Backoff Protocols for Distributed Mutual Exclusion and Ordering*

Gregory Chockler, Dahlia Malkhi
School of Computer Science and Engineering
The Hebrew University of Jerusalem
Jerusalem, Israel 91904
{grishac,dalia}@cs.huji.ac.il

Michael K. Reiter
Secure Systems Research Department
Bell Labs, Lucent Technologies
600 Mountain Ave., Murray Hill, NJ 07974 USA
reiter@research.bell-labs.com

Abstract

We present a simple and efficient protocol for mutual exclusion in synchronous, message-passing distributed systems subject to failures. Our protocol borrows design principles from prior work in backoff protocols for multiple access channels such as Ethernet. Our protocol is adaptive *in that the expected amortized system response time—informally, the average time a process waits before entering the critical section—is a function only of the number of clients currently contending and is independent of the maximum number of processes who might contend. In particular, in the contention-free case, a process can enter the critical section after only one round-trip message delay. We use this protocol to derive a protocol for ordering operations on a replicated object in an* asynchronous *distributed system subject to failures. This protocol is always safe, is probabilistically live during periods of stability, and is suitable for deployment in practical systems.*

1 Introduction

In this paper we describe a very simple, distributed mutual exclusion protocol by which a process can gain the right to execute for a fixed time interval Δ without interference from other processes. Our protocol is directly inspired by *backoff* protocols for multiple access channels; the collision detection protocol of Ethernet is the most well-known example. In Ethernet, a process wishing to send on an (apparently) empty channel simply does so. If it detects that its send collided with another process', it "backs off" for a random delay and tries again later. Here we use similar principles to derive a mutual exclusion algorithm for synchronous message-passing systems that is deterministically safe and that ensures entry to the "critical section" with probability one.

The performance of our protocol can generally be characterized in terms of *amortized system response time* [5]. The amortized system response time is the mean delay that each of t processes incurs before entering the critical section, assuming that all t (and no others) start contending at the same time. We prove an upper bound on the expected amortized system response time of $O(\Delta t)$, thereby showing that our protocol is *adaptive* in that the amortized system response time is independent of the maximum number of processes that might contend. In addition, in the case of no contention, the delay a process incurs before entering the critical section is merely one round-trip message delay on the network, and thus is independent of Δ.

Fault tolerance is a feature of our protocol. We present our protocol in a system model with distinct clients and servers, motivated by the system in which we have implemented it, described later. Clients, which contend for mutual exclusion, may crash without affecting the protocol. In particular, since a client is granted exclusion for a fixed time period Δ—and there is no designated "unlock" operation that a client must perform—a client's failure after it succeeds in gaining exclusion does not preclude other clients from subsequently gaining exclusion after the Δ time period expires. Moreover, our protocol masks the arbitrary (Byzantine) failure of a threshold number of servers.

We use this mutual exclusion protocol to develop a protocol by which operations on a replicated object can be serially ordered. Despite the fact that this ordering protocol is deterministically safe even in an *asynchronous* system—and our mutual exclusion protocol is not—the mutual exclusion protocol is key to ensuring that operations are ordered and complete (with probability one) once the system stabilizes. Our ordering protocol orders arbitrarily many operations on the object as soon as a single contender gains access to the critical section.

Aside from always-safe operation ordering, we have found our mutual exclusion protocol useful for other tasks within the system that motivated it, called Fleet [16]. Fleet supports highly available, shared data for clients using an

***To appear in** ICDCS 2001.

infrastructure of servers that may suffer arbitrary (Byzantine) failures. In order to detect the presence of faulty servers, statistical fault detection algorithms mine for evidence of faulty servers in the responses they return [1]. Since detection is most accurate when data is accessed sequentially, Fleet attempts to serialize data accesses, and we employ the mutual exclusion protocol described here for this purpose. Our mutual exclusion protocol has the useful property that it remains probabilistically live even during periods of instability (asynchrony) in the system.[1] So, while fault detection may suffer during periods of instability, the nonblocking properties of the Fleet data access protocols are never compromised.

The rest of this paper is structured as follows. We review related work in Section 2 and more precisely state our system model in Section 3. In Section 4 we describe our mutual exclusion protocol, and we outline certain optimizations to it in Section 5. We then develop our ordering protocol based upon it in Section 6. A proof of correctness for our ordering protocol can be found in Section 7.

2 Related work

In Singhal's taxonomy [21], the mutual exclusion protocol we present is a "Maekawa-type" protocol, following [12]. In this class of protocols, a process p_i requests permission to enter the critical section from a set Q_i of processes, such that $Q_i \cap Q_j \neq \emptyset$ for all i, j. Each process in a request set Q_i grants exclusive access to the critical section until it is exited, and p_i is allowed to enter the critical section only if all processes in Q_i grant p_i access. Due to the intersection property of request sets and exclusive locking at each process in a request set, only one process can be in the critical section at any time. Also due to these properties, however, Maekawa-type algorithms are generally prone to deadlock and consequently require extra messages to detect and recover from deadlocks. Deadlock-free Maekawa-type protocols, such as [10, 20], have been proposed by strengthening the constraints on request sets so that for all i, j, either $p_i \in Q_j$ or $p_j \in Q_i$ [21]. However, in our context, this strengthening is not possible because the clients requesting mutual exclusion are distinct from the servers that comprise the request sets. Clients cannot be added to request sets because they are transient and because the population of clients that might contend is not known *a priori*. The protocol that we present here works with Maekawa's original (weaker) intersection property $Q_i \cap Q_j \neq \emptyset$ in the fault model addressed in [12]. At the same time, our protocol is not prone to deadlock.

As discussed in Section 1, we evaluate our protocol based on the amortized system response time that it achieves. This measure was introduced in the context of shared-memory mutual exclusion algorithms [5], where there are examples boasting amortized system response times of $O(t)$ or even $O(1)$ (e.g., [3, 5]). An alternative to using our protocol is to employ one of these algorithms, using a distributed protocol to emulate each shared variable it uses (e.g., [15]). While the resulting algorithm would have superior amortized system response time (asymptotically), the performance in practice would be far worse than our protocol in the main case we care about—i.e., contention-free performance—due to the overheads of the variable emulation protocols. This also holds for backoff-style mutual exclusion algorithms explored for the shared-memory setting (e.g., [2], which assumes even stronger objects than shared variables).

The manner in which we build upon our mutual exclusion protocol to order operations on data objects in an asynchronous system is similar to work of Fetzer and Cristian on consensus in the *timed asynchronous model* [6], and the works of Lamport on Paxos [11] and of Keidar and Dolev on *extended 3-phase commit* (E3PC) [8]. These works compose a (not necessarily safe) mutual exclusion protocol with a commit protocol to derive solutions to problems equivalent to our ordering problem. Paxos and E3PC, while building on mutual exclusion protocols, do not propose mutual exclusion implementations of their own. Fetzer and Cristian employ a mutual exclusion protocol that rotates the *preference* for access to the critical section among the possible contenders in sequence, but enables the next preferred contender to be bypassed without delay if that contender is unavailable. As such, its mutual exclusion primitive is also adaptive in the sense above. In order to achieve this, however, the protocol relies on clock synchronization among the participating servers. Clock synchronization is not required by our protocol. More generally, however, our work contributes relative to all the above by providing a new and efficient mutual exclusion primitive, and by admitting arbitrary (Byzantine) server failures in our ordering protocol.

3 System model

Our system model divides the set of processes into *clients* and *servers*. We assume a fixed, known set U of n servers and an arbitrary, finite, but unknown number of clients. The protocol of a process is described in terms of *event handlers* that are triggered by the arrival of events such as the receipt of a message, the tick of a local clock, or input from a calling application. Once triggered, we assume that the event handler runs to completion without delay.

[1] Formally, probabilistic liveness during periods of instability holds only if the scheduling adversary is nonadaptive. That is, for any execution, the scheduler chooses the distribution from which message delays will be drawn before the protocol execution begins; it cannot change this distribution in response to events in the execution. We omit further discussion of this issue here, except to note that in practice, this is an assumption we are willing to adopt for Fleet.

Processes that obey their protocol specifications and receive (and handle) infinitely many events in an infinite run are called *correct*. Other processes are called *faulty*. Up to a threshold b of servers may fail, and may do so arbitrarily (Byzantine failures); i.e., the event handlers of a faulty server may not conform to their specifications. While any number of clients may fail, clients are assumed to fail only by crashing, i.e., simply by no longer receiving events. This restriction of client failures to crashes may seem unrealistic when servers are presumed to fail arbitrarily. However, typically little can be done to protect an application from Byzantine clients, as such clients can always corrupt an object's data by submitting requests with incorrect content. One way of dealing with this in practice is to allow the creator of an object to prohibit untrusted clients from modifying objects using access control mechanisms that remain in force at correct servers provided that b or fewer servers fail (even arbitrarily). Thus, our assumption of client crashes in practice reduces to an assumption about the clients trusted to modify that object by its creator.

We assume that the local clock of each correct process ticks at the same rate as real time, so that a process can accurately measure the passage of a chosen, real-time timeout period. Since the timeout periods involved in our protocols would be very short in practice, this is a reasonable assumption. We do *not* assume that clocks at processes are synchronized.

Processes communicate by message passing. We assume that communication channels provide at-most-once message transmission: if p_1 and p_2 are correct, then p_2 receives any message m from p_1 at most once, and then only if p_1 sent m to p_2. (Obviously, we further assume that p_1 never sends the same message twice to p_2, which it can implement by, e.g., including a unique sequence number in the message.) There is a globally known constant δ. We say that a run is *stable at real time* T if for each correct client, there exists a quorum of correct servers such that any message sent at time $T' \geq T$ between that client and a server in that quorum arrives by time $T' + \delta$. The definition of quorum that we use will be given in Section 4. A run is *synchronous* if it is stable at the time the run begins. Though processes communicate by message passing, we present our protocols in terms of remote operation invocations on servers, for simplicity of presentation. In synchronous runs, we also neglect the processing time of such remote invokations and assume that they complete instantaneously. We will return to explicit message passing events when necessary to prove correctness.

4 Mutual exclusion

In this section we present our mutual exclusion protocol by which clients can contend for the opportunity to run for Δ time units without interference by other clients. More precisely, there is an operation contend that a client can invoke. When the invocation returns at the client, the client then has Δ time units in which to execute in isolation. After Δ time units pass, however, another client's contend operation may return. As discussed in Section 1, in addition to requiring mutual exclusion, we will also be concerned with the system response time.

The idea of the protocol is for clients to access servers simply to find out whether other clients are simultaneously contending. In order to provide mutual exclusion, every pair of clients must access $2b + 1$ correct servers in common. If a client detects that another client is contending, it backs off for a random delay, chosen from a distribution that adapts to the number of contending clients. Intuitively, clients thus delay an amount of time proportional to the number of simultaneously contending clients, while eventually, when they sufficiently space their contentions, each succeeds. More precisely, the protocol is probabilistically live in a synchronous system, i.e., with probability 1 some client's contend operation returns.

The requirement that any two clients access at least $2b+1$ common correct servers can be satisfied if each client queries the servers according to a special variant of *masking quorum systems* [14]. The specific variant we need is a system $\mathcal{Q} \subseteq 2^U$ such that (i) for any quorums $Q_1, Q_2 \in \mathcal{Q}$: $|Q_1 \cap Q_2| \geq 3b+1$, and (ii) for every $B \subseteq U$ with $|B| = b$, there exists $Q \in \mathcal{Q}$ such that $Q \cap B = \emptyset$. Property (i) ensures that if each client queries a quorum of servers, then any two clients' queries intersect in at least $2b + 1$ correct servers. Property (ii) ensures that some quorum is always available. These quorum systems are a special variants of the request sets used in Maekawa's protocol [12] (see Section 2) to address up to b arbitrary server failures. Using a straightforward adaptation of Corollary 4.4 in [14], we find that our requirement of quorums implies $n > 5b$.

The protocol, then, is extremely simple; see Figure 1. In this figure, "$||$" denotes concurrent invocation of statements, and "$d \leftarrow_R S$" denotes the selection of an element of set S uniformly at random and assignment of that element to d. At a high level, the protocol executes as follows. When presented with a request from a client, the server returns FREE if it last returned FREE over $\Delta + 2\delta$ time ago; otherwise it returns LOCKED. To contend, a client collects FREE or LOCKED responses from a quorum and succeeds if at most b responses are LOCKED. If more than b responses are LOCKED, then it delays for some random duration in the interval $[(\Delta + 4\delta) \ldots 2^s(\Delta + 4\delta)]$ before trying again, where s is a "retry value" that records the number of times the client has previously queried servers in this contend operation. That is, clients employ an exponential backoff strategy: the expected duration of a client's delay is proportional to twice its delay during its last retry.

```
contend ():

$s \leftarrow 0$;
repeat [
    $Q_1 \leftarrow \emptyset$;
    $||_{u \in U}$ [ $status_u \leftarrow u.\mathsf{try}()$;
          $Q_1 \leftarrow \{u\} \cup Q_1$;
    ] until ($\exists Q \in \mathcal{Q} : Q \subseteq Q_1$);

    if ($|\{u : status_u = locked\}| \leq b$)
          return;
    else
          $s \leftarrow s + 1$;
          $d \leftarrow_R [(\Delta + 4\delta) \ldots 2^s(\Delta + 4\delta)]$;
          sleep($d$);
] until (false);
```

(a) Client program

```
try ():

if (clock() $-$ lastGranted $> \Delta + 2\delta$)
    lastGranted $\leftarrow$ clock();
    return FREE;
else
    return LOCKED;
```

(b) Server program

Figure 1. Mutual exclusion protocol

The correctness of this protocol is proved easily in the following lemma:

Lemma 1 *If the system is synchronous, and a client's contend operation returns at real time T, then no other client's contend returns in the interval $[T, T + \Delta]$.*

Proof : Suppose that at client c, a contend returns at time T. By assumption, this implies that c invoked try on servers no earlier than $T - 2\delta$. Consider any other client c' whose contend returns in the interval $[T, T + \Delta]$. Again, our assumption implies that c' invoked try on servers no earlier than $T - 2\delta$. Then all correct servers in the quorum Q that c queried, except at most b, responded to c in the time frame $[T - 2\delta, T]$ with the response FREE. Similarly, all correct servers in the quorum Q' that c' queried responded to c' in the time frame $[T - 2\delta, T + \Delta]$ with FREE. Since Q and Q' intersect in at least $2b + 1$ correct servers, at least one such correct server must have responded FREE to both c and c'. Since any correct server returns FREE to only one client in any $\Delta + 2\delta$ time period, this is a contradiction.□

As discussed previously, the measure of quality on which we focus for our mutual exclusion protocol is amortized system response time. The following lemma implies that the expected amortized system response time is $O(\Delta t)$.

Lemma 2 *If the system is synchronous, t clients contend, and none of these clients fails, then all client's contend operations return in expected $O(\Delta t)$ time.*

Proof : (Sketch.) Let T denote a time by which all t clients are contending, and let $R_k = \sum_{s=1}^{k} 2^s(\Delta + 4\delta)$. At time $T + R_k$, every client c_i whose contend operation has not returned has a retry value s_i that satisfies $s_i \geq k$, since the s'th retry of a client is made within at most $2^s(\Delta + 4\delta)$ time units after the previous try. Let $t_k \leq t$ denote the number of number of clients whose contend operations have not returned by time $T + R_k$, and denote them $c_1, \ldots, c_{t_k}$. For each such client's first attempt after $T + R_k$ to fail, the time at which it attempts must follow some other client's by at most $\Delta + 4\delta$. This happens with probability at most $(t_k - 1)/2^k$.

Let X_i^k, $1 \leq i \leq t_k$, be an indicator random variable such that $X_i^k = 0$ if c_i's first attempt after time $T + R_k$ succeeds; $X_i^k = 1$ otherwise. By the analysis above, $P[X_i^k = 1] \leq (t_k - 1)/2^k$. Let $\bar{X}^k = \sum_{i=1}^{t_k} X_i^k$; $\bar{X}^k$ is the number of clients whose attempt following time $T + R_k$ fails. By linearity of expectation,

$$E[\bar{X}^k] \leq t_k \frac{t_k - 1}{2^k}.$$

Since $E[\bar{X}^k] \geq xP[\bar{X}^k \geq x]$, it follows that for any constant $c > 1$,

$$\frac{1}{c} \;\geq\; P\left[\bar{X}^k \geq cE[\bar{X}^k]\right] \;\geq\; P\left[\bar{X}^k \geq ct_k \frac{t_k - 1}{2^k}\right]$$

So, for example, when $k \geq \log_2(2c(t-1))$ we have

$$\frac{1}{c} \;\geq\; P\left[\bar{X}^k \geq \frac{t_k}{2}\right].$$

It follows that the expected k by which all clients succeed is $O(\log t)$, meaning that the expected time by which all clients' contend operations return is at most

$$T + \sum_{s=1}^{O(\log t)} 2^s(\Delta + 4\delta) \;=\; O(\Delta t).$$

□

Corollary 1 *If the system is synchronous, then the expected amortized system response time [5] with t contending clients is $O(\Delta t)$.*

Corollary 2 *If the system is synchronous and some correct client invokes contend , then with probability 1, some client's contend invocation returns.*

In the mutual exclusion protocol as presented in Figure 1 and analyzed in Lemma 2, client backoff was exponential as a function of the number of retries in its contend operation. Even though exponential backoff yields $O(\Delta t)$ amortized system response time, analysis of backoff strategies in the context of multiple access channels shows that it performs less well in other measures than various polynomial backoff strategies (e.g., [7]). While this analysis does not apply to our case directly, we expect that similar properties hold in our setting, and thus in practice it may be preferable to experiment with other backoff strategies.

5 Improvements and optimizations for the mutual exclusion protocol

In this section we sketch several possible improvements and optimizations to the mutual exclusion protocol. The implementation of the proposed ideas and the assessment of their practical implications are the subject of our ongoing work.

Avoiding backoff by breaking symmetry If the application is such that one client c repeatedly contends with little delay between contentions, then we can improve c's response time if c does not back off between consecutive try attempts. The backoff protocol will adequately space the other client's retries, and c's asymmetric strategy will enable it to gain mutual exclusion quickly.

Enqueing client requests In this optimization, each server maintains an internal data structure, called *delayed reply list*, where it records IDs of the clients whose try requests arrive while the server is locked. As soon as the server's status becomes FREE, it goes through the records in the delayed reply list and sends FREE to the client with the lowest ID and LOCKED to everyone else. This optimization may allow the lowest ranking contending client a smooth entry to the critical section, without backoff.

Making mutex safe It is possible to make the protocol safe even during instability periods if clients disregard those replies to their try() requests that arrive after 2δ time units. However, in practice, this optimization can negatively affect the throughput of the applications whose implementation does not require the underlying mutex to be safe (e.g., the operation ordering presented in Section 6).

Parameterized contend In order to allow for better adaptation to changing system conditions and to application needs, it is possible to make Δ a parameter of the contend operation (and, consequently, of the try request) instead of being a system-wide constant. Both safety and the expected delay become parameterized by the actual Δ's employed.

6 Operation ordering

As discussed in Section 1, one of the main applications for the mutual exclusion protocol of Section 4 is a protocol for serializing operations on replicas of an object in a distributed system. In order to perform an operation o on the replicated object, a client application *submits* the operation for execution. The properties that our ordering protocol satisfies are the following:

Order There is a well-defined sequence in which submitted operations are applied, and the result of each operation that returns is consistent with that sequence.

Liveness If a run is eventually stable, then every operation submitted by a correct client is performed with probability one, and if performed, its result is returned to the client.

Due to the Order and Liveness properties, our protocol emulates *state machine replication* [19]. Among others, our implementation supports the following distinct features: First, the ordering responsibilities are delegated to the clients, which are not Byzantine by assumption. This way, we need not employ digital signatures or signature-like cryptographic constructions, thus improving the performance and scalability of the protocol. Second, our protocol makes progress by updating only quorums of replicas, which helps to achieve better load balancing and enhances scalability. Third, our protocol supports nondeterministic operations, since each operation is applied at a client and the resulting object state is then copied back to servers.

Some modern protocols for implementing state machine replication in Byzantine environments (e.g., [17, 9, 4]) assume a less restricted failure model by allowing arbitrary client failures. In these solutions, clients do not actively participate in the protocol, but serve merely as users that inject new operations into the server universe and collect responses. While this approach prevents Byzantine clients from interfering with the ordering protocol, it does not prevent attacks in which faulty clients corrupt object's data by submitting operations with arbitrary parameter values. Thus, in practice, the added value of providing protection against Byzantine client failures in terms of the system security guarantees is outweighed by the performance and scalability gain resulting from delegating ordering responsibilities to the clients.

The detailed client and server programs are shown in Figure 2 and Figure 3 respectively. The client program for submit(o) consists of two threads executed concurrently. The first thread, described in lines 2.3–7, simply submits the operation o to the servers for execution and awaits responses. The second thread, lines 2.8–32, invokes operations to create a new state and commits states in a serial order; we call this the *ordering thread*. If f and g are functions, then $f|g$ denotes a function such that $(f|g)(o) = g(o)$ if $g(o) \neq \perp$ and $f(o)$ otherwise; see line 3.24. The following subsections contain details about operations, states, and *ranks* that are essential to understanding the ordering thread.

6.1 Operations and states

Our protocol works by applying an operation to a *state* to produce a new state and a return result. A client submits an operation o to be performed by invoking $\mathsf{submit}(o)$. For simplicity of presentation, we assume that the same operation is never submitted by two distinct clients or twice by the same client. In practice, enforcing such uniqueness of operations can be implemented by each client labeling each of its operations with the client's identifier and a sequence number.

A state, denoted by σ (possibly with subscripts and/or superscripts), is an abstract data type that has the following interfaces:

- $\sigma.\mathsf{version}$ is an integer-valued field. It denotes the "version" of the state. This field can be set by the protocol manipulating the state.

- $\sigma.\mathsf{doOp}(o)$ applies the operation o to the state σ, performing any modifications on σ in place.

- $\sigma.\mathsf{response}(o)$, if defined, is the return result for operation o.

- $\sigma.\mathsf{reflects}(o)$ indicates whether $\sigma.\mathsf{doOp}(o)$ was previously executed.

A state's interfaces are assumed to satisfy the following properties. First, $\sigma.\mathsf{reflects}(o) = \mathrm{true}$ iff $\mathsf{doOp}(o)$ was invoked on some previous state. In practice, this can be implemented by recording within the state the highest operation sequence number already performed for each client. Second, if σ is the result of applying operations (via doOp) to a prior instance σ' such that $\sigma'.\mathsf{reflects}(o) = \text{false}$, $\sigma.\mathsf{reflects}(o) = \mathrm{true}$, and $\sigma.\mathsf{version} = \sigma'.\mathsf{version} + 1$, then $\sigma.\mathsf{response}(o)$ is defined and returns the result for operation o. Note that by this assumption, $\sigma.\mathsf{response}(o)$ can be eliminated ("garbage collected") when $\sigma.\mathsf{version}$ is incremented. In this way, the size of σ can be limited.

Aside from the instance of garbage collection just mentioned, we do not further elaborate on garbage collection here. The primary data structures that grow in our protocol as presented in Figures 2 and 3 are (i) the record of which client operations have been performed (to compute $\sigma.\mathsf{reflects}(o)$) and (ii) a $\mathsf{response}$ function maintained at each server that records the response for each client operation (see lines 3.3–4,24). In practice, eliminating unnecessary data from these structures can be achieved, for example, by propagating information among servers in the background (e.g., using the techniques of [13]) to convey when information about a given operation can be purged from the system. Other optimizations are possible, e.g., that trade off passing complete states versus update suffixes.

6.2 Rank

Each client executes the ordering thread of our protocol with an associated integer called its *rank*. We assume that no two clients ever adopt the same rank, which can be ensured, e.g., if each client's rank is formed with its identifier in the low-order bits. When invoking an operation on a server u in our protocol, a client always sends its current rank as an argument to the invocation; this rank is denoted by r in $u.\mathsf{get}(r)$, $u.\mathsf{propose}(\sigma, r)$ and $u.\mathsf{commit}(\sigma, r)$ invocations. A server responds to only the highest-ranked client that has contacted it. In particular, if a server u is contacted by a client with a lower rank than another client to which it has already responded, then it throws a RankException that notifies the client of the higher rank under which another client contacted it. In order to get u to respond to it, the client will have to abort its current protocol execution, adjust its rank, and try again (starting at line 2.8).

The precise criteria that dictate when a client aborts its protocol run to adjust its rank are important to the liveness of our protocol. On the one hand, if a client aborts its protocol run based upon receiving a single RankException, then the client risks being aborted by a faulty server who in fact was not contacted by a higher ranking client. On the other hand, if the client requires $b + 1$ RankExceptions in order to abort, then the client may not abort even though b correct servers have been contacted by a higher-ranking client and thus will refuse to return responses to this client.

Our solution to this issue therefore mandates that the quorum system $\mathcal{Q}$ we employ satisfy the following property: For every $B_1, B_2 \subseteq U$ with $|B_1| = |B_2| = b$, there exists $Q \in \mathcal{Q}$ such that $Q \cap (B_1 \cup B_2) = \emptyset$. This restriction enables the client to complete its protocol run using quorums provided up to b correct servers respond with RankException. Consequently, whereas original masking quorum systems existed as long as $n > 4b$ [14], this stronger constraint limits their existence to systems in which $n > 6b$ (see Corollary 4.4 in [14]). When a client is forced to adjust its rank due to receiving $b + 1$ RankExceptions, it does so by choosing a value larger than the maximum of all ranks reported by those RankExceptions.

We note that an alternative approach would be for clients to digitally sign (e.g., [18]) their ranks using a key available only to clients allowed to access the object (or a subset of

```
submit ($o$):

    $waiting \leftarrow$ true;

    || $V_1 \leftarrow \emptyset; Q_1 \leftarrow_R \mathcal{Q}$;
       $\|_{u \in Q_1}$ [ $\rho_u \leftarrow u.\mathsf{submit}(o); V_1 \leftarrow \{\rho_u\} \cup V_1$;
       ] until ($\exists \rho : |\{\rho_u \in V_1 : \rho = \rho_u\}| \geq b+1$);
       $waiting \leftarrow$ false;
       return $\rho : |\{\rho_u \in V_1 : \rho = \rho_u\}| \geq b+1$;

    || repeat [
           contend();

           $Q_2 \leftarrow \emptyset$;
           $\|_{u \in U}$ [ $\langle \sigma^c_u, \sigma^{pc}_u, proposer_u, pending_u \rangle \leftarrow u.\mathsf{get}(r)$;
                   $Q_2 \leftarrow \{u\} \cup Q_2$;
           ] until ($\exists Q \in \mathcal{Q} : Q \subseteq Q_2$);

           $\Sigma^c \leftarrow \{\sigma' : |\{u : \sigma' = \sigma^c_u\}| \geq b+1\}$;
           $\sigma^c \leftarrow \sigma : \sigma.\mathsf{version} = \max_{\sigma' \in \Sigma^c}\{\sigma'.\mathsf{version}\}$;
           $\sigma^{pc} \leftarrow \mathsf{choose}(\{\langle \sigma^{pc}_u, proposer_u \rangle : \sigma^{pc}_u \neq \bot\})$;
           $completed \leftarrow \max\{completed,$
               $\max\{v : |\{u : \sigma^{pc}_u.\mathsf{version} > v\}| \geq b+1\}\}$;

           if ($\sigma^c \neq \bot \wedge \sigma^c.\mathsf{version} > completed$)
               $\sigma \leftarrow \sigma^c$;
           else if ($\sigma^{pc} \neq \bot \wedge \sigma^{pc}.\mathsf{version} > completed$)
               $\sigma \leftarrow \sigma^{pc}$;
           else
               $pending \leftarrow \{o : |\{u : o \in pending_u\}| \geq b+1\}$;
               $\sigma \leftarrow \mathsf{apply}(pending, \sigma^c)$;

           $Q_2 \leftarrow \emptyset$;
           $\|_{u \in U}$ [ $u.\mathsf{propose}(\sigma, r); Q_2 \leftarrow \{u\} \cup Q_2$;
           ] until ($\exists Q \in \mathcal{Q} : Q \subseteq Q_2$);

           $Q_2 \leftarrow \emptyset$;
           $\|_{u \in U}$ [ $u.\mathsf{commit}(\sigma, r); Q_2 \leftarrow \{u\} \cup Q_2$;
           ] until ($\exists Q \in \mathcal{Q} : Q \subseteq Q_2$);

           $completed \leftarrow \max\{completed, \sigma.\mathsf{version}\}$;

       ] until ($waiting =$ false);
```

```
choose($\{\langle \sigma^{pc}_u, proposer_u \rangle\}_u$) :

    $S[1, 2, \ldots] \leftarrow \{\langle \sigma^{pc}_u, proposer_u \rangle\}_u$ sorted
        in descending order by
        $\langle \sigma, proposer \rangle > \langle \sigma', proposer' \rangle \Leftrightarrow$
        $(\sigma.\mathsf{version} > \sigma'.\mathsf{version} \vee$
         $(\sigma.\mathsf{version} = \sigma'.\mathsf{version} \wedge$
          $proposer > proposer'))$;

    $i \leftarrow 1$; $count \leftarrow [0, 0, \ldots]$;
    repeat [ $\langle \sigma, proposer \rangle \leftarrow S[i]$;
             $count[\sigma] \leftarrow count[\sigma] + 1$;
             $i \leftarrow i + 1$;
    ] until ($\exists \sigma : count[\sigma] \geq b+1 \ \vee \ i > |S|$);

    if ($\exists \sigma : count[\sigma] \geq b+1$)
        return $\sigma : count[\sigma] \geq b+1$;
    else
        return $\bot$;
```

```
apply($pending, \sigma'$):

    repeat [ $o \leftarrow_R pending$;
             $pending \leftarrow pending \setminus \{o\}$;
             if ($\sigma'.\mathsf{reflects}(o) =$ false)
                 $\sigma'.\mathsf{doOp}(o)$;
    ] until ($pending = \emptyset$);
    $\sigma'.\mathsf{version} \leftarrow \sigma'.\mathsf{version} + 1$;
    return $\sigma'$;
```

Figure 2. Client side of ordering protocol

them designated to execute the ordering protocol). When a server throws a RankException to a client, it passes the highest rank under which any client has contacted it, including the digital signature on that rank from that client. The client receiving the RankException can verify the validity of the rank by verifying the digital signature on it. In this implementation, a client can abort its protocol run based on a single RankException with which the client receives a validly signed rank, since a faulty server cannot forge signatures. This approach imposes overheads in terms of key

```
submit($o$):
    $pending \leftarrow pending \cup \{o\}$;
    sleep until ($\mathsf{response}(o) \neq \bot$);
    return $\mathsf{response}(o)$;

get($r$):
    if ($r > maxRank$)
        $maxRank \leftarrow r$;
        return $\langle \sigma^c, \sigma^{pc}, proposer, pending \rangle$;
    else
        throw RankException;

propose($\sigma, r$):
    if ($r \geq maxRank$)
        $maxRank \leftarrow r$;
        $proposer \leftarrow r$;
        $\sigma^{pc} \leftarrow \sigma$;
        return;
    else
        throw RankException;

commit($\sigma, r$):
    if ($r \geq maxRank$)
        $maxRank \leftarrow r$;
        $\sigma^c \leftarrow \sigma$;
        $pending \leftarrow pending \setminus$
                    $\{o : \sigma.\mathsf{reflects}(o) = \mathrm{true}\}$;
        $\mathsf{response} \leftarrow \mathsf{response} | \sigma.\mathsf{response}$;
        return;
    else
        throw RankException;
```

Figure 3. Server side of ordering protocol

management and computation, and we therefore opt against it. In particular, digital signatures tend to be relatively intensive to compute and verify. While for a small number of clients, digital signatures can be emulated using message authentication codes, this approach does not scale well.

6.3 Protocol overview

At a high level, the ordering thread of the protocol at a client works by first contending for mutual exclusion, using the protocol of Section 4 (line 2.9). Once this contend returns, the protocol executes similarly to a 3-phase commit protocol. It first invokes get on each server u in some quorum Q^{get} to obtain the states last committed to u (σ^c_u) and last proposed to u (σ^{pc}_u); the rank $proposer_u$ of the client who proposed σ^{pc}_u; and the current set $pending_u$ of pending operations submitted to u. The client then computes the following values:

- σ^c is set to be the state with the highest version number that has been committed to some correct server (i.e., at least $b+1$ servers) in Q^{get} (lines 2.14–15).
- σ^{pc} is set to be the state proposed to some correct server (i.e., at least $b+1$ servers) in Q^{get} by the highest-ranking set of proposers (lines 2.16,33–43).
- *completed* is set to be the highest version number of all states that the responses from the servers in Q^{get} reveal to be committed at a full quorum. In particular, if $b+1$ servers report proposed states σ^{pc}_u with version numbers larger than v, then a state with version v must be committed at a full quorum (line 2.17).

The client chooses which state σ to propose and commit to quorums based on these values. If σ^c has a version number larger than *completed*, then it will propose and commit σ^c to ensure that σ^c gets committed to a full quorum (line 2.19). Its second choice will be to propose and commit the proposed state σ^{pc} if its version number is larger than *completed* (line 2.21). Otherwise, it creates a new state by applying operations to σ^c (lines 2.23–24,44–51), and proposes and commits that state.

The protocol ensures that each newly proposed object state σ' is derived from the state σ that has been most recently committed by applying operations in the *pending* sets of correct servers to σ (line 2.24). This is guaranteed as follows. If σ has been committed to a full quorum, then $\sigma^c = \sigma$ at each correct server in that quorum. This implies that any client that succeeds in invoking get at a full quorum evaluates σ^c to σ, and applies any pending operations to it. If, on the other hand, σ has not been committed at a full quorum, then it is possible for clients to evaluate σ^c to a prior state. However, since σ must be proposed to a full quorum before it is committed, any client that invokes get on a full quorum evaluates σ^{pc} to σ. The client will therefore complete the commitment of σ (bypassing σ^c since *completed* $\geq \sigma^c.\mathsf{version}$) and then continue by applying new operations to σ to derive σ'.

Rank is used to break ties between clients that attempt to propose different states simultaneously. Suppose that p and q each invoke get on a quorum of servers and obtain $\sigma^c = \sigma$ as above. If p succeeds to propose the new state σ' at some full quorum, then the protocol ensures that p's rank is higher than q's rank (otherwise, RankException would be thrown by each correct server in the intersection). Note that, even though q's rank is lower than p's, it might succeed in proposing a new state to some servers (but not to a full quorum) before intersecting with p's quorum and incurring RankExceptions. Nevertheless, p's proposed state and q's proposed state must have the same version, and hence, the choose subroutine will correctly identify p's proposal as the complete one.

7 Correctness proof of the ordering protocol

In this section we prove that Order and Liveness are satisfied by our protocol. Let $\mathcal{M}$ denote a finite set of methods that for any object state σ, a rank r and an operation o, consists of $\mathsf{get}(r)$, $\mathsf{propose}(\sigma, r)$, $\mathsf{commit}(\sigma, r)$ and $\mathsf{submit}(o)$. We assume that any method $\mu \in \mathcal{M}$ can be invoked at a server u at most once throughout the execution. In practice such a requirement can be easily enforced using unique method identifiers composed of client identifier and the sequence number. Let $\mu.rank$ be the rank with which method μ is invoked.

We consider the following system events: For a client p, let $p.\mathsf{send}(u, \mu)$ be the client event that sends the method invocation $\mu \in \mathcal{M}$ to server u, and let $p.\mathsf{ret}(u.\mu, \rho)$, be the client event triggered by the reply of the server u with return value ρ to a previously sent method μ.

A server event is a computation performed upon receiving get , $\mathsf{propose}$, commit , or submit invocations from a client. The event that occurs at a server u as a result of the invocation of a $\mu \in \mathcal{M}$ is denoted $u.\mu$. The code executed by a correct server upon reception of such an invocation is the code of the corresponding server method (see Figure 3). This code executes to completion (return) atomically, with the exception of submit ; submit executes atomically until the sleep command, and its return constitutes a separate event. A faulty server can perform arbitrary computation steps upon reception of client invocations.

We model the system execution as a countable set H of events partially ordered by $\xrightarrow{H}$ relation induced by the natural order of the method invocations and returns. We define the *causal cone* of an event e in H, denoted $\mathsf{ccone}(e, H)$, to be the subset of H such that $\forall e' \in H$, $e' \in \mathsf{ccone}(e, H)$ iff $e' \xrightarrow{H} e$.

If a client p invokes method μ on every server $u \in S \subseteq U$ in H, then we will unite all $p.\mathsf{send}(u, \mu)$ events into a single event called the *client invocation* of a method μ, denoted $p.\mu$. We will write $p.\mu \xrightarrow{H} u.\mu$ for some $u \in U$ iff $p.\mathsf{send}(u, \mu) \in p.\mu$. We also define the *server invocation* of a method μ to be simply the event $u.\mu$ that occurs in H. A server invocation $u.\mu$ is called *complete* in H, if $u.\mu$ does not result in RankException. A client invocation $p.\mu$ is called *complete* in H if there exists a quorum Q such that for each correct server $u \in Q$, $u.\mu$ is complete. We assume that each history begins with the complete $\mathsf{propose}$ and commit invocations with the initial object state σ^0 and the rank 0 as the arguments.

We first set out to prove the Order property of the protocol. We start with proving simple facts that correlate the causal order of $\mathsf{propose}$ events with their ranks:

Lemma 3 *Let* $p.\mathsf{propose}(\sigma_1, r_1)$ *be complete in* H *and* $q.\mathsf{propose}(\sigma_2, r_2) \in H$. *If* $r_1 < r_2$, *then* $p.\mathsf{propose}(\sigma_1, r_1) \xrightarrow{H} q.\mathsf{propose}(\sigma_2, r_2)$. *Otherwise,* $q.\mathsf{get}(r_2) \xrightarrow{H} p.\mathsf{propose}(\sigma_1, r_1)$.

Proof : The result is straightforward from the protocol and the definition of the causal order. □

Corollary 3 *If both* $p.\mathsf{propose}(\sigma_1, r_1)$ *and* $q.\mathsf{propose}(\sigma_2, r_2)$ *are complete in* H, *then they are ordered by the* $\xrightarrow{H}$ *relation.*

For the following lemma, we introduce the following definition. For any $p.\mathsf{propose}(\sigma, r)$, we define its *closest complete* $\mathsf{propose}$ to be $p'.\mathsf{propose}(\sigma', r')$ such that (i) $p'.\mathsf{propose}(\sigma', r')$ is complete; (ii) $p'.\mathsf{propose}(\sigma', r') \xrightarrow{H} p.\mathsf{propose}(\sigma, r)$; and (iii) there does not exist a complete $p''.\mathsf{propose}(\sigma'', r'')$ such that $p'.\mathsf{propose}(\sigma', r') \xrightarrow{H} p''.\mathsf{propose}(\sigma'', r'') \xrightarrow{H} p.\mathsf{propose}(\sigma, r)$. Note that any $p.\mathsf{propose}(\sigma, r)$, other than the propose of σ^0 at system initialization, has a closest complete $\mathsf{propose}$, and that its closest complete $\mathsf{propose}$ is unique by Corollary 3.

Lemma 4 *Consider any* $q.\mathsf{propose}(\sigma_2, r_2)$. *If* $p.\mathsf{propose}(\sigma_1, r_1)$ *is its closest complete* $\mathsf{propose}$ *then all the following results are true:*
(1) $\sigma_2.\mathsf{version} = \sigma_1.\mathsf{version}$ *or* $\sigma_2.\mathsf{version} = \sigma_1.\mathsf{version}{+}1$;
(2) if $\sigma_2.\mathsf{version} = \sigma_1.\mathsf{version}$, *then* $\sigma_1 = \sigma_2$;
(3) if $\sigma_2.\mathsf{version} = \sigma_1.\mathsf{version} + 1$, *then there exists a complete* $p'.\mathsf{commit}(\sigma_1, r')$ *in* $\mathsf{ccone}(q.\mathsf{propose}(\sigma_2, r_2))$, *and* σ_2 *is the result of applying operations in* $\{o : \sigma_2.\mathsf{reflects}(o) \wedge \neg\sigma_1.\mathsf{reflects}(o)\}$ *to* σ_1 *in some sequential order.*

Proof :(Sketch) We prove the result by induction on $\mathsf{ccone}(q.\mathsf{propose}(\sigma_2, r_2))$. That is, we suppose the result holds for any $q'.\mathsf{propose}(\sigma', r') \in \mathsf{ccone}(q.\mathsf{propose}(\sigma_2, r_2))$, and we prove the result for $q.\mathsf{propose}(\sigma_2, r_2)$. Let $\bar{p}.\mathsf{propose}(\bar{\sigma}, \bar{r})$ be the first complete $\mathsf{propose}$ invocation in the causal chain leading to $p.\mathsf{propose}(\sigma_1, r_1)$ such that $\bar{\sigma}.\mathsf{version} = \sigma_1.\mathsf{version}$. By the induction hypothesis for 4.2, $\bar{\sigma} = \sigma_1$.

According to the protocol, the value of σ_2 is computed based on the values of $\langle \sigma_u^c, \sigma_u^{pc}, \mathit{proposer}_u, \mathit{pending}_u \rangle$ returned by each server u in some quorum $Q^{\mathsf{get}(r_2)}$ in response to $q.\mathsf{get}(r_2)$ invocation. Furthemore, if u is correct, then the value of each σ_u^c, σ_u^{pc} and *proposer*$_u$ is determined by some (not necessarily complete) propose invocation $p'.\mathsf{propose}(\sigma', r') \in \mathsf{ccone}(q.\mathsf{get}(r_2))$. By applying results of Lemma 3, Corollary 3 and the induction hypothesis, we conclude the following: If $r' \geq \bar{r}$, then the closest complete propose of $p'.\mathsf{propose}(\sigma', r')$ is either $\bar{p}.\mathsf{propose}(\bar{\sigma}, \bar{r})$ or the one that causally follows $\bar{p}.\mathsf{propose}(\bar{\sigma}, \bar{r})$. Therefore, either $\sigma' = \sigma_1$, or σ' is a state that extends σ_1 with some previously submitted operations. Otherwise, if $r' < \bar{r}$, then the closest complete propose of $p'.\mathsf{propose}(\sigma', r')$ causally precedes $\bar{p}.\mathsf{propose}(\bar{\sigma}, \bar{r})$ and therefore, $\sigma'.\mathsf{version} \leq \bar{\sigma}.\mathsf{version}$.

Once we know the possible values of σ_u^c, σ_u^{pc} and *proposer*$_u$ as returned by $q.\mathsf{get}(r_2)$, and given that any two quorums intersect by at least $2b + 1$ servers, we derive that the value of σ_2 computed in lines 2.14–24 satisfies the lemma results.□

Theorem 1 (Order) *There is a well-defined sequence in which submitted operations are applied and the result of each operation that returns is consistent with that sequence.*

Proof :(Sketch) By Lemma 4 for each committed state σ, σ is derived by applying a block of pending operations (line 2.24) to a previously committed state σ' such that $\sigma'.\mathsf{version} = \sigma.\mathsf{version} - 1$ and for no state $\sigma'' \neq \sigma'$, $\sigma''.\mathsf{version} = \sigma'.\mathsf{version}$. Since the operations within each such pending block are applied in a serial order (lines 2.44–51), there is a well-defined sequence in which operations are applied.□

Theorem 2 (Liveness) *If a run is eventually stable, then every operation submitted by a correct client is performed with probability one, and if performed, its result is returned to the client.*

Proof :(Sketch) Once the system is stable, eventually some correct client q returns from its invocation of contend with probability one. This client executes for sufficiently long (if Δ is chosen adequately) in isolation of other clients. It either commits an existing state σ to a full quorum or else extends σ with operations in *pending* and proposes and commits the new state σ' at a full quorum.□

References

[1] L. Alvisi, D. Malkhi, E. Pierce and M. Reiter. Fault detection for Byzantine quorum systems. In *Proceedings of the 7th IFIP International Working Conference on Dependable Computing for Critical Applications*, pages 357–371, January 1999.

[2] T. E. Anderson. The performance of spin-lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1(1):6–16, January 1990.

[3] R. Alur and G. Taubenefeld. Fast timing-based algorithms. *Distributed Computing* 10(1):1-10, 1996.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[5] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing* 8(1):1–17, 1994.

[6] C. Fetzer and F. Cristian. On the possibility of consensus in asynchronous systems. In *Proceedings of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems*, December 1995.

[7] J. Hastad, T. Leighton, and B. Rogoff. Analysis of backoff protocols for multiple access channels. *SIAM Journal of Computing,* October 1995.

[8] I. Keidar and D. Dolev. Increasing the resilience of distributed and replicated database systems. *Journal of Computer and System Sciences* 57(3):309–324, December 1998.

[9] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st IEEE Hawaii International Conference on System Sciences*, pages 317–326, January 1998.

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

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

[12] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems* 3(2):145–159, May 1985.

[13] D. Malkhi, Y. Mansour, and M. K. Reiter. On diffusing updates in a Byzantine environment. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, October 1999.

[14] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing* 11(4):203–213, 1998.

[15] D. Malkhi and M. K. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 51-58, October 1998.

[16] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering* 12(2):187–202, March/April 2000.

[17] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems* (Lecture Notes in Computer Science 938), pages 99–110, Springer-Verlag, 1995.

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

[19] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22(4):299–319, December 1990.

[20] M. Singhal. A class of deadlock-free Maekawa-type mutual exclusion algorithms for distributed systems. *Distributed Computing* 4(3), February 1991.

[21] M. Singhal. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing* 18(1):94–101, May 1993.