Experience with Distributed Replicated Objects: The Nile Project

Aleta Ricciardi, Michael Ogg, Fabio Previato

TR TR-PDS-1997-012 December 1997

Parallel & Distributed Systems group

Department of Electrical & Computer Engineering

University of Texas at Austin

Austin, Texas 78712

Experience with Distributed Replicated Objects: The Nile Project*

Aleta Ricciardi Michael Ogg† Fabio Previato‡

University of Texas at Austin

aleta@ece.utexas.edu

December 17, 1997

Abstract

The goal of the NILE project is to develop an inexpensive, scalable, fault-tolerant, widely distributed job processing environment. On the systems side, NILE must manage and provide transparent access to hundreds of commodity processors spread across the United States, and a distributed database that will exceed 100 terabytes. These are scales not commonly encountered by projects concerned with fault tolerance. On the software engineering side, NILE must be easily maintained, outlive its development phase, and be able to incorporate, or even migrate to, new software components; these requirements led us to the CORBA standard. While CORBA does not yet include a fault tolerance specification, only a small number of experimental ORBs support object fault tolerance through replication. Our experiences over two years of building NILE have taught us a great deal that may be of use to designers of ORBs that will support object replication and fault tolerance.

1 The NILE Project

The NILE collaboration is building a self-managing, highly-available job processing environment, able to provide transparent access to hundreds of heterogeneous commodity processors

*The Nile Project is funded by the National Science Foundation under grant PHY-9318151.

†The first two authors are also partially supported by DARPA under contract F30602-96-1-0313, and the Texas Higher Education Coordinating Board under contract ARP-003658-260.

‡Current address: Nortel plc; Advanced Technology; London Rd; Harlow, Essex CM17 9NA; UK

and a very large distributed database. The first deployed system, and the genesis of the NILE project, will be for the CLEO high energy physics (HEP) experiment [5] where the computing resources are spread across the United States at 24 collaborating institutions, and the distributed data base exceeds 100 terabytes. Because it is managing and providing transparent access to distributed resources, NILE can be considered a distributed operating system for a global virtual computer composed of a multitude of resources spread over a large geographic area. Consequently, while originally developed to address CLEO's computing needs, NILE is not specific to CLEO, and could equally well be used for any large-scale processing. NILE's modular object structure means that any special purpose algorithms, such as scheduling, can easily be dropped in. NILE is currently being tested by other HEP collaborations and in other target application domains.

NILE must outlive its development phase, be able to adapt to and scale with changes in computing needs, be easily maintained, and be able to incorporate new software components as they become available. These concerns led us to the CORBA standard,[1]. Pragmatic issues such as cost, scalability, and institutional autonomy led us to a widely distributed global architecture, hierarchically composed of tens of geographically proximate sites. Each site implements the fault-tolerant aspects required of NILE, but inter-site coordination is loosely-coupled. Unfortunately, CORBA does not (yet) specify how to interact with replicated objects and does not include reliability specifications. Thus, NILE has been grappling with many critical concepts in distributed object technology and in building large distributed systems.

To program replicated objects we have been using Electra [6], an ORB built on top of a reliable communication substrate; currently, Electra has an adapter for Isis [4], though unfortunately Stratus, Inc. is no longer distributing Isis. In principle adapters for any transport subsystem, such as Ensemble [8], Transis [1] or even TCP/IP, could be developed, though there are subtle difficulties, discussed in Section 3, with middleware not supporting *primary-partition* [3] semantics.

In this paper we discuss our experiences with Electra and the general issues in programming replicated distributed objects. It seems that programming with replicated objects still requires a fundamental understanding of issues in fault-tolerant distributed computing. In particular, adding replication to CORBA objects introduces implementation problems regarding efficiency and correctness. While our experience is specific to one ORB[2], we believe

[1] Common Object Request Broker Architecture, standardized by the Object Management Group, `http://www.omg.org/`.

[2] At the time we began using Electra, Orbix+Isis was the only other available ORB, but only CORBA v1.0 compliant, supporting replicated objects. Iona is no longer distributing it.

the issues we have encountered are representative of any ORB that will attempt to support replicated objects.

1.1 Target Applications

There is a large class of computing problems, particularly in science and engineering, that can be parallelized, and where there is very little, or in many cases, no coupling between the parallel entities. Examples include database search and processing, many simulations, video image processing, and high energy physics. This last category was the original motivation for NILE, although NILE is by no means restricted to it.

In HEP, a particle collision is called an *event*, and the data representing an event typically vary from kilobytes to megabytes. Most modern experiments record 10^9 or more events, so that the total data ranges from terabytes to petabytes. Since each event is stochastically independent from any other, an entire data set can be arbitrarily parallelized down to the event level. When the data are processed on several independent processors, an additional task, the *collection* phase, is needed to collect and present the results after all earlier tasks have completed.

There are several computing tasks in HEP, each one of which can be parallelized in this manner. They include simulation (or Monte Carlo) which generally is very CPU intensive, physics analysis which generally is very I/O intensive, and reconstruction (which converts the raw data into "data products") which is both CPU and I/O intensive.

We define a *job* as the complete computation, and a *subjob* as a parallelized component. The collection task is a special case of a subjob. These terms are used within NILE whatever the actual application might be.

1.2 Requirements and Constraints

In designing the software architecture of NILE, we drew three significant conclusions, in order of importance, from analyzing the desired functionality, requirements, and constraints.

1. NILE *will be based on the CORBA standard.* The NILE system must outlive its development phase by many years. It should therefore be easy to maintain. It should also be able to incorporate improved software components and additional functionality as these become available. Since we cannot impose static, collaboration-wide hardware requirements, NILE must run on a wide variety of platforms. NILE cannot be dependent on a single vendor.

2. The NILE *system must be highly-available.* Viewing NILE as a type of operating system means much of its functionality, and therefore many of its objects, must be highly available. Of special concern are the objects that represent NILE sites to users and to other NILE sites, objects that manage resources, objects that oversee the execution of users' jobs (though not the jobs themselves), and auxiliary objects on which these depend. Performance concerns dictate that the strategy for providing availability (*i.e.*, active, passive or temporal replication) should be carefully selected.

3. The NILE *system should encompass all an application's sites and their resources.* Jobs issued by any legitimate user, from any site, should be able to use resources located at any site. However, sites must be able to retain autonomy over local resources, and network partitions should not prohibit work at individual sites that have sufficient resources.

2 NILE Architecture

In NILE's software architecture, shown in Figure 1, each participating local-area network is a *site*. Each site has a local set of users, and computational and storage resources. These resources are available to process *jobs* that have been submitted by local users, or legitimate users local to other NILE sites. We focus here on the site-local objects and functionality, but local-area NILE was designed to adapt easily to a wide-area setting [2]. For instance, object groups that must be tightly coupled are not spread across the WAN; conversely, only state information that changes infrequently would be shared at all sites. The NILE design documents describe these criteria in detail [7].

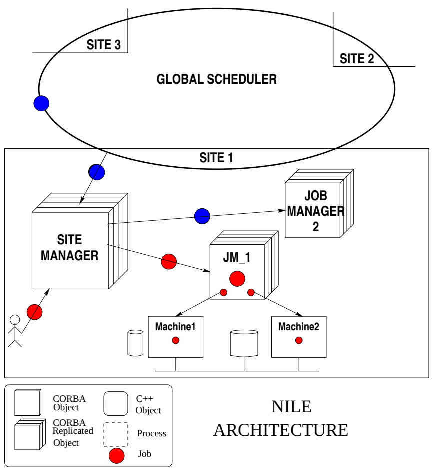

Figure 1: The NILE global architecture.

Each job is divided into a list of *subjobs* that are run in parallel. The breakdown depends on the availability of resources and the job's computational and data requirements. There is a bias toward using local resources, but the great disparity in sites' resources will require many jobs to be processed at sites other than the user's home site, possibly with subjobs at multiple sites.

NILE provides an abstraction of the computational and data resources as a single computing facility, and keeps users unaware of the underlying system complexity. In this way, NILE is building a global virtual computer. Like a conventional computer, NILE has a *Control*

System (OS), a *Data Repository* (storage system) and a *User Interface*. NILE provides this simple model of computation from a set of independent machines with different architectures and OS, each of which is subject to failures. Our architecture ensures that each site remains operational,[3] and that every job executes to completion, despite the failure of NILE objects and processors on which subjobs are executing. The next section describe the NILE Control System objects, so that their interactions and fault tolerance characteristics can be clearly understood for the discussion in Section 3.

2.1 NILE Control System

The *Nile Control System* (NCS) refers to the collection of NILE system objects running at each site that perform all the tasks necessary to execute jobs, manage local resources, and represent that site within the entire NILE system. The NCS consists of a set of distributed, replicated CORBA objects, as shown in Figure 2.

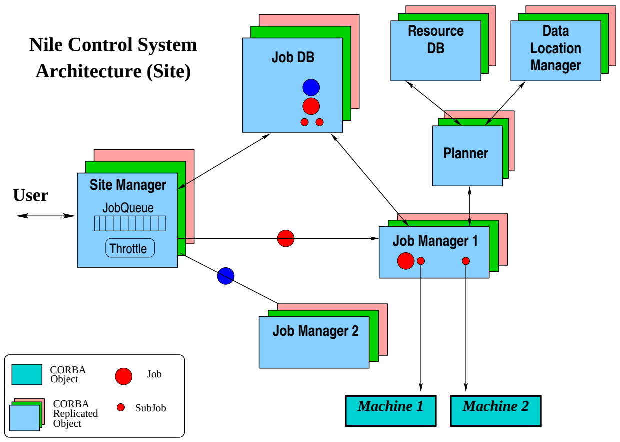

Figure 2: The NILE Control System Architecture.

• The *SiteManager* is the main interface between local users and the NCS. It is also the site's interface to the rest of the NILE system. It receives requests about jobs (submission, control, status) and arranges for appropriate processing by creating or contacting other objects.
• Because the SiteManager is stateless, the *JobDataBase* (JobDB) stores the state of jobs that have entered NILE (pending, processing or completed), together with context information describing output locations.

[3]Whether a site is reachable depends on the wide-area network, which is not under our control. This aspect, in fact, led us to sites for high-availability, meaning that inter-site unresponsiveness would be due solely to the larger network.

• For each job submitted to it, the SiteManager creates a *JobManager* which is responsible for managing and processing the job. It obtains an execution plan from the *Planner* and applies it by dividing the job into a set of smaller subjobs to be executed in parallel. The JobManager monitors the state of each subjob. If a subjob fails, it can recursively request a new plan and split the failed subjob into smaller subjobs. When all subjobs have completed, the JobManager starts a special *Collection Subjob* to collect and assemble the results; this keeps the JobManager completely general as it maintains the separation between the JobManager and the specifics of any job. If the execution plan involves resources at another site, the JobManager will pass the relevant subjobs to the appropriate SiteManagers.
• The *ResourceDB* and *DataLocationManager* (DLM) store information regarding available processors and data within the NILE system. The Planner uses this information in determining the breakdown into subjobs and execution plans.

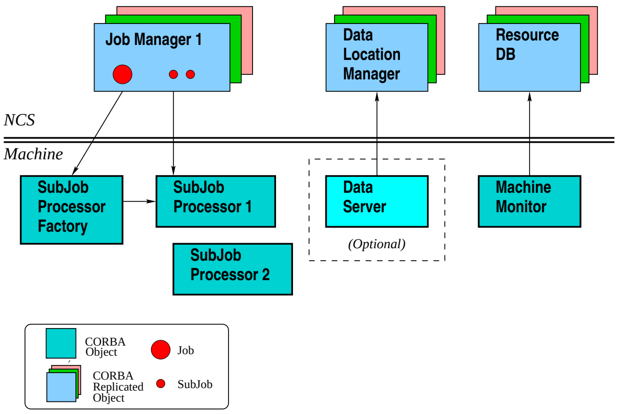

Figure 3: NCS Machine Architecture

• The *SubJobProcessor* creates a context and runs a given subjob, controlling its execution. The SubJobProcessor signals the JobManager when a subjob has terminated. Each physical machine (see Figure 3) has a *SubJobProcessorFactory* that creates and destroys SubJobProcessor objects for that machine.
• Each processor has a *ResourceMonitor*, which transmits relevant usage and availability information to the ResourceDB. Each data server reports its metadata to the DLM. A physical machine may be either a processor or data server or both.

3 Implementation Issues

An *object group* extends a distributed object by allowing the object to be replicated in a manner that is transparent to clients of the object (group) and, to some extent, applications programmers (Figure 4). To invoke a remote method call (RMC) on a non-replicated object, a client must have a reference to it; when an object is replicated, the reference is to the group of replicas. Transparency is obtained because this reference is indistinguishable from a reference to a non-replicated object, and clients use it in the same way. The Electra runtime system (RTS) multicasts the client RMC to the group, and returning replies (one by default, but more if specified) to the client. Transparency is maintained only if the object group behaves indistinguishably from a non-replicated object. In fault-tolerant distributed computing, such group behavior said to obey *primary partition, virtually synchronous* [3] semantics.

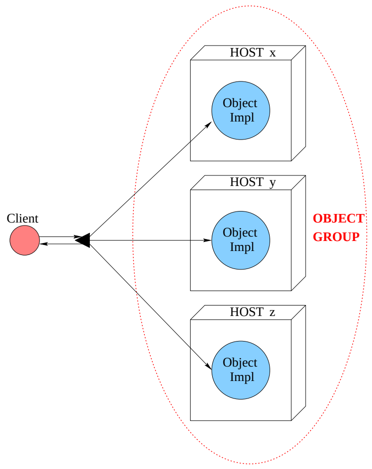

Figure 4: Object group and multicast of client RMC.

To implement virtually synchronous object groups, Electra's RTS relies on the following services: consistent failure detection, consistent group membership changes, state transfer, object monitoring, and request atomicity and order.
In the version we have been using, and the basis of the following discussion, these services are provided by layering Electra on top of Isis (see Figure 5). To use Electra with other group-based middleware, only the Electra adapter needs to be written; however, if primary partition semantics are not supported, for example in Ensemble [8] or Transis [1], an additional layer beneath the adapter is needed to provide the primary partition semantics Electra assumes.

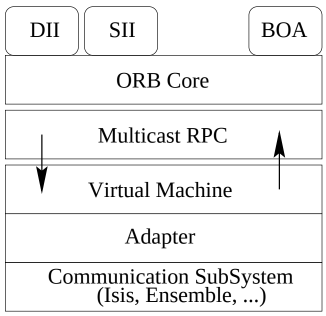

Figure 5: Electra layers.

It can be argued that the need for primary-partition semantics is inherent to the goals of encapsulation: the implementation of an object should remain hidden to clients of the object. Implicit, here, is the notion that an object is linear, meaning encapsulation is lost if an object group cannot maintain the illusion of being a single (fault-tolerant) linear thread of execution. This is only ensured by primary-partition group semantics. Permitting progress in minority partitions can lead to the object appearing to contradict itself, or "forget" its own history. In these cases it will be evident that the object is in fact a group of objects (that are not tightly controlled).

3.1 Observations from and for Object Replication

Using Electra on top of Isis to build NILE we have seen that:

- Object groups do not absolve applications programmers from knowing how to solve basic problems in fault-tolerant distributed computing.
  This need arises from efficiency and correctness concerns. It is unlikely that any implementation of object groups, whether layered on top of an appropriate transport layer or provided as an external service, can afford to assume pure state machine replication is necessary in every instance;[4] the choice of replication strategy (active, passive, or temporal) should be specific to each application. However, once programmers have a choice of strategies, they become responsible for ensuring the required group semantics. We discuss this in Section 3.2.

- While CORBA facilitates complex interactions among objects, complex interactions among (object) groups are difficult and expensive to implement.
  Objects are never statically categorized as server or client, but instead assume these roles as consequences of their interactions with other objects. When an object group begins acting as a client of another object (group), it becomes difficult to ensure a single, clean line of execution without either group synchronization mechanisms, or an enforced structure on the group that inherently implements any necessary synchronization. That is, an object group should be able to assume different group semantics depending on whether the object group is acting as server or client. We discuss this in detail in Section 3.2.2.

- Object groups can expose subtle, low-level artifacts of the underlying process-oriented execution model.
  The replica consistency problems we describe in Section 3.3 arise from thread preemption and (re)scheduling; they are not problems within Isis. Indeed, the Isis documentation notes these caveats and Isis provides mechanisms to avoid potential problems. Instead, it is our opinion that the problems derive from trying to hide the fact

[4] Our third point describes how this is also a difficult model in which to program multi-object interactions.

that the object-oriented programming model is built on a process-oriented execution model.

In analyzing the source of the problems described in the next sections, we do not believe they are wholly attributable to Isis or Electra. The manifestations may be peculiar to these, but the sources, which will be present in any ORB supporting replication, seem to be the desire to hide the implementation of replication, and the inherent mismatch between programming and execution models.

3.2 Exposing Groups and Replication

The object group paradigm contains the conflicting goals of implementation simplicity and implementation efficiency. On the one hand, ORBs supporting replication should attempt to simplify group-based programming so that no specialized knowledge is needed. One way of accomplishing this is to reduce the choices for programmers and implement only the strongest of all semantics for groups to ensure correctness.[5] On the other hand, to achieve acceptable performance, programmers will want to tailor group replication strategies and RMC delivery semantics for individual applications. In the absence of tools that would assist programmers in writing correct and efficient applications, knowledge of canonical problems in fault-tolerant distributed computing is inescapable.

3.2.1 Multicast Protocol Policy for a Group

To improve performance of applications, Electra allows the programmer to specify a *protocol policy* for each object group. This feature provides needed flexibility at the expense of programming simplicity. Programmers indicate which multicast primitive the RTS should use to deliver RMCs. With Isis as the underlying transport, the choices are its FBcast, CBcast, ABcast, and GBcast; their delivery semantics, and therefore implementation cost and degree of application inhibition, vary a great deal. To use this feature correctly, and to exploit the efficiency gains, programmers need to know the difference between the protocols and understand whether their applications can safely use weaker primitives or do, in fact, require stronger ones. That is, programmers must be well aware of the multicast communication layer.

Since the protocol policy selected applies to every method of an object group, correctness decrees that the method with the most stringent semantic requirements dictate the policy

[5] This has the added benefit of more easily conforming to standards, and was an important consideration for Electra.

for that object and all its methods. It may therefore be desirable to further sacrifice programming simplicity, and gain efficiency by allowing a protocol policy to be selected for each method of an object.

3.2.2 Flexibility in Choice of Replication Strategy

While Isis provides a *Coordinator-Cohorts* (cocos) group structure for passive replication, it is not part of the present Electra system, which provides only peer-group (*i.e.*, active) replication. A cocos group is commonly used to avoid all group members performing costly computations, and to minimize the cost of communication between groups. Ideally, NILE's JobManagers and SiteManagers should be organized as cocos groups as both communicate with many other NILE objects or object groups, some of which will be at remote sites.

The difficulties that arise in trying to provide exactly-once semantics for any replicated server become compounded when a server group then becomes a client of another object (group). It is usually the case that the RMC should be invoked only once on the second server group (not by every replica in the server-cum-client group), and so a cocos structure makes sense for client groups. However, exactly-once semantics cannot be ensured without making the RMC and cohort update atomic; atomicity should also be provided for the server's response and cohort update. Said another way, if a temporary group could be formed, consisting of the cohorts and the server (group), then the atomicity of the underlying communication substrate might help in solving the problem; of course, the RMC intended for the server group would make no sense to the cohorts of the client. Suffice it to say, this is a significant obstacle and almost impossible to solve using only object group communication.

We expect this scenario to be common in many object group applications. In NILE, the JobManager group, which is a server for the SiteManager, must create exactly one SubJobProcessor object for each subjob. It does this by calling the SubJobProcessorFactory, thereby becoming a client of it. As this interaction is responsible for launching subjob execution, exactly-once semantics are particularly important (including some subjobs twice, or not at all, can drastically affect many computations). The same issues arise when the JobManager group creates the collection subjob. We have built a cumbersome cocos framework that allows a single replica of an object group to execute a set of operations then notify the rest of the group with results or failures, but we have not solved the problem of atomic notification of outgoing RMC.

3.2.3 Synchronization Between Replicas

Our experience has shown that at least two synchronization primitives should be available to programmers.

Barrier synchronization is important for situations in which each replica computes independently until it reaches a point where the next step cannot be taken until all replicas have reached that point in their own computations. In NILE this is evident in the creation of the JobManager group. Each SiteManager replica creates a JobManager replica for each job submitted, but it happens that the JobManager replicas should not begin executing until the group is completely formed.[6] That is, the SiteManager should invoke a `start()` RMC for the JobManager, but only after all SiteManager replicas have reached a 'JobManager-created' barrier.

Coordinator-Cohort is also a kind of synchronization. In the usual case, every replica executes some code, then all but the coordinator stop computing until the coordinator informs them of results of an expensive or unique computation (or is determined to have failed). All replicas then continue again with the computation. In this manner, every replica synchronizes itself with the coordinator.

Working only within the object group paradigm provided by Electra, it is very hard to build such primitives, precisely because the underlying communication layer is inaccessible (*e.g.*, it is impossible to address an RMC to a single object replica of a group). However, these common and useful methods should be provided by the ORB.

3.3 Correctness Issues

Consistency between active object replicas is based upon the virtual synchrony (VS) model of execution. Roughly speaking, VS ensures that all object replicas observe the same events in the same order or logical time. This requires that the order in which any non-deterministic events are to occur be agreed upon by all replicas. Such events include delivery of unicast and multicasts (*i.e.*, RMC delivery), failure suspicions and changes in the composition of the object group. In response to each of these events an action sequence (thread) is executed. Since the event order and the response code is the same for each replica, the computation will be consistent. VS assumes that each response (thread) cannot be interrupted by a new event. Usually this is achieved with non-preemptive threads, which unfortunately impose a heavy performance penalty in addition to deadlock problems. For these reasons, Isis adopted a hybrid solution in which the system can re-schedule a thread that is performing some

[6] The reasons for this are beyond the scope of the paper.

blocking operation (*e.g.*, I/O, communication); a thread can also explicitly release control. Now, however, programmers are responsible for avoiding or handling situations that could lead to replicated objects' states becoming inconsistent. While Isis provides mechanisms to avoid potential problems, in the interest of standards conformance, Electra does not make these visible.

3.3.1 The State Transfer Problem

To maintain consistency among object replicas when a new object joins an existing group, Electra transfers the state of an object inside the group to the new object. To ensure that the transfered state is itself correct (*i.e.*, consistent with the other replicas' states) Electra, via Isis, completes all pending RMC deliveries, and avoids delivering new RMCs until the transfer is completed. As RMCs are multicasts, this is precisely virtual synchrony.

However, we have experienced situations where these restrictions and precautions are not strong enough to maintain state consistency. In many situations, not only must pending RMC deliveries be completed, but also all *on-going threads* should be completed before the transfer can take place. If, for any reason, a thread relinquishes control before it terminates, and if the state transfer occurs during this pause, the transfered state may be incorrect when the threads do complete; Figure 6 demonstrates such a scenario. There the returning thread subsequently modifies the replicated state, making the state of the new member inconsistent with the state of the previously-existing members. We have not found a clean way to avoid the problem in Electra.

```
class Counter { // example code causing state transfer problem
    public: Long number() { n = numSrv->getNextNum();
                            cache = n; // indicated by ○ in diagram below
                            return n; // indicated by □ in diagram below
                            }
    private: Long cache;
}
```

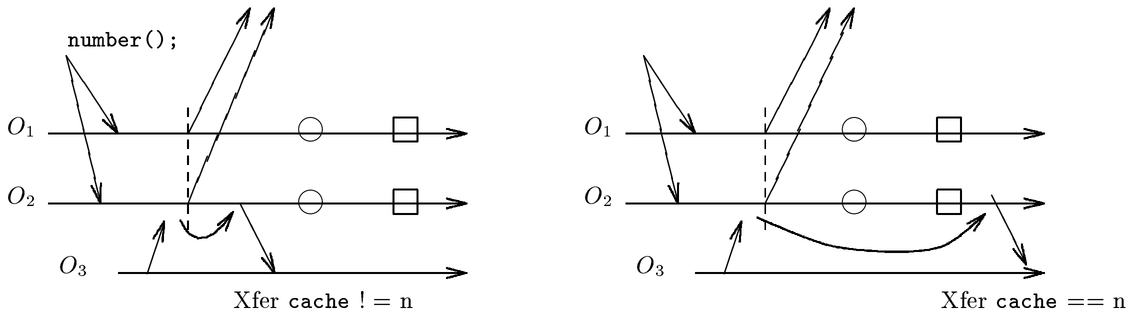

Figure 6: Problem and solution for state transfer problem.

In NILE, this can occur when the SiteManager receives a new job submission. For scheduling purposes, the SiteManager maintains a counter of the number of jobs waiting to be processed. When the SiteManager is ready to process a new job, it invokes a 'start' RMC for the proper JobManager then decrements the job counter. However, when it issues this RMC, the SiteManager releases control; should a new SiteManager replica join at that point, it receives a job counter value that will later be incremented by the other SiteManager replicas.

3.3.2 Thread Rescheduling Order

The cause of the thread rescheduling problem is related to that described above, with similar results in inconsistent state. In Electra, each message received by an object generates a new thread; for object groups this means one thread per replica, per RMC. Multiple invocations of an RMC result in multiple threads per replica. To ensure the replicated object's state is consistent, the threads *of each replica* must be executed in the same order. Again this is usually true, but if a thread releases control before it terminates, then the rescheduling order of the threads at one replica is not guaranteed to be the same as the rescheduling order of the threads at any other replica. The effect is the same as if the RMCs had been issued with an unordered multicast, rather than an ordered atomic multicast. It is possible for the Electra programmer to handle this situation using semaphores.

The code segment and execution sequence shown in Figures 7 and 8 generate such a scenario. There, O_1 and O_2 are replicas of object O. Upon receiving the first RMC, the threads t_{11} and t_{21} are generated by O_1 and O_2 respectively; receiving the second RMC generates threads t_{12} by O_1, and t_{22} by O_2. The RMC being executed then issues an RMC on another object thus releasing execution control. At this point, the rescheduling order within O_1 can be different from that within O_2. If so, the values returned by the RMC can be inconsistent between replicas, even though the RMCs were delivered at both replicas in the same order.

```
read(state);  AnotherObject.anotherRMC();  return(++state);  // RMC of object O
```

O_1 replica. Execution sequence for threads

t_{11}: `read(state); AnotherObject.anotherRMC(); // suspend`

t_{12} `read(state); AnotherObject.anotherRMC(); // suspend`

t_{11}: `return(++state);`

t_{12}: `return(++state);`

O_2 replica. Execution sequence for threads

t_{21}: `read(state); AnotherObject.anotherRMC(); // suspend`

t_{22} `read(state); AnotherObject.anotherRMC(); // suspend`

t_{22}: `return(++state);`

t_{21}: `return(++state);`

Figure 7: Code and execution sequence for RMCs on two replicas of object O.

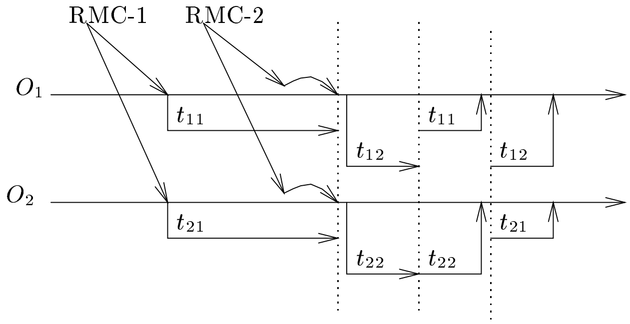

Figure 9: Execution diagram for two RMCs invoked on two replicas of object O.

If the initial value of `state` is 0 in both replicas, then threads t_{11} and t_{21} (spawned by RMC-1 to O) will return 1 and 2 respectively, while threads t_{12} and t_{22} (spawned by RMC-2) will return 2 and 1 respectively.

4 Conclusion

Electra has proven to offer many advantages for building a highly-available distributed object system. Designers and programmers can concentrate on the interaction of CORBA objects, yet still obtain the benefits of fault tolerance through replicated objects. Nevertheless, in complex systems such as NILE, there are still some areas where it is necessary to be aware of the underlying communication layer so that subtle traps can be avoided. Also, although a CORBA-compliant ORB allows a system to be built without regard to the lower layers, when implementing a system with *replicated* CORBA objects, some critical and complex aspects of the group model should and necessarily do appear to the programmer.

Our experience in building a large, complex system in which many facets of fault tolerance co-exist and interact, leads us to the following observations.

1. It may be desirable to allow object groups to dynamically and transparently assume different organizational strategies depending on whether they are acting as server or client. This could improve both execution performance and the simplicity of programming.

2. In deference to standards conformity, it is desirable to restrict RMC delivery to only total order multicasts. However multiple implementations of total order multicast can exist, the use of which will vary depending on the organization of the group. Through this, rather than by supporting multiple semantics for RMC delivery, the desired performance can be achieved.

3. Tools to assist in programming object groups are very desirable. The mismatch between the process-oriented execution model and the object-oriented programming model means that knowledge of subtle aspects of group coordination and fault tolerance is probably inescapable.

Acknowledgements

We would like to thank Silvano Maffeis for his help with and support of Electra and many useful, interesting discussions.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd FTCS*, pages 76–84. IEEE, 1992.

[2] A. Amoroso, K. Marzullo, and A. Ricciardi. Wide-Area Nile: A Case Study of a Wide-Area Data-Parallel Application. Technical Report CS97-566, UC San Diego, 1997.

[3] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *11th Symposium on Operating System Principles*, pages 123–138, November 1987.

[4] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[5] D. Miller et al. CLEO Collaboration. The CLEO III Detector. Technical report, Cornell preprint CLNS 94/1277, 1994.

[6] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming.* PhD thesis, Universität Zürich, 1995.

[7] M. Ogg, G. Sharp, and Nile Collaboration. Nile Design Requirements. `http://www.nile.utexas.edu/Nile/docs/Online`.

[8] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, 1996.