5.7, all the clients initially prefer the same value in round r and, by Lemma 5.2, all clients decide by the end of round $r+1$. By Lemma 5.6, at most $(c+1)c$ rounds are multivalent. For any nonmultivalent round r, since the value of the coin for round r is revealed only after some client, say p, has completed its second justified append for round $r-1$ (by Stipulation 5.1), the probability of the coin value for that round returning p's initial preference for that round is $1/c$. Therefore, all correct clients decide in an expected $O(c^2)$ rounds. □

For the next lemma, we need to introduce some additional terminology. Consider the sequence of invocations of Last at a correct client. Let a *v-sequence* be a sequence of consecutive Last invocations that each assigns $\texttt{leaderVals} = \{v\}$. Similarly, let a *$\perp$-sequence* be a sequence of consecutive invocations that each assigns leaderVals to be a set of size two or more or a set containing $\perp$. Sequences are maximal, i.e., a sequence of one type is never directly followed by a sequence of the same type. Let the *length* of a v sequence or a $\perp$ sequence be the number of invocations in the sequence and let the *round* of the sequence be the value of leaderRound immediately following the last invocation in the sequence.

Lemma 5.9. *Let s_1, s_2, and s_3 be consecutive sequences at a correct client, where s_2 is a v-sequence. Then, the round of s_3 is larger than the round of s_1.*

Proof. Let p be the correct client at which s_1, s_2, and s_3 are executed. Let r be the round of s_1 and let r' be the value of leaderRound immediately following the first invocation of Last in s_2. If $r' > r$, then we are done. Otherwise (i.e., $r' = r$), this implies that some client has executed a justifiable append in round $r+1$ (causing clients in round r to prefer its value as the leaders' value) and in p's next invocation of Last, it will observe this append, i.e., leaderRound will equal some $r'' > r$. Therefore, s_2 (and, transitively, s_3) or s_3 (if s_2 is of length one) has a higher round than s_1. □

Theorem 5.9. *Each correct client decides in expected $O(nc^4)$ array operations.*

Proof. We first argue that each correct client completes each round in an expected $O(c^4)$ append and Last operations. Let p be a correct client executing in round r. First note that a sequence (of any type) of length four causes p to decide or move to round $r+1$. So, in the worst case, the values returned by Last at p in round r consist of alternating v-sequences (for some v) and $\perp$-sequences, where each sequence is of length three or less. By Lemma 5.9, however, the rounds of every fourth such sequence are monotonically increasing. Since by Lemma 5.8 there is an expected $O(c^2)$ number of rounds in the protocol, there is an expected $O(c^2)$ number of (constant-length) sequences in each clients round and, hence, every client completes the protocol in an expected $O(c^4)$ number of append and Last operations.

To complete the proof, we are left with analyzing the complexity of the Last operations. By the previous argument, each array grows to an expected $O(c^4)$ length. Therefore, over the course of an entire protocol run, all Last operations executed by each client require an expected $O(nc^4)$ read operations, presuming that a Last operation does not reread array elements observed in a previous Last operation by the same client. This yields a total expected number of read and append operations of $O(nc^4)$. □

5.6 From Binary Consensus to Multivalued Consensus

The consensus object implementation described and analyzed in this section allows a client to propose any value to the object and, thus, is called *multivalued* in distributed computing parlance. A commonly studied variation on the multivalued consensus problem is the *binary* consensus problem, in which each client is allowed to propose only a single bit (i.e., 0 or 1) to the object. In the case that the consensus protocol of Fig. 5 is used for binary consensus, it is not difficult to verify that there are no multivalent coin operations. This yields an expected running time of only $O(n)$ TAOA operations for binary consensus.

Binary consensus is interesting also because it can be used to construct a multivalued consensus object in our environment. Assume that the initial value of any client is at most ℓ bits in length. We can use a different binary consensus object to agree on each bit v_i of the final decision value v, for a total of ℓ binary consensus objects $o_1, \ldots, o_\ell$. The algorithm is shown in Fig. 7. In this figure, $o_i(0)$ denotes the proposal of 0 to the ith binary consensus object (and similarly for $o_i(1)$), which returns the ith consensus value. A precise specification of this algorithm requires us to restrict the initial append by each process in the ith instance of binary consensus to be consistent with the ith bit of some client's initial ℓ bit value that is not already precluded by $v_1 \ldots v_{i-1}$; otherwise, the append is not justified.

The total expected running time for the multivalued consensus protocol of Fig. 7 is $O(\ell n)$. Thus, this protocol may be advantageous to using that of Fig. 5 for multivalued consensus when c^4 is expected to exceed ℓ.

```
(1)   for $i = 1 \ldots \ell$
(2)       if $v_1 \ldots v_{i-1} 0$ is a prefix of some client's initial value
(3)           $v_i \leftarrow o_i(0)$
(4)       else
(5)           $v_i \leftarrow o_i(1)$
(6)   return $v_1 \ldots v_\ell$
```

Fig. 7. Multivalued consensus from binary consensus.

6 Conclusion

In this paper, we proposed an architecture for the construction of survivable and scalable data repositories and gave an overview of its implementation in the Fleet system. The distinguishing features of Fleet are its implementation of strong data abstractions in a scalable way using untrusted servers and clients. The applications for which we are targeting Fleet include critical components of large scale public-key infrastructures, publishing and dissemination services, and national election systems. We also argued for the scalability of Fleet based on the efficiency of data access protocols and the novel use of quorum systems at the core of Fleet.

A central contribution of this paper is our description of a survivable and scalable consensus object which, in many ways, is the most powerful abstraction that we have designed for the Fleet architecture thus far. Our consensus object is a powerful abstraction, allowing individual clients to obtain a consensus value without waiting for other clients to invoke the object. Several of the enabling mechanisms we have developed in our protocol are of general value in themselves: The TAOAs can be used in other protocols to support nonmalleable communication among clients when Byzantine failures are a concern and the distributed coin-flipping technique of Section 5.4 can be useful in other randomized protocols.

Acknowledgment

This material is based upon work supported by the Air Force under contract No. F30602-99-C-0165. Any opinions, finding and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United State Air Force. Portions of this material previously appeared in [28], [29].

References

[1] L. Alvisi, D. Malkhi, E. Pierce, and M. Reiter, "Fault Detection for Byzantine Quorum Systems," *Proc. Seventh IFIP Int'l Working Conf. Dependable Computing for Critical Applications,* pp. 357–371, Jan. 1999.

[2] R.J. Anderson, "The Eternity Service," *Proc. Pragocrypt '96,* 1996.

[3] J. Aspnes and M. Herlihy, "Fast Randomized Consensus Using Shared Memory," *J. Algorithms,* vol. 11, pp. 441–461, 1990.

[4] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message-Passing Systems," *J. ACM,* vol. 42, no. 1, pp. 124–142, Jan. 1995.

[5] J.K. Bennet, J.B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. Second ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 168–176, Mar. 1990.

[6] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," *Proc. Third USENIX Symp. Operating Systems Design and Implementation,* Feb. 1999.

[7] B. Chor and C. Dwork, "Randomization in Byzantine Agreement," *Advances in Computing Research, Randomness in Computation,* S. Micali, ed., vol. 5, pp. 443–497, JAI Press, 1989.

[8] Y. Desmedt and Y. Frankel, "Shared Generation of Authenticators and Signatures," *Advances in Cryptology—CRYPTO '91 Proc.,* J. Feigenbaum, ed., pp. 457–469, 1992.

[9] A. De Santis, Y. Desmedt, Y. Frankel, and M. Yung, "How to Share a Function Securely," *Proc. 26th ACM Symp. Theory of Computing,* pp. 522–533, May 1994.

[10] W. Diffie and M.E. Hellman, " New Directions in Cryptography," *IEEE Trans. Information Theory,* vol. 22, no. 6, pp. 644–654, Nov. 1976.

[11] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374–382, Apr. 1985.

[12] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson, "The Digital Distributed System Security Architecture," *Proc. 12th NIST/NCSC Nat'l Computer Security Conf.,* pp. 305–319, Oct. 1989.

[13] J. Garay, R. Gennaro, C. Jutla, and T. Rabin, "Secure Distributed Storage and Retrieval," *Proc. 11th Int'l Workshop Distributed Algorithms,* pp. 275–289, 1997.

[14] D K. Gifford, "Weighted Voting for Replicated Data," *Proc. Seventh ACM Symp. Operating Systems Principles,* pp. 150–162, 1979.

[15] Int'l Telegraph and Telephone Consultative Committee (CCITT), "The Directory—Authentication Framework, Recommendation, X. 509," 1988.

[16] K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith, "The SecureRing Protocols for Securing Group Communication," *Proc. 31st Ann. Hawaii Int'l Conf. System Sciences,* pp. 317–326, Jan. 1998.

[17] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558–565, July 1978.

[18] L. Lamport, "On Interprocess Communication (Part II: Algorithms)," *Distributed Computing,* vol. 1, pp. 86–101, 1986.

[19] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Programming Languages and Systems,* vol. 4, no. 3, pp. 382–401, July 1982.

[20] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in Distributed Systems: Theory and Practice," *ACM Trans. Computer Systems,* vol. 10, no. 4, pp. 265–310, Nov. 1992.

[21] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Computer Systems,* vol. 7, no. 4, pp. 321–359, Nov. 1989.

[22] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler, "Implementation of Argus," *Proc. 11th ACM Symp. Operating Systems Principles,* pp. 111–122, 1987.

[23] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira, "Safe and Efficient Sharing of Persistent Objects in Thor," *Proc. SIGMOD '96,* pp. 318–329, June 1996.

[24] P.V. McMahon, "SESAME V2 Public Key and Authorization Extensions to Kerberos," *Proc. 1995 Internet Soc. Symp. Network and Distributed System Security,* pp. 114–131, Feb. 1995.

[25] D. Malkhi, "Quorum Systems," *The Encyclopedia of Distributed Computing,* J. Urban and P. Dasgupta, eds., Kluwer Academic, to be published.

[26] D. Malkhi, Y. Mansour, and M.K. Reiter, "On Diffusing Updates in a Byzantine Environment," *Proc. 18th IEEE Symp. Reliable Distributed Systems,* Oct. 1999.

[27] D. Malkhi and M. Reiter, "Byzantine Quorum Systems," *Distributed Computing,* vol. 11, no. 4, pp. 203–213, 1998.

[28] D. Malkhi and M. Reiter, "Secure and Scalable Replication in Phalanx (extended abstract)," *Proc. 17th IEEE Symp. Reliable Distributed Systems,* pp. 51–60, Oct. 1998.

[29] D. Malkhi and M. Reiter, "Survivable Consensus Objects (extended abstract)," *Proc. 17th IEEE Symp. Reliable Distributed Systems,* pp. 271–279, Oct. 1998.

[30] D. Malkhi, M. Reiter, and A. Wool, "The Load and Availability of Byzantine Quorum Systems," *Proc. 16th ACM Symp. Principles of Distributed Computing,* pp. 249–257, Aug. 1997.

[31] D. Malkhi, M. Reiter, A. Wool, and R. Wright, "Probabilistic Quorum Systems," brief announcement appears in *Proc. 17th ACM Symp. Principles of Distributed Computing,* submitted for publication, pp. 321, June 1998.

[32] D. Malkhi, M. Reiter, and R. Wright, "Probabilistic Quorum Systems," *Proc. 16th ACM Symp. Principles of Distributed Computing,* pp. 267–273, Aug. 1997.

[33] "Group Comm.," *Comm. ACM,* D. Powell, ed. vol. 39, no. 4, Apr. 1996.

[34] M.K. Reiter, "Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart," *Proc. Second ACM Conf. Computer and Comm. Security,* pp. 68–80, Nov. 1994.

[35] M.K. Reiter, M.K. Franklin, J.B. Lacy, and R.N. Wright, "The Ω Key Management Service," *J. Computer Security,* vol. 4, no. 4, pp. 267–287, 1996.

[36] R. Rivest, A. Shamir, and L. Adleman, " A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Comm. ACM,* vol. 21, no. 2, pp. 120–126, Feb. 1978.

[37] F.B. Schneider, " Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys,* vol. 22, no. 4, pp. 299–319, Dec. 1990.

[38] S.K. Shrivastava, P.D. Ezhilchelvan, N.A. Speirs, S. Tao, and A. Tully, "Principal Features of the VOLTAN Family of Reliable Node Architectures for Distributed Systems," *IEEE Trans. Computers,* vol. 41, no. 5, pp. 542–549, May 1992.

[39] FIPS180-1, "Secure Hash Standard," *Federal Information Processing Standards Publication 180-1,* US Dept. of Commerce/NIST, National Technical Information Service, Apr. 1995

[40] R.H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. Database Systems,* vol. 4, no. 2, pp. 180–209, 1979.

[41] A. Wool, "Quorum Systems in Replicated Databases: Science or Fiction?" *IEEE Technical Committee on Data Eng.,* vol. 21, no. 4, pp. 3–11, Dec. 1998.

Dahlia Malkhi received her PhD, MSc degrees in computer science and a BSc degree in mathematics and computer science in 1994, 1988, 1985, respectively, from the Hebrew University of Jerusalem, Israel. She was a member of the Secure Systems Research Department at AT&T Labs–Research in Florham Park, New Jersey, from 1995 to 1999. She is currently a faculty member of the School of Computer Science and Engineering at the Hebrew University of Jerusalem, Israel. Her research interests include all areas of distributed systems and security.

Michael Reiter received the BS degree in mathematical sciences from the University of North Carolina in 1989, and the MS and PhD degrees in computer science from Cornell University in 1991 and 1993, respectively. From 1993 to 1998, he was a member of AT&T Labs–Research. In 1998, he moved to his present position as department head of the Secure Systems Research Department in Bell Labs, Lucent Technologies. His research interests include all areas of computer and communications security, electronic commerce, and distributed computing.