Replicated Invocations in Wide-Area Systems: A Possible Solution

Arno Bakker Maarten van Steen

Department of Mathematics and Computer Sciences,
Vrije Universiteit Amsterdam,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
{arno,steen}@cs.vu.nl

Keywords: distributed objects, active replication, duplicate invocation problem, wide area systems

Abstract

Using the active replication strategy to replicate objects introduces some problems when the objects invoke methods of other objects. In addition to describing what these problems are we present a possible solution. The objects we are interested in are massively replicated objects with replicas distributed over a large geographical area.

1 Introduction

In object-oriented distributed systems objects are often replicated to improve reliability or increase performance. One popular replication strategy is active replication, in which all replicas of an object carry out the method invoked by a client. This strategy introduces some conceptual and practical problems when actively replicated objects start invoking methods of other objects.

In this paper we will present a number of problems related to actively replicated objects acting as clients and invoking other objects' methods. The objects we consider may have a very high replication degree and their replicas may be distributed over a large geographical area. This means that communication between replicas is slow. Due to the potentially large number of objects in the system, communication is also considered expensive, especially communicating with groups of machines. We will not only present the problems, but also a possible solution.

System Model and Assumptions

We assume that objects use active replication as part of the state machine approach to making them fault-tolerant [1]. The machines on which they are located are assumed to suffer only from fail-stop type failures. Method invocation is synchronous, so a client (object) can invoke only one method at a time. It invokes methods on (other) distributed objects by doing an invocation on a proxy of the object that is present in the client's address space. This proxy will distribute the invocation to (a subset of) the replicas of the called object, which will carry out the call.

In case of a state-modifying operation (which we will call a *write* or *write-type operation* from now on), the invocation is forwarded to all replicas of the object. In case of a non-state modifying operation (a *read* or *read type operation*) the invocation is forwarded to just one replica. Note that the latter already is an optimization to straightforward active replication. Special care will have to be taken to ensure that these read-type operations are also made fault-tolerant. Write-type operations modify only the state variables of the object; we assume they have no other (side) effects. Since we are using a state machine approach, write-type operations are carried out by each replica in the same global order.

The proxy of the object was installed in the client's address space as part of a process called *binding*. Part of this binding process is the selection of one replica to which the read-type operations will be directed. There may be several selection criteria, but for now we assume that it will be proximity to the client's machine. The proxy is said to be "connected to" this replica. For our discussion we will assume that all binding has been done.

We currently consider only the case in which one object calls one other object. So there is one single, but replicated client calling one single, but replicated (server) object. It is our intention to extend our solution to handle multiple, different clients invoking methods on the same object at the same time.

2 Problems

There are four major problems that need to be solved in order to make method invocations between actively replicated object work correctly and efficiently.

2.1 Problem 1: Duplicate Invocations

When trying to make objects fault tolerant the invocation of methods of other objects also has to be made fault tolerant. The way fault tolerance is achieved with active replication is doing things multiple times. That is why, in principle, each operation is carried out at each of the object's replicas. If this operation involves calling another object this will naturally be done multiple times. Each individual replica makes the invocation on the object to call. The invocation can thus be made to withstand $n-1$ (node-)failures, where n is the replication degree of the caller. However, there is one major drawback. If the operation is not idempotent this repeated invocation of the other object's method will corrupt the state of this other object. As a result replicas will get different answers to the same request which also causes problems, this time at the calling side.

2.2 Problem 2: Large Number of Requests

If the fault-tolerance degree of the calling object is high, there will be many invocations per conceptual call. Each of the replicas of the calling object will have called the proxy of the destination object in its address space. Suppose that the method invoked is a write-type operation. Active replication dictates that all state-modifying operations should be sent to all replicas. Therefore each of the called proxies would forward the (reification of the) invocation to the replicas of the called object. The result of which would be that each replica would receive the invocation n times, possibly causing the machine it is on to crash. Furthermore, the excessive amount of network traffic (n times m messages, where m is the number of replicas of the called object) would lead to congestion and other problems in the network.

2.3 Problem 3: Large Number of Replies

A fault-tolerant object which uses straightforward active replication will also send redundant replies. Each replica will send the answer to the proxies that requested it by forwarding the invocation to this replica. Since all proxies will have requested the answer from all replicas (see above) we will have m times n reply messages, again leading to congestion and possibly machine crashes. Note that in our model this problem occurs only with write-type operations. Reads are sent only to the replica to which the proxy is connected and receive their reply in a normal single request/single reply fashion.

2.4 Problem 4: Returning The Same Answer

Since each of the invocations is just an instance of one and the same conceptual call, all these duplicate requests should return the same answer. But due to delays and differences in machine speed there may be a long time between the delivery of the first and the delivery of the last invocation to the called object. If there are other invocations that are concurrent with this call, returning the same answer may become a problem.

Consider the example shown in Figure 1. When the client invokes method `w1()` on A, the write call will be distributed to all replicas. Each of the replicas will execute the method and, as a consequence, call `B.rd()`, a read-type operation on B. The time at which each replica makes the call on B depends on the transmission time of the original write on A and the speed of the machine the replica runs on. Assume that the machines hosting A_1 and A_2 are far apart, or that A_2 runs on a slow machine.

Reads are processed by the replica to which pB_1 is connected (in this case the nearest replica), so A_1 will get the answer to the read fairly quickly, allowing it to continue. A_1 subsequently calls method `w2()` of B. This write has to be distributed to all replicas of B and the proxy pB_1 takes care of that. Due to delays and machine-speed differences, the write on B might arrive at B_2 before the previous read call (`B.rd()`) done by pB_2 (as part of A_2's execution of the `w1()` call).

If the write is done without further consideration, pB_2 might get a different result to its `B.rd()` call than pB_1 did before him. Since the requests are instances of the same conceptual call, they should have gotten the same answer.

3 A Possible Solution

In this section we will present a possible solution to the problems described. We do not claim that this is the only or most optimal solution, but we are confident that it will work well in a large-scale distributed system and intend to verify this by running experiments on ASCI's distributed supercomputer.

In the design of this solution we have made several assumptions:

- Communication is atomic and reliable: when a node sends a message it will be delivered to all its destinations.
- There are no network partitions.

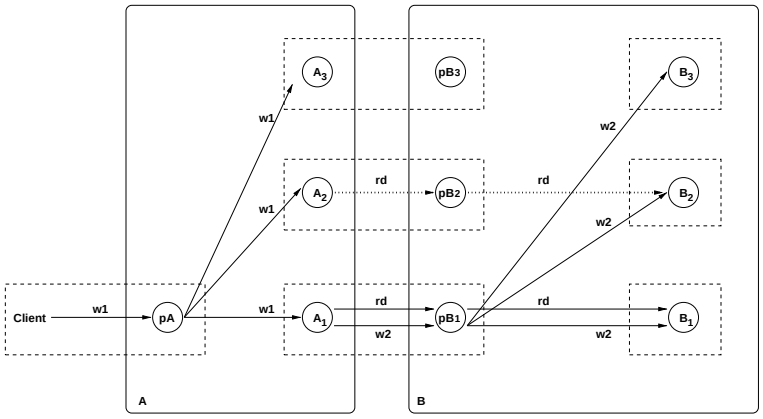

Figure 1: Intra-object concurrency. The code executed is shown on the right.

- Replicas are deterministic, that is, invoking the same method on replicas with the same version of the state returns the same result.
- Proxies have sufficient resources to store all information we intend to store on them.

When storing information at a proxy we will explicitly take into account the amount of data stored, so this latter assumption is no passport for letting us store massive amounts of information at each proxy. It just allows us to keep the presentation of the solution more focussed.

3.1 Solving Problem 1: Call IDs

The first thing to realize is that duplicate invocation problem is caused by the inability of the called object to recognize duplicate invocations as one conceptual method invocation. In systems without actively replicated objects there is always a one-to-one correspondence between a method invocation at the implementation level and a method invocation at the conceptual level. In systems with actively replicated objects this one-to-one correspondence no longer exists. Active replication causes a conceptual method invocation to be implemented as multiple redundant invocations.

One solution to this problem therefore is restoring this one-to-one correspondence between conceptual and implementation-level invocation. This solution was chosen by Mazouni et al. [2]. Each replica is connected to an *encapsulator* which intercepts all the invocations done by the replica. All encapsulators belong to the same communication group and have knowledge of the group membership. This allows them to deterministically elect one of them as coordinator based on local knowledge. This coordinator then does the actual invocation on a proxy of the object to call.

With large numbers of replicas per object and possible node failures, having complete and up-to-date knowledge about membership at each replica site in a wide-area system is no longer feasible. This makes electing a coordinator much more difficult, since the replicas have to explicitly achieve consensus.

We chose to simply let each duplicate invocation carry a call ID, identifying it as an duplicate of one single conceptual call. However, called objects will now have to explicitly allow their methods to be called by means of duplicate invocations, identifiable as duplicates by their call ID. In our opinion the advantages of this solution in a wide-area system fully compensate for this limitation.

We are still working on the exact details of how each of the invocations can be labeled with the same call ID.

3.2 Solving Problem 2: Forward Once

When an actively replicated object *A* invokes a write-type method of another object *B*, each of the replicas of *A* will invoke this method on a proxy of *B*. If *B* also happens to be actively replicated each of these proxies forwards (multicasts) the reification of this invocation to the replicas of *B*. The problem we solved in the previous section is that the replicas of *B* will not carry out the operation more than once, because duplicate invocation requests are recognizable by their call ID. What we have not solved are the problems caused by the potentially large amount of network traffic.

What is required by the active replication strategy of *B* is that each of its replicas receives one copy of the invocation. One possible solution is, given the reliable communication assumption, to let one of the proxies forward the call to the group of replicas. But that would reduce the fault tolerance of *A*'s invocation to one. Therefore we would also have to make sure that if that proxy failed to forward the call, the others would take over.

Another solution would be to let each proxy send only to a subset of the replica group. That will require

some administration to make sure that all replicas will receive a copy of the invocation. Maintaining such an administration in a wide-area environment with node failures is not an easy task, we therefore chose the former solution.

Detecting Proxy Failure The first question to answer is: how do the other proxies detect the failure of the designated forwarder? Instead of relying on a separate failure detection system, we let the others detect failure by detecting the not-forwarding of the call message. This requires that the designated forwarder also forwards the call to the group of proxies working for the calling object (*A*). The proxies that are not forwarding the call set a timer and if they do not receive the call message before the timer expires, they will forward it themselves.

The exact values of the timers will, of course, have great impact on the performance of the system when node failures occur. If the timers all expire at the same time and there is no message, we may have *n* proxies forwarding the call. If the timers are set too high the performance of the system will be bad in case of failures.

For optimal performance the timers would be set such that the proxy closest to the designated forwarder would time-out as soon as it is clear that no forwarding was done. To best handle subsequent failures, the proxy closest to this proxy would set its timer such, that if the designated forwarder did not forward *and* the next proxy in line did not forward, it would time-out at the earliest possible time and do the forwarding. It is clear that calculating these ideal values for all proxies requires some knowledge about relative position of the members of the proxy group[1]. Since this group can have a very large membership, it is impossible to have this information available at each proxy. This means that the calculation cannot be done locally.

So calculating the optimal values does not seem to be an option. The question is how much deviating from the optimal timer values affects the performance and how we could calculate semi-optimal values which will yield good performance in the majority of cases. This is still a topic of research.

Choosing a Designated Forwarder The second problem to solve is choosing the one proxy that is to forward the call (the designated forwarder). The easiest way is having the client object assign one. Electing one is out of the question, because that would require achieving consensus over a wide-area network.

How does the client object choose the first proxy? Recall that we are dealing with a write operation on the calling object which, in turn, invokes a write operation on another object by means of multiple invocations. When the write operation on the calling object is distributed to its replicas, the proxy forwarding the request names one of the replicas in the message. An obvious candidate would be the replica that it is connected to, since that will probably be in the vicinity. When the forwarded write request arrives at a replica, it checks to see if it is the one who was named in the message and records this fact. If the replica, in the execution of the write method, has to call other objects it will check if it was named in the original message, and if so, make the invocation on the proxy of the object to call. The invocation not only carries a call ID (which it has to since it is part of a replicated invocation) but also a bit indicating that the proxy called is the one that should forward the invocation to the replicas of the called object[2].

3.3 Solving Problem 3: Proxy Group

To avoid problem 2 we chose to let just one proxy forward write operations to the replica group. As a result the replicas see only the invocation request from one proxy. This was our intention but the side-effect is that replicas no longer automatically know who to send the answer to. While there was only one sender, there are a lot of receivers interested in the answer, in particular all proxies that have or will receive the write invocation from their user.

One idea is to let all proxies of a particular client join a multicast group and sending the answer to that group (a solution which avoids application-level administration) and use the same scheme that was used to solve the number-of-requests problem to solve this number-of-replies problem. Assigning a designated replier is not as easy, however.

We therefore chose another solution. Remember that each proxy is connected to a particular replica which it uses to get answers to read-type operations. Each of those replicas could keep track of which proxies are connected to it. If it received a request for a write-type operation, it would execute the method and send the reply only to those proxies that are connected to it. In this way, the answer to a write-type operation will be delivered to all interested proxies, but with less system-wide communication.

Proxies connected to a replica will, by assumption, be relatively close to that replica, making communication and maintenance of the administration of which proxies connected to it not extremely difficult or expensive.

[1] Although we have not shown it here, it also requires knowledge about the position of the replica which is used by each proxy to handle the reads it receives.

[2] Note that in principle this bit just means "you go first". If the called object were not actively replicated, but used a primary-backup type of scheme, this first proxy should try to deliver the message to the primary.

Replica Failure However, if a replica fails, its connected proxies should still get the answer, or else they will be blocked forever. Not only write-type operations suffer from this problem. Remember that read-type operations are forwarded only to one replica, the replica the proxy is connected to. If it fails the proxy cannot get the answer to its client's read.

The obvious solution in both cases would be to ask another replica, since it is its job to provide answers to proxies. However, this could slow down the progress of other replicas as they would have to delay any subsequent operations until, in the extreme case, all the proxies used by a particular client had received the answer. This is obviously not acceptable.

We can solve the problem of making replicated read and replicated write operations fault tolerant. By using the mechanisms which we also use to solve problem 4 we can ensure that proxies will get an answer without stalling other replicas. Making normal (nonreplicated) reads fault tolerant introduces some new problems and is still a topic of research.

Proxy Failures One other issue is the effect of proxy failure. The crash of a proxy pB_x most likely also implies the crash of its user, replica A_x, since they are in the same address space and it is reasonable to assume that an address space crashes as a whole. The replica to which proxy pB_x was connected (replica B_x) needs to detect the failure of proxies and update its administration of how to reach its connected proxies. If proxies are relatively close-by, as we have assumed, this will not be such a problem.

3.4 Solving Problem 4: Shifting Responsibility

The problem of returning the same answer to all proxies can be solved in a number of different ways. Note that given time, the `B.rd()` call will arrive at B_2, after which we are allowed to execute `w2()`. In principle, all that is required is that B_2 knows about the operations still underway so it will postpone the write.

However, waiting for a (very) long time is generally not acceptable. Faster solutions can be devised if we start making more use of the proxies used by *A*. Instead of delaying write operations at the replicas because there are still duplicates of previous read requests underway, we could make sure that the proxies are able to answer the read requests without any further help from the replicas. By partly freeing the replicas from their normal responsibility of providing answers, they could be allowed to proceed more quickly. This shifting of responsibility from server-side to client-side was already suggested by Cooper [3].

Pushing Answers to Proxies When a proxy is designated to forward a write operation to the replicas, it adds to this write message all the answers to previous replicated read requests received since the last write operation. Each answer is labeled with the call ID of the replicated read to which it is the answer. Recall that the write message is also forwarded to the proxy group as part of the solution to problem 2. Each proxy will therefore receive the responses to all requests that it will be asked to answer. It is likely that it will already have seen and answered—by forwarding the request to its connected replica and returning its response— part of these requests by the time it receives the write message. The proxies will store the answers to requests they have not seen yet and discard the rest. This approach has the advantage that there is no superfluous processing and communication at the replicas.

It can be shown that it is sufficient to send the answers to previous replicated read requests received since the last write operation. Basically the write acts as some sort of barrier, making sure that proxies will not ask replicas for answers to previous reads. We have also worked out what to do when the set of answers grows large and it is no longer feasible to store it at the forwarding proxy.

Sending all answers is not the only option. Another solution is to install a copy of the state at each proxy as it was before the write. Each proxy would be able to compute the answers from that, given the code of the object's methods. In effect, this would mean turning a proxy into a (temporary) replica. Since the state can be large this will not work in general, but it may provide an alternative for keeping all answers around. If the set of answers becomes too large to remember, the proxy can decide to discard the old answers and send a copy of the state along instead.

4 Related Work

The most extensive work on solving the problems introduced by actively replicated clients was done by Mazouni et al. [2][4] at the Swiss Federal Institute of Technology. They solve the duplicate invocation problem by adding a meta-layer of encapsulators which ensures that only one real invocation is done on the object.

For sending just a single reply to a client they use the same mechanism. When a replica returns an answer, this reply is intercepted by the replica's encapsulator. The encapsulator who is coordinator returns the answer to a proxy of the client in its address space. This proxy multicasts the answer to all client replicas. Because they send the result of an invocation to all the client's replicas at once they do not have problems returning the same answer (Problem 4).

While their solution for reducing network traffic is

similar to ours, their solution is less suitable for massively replicated objects with widely distributed replicas, because it requires the election of a coordinator.

The DistView toolkit [5] uses a solution very similar to that of Mazouni et al., designed for widely distributed objects, but it also depends on complete group membership knowledge. In a proposal by Isis Distributed Systems Inc. to extend the Object Request Broker concept with support for object groups [6], replicas coordinate to make sure the server object is invoked only once. How this coordination is done is not made explicit. The proposal also does not address the problem of returning the same answer. This latter problem is recognized, in a somewhat different form, by Mark Little [7], but his solution (reaching agreement) is not suitable for wide-area systems.

Cooper [8][3] was one of the first to describe the duplicate invocation problem caused by actively replicated clients. His fault-tolerant remote procedure call mechanism is targeted towards local area networks and is meant to use the efficient multicast facilities provided by these networks.

5 Conclusions

We presented four major problems that need to be solved to allow actively replicated objects to call other (actively replicated) objects from their methods. (1) The duplicate invocation of an object's methods can corrupt its state and the (replicated) state of the calling object. The amount of requests (2) and replies (3) generated by a massively replicated object can cause serious problems in the network and prevents the system from scaling to large numbers of fault-tolerant objects. (4) Due to the geographical distribution of replicas there may be multiple operations working on the same object, even with a single client, resulting in complex concurrency problems.

The solution we presented might solve all these problems very efficiently. However, there are still a number of open issues which require further research. The administration of call IDs, the effects of semi-optimal timer values on the performance of a replicated invocation when nodes in the system fail and making nonreplicated reads fault tolerant.

References

[1] F.B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[2] K. Mazouni, B. Garbinato, and R. Guerraoui. Filtering Duplicated Invocations Using Symmetric Proxies. In *Proc. of the Fourth IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS)*, Lund, Sweden, August 1995.

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

[4] K.R. Mazouni. *Étude de l'Invocation Entre Objets Dupliqués Dans un Système Réparti Tolérant Aux Fautes*. PhD thesis, École Polytechnique Fédérale de Lausanne, 1996. Thèse No. 1578.

[5] Gwobaw A. Wu and Atul Prakash. Distributed Object Replication Support for Collaborative Systems. Technical Report CSE-TR-276-96, Dept. Electrical Engineering and Computer Science, University of Michigan, Ann Harbor, MI, USA, 1996.

[6] Inc. Isis Distributed Systems. Object Groups: A response to the ORB 2.0 RFI. OMG Document 93-04-11, Object Management Group, April 1993.

[7] Mark C. Little. *Object Replication in a Distributed System*. PhD thesis, The University of Newcastle upon Tyne Computing Laboratory, September 1991.

[8] E.C. Cooper. Replicated Procedure Call. *ACM Operating Systems Review*, 20(1):44–56, January 1986.