Optimization Techniques for Replicating CORBA Objects[*]

Xavier Défago defago@lse.epfl.ch
Pascal Felber[†] pfelber@us.oracle.com
André Schiper schiper@lse.epfl.ch

Laboratoire de Systèmes d'Exploitation
École Polytechnique Fédérale de Lausanne, Switzerland

Abstract

The CORBA Object Group Service (OGS) is a new CORBA service that provides support for fault-tolerance through the replication of CORBA objects.

In this paper, we present several optimization techniques that are used to improve the performance of OGS. For each optimization, we analyse the impact on the throughput and the response time of OGS.

The optimization techniques presented in this paper are quite generic and can be applied to many fault-tolerant distributed algorithms based on consensus.

1. Introduction

Distributed computing is one of the major trends in the computer industry. To answer the growing demand in distributed technologies, several middleware environments (e.g., RPC, CORBA, DCOM) have emerged during the last few years that greatly simplify the development of distributed applications and systems. The *Common Object Request Broker Architecture* [14], also known as CORBA, is currently a popular environment for developing distributed applications and systems. CORBA is a standard implemented by different vendors. As its basic abstraction, CORBA provides the notion of distributed objects.

The growing popularity of distributed computing rises new kinds of concerns such as the dependability. A typical dependability problem is the availability of a service. This availability can be significantly reduced if the service is vulnerable to the failure of some of its parts. Ironically, while distribution is a factor that makes distributed applications more vulnerable, it can also be used to make them *more reliable* than their individual parts.

Redundancy is a commonly used technique to make a system more reliable than its components. To support the development of reliable services, specialized middleware environments (e.g., Isis [2], Horus [16], Totem [13], Transis [1], Consul [12], Phoenix [10]) have been developed. These environments greatly reduce the complexity of building reliable distributed systems by providing all the necessary support for replicating services.

Despite its convenience and popularity, CORBA still does not provide enough support for dependable systems. To overcome this lack of support, we have developed a CORBA service that addresses the dependability requirements of distributed applications [5, 6]. This work was done in the context of the European ESPRIT projects OpenDREAMS and OpenDREAMS-II, which aim at providing an open, distributed, reliable framework for supervision and control systems. Our service uses replication techniques to make CORBA objects fault-tolerant and highly available.

In this paper, we discuss several optimization techniques to improve the performance of the replication service. The optimizations presented in this paper are algorithm-based and, as such, quite generic. We analyse the impact of each independent optimization on the throughput and the response time of our replication protocol.

The rest of this paper is structured as follows. Section 2 presents background concepts about CORBA, object replication, and group communication protocols. Section 3 details the techniques that we have used to measure the performance of our replication service. Section 4 describes the optimization techniques that we have used to optimize the failure-free case. Section 5 describes two important extensions that allow a significantly better reaction to failures. Finally, Section 6 presents some concluding remarks.

[*] Research supported by OFES under contract number 96.0454, as part of the ESPRIT Project OpenDREAMS-II (number 25262).

[†] Currently affiliated to *Oracle Corp.*, Portland, Oregon.

2. Background

2.1. Distributed Objects and CORBA

The *Object Management Architecture* (OMA) [14], specified by the *Object Management Group* (OMG), is a conceptual infrastructure for building portable and interoperable software components, based on open and standardized interfaces.

Commercially known as CORBA, the *Object Request Broker* (ORB) is the communication core of the OMA. The ORB enables objects to transparently invoke remote operations in a distributed environment. The ORB also provides the environment for managing objects, advertising their existence, and describing their metadata. Clients use *object references* to identify remote objects and to invoke their operations.

The *Object Services* are a collection of interfaces and objects supporting basic functionalities useful for most CORBA applications. A CORBA service is basically a set of CORBA objects. These objects can be invoked through the ORB by any CORBA client. Services are not related to any specific application but are basic building blocks, usually provided by CORBA environments. Several services have been designed and adopted as standards by the OMG.

2.2. The Object Group Service

Neither the ORB nor the existing services provide yet the necessary tools for building highly available applications. This can be considered a major limitation for the use of CORBA in many of today's applications such as finance, process control and telecommunications.

To overcome this limitation, we have developed an *Object Group Service* (OGS) that provides the ability to group CORBA objects and invoke them as a single fault-tolerant entity [6]. Group invocation can be performed with various properties, such as atomicity and total order. Atomicity ensures that either all non-crashed members of a group receive an invocation or none of them does. The total order ensures that all members of a group receive invocations in the same order. The group paradigm has proved to be very useful in supporting highly available applications through replication. The replicas of an object are gathered inside a group. Thanks to the atomicity and total order properties, invocations made to the group hide the replication by ensuring that all replicas behave as a single copy. OGS was designed as a new CORBA service that complements the collection of CORBA services such as the *Object Transaction Service*, the *Object Persistence Service*, the *Event Service*, and the *Life Cycle Service*.

The service approach taken by OGS stands in contrast with other attempts at providing support for fault-tolerance in CORBA. In Orbix+Isis ([8]) and Electra ([9]) the group communication platform is integrated into the ORB. In Eternal/Realize ([11]), invocations are intercepted and passed over to the group communication platform.

2.3. The Consensus Problem

The consensus problem is a central problem in the context of distributed systems. More specifically, this problem is central for the support of replicating objects in OGS: consensus is used to implement total order [3] and to solve the group membership problem.

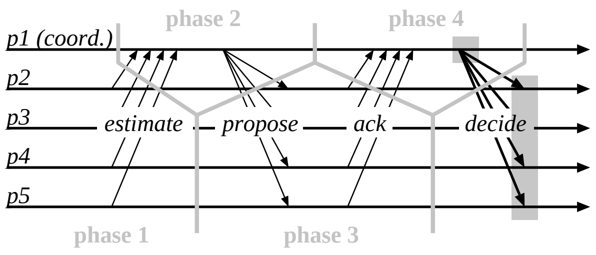


Figure 1. Consensus algorithm when no failure occurs. ($n = 5$ processes)

The consensus problem is defined on a set Π of processes. Every process $p_i \in \Pi$ starts with an initial value v_i, and all correct processes must agree on a common value v that is the initial value of one of the processes. OGS relies on the consensus algorithm using the failure detector $\diamond\mathcal{S}$, proposed by Chandra and Toueg [3]. The algorithm is based on the rotating coordinator paradigm. In each round, one of the processes is the coordinator. Figure 1 depicts the communication schema of the algorithm when no failure occurs. On the figure, a gray rectangle represents the time when a process decides. More details on the algorithm can be found in [3].

3. Performance of OGS

3.1. System Configuration

We have performed our measurements with the C++ version of OGS, compiled with VisiBroker 3.2. The testing took place on a local 10 Mbit Ethernet network, interconnecting 13 Sun SPARCstations running Solaris 2.5.1 or 2.6. The test were carried under normal load conditions. Among these workstations, there were four Sun UltraSPARC 30 (250 Mhz processor, 128 MB of RAM), and nine Sun UltraSPARC 1 (170 Mhz processor, 64 MB of RAM). All the client and server applications were located on different hosts.

3.2. Test Scenarios

For each optimization, we have measured the average response time as seen by a client, and the maximal throughput that the replicated service can handle. All measures have been made with the assumption that no failure occurs. We have evaluated the gain in performance of each optimization by comparing with the results obtained without optimization.

The response time is the time elapsed between the emission of a request by a client and the reception of the corresponding reply. The average response time of the service is measured in the following manner: a single client issues a fixed number of totally ordered invocations (typically 100). The client serializes the invocations by waiting for the reply from its request before issuing the next request. The total time is divided by the number of invocations issued, to obtain the average response time of a single invocation.

The maximal throughput is the number of requests that OGS can totally order in a fixed period of time. The maximal throughput was measured in the following way: many clients[1] issue totally ordered invocations to the service. Since there are many clients, many invocations are performed concurrently. The number of clients is increased until the service has reached its maximal throughput. Unlike the measures of the response time, the throughput of OGS benefits from the ability of the total order algorithm to order several requests in a single execution of the consensus algorithm.

The measures have been made with different group sizes; the tests have involved 3, 5, and then 7 replicas. For all measures, the client waits for a single reply from the service.

3.3. Reference measures

To provide a base for comparison, we have first measured OGS without any optimization. The results that we have obtained are illustrated in Table 1.

	resp. time [ms]			thput [msg/s]		
$n =$	3	5	7	3	5	7
no opt.	14.9	27.0	49.2	190	110	69

Table 1. Reference measures.

4. Improving Performance

In this section, we present three optimizations to the consensus algorithm that help improve the response time of the service. The first optimization aims at reducing the network congestion by reducing the number of messages generated by the consensus algorithm. The second optimization consists in removing one phase of the algorithm and thus reduces the number of communication steps of the algorithm. The last optimization applies to the common case where there are exactly three replicas.

[1]For the convenience of tests, there is only one client *process* that holds many client *threads*. Since the client part of the ORB becomes a bottleneck, the results obtained for the throughput are probably underestimated.

4.1. Optimization 1: Reducing network congestion

In the original consensus algorithm, as presented by Chandra and Toueg, the processes proceed to the next round directly after they have send an acknowledgement (*ack* or *nack*) to the coordinator. In the case where no failure occur, proceeding directly to the next round generates many unnecessary messages. The messages sent on behalf of round 2 are represented in gray on Figure 2.

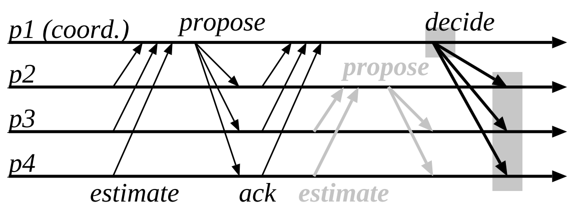


Figure 2. Optimization 1: reducing network congestion.

To circumvent this problem, we modify the algorithm so that the participants proceed to the next round only after some delay d, or earlier if they suspect the coordinator. The value of d is chosen such that it is greater than the usual round-trip time. The delay d should ensure that there is a high probability for the decision message to arrive before any participant has a chance to proceed to the next round.

With this modification, the algorithm generates fewer messages. As a result, the performance of the consensus algorithm is improved in the common case where no crash occur. Our observations have shown that the number of messages generated by n replicas can be reduced by as much as $2 * n - 3$ messages.

It is important to note that this optimization may delay the decision of the consensus in the case of an actual crash of the coordinator. This is however not a problem since this cost is rendered negligible by the optimization presented in Section 5.

	resp. time [ms]			thput [msg/s]		
$n =$	3	5	7	3	5	7
no opt.	14.9	27.0	49.2	190	110	69
opt. 1	13.8	25.0	42.9	201	113	75

Table 2. Performance measurements for optimization 1

As mentioned earlier, optimization 1 significantly reduces the number of messages generated by the algorithm. Although these messages do not lie on the critical path, they load the network and the processes unnecessarily. As shown in Table 2, these messages account to about 10% of the overhead for the response time, and around 5% for the throughput (see also Fig. 5).

4.2. Optimization 2: Removing one phase

As explained in [15], in the first round of the consensus algorithm, the coordinator does not need to receive the estimates of the other processes and can just start by proposing its own initial value. In other words, if the coordinator of the first round tries to impose its own value, it is possible to remove one communication step to the algorithm.

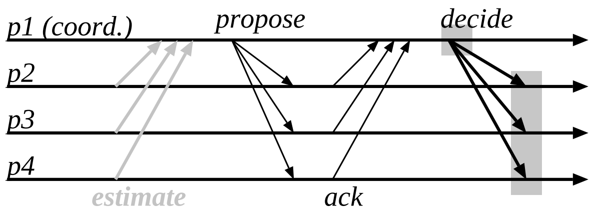


Figure 3. Optimization 2: removing the first phase of the first round.

The second optimization that we introduce consists in having all processes start the consensus in the second phase of the first round. As a result of this optimization, the consensus takes only three communication steps instead of four, and generates fewer messages. This optimization is illustrated in Figure 3.

	resp. time [ms]			thput [msg/s]		
$n =$	3	5	7	3	5	7
no opt.	14.9	27.0	49.2	190	110	69
opt. 2	10.9	21.8	42.0	208	117	78
opt. 1+2	10.0	19.5	39.7	213	123	80

Table 3. Performance measurements for optimization 2.

Optimization 2 reduces the number of messages to a lesser extend than optimization 1 ($n - 1$ messages). However, it also reduces the number of communication steps. The measures illustrated in Table 3 show that the ability to reduce the number of steps has a significant impact on both the response time and the throughput. Besides, since optimization 1 and optimization 2 are independent, the combination of the two optimizations accounts for as much as 33% improvement for the response time and 12% for the throughput.

4.3. Optimization 3: Specializing for 3 replicas

The optimization presented in this section is specialized for the case of three replicas. In this case, the system tolerates the crash of *one* replica. We believe that optimizing this case is worthwhile since it is commonly encountered in practice. The optimization is introduced in OGS by testing the size of the group before starting a consensus algorithm.

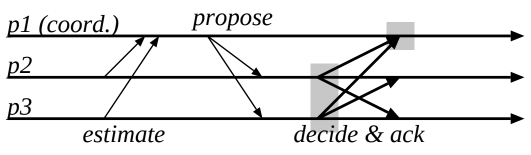


Figure 4. Optimization 3: specializing for 3 replicas.

This optimization is based on the observation that two processes form a majority in a group of three processes. In the context of the consensus, a process that accepts the proposition made by the coordinator (phase 3) already knows that the proposed value will be the decision value. Indeed, the process and the coordinator already form a majority together. As a result, the process is in a position where it can decide. This optimization is illustrated in Figure 4 where p_2 and p_3 decide as soon as they receive the proposition from the coordinator p_1.

	resp. time [ms]	thput [msg/s]
no opt.	14.9	190
opt. 3	12.4	208
opt. 1+2+3	9.8	224

Table 4. Performance measurements for optimization 3 ($n = 3$).

The measures concerning optimization 3 have been performed for the case of three replicas. This optimization reduces the number of communication steps as seen by the client, but does not really reduce the number of messages. As illustrated in Table 4, optimization 3 gives much better results than optimization 1, but lower results than optimization 2. The combination of the three optimizations gives very good results. It is however interesting to note that optimization 1 has little impact in this context since the participants decide before the coordinator.

The improvement obtained by each of the optimizations are illustrated in Figure 5. The gain obtained for the throughput amounts to as much as 18%, as illustrated in Figure 5(a). But, Figure 5(b) shows that the improvement of the response time is even more impressive, since it amounts to as much as 34% of the general algorithm.

5. Improving Reaction to Failures

The optimizations presented in the previous section aim at improving the response time in the case where no failure occur. In this section, we present a modification that dramatically reduces the time needed to react to the crash of a replica. The concepts on which this modification is based were first presented in [4]. It has been implemented successfully in the context of OGS.

5.1. Two time-out group membership

A slow reaction to failure is a common problem of group communication platforms (e.g., Isis [2], Horus [2, 16], Consul [12], Transis [1], Phoenix [10]) for asynchronous systems. In such platforms, the failure detection mechanism is based on a time-out to detect processes' crash. The purpose of the time-out is (1) to define the membership of the groups, and (2) to ensure that the algorithms (e.g, the replication algorithm) do not wait for a message from a crashed process.

As illustrated on Figure 6, the choice of an adequate value for this time-out leads to the following trade-off.

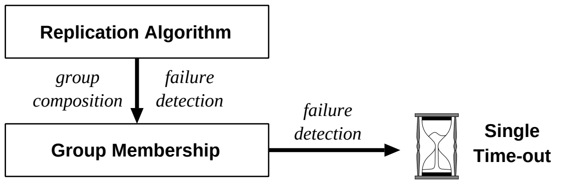


Figure 6. One time-out approach.

A large value causes long black-out periods of the replication algorithm, in the case of a crash. On the other hand, a small time-out value increases the probability of incorrectly removing a correct replica. Due to the prohibitive cost of an incorrect removal (the replica will have to rejoin the group), a large value is usually preferred, typically in the order of a few minutes. However, a black-out period in the order of a minute is often unacceptable, especially for time-critical applications [7].

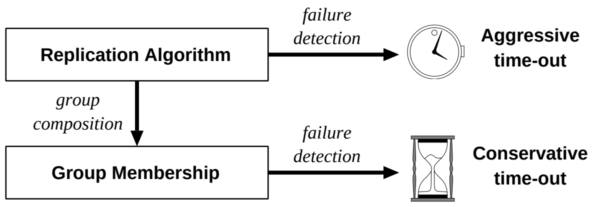
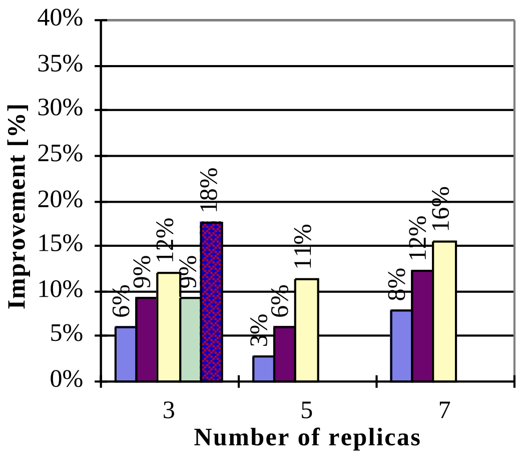


Figure 7. Two time-out approach.

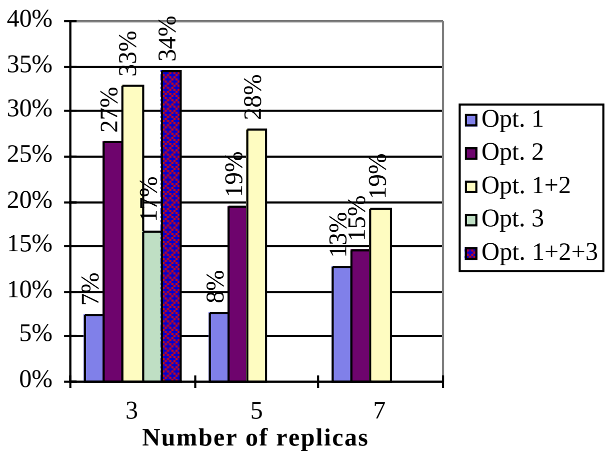
The solution for the next version of OGS makes use of two time-out values in order to overcome the trade-off. This concept is illustrated in Figure 7. An aggressive value is used by the replication algorithm to react quickly to failures. This value is typically in the order of 50 ms on a LAN (local area network). The group membership relies on a second time-out value that is large enough to avoid the problem of incorrectly removing correct replicas. In OGS, this conservative time-out value is in the order of 10 minutes.

Due to the difficulty of measuring the performance in the case of a crash, we do not give figures for this case. The difficulty stems from the large difference that exists between different runs with one crash. Indeed, if a process different from the coordinator crashes, the response time is not different from the failure-free case. On the other hand, if the coordinator crashes before it can send its proposition, the response time is much larger.

In the latter case, the response time is extended by the duration of an extra round and by the time needed to detect the crash. Let us take an example and assume that one round of the consensus algorithm takes about



(a) Throughput.



(b) Response time.

Figure 5. Improvement over the standard algorithm.

15 ms (see Table 1), and that the time-out value is 60 sec. The response time if the coordinator crashes is mostly defined by the time needed to detect the crash; a little more than 60 sec. Let us now assume that the time-out value is 50 ms. In other words, the time-out value is in the same order of magnitude than the time needed to finish one round of the consensus. As a result, the duration of a round becomes also determinant and the response time to expect is around 80 ms. The gain in performance in such a case is three orders of magnitude.

6. Conclusion

In this paper, we have presented four optimization techniques that make it possible to improve the response time of replicated invocations in the context of OGS. These optimizations are applicable in the context of group communication services based on the consensus.

Three of the optimization techniques aim at improving the case where no failure occur. We have shown that a combination of these three techniques makes it possible to reduce the average response time of the system by more than one third, and to increase the throughput by almost 20%.

The two time-out group membership significantly improves the reaction to failures. By decoupling the problem of failure detection from the membership service, we improve the reaction to failures by as much as three orders of magnitude. This, of course, has a significant impact on the response time in the event of a crash.

Despite the strong improvement that we have been able to obtain, we believe that there is still room for optimizations and we are still investigating this issue.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: a communication sub-system for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, Boston, MA, USA, July 1992.

[2] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1993.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[4] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, West Lafayette, IN, USA, Oct. 1998.

[5] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867.

[6] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[7] R. Guerraoui and A. Schiper. Consensus: the big misunderstanding. In *Proceedings of the 6th IEEE Com-*

puter Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6), pages 183–188, Tunis, Tunisia, Oct. 1997. IEEE Computer Society Press.

[8] IONA and Isis. *An Introduction to Orbix+Isis*. IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994.

[9] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, University of Zurich (Switzerland), Feb. 1995.

[10] C. P. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, Sept. 1996.

[11] P. M. Melliar-Smith, L. E. Moser, V. Kalogeraki, and P. Narasimhan. The Realize middleware for replication and resource management. In N. Davies, K. Raymond, and J. Seitz, editors, *Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 123–138, The Lake District, UK, Sept. 1998. Springer-Verlag.

[12] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal*, 1(2):87–103, 1993.

[13] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulis, and T. P. Archambault. The Totem system. In *Proceedings of the 25rd International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 61–66, Pasadena, CA, USA, 1995.

[14] OMG. *The Common Object Request Broker: Architecture and S pecification*. OMG, Feb. 1998.

[15] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[16] R. van Renesse, K. P. Birman, B. B. Glade, K. Guo, et al. Horus: A flexible group communications system. Technical Report TR95-1500, University of Cornell (NY), Mar. 1995.