Nested Objects in a Byzantine Quorum-Replicated System
(Extended Abstract)

Charles P. Fry* Michael K. Reiter†

Abstract

Modern distributed, object-based systems support nested method invocations, whereby one object can invoke methods on another. In this paper we present a framework that supports nested method invocations among Byzantine fault-tolerant, replicated objects that are accessed via quorum systems. A challenge in this context is that client object replicas can induce unwanted method invocations on server object replicas, due either to redundant invocations by client replicas or Byzantine failures within the client replicas. At the core of our framework are a new quorum-based authorization technique and a novel method invocation protocol that ensure the linearizability and failure atomicity of nested method invocations despite Byzantine client and server replica failures. We detail the implementation of these techniques in a system called Fleet, and give preliminary performance results for them.

Keywords: Distributed systems, Quorum systems, Byzantine failures, Replication, Fault tolerance

1. Introduction

In this paper we present the design and implementation of a framework to support Byzantine fault-tolerance [14] in a distributed, object-based system. In modern object-based systems, it is commonplace that objects are passed as arguments to and can invoke methods on other objects. A goal of our framework is to support these natural models of object interaction seamlessly from the programmer's perspective, while utilizing object replication and Byzantine fault-tolerant method invocation protocols to mask the Byzantine (arbitrary) failure of a limited number of replicas of each object.

* Department of Electrical & Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA; cfry@ece.cmu.edu

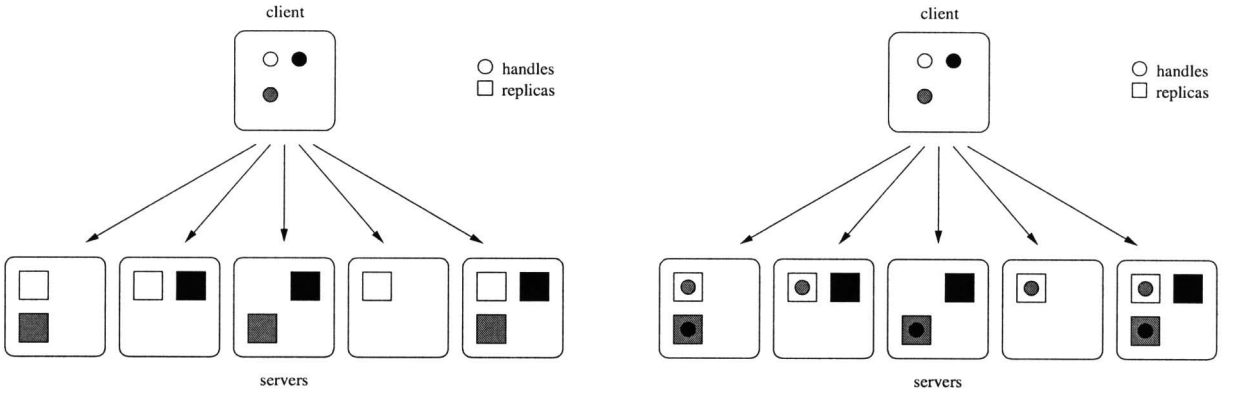
† Department of Electrical & Computer Engineering, Department of Computer Science, and CyLab, Carnegie Mellon University, Pittsburgh, PA, USA; reiter@cmu.edu

The model of object interaction that our framework supports is motivated by that of Java remote objects. A Java *remote object* is one that can be invoked from outside the *Java Virtual Machine* (JVM) in which it resides, via a protocol called *Remote Method Invocation* (RMI). The client JVM of a remote object holds a proxy for the remote object, called a *stub*, that implements the same interface as the remote object. The client program can invoke methods on the remote object by invoking them on the stub, and can pass the stub as a parameter to other, possibly remote, objects. Those objects that then hold the stub can invoke methods on the remote object, as well. This mechanism thus provides location transparency for calls to the remote object.

In this work, we consider a system we are implementing whereby a serializable Java object can be dynamically exported outside the JVM in which it was created, and replicated to a number of other server JVMs, yielding a *distributed object*. After this operation, the client JVM is left with a *handle* (conceptually similar to a *stub*, but functionally different), again that implements the same interface as the original object; see Figure 1(a). Method invocations on the handle are translated to method invocations on a set (*quorum* [15]) of replicas for the distributed object. Like RMI stubs, handles can be passed as parameters to method invocations, potentially on other objects that have been distributed in this way, resulting in object nesting; see Figure 1(b). Those JVMs that hold a handle for the distributed object are called *clients*; however, clients of one distributed object can be servers for other distributed objects, so when we refer to a client or a server, it indicates the role in which it is participating at that time.

With nested objects, it is no longer desirable to allow unfettered access to a distributed object by any client that possesses a handle for that object. In particular, we cannot allow a single client replica to perform arbitrary operations on another distributed object: doing so could result in duplicate method invocations when other client replicas perform the same method invocation, and Byzantine-faulty client replicas could corrupt the embedded object, from the application perspective, by invoking incorrect methods on it. Instead, the central goals of our framework are to ensure that (i) only those method invocations endorsed by correct repli-

Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)
1060-9857/04 $ 20.00 IEEE



(a) Distributed object components (b) Nested distributed objects

Figure 1. Distributed objects

cas of the calling object are performed, and that (ii) the method invocation protocol itself is robust to a limited number of Byzantine faulty client replicas and server replicas.

For (i), we propose an authorization framework for nested method invocations. This framework authorizes method invocations on distributed objects from single trusted clients as well as from quorums of individually untrusted clients. When object o_1 is passed to o_2, authorization for quorums of o_2's replicas to invoke o_1 is transparently delegated with the use of delegation keys and certificates. For (ii), we develop a new quorum based method invocation protocol that ensures linearizable [10] object invocations and failure atomicity of method invocations at arbitrary depths, again despite Byzantine failures of a limited number of replicas of each distributed object. This protocol makes no assumptions on message transmission times, i.e. it is designed to function correctly in an asynchronous environment.

We have implemented our framework within a significant restructuring of the Fleet system [16]. The initial Fleet system from which we began this implementation did not support nested objects seamlessly, provided no tolerance for Byzantine client invocations, and did not implement an authorization scheme. As such, our framework is a significant advance in this context.

2. Related Work

To our knowledge, the only prior system to support linearizable access to nested objects in the face of Byzantine faults is Immune [21]. Immune takes a different approach in that it implements every distributed object using *state machine replication* [23], in which every method invocation is executed by every object replica. Nested method invocations are performed with relative simplicity in the Immune system, as its use of Byzantine fault-tolerant atomic broadcast (specifically [11]) ensures that all client and server replicas receive every communication in the same order. In contrast, our approach implements quorum-based access: each method invocation involves accessing only a randomly selected quorum of replicas, which can be relatively small, e.g., $O(\sqrt{bn})$ replicas for a distributed object with n replicas and tolerating b Byzantine replica failures [17]. While offering improved scalability and load dispersion, the quorum approach introduces challenges not present in the state machine approach, notably the absence of atomic multicast among all client and server replicas to coordinate invocations. Finally, our authorization framework has no analog in Immune, which permits any replicated object to invoke arbitrary methods on another replicated object.

Benign fault-tolerant nesting of transactions has previously been a topic of study in the context of database systems [18]. Nested transactions achieve concurrency atomicity in the form of serializability [19]. While necessary for transactional systems that must apply multiple operations on potentially many objects, serializability is both *non-local* and *blocking*, thus requiring global coordination and locking to enforce it. Serializability was recently studied in the context of JavaSpace transactions [20]; however, while there can be nested transactions in JavaSpaces, there is no analog to our nested method invocations as JavaSpaces contain passive objects that can only be read and written, rather than remote objects on which methods can be invoked.

Linearizability, the form of concurrency atomicity that we pursue in this work, provides concurrency atomicity of single operations performed on a single object only [10]. Though weaker than serializability, we have opted for linearizability for two reasons. First, returning to the object sharing model that motivates our work, linearizability is

the concurrency atomicity property that is achieved by Java RMI (though not in a fault-tolerant way), provided that the remote object processes requests sequentially. As such, our system will be suited to applications already utilizing that model. Second, linearizability can be enforced locally, and avoids locking, which is problematic when a node responsible for unlocking an object fails. Nevertheless, we intend to explore serializability in future work.

Failure atomicity is a second property that is often considered in conjunction with concurrency atomicity, especially in transactional systems such as databases, although it is normally limited to benign failures. Our approach achieves failure atomicity in the face of Byzantine failures, as detailed in Section 4.

3. Authorization Framework

As discussed in Section 1, object nesting in combination with Byzantine failures requires that we depart from a model in which simply possessing a handle for a distributed object is sufficient to invoke methods on it. Otherwise, a faulty object replica which was given a handle for another object through servicing a method invocation would then be able to invoke arbitrary methods on that object. Our goal is to allow the creator of a distributed object to invoke methods on that object, and to delegate that authority to another client object, which itself may be replicated in order to withstand the Byzantine failure of some of its replicas. We anticipate that this delegation will most commonly occur automatically when a handle to one distributed object is passed as a parameter to a method invocation on a second distributed object.

3.1. Assumptions

As discussed previously, the environment that we consider executes methods on a distributed object at a *quorum* of its replicas, and the set of allowable quorums constitutes a *quorum system* for the distributed object. For the purposes of the present section, we are not concerned with the structure of the quorum system, except for one assumption: the quorum system is formed based on an assumed maximum number b of its replicas that will suffer Byzantine failures. For example, it is necessary that each quorum be larger than b, lest operations be performed at a quorum of only faulty servers, and it is also necessary that any b failures leaves a quorum available, so that operations can be completed. Such quorum constructions can be found in, e.g., [15, 17]. We assume that communications to and from servers are protected using standard cryptographic techniques.

At a high level, our strategy will be to permit only a number of replicas of size $b_1 + 1$ or greater of a distributed object o_1 to invoke methods on another distributed object o_2. Here, b_1 denotes the number of replica failures that the quorum system for o_1 was designed to survive. In this way, any invocation by any replica of o_1 that the correct replicas of o_2 accept is corroborated by a correct replica of o_1. For the purposes of this section, we treat trusted individual clients that are permitted to invoke methods on a distributed object, e.g., the creator of the object or another client to which it explicitly passes the handle through an out-of-band mechanism, as a special case of a distributed object o_1 with $n_1 = 1$ replicas and $b_1 = 0$ faults.

3.2. Delegation

The starting point for method invocation authorization is the principal that originally creates a distributed object, i.e., the client that exports the object from its JVM to make a new distributed object. When the distributed object is created, its creator generates a new private digital signing key S and corresponding public verification key V (e.g., [22, 12, 1]) and deploys V with each replica, as the "root" key for the distributed object. The creator can optionally deploy additional root public keys with each replica, though here we restrict our attention to a single root key.

The correct replicas of this distributed object will henceforth only permit method invocations bearing digital signatures that can be verified with V, or for which the root key has delegated authority (perhaps transitively). This delegation can occur in two ways. The most straightforward is explicit delegation by the application, in which the object creator certifies a public key provided by another potential client as being authorized to access the distributed object. This form of delegation closely follows that of, e.g., Gasser and McDermott [9], and will not be detailed here.

The second and more complex form of delegation occurs implicitly, when objects are nested. To support this form of delegation, each handle contains a private signature key called the *handle key*—the handle key for the initial handle is the private root key of the handle—plus a set of *statements* (certificates) regarding the keys for which that handle key bears authority. To fully describe how delegation works, we need to consider two cases: one in which a handle for one distributed object is passed as a parameter into a method call on another distributed object, and one in which a method call on one distributed object returns a handle of another distributed object.

Consider an object o_0 with n_0 replicas, each replica r_0^i of which holds handles h_1^i and h_2^i for distributed objects o_1 and o_2, respectively. Figure 2(a) shows an instance in which $n_0 = 1$, as would be the case if o_0 were a non-replicated client. Let S_2^i denote the private handle key for h_2^i. If r_0^i passes h_2^i in a method call parameter to h_1^i, then h_2^i makes a copy h_2^{ij} of itself for replica r_1^j (i.e., the j-th replica of o_1), except h_2^{ij} is equipped with a newly gener-

ated public/private key pair (V_2^{ij}, S_2^{ij}) before being sent to r_1^j. The results of this delegation in the case of Figure 2(a) are shown in Figure 2(b).

After generating a new public/private key pair for each replica of o_1, a new statement must be constructed authorizing a non-faulty set of o_1's replicas to invoke methods on o_2. Let $\mathcal{V}_2^i$ denote the set $\{V_2^{i1}, \ldots, V_2^{in_1}\}$ of public keys created for o_1's replicas, where n_1 is the number of replicas of o_1, and let b_1 be the number of faults the quorum construction of o_1 is designed to mask. Then for each replica r_1^i of o_1, the statement set of h_2^{ij} is augmented with a statement of the form

$$V_2^i \text{ says } (b_1 + 1 \text{ of } \mathcal{V}_2^i) \Rightarrow V_2^i, \tag{1}$$

i.e., a certificate signed by S_2^i stating that any subset of $b_1 + 1$ keys in $\mathcal{V}_2^i$ is authorized with the same privileges as V_2^i. ("Says" and "speaks for" ($\Rightarrow$) are common formalisms for expressing credentials, e.g., [13, 8, 2, 4, 3].) Verifying a "signature" σ on a message m with $(b_1 + 1 \text{ of } \mathcal{V}_2^i)$ means verifying that at least $b_1 + 1$ of the public keys in $\{V_2^{i1}, \ldots, V_2^{in_1}\}$ can be used to verify signatures in σ (a set) for m.

Finally, once at least $b_0 + 1$ of handles $h_2^{1j}, h_2^{2j}, \ldots, h_2^{n_0 j}$ have been deployed at r_1^j, r_1^j can coalesce these handles into a single handle $\hat{h}_2^j$ by generating a new public/private key pair $(\hat{V}_2^j, \hat{S}_2^j)$ and the certificate[1]

$$\left(\bigwedge_i V_2^{ij}\right) \text{ says } \left(\hat{V}_2^j \Rightarrow \bigwedge_i V_2^{ij}\right) \tag{2}$$

In this way, requests issued by $\hat{h}_2^j$ to o_2 replicas will need only sign with $\hat{S}_2^j$, versus each of $S_2^{1j}, \ldots, S_2^{n_0 j}$. Of course, the statement set of $\hat{h}_2^j$ contains both (2) and the union of the statement sets of $h_2^{1j}, \ldots, h_2^{n_0 j}$, which includes (1).

Safety Consider a method invocation m executed by only faulty handles $\{\hat{h}_2^j\}$, i.e., by at most b_1 replicas $\{r_1^j\}$. Each such invocation is signed by $\hat{S}_2^j$, and so each replica r_2^k that sees this invocation can determine that $\hat{V}_2^j$ says m and thus that

$$\left(\bigwedge_i V_2^{ij}\right) \text{ says } m$$

by (2). However, since there are at most b_1 j's for which this holds true, it is not possible to infer that $(b_1 + 1 \text{ of } \mathcal{V}_2^i)$ says m, or thus that V_2^i says m for any correct r_0^i. Consequently, if any invocation m' to o_2 by o_0 requires $b_0 + 1$ replicas of o_0 to submit m', then any invocation m to o_2 by o_1 will not succeed if only b_1 replicas of o_1 submit m, and safety follows by induction.

Cost The latency of the above delegation is dominated by the costs of (i) generating the n_1 private key pairs $(S_2^{i1}, V_2^{i1}), \ldots, (S_2^{in_1}, V_2^{in_1})$ at r_0^i; (ii) generating the private key pair $(\hat{S}_2^j, \hat{V}_2^j)$ at r_1^j; and (iii) generating the digital signatures for (1) and (2) at r_0^i and r_1^j, respectively. For digital signature schemes for which key generation is costly, notably RSA [22], it would be necessary to pre-generate these keys in the background and store them for use when needed. For this reason, it would be preferable to use a digital signature scheme, such as DSA [12] or ECDSA [1], for which key generation is very efficient [24].[2]

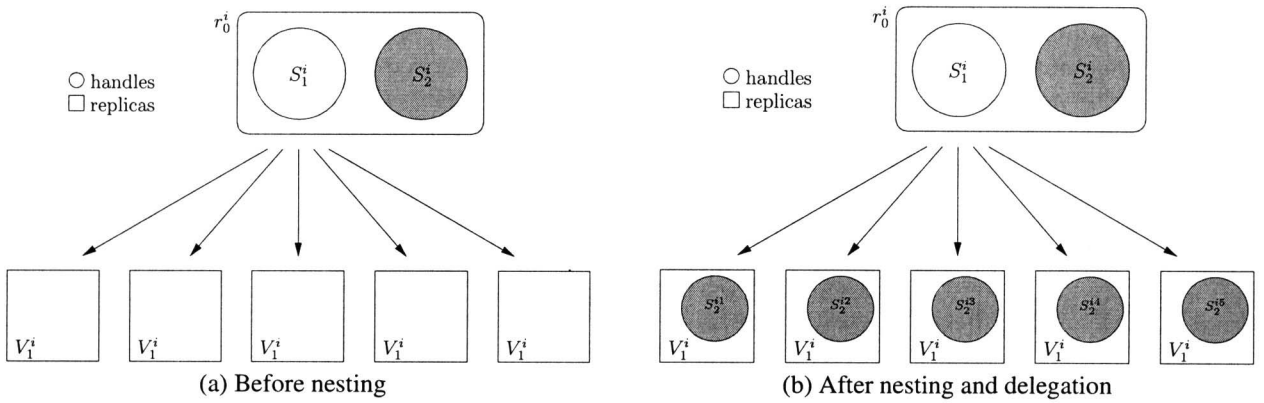
A method invocation following this delegation, i.e., in which object o_1 invokes a method on object o_2, requires each invoking replica r_1^j to digitally sign its request with $\hat{S}_2^j$. Replica r_2^k, upon receiving such a request, must verify not only its signature but also the signatures of the statement sets forwarded with the request. This cost is particularly important since as nesting depth increases, the size of generated statement sets grows. While r_2^k will incur this cost on the first method invocation, caching the verification status of statements (certificates) should significantly decrease the cost of subsequent method invocations. Nonetheless, nested method invocations performed after one object has been nested in another will be dominated by the cost of signature verification, and would thus benefit from a digital signature scheme, such as RSA, where verification was very efficient [24]. As keys can be pre-generated, but not pre-verified, the cost of signature verification is the determining factor in selecting a signature algorithm for use in a practical system.

Returning Handles Our framework accommodates returning a handle from a method invocation on a distributed object in a very similar way. For this, we continue from the above example, and consider that after the above has transpired, replicas of o_0 invoke a method on their corresponding handles for o_1, and in doing so should obtain handles for o_2 that should permit the o_0 replicas to invoke methods on o_2 in the future. (o_0 need not be the same object as in the preceding example, though to simplify notation we consider it to be.) Note that for r_1^j to perform the invocation, it must be invoked by $b_0 + 1$ replicas of o_0, and it will return a copy $\hat{h}_2^{ij}$ of its handle $\hat{h}_2^j$ to the handle h_1^i in r_0^i. Again, however, r_1^j will replace $\hat{S}_2^j$ in $\hat{h}_2^{ij}$ with a newly generated handle key $\hat{S}_2^{ij}$, and add

$$\hat{V}_2^j \text{ says } (b_0 + 1 \text{ of } \hat{\mathcal{V}}_2^j) \Rightarrow \hat{V}_2^j \tag{3}$$

1 The handles $h_2^{1j}, h_2^{2j}, \ldots, h_2^{n_0 j}$ need not all be deployed to r_1^j, and some may not be due to failures. Thus, at any point in time this certificate will concern only the keys S_2^{ij} that r_1^j has received so far, and can be updated when another handle is received.

2 Key generation in DSA is efficient once certain global parameters are fixed, i.e., primes typically denoted by p and q, and a generator g of a subgroup of order q in the integers modulo p. Similarly, ECDSA is typically defined over a fixed curve, which then allows efficient key generation.



(a) Before nesting (b) After nesting and delegation

Figure 2. Nested objects

to the statement set of $\hat{h}_2^{ij}$, where $\hat{\mathcal{V}}_2^j = \{\hat{V}_2^{1j}, \ldots, \hat{V}_2^{n_0 j}\}$ and b_0 is the number of failures the quorum system for o_0 is designed to mask. Just as r_1^j did in the previous case, when handle h_1^i (in replica r_0^i) receives at least $b_1 + 1$ handles $\hat{h}_2^{ij}$, it coalesces these handles to build a handle $\bar{h}_2^i$ with a new handle key $\bar{S}_2^i$ and a statement set including

$$\left(\bigwedge_j \hat{V}_2^{ij}\right) \text{ says } \left(\bar{V}_2^i \Rightarrow \bigwedge_j \hat{V}_2^{ij}\right) \tag{4}$$

h_1^i then returns $\bar{h}_2^i$ from the method invocation, to r_0^i. As before, if only b_0 replicas of o_0 now attempt to perform a method invocation m on o_2, then no correct replica of o_2 will infer ($b_0 + 1$ of $\hat{\mathcal{V}}_2^j$) says m, and so method m will not be invoked. The cost analysis of this case is similar to that above.

Revocation Distributed objects are typically intended to be long-lasting, and in particular, to outlive the clients that create them. As a result, it is not practical for the certificates created during delegation (i.e., (1)–(4)) to expire; doing so would leave distributed objects stranded with no clients able to access them. As a result, we opt for a different form of revocation, conceptually similar to the approach in Gasser and McDermott [9]: when a JVM no longer requires a reference to a handle and so the handle is garbage collected, its private handle key is deleted. This implies that once the correct replicas of a distributed object o_1 have deleted their handles for another distributed object o_2, the delegation that permits o_1 to invoke methods on o_2 can no longer be exercised.

4. Failure Atomicity

Because they span multiple operations, nested method invocations introduce the possibility that one operation succeeds while the operations it induces fail. *Failure atomicity* in this context involves ensuring that all nested method invocations are successfully performed. To ensure that each nested operation is consistently performed by a full quorum of correct server replicas, it is necessary that each server replica in such a quorum receives a copy of the method invocation request from at least $b_c + 1$ client replicas, where b_c denotes the number of faults that the client object is configured to tolerate. A first step toward attaining this goal is for every client replica to send its method invocation request to the same quorum of server replicas. This is done by using a unique method identifier[3] as input to a public uniform hash function that maps method identifiers to quorums.

Unfortunately, selection of the same quorum by every client replica is not sufficient to guarantee failure atomicity, as malicious server replicas could respond selectively to requesting client replicas, allowing some of them to succeed with their selected quorum, while forcing others to select a new quorum. This could prevent any quorum of correct server replicas from receiving the same request from $b_c + 1$ client replicas.

In addition to selecting the same initial quorum to contact, each calling client replica also needs to ensure that even after any failures that may occur on any of the initial server replicas contacted, including failure to respond to requests, a full quorum of correct server replicas will receive at least $b_c + 1$ requests. We accomplish this using *nonblocking Byzantine quorum systems* [5], which guarantee that even when b_s server replicas fail, a response will still be received from a set of server replicas constituting a standard Byzantine quorum [15]. Because of this strong availability guarantee, none of the calling client replicas will ever need to select another quorum to contact due to failures in

3 This method identifier is constructed not only by encoding the method name and argument values, but also by appending a counter value to the method identifier of the method call that "contains" it. This identifier will be detailed in the full paper.

the first selected quorum. This in turn ensures that a standard quorum of server replicas will all receive requests from and send replies to at least $b_c + 1$ client replicas.

Byzantine quorum systems and non-blocking Byzantine quorum systems are similar in that both require a response from a standard Byzantine quorum. They differ in that the former seeks this response by accessing a standard Byzantine quorum, and will thus be prone to the failure of any selected server, while the latter accesses a non-blocking Byzantine quorum, ensuring that even after the failure of some of those servers, the required response will still be received. We distinguish between the two types of quorums involved in non-blocking Byzantine quorum systems as access quorums, to whom requests are sent, and underlying quorums, from which responses are received.

Previous uses of non-blocking quorum systems allow access quorum members to be contacted incrementally [5]. Once a response is received from an underlying quorum, the client need not contact any remaining members of the access quorum. However, when non-blocking quorum systems are used for nested operations, a client replica cannot stop contacting server replicas merely because it has received a response from an underlying quorum. This is because all client replicas need to work together, ensuring that each client replica receives a response from an underlying quorum. When any given client replica receives such a response, it is possible that some of the responses come from malicious server replicas who only respond to a subset of the client replicas. Thus each client replica must continue to contact a full access quorum even after it has received a response from an underlying quorum in order to ensure that the other client replicas will also be able to receive the necessary response.

5. Operation Ordering

As discussed in Section 1, the model of object interaction that we attempt to mimic in our approach is that offered by Java RMI. The concurrency semantics that most naturally characterize Java RMI, presuming that remote Java objects process each method invocation in isolation, is linearizability [10]. For the distributed objects we consider, owing to their replication and quorum based access, we require a method invocation protocol that implements this property. Prior protocols to achieve this property in systems supporting quorum-based access, notably [7], assume a correct client. This assumption is violated in nested object systems, where clients can be replicas of other distributed objects, some of which may be Byzantine faulty.

The approach we take to method invocations here retains the basic structure of prior quorum-based protocols in permitting a single client to drive the ordering protocol, but does so without trusting that client to make any protocol decisions. Rather, this client is used only as a point of centralization to distribute a set of server messages from which the servers work to make protocol decisions. Forms of misbehavior, notably sending different messages to different servers, cannot cause correct servers to take permanent and conflicting actions. In addition, a faulty client cannot prevent progress by falling silent as another client can unilaterally "take over" driving the protocol.

In order to ensure linearizability, all correct servers must apply operations in the same order. They do this by indirectly communicating state with other server replicas in a quorum, and then making deterministic decisions based upon the view they have of the quorum. When a non-faulty client drives the protocol, it echos the state of each server replica in a quorum to the other server replicas in some quorum. Server state consists of the set of pending operations on that replica, and the last distributed object state committed (σ^c), proposed (σ^{pc}), and suggested (σ^s) on that replica.

5.1. Client Protocol

The client side of the method invocation protocol is shown in Figure 3. This protocol consists of two main parts, shown in lines 2–10 and 11–23 of Figure 3, run concurrently as indicated by the "$\|$" in lines 2 and 11. In the first part, the client request ("*op*") is forwarded to a quorum Q drawn deterministically ("$\leftarrow_D$", line 4) from a nonblocking quorum system $\mathcal{Q}^+$, as described in Section 4. The client signs its request ("$S(op)$", line 6) using its private key (the handle key of Section 3) and sends the request to each member of Q. It collects responses ρ into a set R_1 until it has at least $b_s + 1$ replicas (line 8), where b_s is the number of replica failures that the quorum system $\mathcal{Q}^+$ is designed to withstand.

In parallel, the client drives the ordering protocol as shown in lines 11–23 of Figure 3. The client runs this ordering protocol with a particular *rank* r, which is an integer value assigned anew by the chooseRank method each time the client begins the ordering protocol (line 13). In order to make progress, the same rank must be kept during each iteration, so the standard behavior of chooseRank is to leave the current rank unchanged. However, at any point in time each server will only interact with the highest ranked client that has contacted it so far. As such, when a client receives a RankException from a server (not shown in Figure 3), chooseRank selects a new, larger rank for use in the next round of the ordering protocol. A backoff strategy could be used to avoid clients repeatedly interrupting each other; as this does not effect the underlying protocol, we discuss it in Section 5.4.

The core of this part of the client protocol is that it simply queries each server u in a non-blocking quorum $Q_2 \in \mathcal{Q}^+$ (lines 15-16) by querying u.process(R_2, r) (line 17) where R_2 is the set of valid responses gathered from servers in the

```
1.  submit(op)
2.      ||  waiting ← true
3.          R_1 ← ∅
4.          Q_1 ←_D Q^+
5.          ||_{u∈Q_1}
6.              ρ_u ← u.submit(S(op))
7.              R_1 ← ρ_u ∪ R_1
8.          until (∃ρ : |{ρ_u ∈ R_1 : ρ = ρ_u}| ≥ b_s + 1)
9.          waiting ← false
10.         return ρ : |{ρ_u ∈ R_1 : ρ = ρ_u}| ≥ b_s + 1

11.     ||  R_2 ← ∅
12.         repeat
13.             r ← chooseRank()
14.             R_3 ← ∅
15.             Q_2 ←_D Q^+
16.             ||_{u∈Q_2}
17.                 state_u ← u.process(R_2, r)
18.                 if (valid(state_u, R_2, r, V_u))
19.                     Q_3 ← {u} ∪ Q_3
20.                     R_3 ← state_u ∪ R_3
21.             until (∃Q ∈ Q : Q ⊆ Q_3)
22.             R_2 ← R_3
23.         until (waiting = false)
```

Figure 3. Client side of ordering protocol

previous iteration, until it receives valid responses from an underlying quorum $Q \in \mathcal{Q}$ (line 21) where $\mathcal{Q}$ is the underlying quorum system induced by $\mathcal{Q}^+$ (see Section 4). Here, a response *state* is *valid* if the *state* is consistent with the correct execution of u.process(R_2, r), which includes bearing the current rank r, being properly signed by S_u, and bearing values reflecting correct protocol logic when applied to R_2. Validity is tested in line 18, though this is not shown in the figure to avoid duplicating the server-side logic. Although validity testing by the client can not be relied on to ensure correctness (as the client could be faulty), it is important in allowing correct clients to determine when valid responses have been received from a full quorum, without which the next round may be futile.

A malicious client could fail to send messages to some or all of the server replicas, or at the very worst could send different messages to different sets of server replicas. As we will demonstrate shortly, the only possible effect of such misbehavior is that it might temporarily slow progress, but this cannot result in an incorrect linearization of method invocations.

5.2. Server Protocol

The server side of the protocol, as outlined in Figure 4, is necessarily more complicated as it handles all of the protocol's logic. The process method (on the left of Figure 4) is invoked by client replicas' ordering threads, while the remaining methods (on the right of Figure 4) are intended strictly for internal use by a server replica. For simplicity we do not show the submit method invoked by clients' submit threads; this simply adds the submitted operation to the *pending* set once it has been requested by $b_c + 1$ client replicas, and returns a response to the waiting client replicas when the submitted operation has been ordered and executed.

The bulk of the server-side logic is contained in the process method. This method first confirms that the rank of the client is at least as large as any rank seen so far, and throws a RankException and terminates this run of process if not (lines 2–6). After checking the rank and verifying the signatures of the supplied server replica states (line 9), each server replica takes steps to determine which new object state should be *suggested*, *proposed* or *committed* next. It does this on the basis of the following values, which it derives from the server replica states $\{state_u\}_{u \in Q}$. (Note that the server-side logic is only concerned with underlying quorums, thus all references to quorums in this section refer to underlying quorums; non-blocking access quorums are only required by the client for use when contacting servers.)

- σ_g^s (line 13) is the state last suggested by some quorum of servers, if any;
- σ_g^{pc} (line 17) is the state with the highest version number proposed to some correct server or, if there are multiple such states, the one proposed by the highest-ranked proposer;
- σ_g^c (line 21) is the state with the highest version number committed to some correct server; and
- *completed* is the highest version number for which some quorum has committed a state with that version number (line 24), or for which some correct server has suggested (line 14) or proposed (line 19) a state with a strictly higher version number. We say that a state has completed if its version number is less than or equal to *completed* at some correct server.

In summary, the server takes the following actions, in order. It commits σ_g^c if it cannot determine that it has been committed at a full quorum (lines 25–26). Otherwise, it looks at σ_g^{pc} if it exists and is not already completed (line 27): if σ_g^{pc} has been proposed to a full quorum by this client (lines 18, 28), it commits σ_g^{pc} (line 29). If σ_g^{pc} has not been proposed to a full quorum by this client, the server suggests σ_g^{pc} if it has not already been suggested at a full quorum (line 33) and proposes σ_g^{pc} if it has (line 31). If σ_g^{pc} does

```
1.  process({state_u}_{u∈Q}, r)

2.      if (r > maxRank)
3.          maxRank ← r
4.          σ^s ← ⊥
5.      else if (r < maxRank)
6.          throw RankException(maxRank)

7.      foreach u ∈ Q
8.          ⟨content_u, r_u⟩ ← state_u
9.          if (r_u = r ∧ V_u says state_u)
10.             ⟨σ_u^c, σ_u^pc, σ_u^s, pending_u⟩ ← content_u
11.         else Q ← Q \ u

12.     if (∃Q' ∈ Q : Q' ⊆ Q)
13.         σ_g^s ← σ : ∃Q' ∈ Q : Q' ⊆ {u : σ = σ_u^s}
14.         completed ← max{v : |{u : σ_u^s.version > v}| ≥ b_s + 1}

15.         Σ_g^pc ← {σ : |{u : σ = σ_u^pc}| ≥ b_s + 1}
16.         Σ_g^pc' ← {σ ∈ Σ_g^pc : σ.version = max_{σ'∈Σ_g^pc}{σ'.version}}
17.         σ_g^pc ← σ ∈ Σ_g^pc' : σ.proposer = max_{σ'∈Σ_g^pc'}{σ'.proposer}
18.         Q_pc ← {u : σ_u^pc = σ_g^pc ∧ σ_u^pc.proposer = r}
19.         completed ← max{completed,
                max{v : |{u : σ_u^pc.version > v}| ≥ b_s + 1}}

20.         Σ_g^c ← {σ : |{u : σ = σ_u^c}| ≥ b_s + 1}
21.         σ_g^c ← σ ∈ Σ_g^c : σ.version = max_{σ'∈Σ_g^c}{σ'.version}
22.         Q_c ← {u : σ_u^c = σ_g^c}
23.         if (∃Q' ∈ Q : Q' ⊆ Q_c)
24.             completed ← max{completed, σ_g^c.version}

25.         if (σ_g^c ≠ ⊥ ∧ σ_g^c.version > completed)
26.             commit(σ_g^c)
27.         else if (σ_g^pc ≠ ⊥ ∧ σ_g^pc.version > completed)
28.             if (∃Q' ∈ Q : Q' ⊆ Q_pc)
29.                 commit(σ_g^pc)
30.             else if (σ_g^pc = σ_g^s)
31.                 propose(σ_g^pc, r)
32.             else
33.                 suggest(σ_g^pc)
34.         else if (σ_g^s ≠ ⊥ ∧ σ_g^s.version > completed)
35.             propose(σ_g^s, r)
36.         else
37.             pending_g ← {op : |{u : op ∈ pending_u}| ≥ b_s + 1}
38.             if (pending_g ≠ ∅)
39.                 σ ← addOps(σ_g^c, pending_g)
40.                 suggest(σ)

41.     return S(⟨σ^c, σ^pc, σ^s, pending⟩, r)
```

```
1.  addOps(σ', pending)
2.      repeat
3.          op ←_D pending
4.          pending ← pending \ {op}
5.          if (σ'.reflects(op) = false)
6.              σ'.addOp(op)
7.      until (pending = ∅)
8.      σ'.version ← σ'.version + 1
9.      return σ'
```

```
1.  suggest(σ)
2.      if (σ^s = ⊥)
3.          σ^s ← σ
4.      else
5.          throw RankException(maxRank)
```

```
1.  propose(σ, r)
2.      σ.proposer ← r
3.      σ^pc ← σ
```

```
1.  commit(σ)
2.      if (σ.version > σ^c.version)
3.          σ.applyOps
4.          σ^c ← σ
5.          pending ← pending \
                {op : σ.reflects(op) = true}
6.          response ← response|σ.response
```

Figure 4. Server side of ordering protocol

not exist or is already completed, then the server similarly examines σ_g^s, proposing it if it exists and is not already completed (lines 34–35). If σ_g^s does not need to be acted upon, the server deterministically orders method invocations that are pending on $b_s + 1$ servers—appending them to the current state, but not yet executing them; see below—through the addOps operation, and suggests this state (lines 37–40). We note that in addOps, the predicate σ.reflects(op) indicates whether or not the operation op is incorporated into the state σ; as such, it prevents duplicate invocations. Finally, the server signs its current local states that it sees as committed, proposed, and suggested, as well as all pending invocations and the rank r, and returns this (line 41).

It is important to note that the addOps operation (line 39) does not actually invoke method invocations on the object σ_g^c; rather, they are just appended for later application by applyOps (line 3 of commit). With the introduction of nested operations, the scope of impact of a method invocation on an object is no longer limited to that object. Each invocation can result in a nested method invocation on an-

other object. As such, it is important to withhold performing operations until they are actually committed, as a suggested or even a proposed state may never ultimately be committed.

5.3. Correctness

Space limitations preclude a full proof of linearizability for this protocol. However, we note several important lemmas for the proof. As all of the protocol logic is performed on the server side, all of the following lemmas refer to Figure 4, and all quorum references are to underlying quorums.

The first important lemma is that there is only one object state σ that can be proposed at a rank r; i.e., the proposed state for rank r is unique. Informally, this is true since any proposal at rank r must first be suggested at a full quorum by a client with rank r, and since correct servers suggest only one state per rank (note line 4 of process and line 2 of suggest).

Lemma 5.1. *Suppose that $\langle \sigma_1, r_1 \rangle$ is proposed at a correct server replica, and $\langle \sigma_2, r_2 \rangle$ is proposed at a correct server replica. If $r_2 = r_1$, then $\sigma_2 = \sigma_1$.*

A second important lemma is that once an object state σ is proposed to a full quorum, any higher-ranked proposal for the same version number will also propose that object state. In other words, not only is the object state per rank uniquely defined, but so is the object state proposed at a full quorum per version number.

Lemma 5.2. *Suppose that $\langle \sigma_1, r_1 \rangle$ is proposed at a full quorum, and $\langle \sigma_2, r_2 \rangle$ is proposed at a correct server replica. If $\sigma_2.\mathsf{version} = \sigma_1.\mathsf{version}$ and $r_2 \geq r_1$, then $\sigma_2 = \sigma_1$.*

Third, any state committed at a correct server must be proposed to a full quorum. Combined with the above lemmas, the following lemma thus implies that there is a well-defined sequence of object versions that are committed.

Lemma 5.3. *If σ is committed at some correct server replica by a client with rank r_2, then $\langle \sigma, r_1 \rangle$ for $r_1 \leq r_2$ was proposed at a full quorum.*

Now, linearizability easily follows from the following lemma, which argues that one object state is built from the previous by applying pending operations.

Lemma 5.4. *If σ_2 is suggested at some correct server replica, then it extends the state σ_1 that is committed at a full quorum with $\sigma_2.\mathsf{version} = \sigma_1.\mathsf{version} + 1$, by applying operations in $\{op : \sigma_2.\mathsf{reflects}(op) \wedge \neg\sigma_1.\mathsf{reflects}(op)\}$ to σ_1 in some sequential order.*

5.4. Liveness

Our discussion so far has focused on safety. Liveness is also an important consideration, especially as a Byzantine client could end up driving the protocol. It is thus desirable to tolerate faulty clients while still achieving high throughput with many concurrent clients. A first step towards this is to require clients to sign their ordering requests with the private key of the associated handle. In conjunction with the delegation statements defined in Section 3, this would ensure that the ordering protocol for a given distributed object could only be driven by handles which were authorized to invoke methods on that object.

In addition to ensuring that only authorized handles can drive the ordering protocol, it is also necessary to prevent Byzantine clients from doing all the driving and thus preventing forward progress. We do this with a form of backoff similar to [7], though enforced by server replicas to prevent faulty clients from repeatedly interrupting the protocol for correct clients. While a faulty client can delay the ordering protocol during a single round, enforced backoff ensures that correct clients (who are the majority due to quorum overlap requirements) will regularly be allowed to drive the protocol. Because operations are applied en masse, high average throughput is maintained by the system despite small delays during rounds which are driven by Byzantine clients. We present a complete analysis in the full paper.

5.5. Performance

We have implemented the protocol discussed in Sections 5.1 and 5.2 within the context of a substantial revision of the Fleet system. Fleet is a distributed object store built in Java that implements distributed objects such as those shown in Figure 1(a). However, Fleet previously did not provide support for transparent nesting; in fact, nesting could result in duplicate nested method invocations, even in the absence of failures [16]. As such, our framework provides a useful extension to the Fleet system.

In our experiments, each server and client was equipped with dual Pentium-III 1GHz processors, running Linux 2.4.24 SMP and Java HotSpot™ Server VM 1.4.2. The servers and client were connected by 100Mbps Ethernet and utilized TCP for all communication; multicast was not used. Key generation, signing, and signature verification were all performed natively using the Crypto++ Library. All tests were performed with three signature algorithms: RSA [22] with 1024-bit keys, DSA [12] with 1024-bit keys and ECDSA [1] with 160-bit keys. By way of comparison, we also adapted our protocol to use HMACs [6], using Diffie-Hellman key agreement to establish a shared secret key between each pair of servers.

The non-blocking quorum system in use was the extension of the Paths system [17] proposed by Bazzi [5], and all method invocations were performed by unreplicated clients.[4] In order to measure the overhead introduced by the

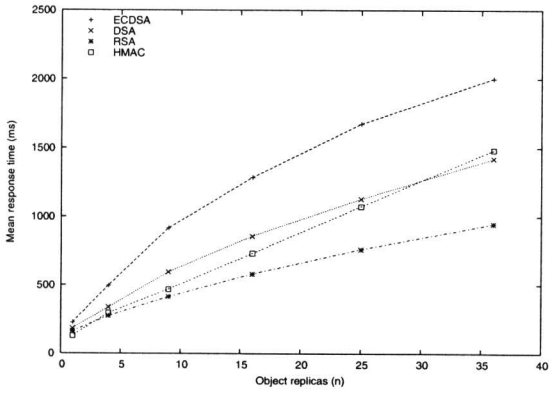


Figure 5. Mean response time

ordering protocol, all method invocations performed inexpensive tasks, such as get and set. Due to the limited set of homogeneous computers at our disposal, we only tested distributed objects having between $n = 1$ and $n = 36$ replicas and with $b = 0$, yielding quorums of sizes from $q = 1$ to $q = 20$.

Preliminary response times as seen by a client for a relatively unoptimized implementation of this protocol are shown in Figure 5. Among the three digital signature algorithms tested here, the use of RSA signatures, whose verification times are notably faster than the other signature algorithms [24], yielded the lowest mean response time. This result reflects the fact that each server replica in a quorum must sign its own state, but verify signatures on states from every server in a quorum. Note that the response time growth rate is sublinear while b stays constant, due to the fact that quorums in the chosen quorum system are asymptotically of size $O(\sqrt{bn})$.

Prior to analyzing the performance of HMACs, it is necessary to understand how their implementation differs from that of signatures. As part of the ordering protocol, each server replica signs its local state and sends it to the driving client replica (Figure 4, line 41), who then distributes it to a quorum of server replicas (Figure 3, lines 16–17). When using HMACs instead of signatures, it is necessary for each server replica to include an HMAC for every other server replica, and for the driving client replica to include the appropriate set of HMACs for each server replica that it contacts. HMACs thus consume more network bandwidth between server replicas and the driving client replica, as the number of server replicas grows. Considering this, it is unsurprising that the response time using HMACs does not scale as well as signatures as n increases.

4 While replicated clients are an important component of our work, the ordering protocol is driven by a single client replica at any given time.

Another difference resulting from HMACs is that the client is unable to verify them (unlike signatures). As such, a faulty server that returns invalid HMACs will not be readily detectable to a client, and may impinge on the client's ability to complete the protocol. In such circumstances, the client can "fall back" to a signature-based protocol to better enforce progress.

6. Conclusions and future work

The delegation architecture which we have presented can be used to arbitrarily nest distributed objects, while ensuring correct system behavior despite a limited number of Byzantine failures in the object replicas. This is accomplished through the creation of delegation keys and delegation certificates that work together to authorize the invocation of methods on nested objects. Method invocation capabilities on a specific distributed object can be delegated to other clients explicitly, and are implicitly delegated to permit nested method invocations by sufficiently large groups of replicas to succeed transparently.

Through the use of non-blocking Byzantine quorum systems, failure atomicity can be ensured for nested object interactions. Specifically, at every level within a chain of nested method invocations, enough server replicas will be activated to succeed at performing the next invocation in the chain. Further, as long as non-blocking quorums are used at each level in such a chain, the success of a method invocation at one level will not be followed by the failure of the parent operation.

Finally, we presented a novel protocol to achieve linearizability of nested method invocations. Our protocol tolerates Byzantine faulty clients, as is necessary when methods are invoked from other distributed objects that themselves must withstand Byzantine faults. We detailed this protocol, argued its correctness, and described its implementation and performance in a distributed object system.

An extension that we are presently studying is the nested *creation* of distributed objects, i.e., where one distributed object creates another. Our framework provides a basis for supporting distributed object creation, but requires us to address additional issues. We will report on this progress in future work.

References

[1] ANSI X9.62, *Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, American National Standards Institute, 1999.

[2] A. W. Appel and E. W. Felten. Proof-Carrying Authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, November 1999.

[3] D. Balfanz, D. Dean, and M. Spreitzer. A security infrastructure for distributed Java applications. In *Proceedings of 2000 IEEE Symposium on Security and Privacy*, May 2000.

[4] L. Bauer, M. A. Schneider, and E. W. Felten. A General and Flexible Access-Control System for the Web. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[5] R. A. Bazzi. Access cost for asynchronous Byzantine quorum systems. *Distributed Computing* 14(1):41–48, 2001.

[6] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 1–15, 1996.

[7] G. Chockler, D. Malkhi, and M. K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 11–20, April 2001.

[8] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. SPKI Certificate Theory, September 1999. RFC2693.

[9] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 20–30, May 1990.

[10] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12(3):463–492, 1990.

[11] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security* 4(4), November 2001.

[12] D. W. Kravitz. Digital signature algorithm. U.S. Patent 5,231,668, 27 July 1993.

[13] B. Lampson, M. Abadi, M. Burrows and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems* 10(4):265–310, November 1992.

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

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

[16] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the Fleet system. In *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition*, Vol. II, pages 126–136, June 2001.

[17] D. Malkhi, M. K. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. *SIAM Journal of Computing* 29(6):1889–1906, 2000.

[18] J. E. B. Moss. Nested transactions: An approach to reliable distributed computing. Ph.D. Thesis, Massachusetts Institute of Technology, May 1981.

[19] J. E. B. Moss. An Introduction to Nested Transactions. COINS TR 86-41, University of Massachusetts, Department of Computer Science, September 1986.

[20] N. Busi and G. Zavattaro. On the Serializability of Transactions in JavaSpaces. In *Proc. of International Workshop on Concurrency and Coordination*, Electronic Notes in Theoretical Computer Science, Vol. 54, July 2001.

[21] P. Narasimhan, K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith. Providing support for survivable CORBA applications with the Immune system. In *Proceedings of the 1999 IEEE International Conference on Distributed Computing Systems*, pages 507–516, May 1999.

[22] 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, Feb. 1978.

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

[24] M. J. Wiener. Performance Comparison of Public-Key Cryptosystems. *CryptoBytes* 4(1):1–5, 1998.