Replicated Invocations in Wide-Area Systems

Arno Bakker* Maarten van Steen Andrew S. Tanenbaum
{arno,steen,ast}@cs.vu.nl

1 Introduction

In many object-oriented distributed systems a client invokes a method of a distributed object through a call on a local proxy of that object. If a client is replicated, such a call may possibly be done by several of its replicas. Consequently, what is conceptually a single method invocation may result in several identical calls on the same object. Such an invocation is said to be *replicated*, the individual calls are referred to as the *copies* of the invocation.

The problems associated with replicated invocations are well known [3][4]. However, existing solutions depend on global coordination, or, for example, assume group communication is fast and efficient, so that they can be applied only in local-area networks.

In this paper we present the problems associated with replicated invocations in the context of wide-area systems and present a new solution to these problems. Our solution allows massively and widely replicated objects to efficiently call other massively and widely replicated objects using replicated invocations. Our solution is scalable in the sense that it uses local communication whenever possible, requires no complete knowledge of group memberships and puts only a small load on replicas of the called object.

For the moment we limit ourselves to a single replicated object making replicated invocations on a single replicated object. We assume that both calling and called object use active replication [6]. Nodes in the system are assumed to be fail-stop. Only state-modifying operations (*write methods*) are carried out by all replicas. Read methods are sent to a single (nearby) replica. For this purpose each proxy is (implicitly) connected to the nearest replica of the object it represents. Write methods modify only the (volatile) state variables of the object; we assume they have no other side effects. Write methods are guaranteed to be carried out by each replica in the same global order.

The remainder of the paper is organized as follows. Section 2 describes the problems introduces by replicated invocations in a wide-area context. Section 3 gives an overview of our solution as it would operate in the absence of node failures. Section 4 describes the fault tolerance of the solution. Section 5 discusses related work and we finish by presenting conclusions and future work.

2 Problem Description

There are three major problems associated with replicated invocations.

Problem 1: Potential State Corruption If a request is not recognized as a copy of a previous invocation the associated method may be carried out more than once. If the method is nonidempotent the state of the called object will be corrupted.

Problem 2: Returning the Same Answer A single conceptual method invocation has a single conceptual answer. Therefore, each copy of a method invocation should return the same answer. If copies are considered separate invocations, different copies of the same invocation can return different results.

To illustrate this point, consider a replicated object A with a write method MA that successively invokes two methods of a replicated object B, MB_{read} and MB_{write}. Assume that the copies of the MB_{read} invocation are not recognized as copies. The MB_{write} invocation is sent to all replicas of B. Due to variable delays in the network, it may now happen

*Postal address: Arno Bakker, Department of Mathematics and Computer Sciences, Vrije Universiteit, De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands.

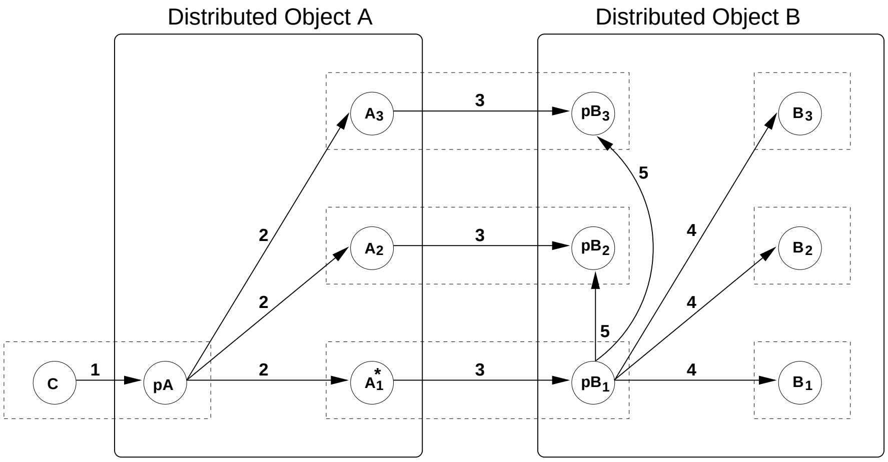

Figure 1: A client C invokes a write method on a distributed object A which, in turn, invokes a write method of a distributed object B. Dotted boxes represent processes. Solid boxes represent object boundaries. The asterisk indicates that replica A_1 is *primary caller*.

that some of the copies of the MB_{read} are delivered to a replica of B *before* it carried out the write and others *after* it carried out the write. Consequently, different copies of the MB_{read} may be carried out on different versions of the state.

Even if copies are recognized, returning the same answer is hard to implement in an efficient and scalable manner. The time between the start of a replicated invocation and its completion can be considerable. Keeping answers to (potentially large numbers of) unfinished invocations for long periods of time is expensive, as is determining if an answer can be removed from the history.

Problem 3: Large Number of Messages If the calling object is actively replicated, each of the calling object's replicas makes the call on the local proxy. If the method invoked is a write method and the called object is actively replicated, the invocation is forwarded to all its replicas. Given that the calling and called object can both have a large replication degree, this could result in an excessive number of request and reply messages.

3 A New Solution

In this section we give an overview of our solution as it would operate in the absence of node failures.

We have made the following assumptions designing this solution:

1. Communication is atomic and reliable: when a node sends a message it will be delivered to all its destinations.
2. There are no network partitions.
3. Proxies are able to use local knowledge to determine which version of the state a read method should use.
4. Replicas are deterministic, that is, invoking the same method on replicas with the same version of the state returns the same result.
5. All clients are bound to the object they will call, that is, they have a local proxy for that object.

Furthermore, we assume that the sets of replicas do not change during the invocations.

3.1 Overview of Failure-Free Operation

3.1.1 Request Phase of the Replicated Invocation

Figure 1 depicts the request phase of a replicated invocation by a distributed object A on a distributed object B. Both A and B are assumed to be actively replicated. The replicated invocation of B's method is the result of the execution of one of A's write methods.

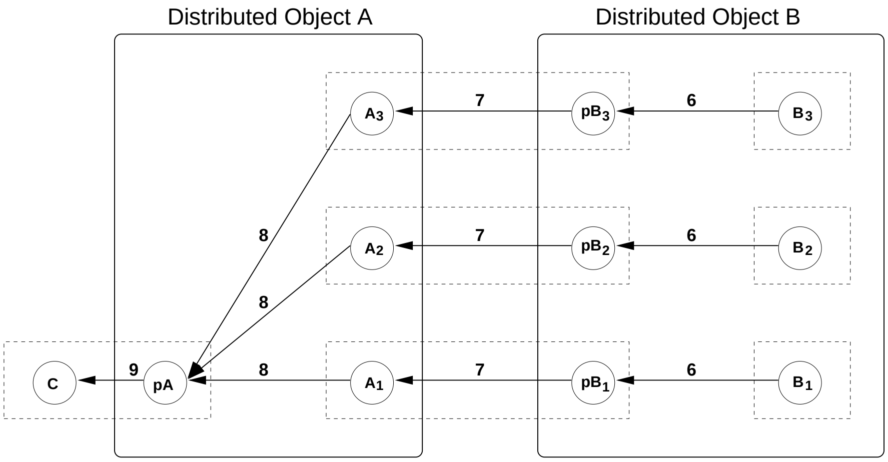

Figure 2: The results of the chain of method invocations started by client C are returned. Again, dotted boxes represent processes and solid boxes represent object boundaries. See Figure 1.

A's write method is invoked by client C through a call on the local proxy of A (pA) (arrow 1 in Figure 1). The local proxy multicasts the invocation request to the replicas of A (arrow 2). In this message it designates one of the replicas as *primary caller*. When the message is delivered, the replicas of A record if they were named primary caller and start to execute the write method. Part of the execution of object A's write method is invoking one of the write methods of object B. The replicas of A invoke this method through a call on a local proxy of B (arrow 3). This call carries extra information, notably a call ID and a flag, indicating whether or not this replica of A was named primary caller in the request message. Each replica of A assigns the same call ID to the call.

The proxy of B that is invoked with the primary-caller flag turned on (pB_1) sends two messages: one invocation request to all replicas of B (arrow 4), and one so-called "hush" message to the other proxies of B used by object A (pB_2 and pB_3) (arrow 5 in Figure 1). The "hush" message contains the call ID of the invocation and results of previous replicated invocations by object A[1]. These results are taken from a local answer log in which each proxy of B records the answers to previous replicated invocations by A, either received directly from a replica of B or sent by another proxy of B in an earlier "hush" message.

The proxies of B that were invoked with the primary-caller flag off check their answer log to see if they already received the answer to this write-method invocation on B. If so, they return it to the replica of A, as described in the next section. Otherwise, the proxies of B block and wait for the "hush" message. When it is received, they block again and wait until they receive the result of the write-method invocation from the replicas of B.

3.1.2 Reply Phase of the Replicated Invocation

Each of the replicas of B carries out the write method as soon as the invocation request (arrow 4 in Figure 1) is delivered. When the replica is finished it returns the result[2] of the method invocation to a subset of the proxies of B (arrow 6 in Figure 2).

The proxies of B record the result in their answer log. If the proxies were waiting for the result of this invocation they return it to their callers, the replicas of A (arrow 7). The replicas of A continue the execution of A's write method. When finished, the replica of A that was named primary caller in the request message immediately returns the result of the write-method invocation on A to the proxy pA (arrow 8). All other replicas of A return the result to pA only if they have not seen any new invocations coming from that proxy for a long time.

Finally, the proxy pA returns the result to client C (arrow 9 in Figure 2).

[1]It contains the result of the last write-method invocation by object A and the results of any subsequent replicated read-method invocations by that object.

[2]A write method is a normal method therefore it may also return a result.

3.2 Replicated Invocation of a Read Method

Assume the method invoked by distributed object A was a read method of distributed object B instead of a write method. In that case, the local proxies of B first consult their answer log to check if they already received the result of this invocation. If not, each proxy sends an invocation request to what it considers the nearest replica of B.

This invocation request contains the version number of the state on which the read should be carried out. The replica of B receiving the request checks if it still has this version of the state. If so, it carries out the method and subsequently sends the result back to the proxy of B. Otherwise, it sends a message saying the method was not carried out because the desired version of the state was no longer available.

In the former case the proxy records the result in its answer log and returns the result to the replica of A. In the latter case the proxy queries the other proxies of B used by A for the result, or uses the result sent in a subsequent "hush" message and returns it to the replica of A.

3.3 Discussion

Preventing State Corruption Write-method invocations are sent to the replicas of the called object (B) only once and can therefore be executed only once, given our assumptions. The call IDs are necessary for two reasons. First, the copies of a replicated invocation need to be identified as copies, so we are able to return the same result to each of them. Second, due to node failures, there may be multiple proxies of B multicasting a write, as discussed in Section 4.2.1.

Returning the Same Answer Call IDs ensure that copies of replicated method invocations are recognized as such. State-version numbers, multicasting previous results (in the "hush" messages), and a global order on writes[3] ensure that all proxies indeed return the same result. Note that this solution has the advantage that a write-method invocation can be delivered to a replica of B even if this replica has not seen all copies of previous read-method invocations. The responsibility for providing answers to these requests is shifted to the proxies of B used by object A. This has the additional advantage that the replicas do not have to allocate resources for this task. This shifting of responsibility from server-side to client-side was already suggested by Cooper [2].

Large Number of Messages Distributing the invocation of a write method of object B takes two messages: one invocation request sent to all replicas and one message containing previous answers to the proxies of B used by object A (the "hush" message). They are both multicast to a potentially large and wide-spread group. This is expensive but one might be able to efficiently combine the two multicasts into one. Since object A depends on object B, it makes sense to install replicas of A and B close together. Consequently, the proxies of B used by A and the replicas of B will be relatively close together. Therefore, the two messages will be multicast to the same parts of the network, making it efficient to combine them into one. The amount of request-related traffic could therefore be relatively small.

The amount of reply-related traffic is also relatively small. Each replica of B replies only to a subset of the proxies of B (arrow 6 in Figure 1[4]), in particular to those proxies of B that consider this replica the nearest replica of B. This means that the reply traffic will be local. The proximity of the proxies to the replica also makes maintaining the set of proxies to reply to relatively easy.

Except for a proxy knowing its nearest replica (and vice versa), our solution requires little explicit knowledge about group membership. We coordinate actions by assigning primary callers, not by election based on global knowledge of a complete and ordered list of group members. The lower-level multicast service used requires complete knowledge about the group membership, of course.

4 Fault-Tolerance of the Solution

In this section we show that our solution to the problems associated with replicated invocations is fault tolerant. A node can host multiple processes. When a node fails, all processes on it are assumed to fail as well. A process can fail independently from other processes on the same node. A process is assumed to fail as a whole, that is, if a process fails all replicas and/or proxies present in that process also fail.

[3] Since we currently only consider a single replicated object calling another single replicated object, creating a global order is simple. We are investigating the effects of having multiple replicated clients calling a single object. See Section 6.

[4] In this case the subsets are all singleton sets.

4.1 Problems Caused by Process Failures

The failure of a process may or may not affect our solution depending on when the failure occurs. In particular, it depends on whether a process fails before or after sending the required messages. There are five different message types, named after their labels in Figures 1 and 2.

Type 2: A method invocation sent by a proxy of A to one (read method) or all (write method) replicas of A.
Type 4: A method invocation sent by a proxy of B to one (read method) or all (write method) replicas of B.
Type 5: A "hush" message sent by a proxy of B to all other proxies of B used by A (sent as part of a write on B).
Type 6: A reply sent by a replica of B to a subset of the proxies of B used by A.
Type 8: A reply sent by the replicas of A to the proxy of A.

In the next sections we investigate the effects of these messages not being sent due to a crash of the sending process. Recall that invocations of write methods are sent to all replicas of an object A, but the invocation of a read method is sent to only one replica of A. As a consequence, when object A, in turn, invokes a read method of an object B as part of the execution of one of A's *write* methods, the read-method invocation will have n copies since the write method of A will be carried out by all n replicas of A.

However, when the read method of B is invoked as part of the execution of one of A's *read* methods, the invocation has only one copy since the read method of A is being carried out by only one replica of A. We call this latter invocation a nonreplicated or normal read-method invocation. Only invocations of read methods can be nonreplicated. We assume that read methods of an object cannot call write methods of other objects.

4.1.1 Type 2 Message Not Sent Due to Process Failure

When the process containing client C and proxy pA fails before sending the method invocation to the replicas of A (see Figure 1), the replica(s) of A will, of course, not carry out the read or write method. But since client C failed along with the process, no further action is necessary.

4.1.2 Type 4 Message Not Sent Due to Process Failure

Type 4 messages can be divided into three subtypes:

4i. a nonreplicated invocation of a read method of B.
4ii. a replicated invocation of a read method of B.
4iii. a (replicated) invocation of a write method of B.

A nonreplicated invocation of a read method is always part of a chain of read-method invocations, in case of Figure 1 $MA_{read} \rightarrow MB_{read}$. Therefore, if the process hosting the single proxy of B that was invoked (proxy pB_1 in Figure 1) fails before sending the Type 4i message, there is a problem. The caller at the start of the chain of invocations (in this case pA) will not get an answer. We call this problem F1.

A replicated invocation of a read method of B has more than one copy. Since reads require only a unicast message to a single nearby replica, we let all proxies of B invoked by A send a request message to what each proxy considers the nearest replica of B. Therefore, if a process hosting one of the proxies fails and the Type 4ii message is not sent, there will be other proxies sending the same request to a replica of B. Therefore the read method will be carried out and a result will be delivered to a replica of A.

A replicated invocation of a write method of B also has multiple copies, but since a write requires a multicast to the whole replica group of B we let only one proxy of B (the one invoked with the primary-caller flag turned on) send the Type 4iii message. If the process hosting this proxy crashes before sending the message, the write on B will not be carried out and object A will never receive an answer. We call this problem F2.

4.1.3 Type 5 Message Not Sent Due to Process Failure

A Type 5 message is one of the mechanisms we introduced to make our solution tolerant to process failures. We discuss the effects of its loss in Section 4.2.2 when we discuss the solution to problem F2.

4.1.4 Type 6 Message Not Sent Due to Process Failure

Type 6 messages can be divided into three subtypes:

6i. result of a nonreplicated invocation of a read method of B (i.e., the reply to a Type 4i message).
6ii. result of a replicated invocation of a read method of B (i.e., the reply to a Type 4ii message).
6iii. result of a (replicated) invocation of a write method of B (i.e., the reply to a Type 4iii message).

If the reply to a nonreplicated read-method invocation on B is not sent, the process hosting the replica of B that was asked to carry out the method failed (that would be the process containing replica B_1 in Figure 2). This means that the proxy of B that sent the request (pB_1) will not receive the result of this invocation. We call this problem F3.

When a Type 6ii message is not sent because a process hosting a replica of B failed, there are still other replicas of B that send the reply to their invoking proxies. As long as one copy of a Type 6ii message is sent, object A will get an answer. However, it is important that all proxies of B used by A receive a copy of a Type 6ii message to preserve the fault tolerance of object A. We will call this problem F4.

Type 6iii messages suffer from the same problem.

4.1.5 Type 8 Message Not Sent Due to Process Failure

Type 8 messages can be divided into two subtypes:

8i. result of a read-method invocation on A.
8ii. result of write-method invocation of A.

Both types of messages carry results from method invocations sent in Type 2 messages.

If the process containing the replica of A that is considered the nearest replica of A by proxy pA fails before sending the result of the read, proxy pA will not get an answer. This problem is equivalent to F1. In both cases the proxy pA does not get the results of an invocation of a read method.

Messages of Type 8ii have natural redundancy (see Figure 2), therefore not sending such a message does not pose a problem.

4.2 Solutions to the Problems Caused by Process Failures

4.2.1 Problem F1: Proxy of A does not receive results of a nonreplicated read-method invocation

The synchronous[5] nature of method invocations ensures there are no successive write operations. Therefore, if the replica of A failed before returning an answer, pA can resend the method invocation to a different replica of A, since there is no risk of the read being carried out on a newer version of the state. However, this other replica may not have the latest version of the state. Proxy pA therefore puts the version number of the state on which the read should be carried out in the invocation message. The read is queued until the prior write-method invocations have been carried out.

4.2.2 Problem F2: Primary proxy of B fails before sending write-method invocation

The basic solution to F2 is to exploit the redundancy of the replicated invocation. The proxy of B that was invoked with the primary-caller flag turned on (pB_1 in Figure 1), multicasts the invocation of the write method of B immediately. The other proxies of B used by A set a timer and wait. If the timer expires before receiving the "hush" message, the proxies multicast the method invocation. If they do receive the "hush" message in time they will regard this as proof that the primary proxy did not fail and successfully sent the method invocation to all replicas of B. More on timers in Section 4.2.6.

To prevent large amounts of network traffic when the primary proxy fails, the "hush" mechanism should be applied recursively. The first non-primary proxy of B that times out and multicasts the method invocation also sends a "hush" message to the remaining proxies of B. A similar solution, but without timers was presented by Jalote in [4]. It is obvious that multiple proxies of B may be multicasting the invocation. The call ID present on all copies of a replicated invocation (to ensure that all copies of a replicated invocation return the same answer) is used by replicas of B to filter out copies of invocations already carried out. See Section 4.2.5.

[5] We do not consider asynchronous method invocations in this paper.

4.2.3 Problem F3: Replica of *B* fails before returning results of a nonreplicated read

Problem F3 is very similar to problem F1 and therefore the same arguments apply and the same solution can be used. Due to the synchronous nature of method invocations proxy pB_1 can, after detecting the failure of the nearby replica, resend the read-method invocation to another replica of *B*. The version number of the state on which the read should be carried out should, of course, be included in the invocation message.

4.2.4 Problem F4: Some replies to a replicated read or write not sent

When a replica of *B* fails during a replicated read (i.e., Type 6ii message is not sent) the proxy of *B* that considered it the nearest replica of *B* will have to find another nearest replica. The proxy might or might not be able to redo the read on the new nearest replica, because successive writes on the object forwarded by another proxy of *B* may have changed the state of this replica. In that case, the proxy queries its neighbouring proxies for the result of this invocation (using, for example, a scoped multicast). This action is concurrent with subsequent write-method invocations by *A* (which might clear a proxy's answer log), but the querying proxy can always fall back on the previous read results in the "hush" message that is sent as part of the subsequent write (see Section 3.1.1).

Redoing a write is, of course, impossible. Therefore the proxy of *B* has to query its neighbouring proxies. To allow it to fall back on a subsequent "hush" message, "hush" messages carry the results of the last write done by object *A*.

4.2.5 Call IDs

Call IDs as passed by the replicas of *A* to the proxies of *B* are monotonically increasing. This makes filtering out copies easy. If the copy has a call ID less or equal than the last call ID processed *and* the operation invoked is a write, the copy is discarded. If it is an copy of a replicated read we check if we still have the version of the state that is required and act accordingly.

The state machine approach requires a total order on writes, which means that a replica, given two writes, should be able to determine which write should be carried out first. And more importantly, if there are other writes which should go first. Having monotonically increasing call IDs is not sufficient, therefore the proxies of *B* include the call ID of the last write done by *A* in the write request to the replicas of *B*.

4.2.6 Timers

For optimal performance during failures, the proxy of *B* closest to the primary should take over as soon as it is clear that the primary did not multicast the write-method invocation. If this proxy, in turn, fails, the proxy closest to it should take over, and so on. Setting the timer values for optimal performance therefore requires detailed knowledge about transmission times between nodes to be available at each proxy. This is not feasible for large numbers of widely distributed proxies (i.e., a massively and widely replicated object *A*). We therefore adopt a very simple solution: proxies choose large random values for their timers.

5 Related Work

In the GARF system [5] the problems associated with replicated invocations are solved as follows. Incarnations of the replicated invocation are intercepted by a meta layer, which elects one invoking replica *coordinator*. To elect a coordinator it must have full knowledge about the group membership. This coordinator makes the actual invocation on the other object. A symmetric approach is used for returning the result: the coordinator of the called object returns the result to a local proxy of the *calling* object, which multicasts it to the calling object's replicas. Maintaining complete knowledge about group membership at each node is not feasible in wide-area systems. Furthermore, using group communication for invoking and returning the results of both read and write methods is inefficient. The Hot Replication algorithm [1] is similar to that used in GARF, but does not use a symmetric approach. The results of reads and writes are multicast to the calling object's replicas by the coordinator of the calling object instead of a proxy.

The DistView toolkit [7] uses a solution also very similar to that of GARF, explicitly designed for widely distributed objects. It is also similar to our solution in that each replica of the calling object calls the other object. This is only exploited if calling and called object are replicated on the same set of nodes. In all other cases method invocations and results are multicast by fixed coordinators, whose election is also based on having complete group membership knowledge at each node.

In Jalote's solution [4] a request or reply is sent to both the replicas of the called object and the proxies of the called object used by the caller. This prevents unnecessary request and reply messages, because the proxies will not multicast the request themselves if they already received it from another proxy. The same holds for replicas and replies. An advantage of this approach is that node failures are transparent to the invocation algorithm. Jalote's solution is targeted towards broadcast local-area networks and will not work correctly in networks with highly variable delays, but the basic ideas are very useful (see Section 4.2.2).

6 Conclusions and Future Work

If a distributed object that is actively replicated invokes methods of another distributed object, each replica of the client object will make the call, resulting in what we have called a *replicated invocation*. In this paper we presented the three problems associated with replicated invocations in the context of wide-area systems, where distributed objects may have many and widely distributed replicas. Replicated invocations can cause state corruption, both in the called object (executing a nonidempotent method multiple times) and in the calling object (returning different answers to different replicas of the caller), and can result in large numbers of request and reply messages.

We presented a novel solution to these problems for a single actively replicated object invoking another single actively replicated object. This solution is efficient and scalable in the sense that (1) it uses nearby replicas and local communication whenever possible, (2) requires no complete knowledge about group membership and (3) reduces the load on replicas by removing part of their responsibility. We have also shown that this solution can tolerate process failures. We will substantiate the efficiency and scalability claims by simulations and experiments on our wide-area test bed.

At the moment our solution is limited to a single replicated client. In the future we will extend our solution to one in which multiple clients can invoke methods of the same object using replicated invocations. Furthermore, we have to look into the issues related to the introduction of new replicas of calling and called object. We are also interested in solving the problems of replicated invocations for objects with weaker consistency models.

References

[1] BEEDUBAIL, G., KARMARKAR, A., GURIJALA, A., MARTI, W., AND POOCH, U. Fault Tolerant Objects in Distributed Systems Using Hot Replication. Tech. Rep. TR95-023, Department of Computer Science, Texas A&M University, USA, Apr. 1995. Revised version of this paper appears in Proceedings of the Fifteenth International Phoenix Conference on Computers and Communications (IPCCC), Phoenix, AZ, March 1996.

[2] COOPER, E. *Replicated Distributed Programs*. PhD thesis, Dept. Computer Science, University of California at Berkeley, Apr. 1985. (Technical Report CSD-85-231).

[3] COOPER, E. Replicated Procedure Call. *ACM Operating Systems Review 20*, 1 (Jan. 1986), 44–56.

[4] JALOTE, P. Resilient Objects in Broadcast Networks. *IEEE Transactions on Software Engineering 15*, 1 (Jan. 1989), 68–72.

[5] MAZOUNI, K., GARBINATO, B., AND GUERRAOUI, R. Filtering Duplicated Invocations Using Symmetric Proxies. In *Proc. of the Fourth IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS)* (Lund, Sweden, Aug. 1995).

[6] SCHNEIDER, F. Implementing Fault-Tolerant Services Using the State Machine Approach. *ACM Computing Surveys 22*, 4 (Dec. 1990), 299–319.

[7] WU, G. A., AND PRAKASH, A. Distributed Object Replication Support for Collaborative Systems. Tech. Rep. CSE-TR-276-96, Dept. Electrical Engineering and Computer Science, University of Michigan, Ann Harbor, MI, USA, 1996.