Remote Method Calling and Object-Group Communication*

Position Paper

ECOOP'93 Workshop on Object-based Distributed Programming

Silvano Maffeis
maffeis@ifi.unizh.ch
University of Zurich Dept. of Computer Science
Winterthurerstr. 190, CH–8057 Zurich, Switzerland
Tel: +41 1 257 4327 Fax: +41 1 363 0035

July 1993

Abstract

Many of today's object-oriented distributed toolkits focus on transactions to synchronize distributed applications. Transaction mechanisms are well suited for synchronizing *data-oriented* applications. In contrast, *process-oriented* applications like various kinds of failure tolerant client-server applications involving object-replication, require a much finer degree of synchronization, realizable by virtual synchrony, atomic broadcast, and message passing mechanisms. Toolkits offering these capabilities, for example Isis or Horus, are low-level and therefore difficult to program. In this position paper we present object-oriented abstractions based on virtual synchrony and atomic broadcast primitives to ease the development of failure-resilient client-server applications. Examples of how such abstractions are being realized in a prototype toolkit called Electra illustrate this paper.

Keywords: Failure Tolerance, Replicated Objects, Object-Group Communication, Object Oriented Distributed Programming, Remote Method Calling (RMC), Horus

1 Introduction

Many of today's object oriented systems, for example Arjuna [11] or Avalon-C++ [3] focus on transaction mechanisms and atomic objects to synchronize distributed, concurrent applications. Such mechanisms are powerful and expressive for data oriented applications, where distributed processes can be synchronized by locking certain data-objects, but not adequate enough for a finer degree of synchronization [1]. A fine degree of synchronization is needed by various kinds of failure tolerant client-server applications, such as failure tolerant filesystems, nameservers and so forth. Other group-oriented systems, *i.e.* systems requiring process-groups and group-multicast mechanisms, also need a much finer degree of synchronization than transactions can offer. Examples of such group oriented systems are CSCW applications and applications requiring object-replication

*This work is supported by *Siemens AG ZFE*, Germany and *Schweizer Bundesamt für Konjunkturfragen*, Grants No. 2255.1 and 2554.1

for failure tolerance. Here, replicated objects can be modeled and accessed using process-groups.

The intent of this position paper is to propose *Remote Method Calling (RMC)*, which essentially is an object-oriented variation of the RPC [10] paradigm, and *causal object-group communication* for providing a fine degree of synchronization between distributed threads of execution. We will also exemplify how these abstractions are realized in an object oriented toolkit for failure tolerant client-server applications called the ELECTRA toolkit [9], which is in development at the University of Zurich.

2 Short Overview of the ELECTRA Toolkit

Currently, a small number of programming toolkits exist, mostly in form of C programming libraries, which aid programmers in building process-oriented, failure tolerant distributed applications. Examples are ISIS [1] and HORUS [13]. However, the methods and tools offered by such toolkits are often difficult to use by people lacking many years of experience in building distributed systems, since most of the provided abstractions are low-level and not expressive enough for modeling real world problems. Furthermore, it is mostly impossible to realize reusable software components for distributed systems directly with such programming libraries.

ELECTRA is a C++ toolkit consisting of a class library for multi-threaded services (*active* C++ objects), a class library for *passive* objects which can be communicated between such services, a *service definition language (SDL)* for declaring multi-threaded services using a syntax very similar to C++, a distributed runtime environment and runtime monitoring tools.

The goal of ELECTRA is to ease the development of failure tolerant distributed applications by providing class libraries of reusable components and powerful object-oriented abstractions, such as object-groups, type safe object-group communication and Remote Method Calling.

ELECTRA builds upon the HORUS toolkit [13] developed at Cornell University, Ithaca, NY. HORUS provides primitives for managing threads, for causal broadcast, for consistent views on failures and process-group membership and so forth. The primitives are provided in form of a portable and efficient C library, but the abstraction level is low. For example, the communication in HORUS mainly takes place by communicating buffers of unstructured data. Abstract data types must manually be marshaled by the programmer before they can be communicated.

3 Remote Method Calling

ELECTRA allows an application to send messages to a remote service. These messages may include the standard C++ base types and C++ objects and may return standard C++ base types and C++ objects.

Communication with a remote service is realized using conventional C++ syntax by invoking the methods of a local stub object, i.e. a *proxy*. This proxy is bound to a remote instance of the service where method invocations on the proxy are transparently transmitted. The first parameter to a proxy instantiation is the identification name of the service, which obeys to the format <Domain>:<Path>. The name "cs.cornell.edu:/campus/nameserver" for example refers to the service "/campus/nameserver" running in the domain "cs.cornell.edu".

A special ELECTRA service known as the *master-trader* manages the identification names for a domain. Such a domain usually comprises one LAN.

ELECTRA applications transparently query their master-trader whenever a proxy object is instantiated. The return value of the query is the physical network address of the service to be bound, called the service's *entity ID (EID)*.

ELECTRA supports three kinds of RMC: asynchronous mode, synchronous mode and *promises* [7]. In asynchronous mode, an *upcall* method can be specified. This method will be invoked as an own light-weight process when the RMC returns. The following code fragment demonstrates ELECTRA RMCs:

```
Nameserver ns( "ai.mit.edu:/pub/ns1", BIND );    // bind to a remote service.
Promise promise;                                  // a "promise" object. Used later.
Name name; Address address; ...                   // parameters.

ns.install( name, address, SYNC );                // synchronous RMC.
ns.install( name, address, ASYNC );               // asynchronous RMC.
ns.install( name, address, ASYNC, method );       // with "upcall" method.
ns.lookup( name, address, PROMISE, promise );     // with "promise".
...
promise.wait();                                   // explicit synchronization. Return
                                                  // values are available afterwards.
```

4 Replication

To achieve high availability despite host and network failures, an ELECTRA service can be replicated over several hosts. The replicas belong to the same *object-group*. RMCs to a replicated service are transparently broadcasted to all of the replicas. The programmer still works with a single proxy, which now represents a group of services.

Consistency of the replicated state can be guaranteed by forcing RMCs to occur in the *same order* at all replicas, which is known as *atomically ordered broadcast* [2]. However, some applications can preserver consistency with a weaker, less expensive *causal ordering* [6, 12] using *causal broadcast*.

4.1 Benefits of Object-Groups

Object-groups provide for replication transparency, meaning that client applications (mostly) need not to know whether they communicate with a single instance of a service or with an object-group [5]. Further advantages of object-groups are:

- A nonreplicated object can easily be replaced by an object-group whenever the object becomes overloaded. Parallelism and load sharing can thus be increased step by step, provided that the toolkit offers the required abstraction mechanism which allow to communicate with an object-group as if it were a non-replicated object.
- Availability and fault tolerance are increased by using object-groups to implement critical services. The service then remains operational as long as at least one of the group members is operational.
- Object-groups can be used as reusable building blocks for complex distributed systems. New (replicated) services can be realized by deriving them from existing services by means of class inheritance.
- Objects encapsulate an internal state and make it accessible through a set of well-defined methods. State-encapsulation eases the realization of state-checkpointing and state-migration. Both issues are important for building scalable, failure tolerant applications.

4.2 Declaring Replicated Services

In this section we show how replicated services can be defined in ELECTRA-SDL (Service Definition Language). The following SDL code fragment declares a replicated NameServer which is derived from the super-service ElectraService:

```
replicated service NameServer: public ElectraService {
        cbcast      install( IN Address a, IN Name n );
        cbcast      remove( IN Address a );
        method      lookup( IN Address a, OUT Name n );
}
```

The broadcast-specifier cbcast indicates that install() and remove() operations shall be broadcasted to the members of the object-group using causal broadcast. The method specifier indicates that lookup() requests are not to be broadcasted since they do not alter the service's state. A lookup() can be issued on any replica, preferably on one where fast access is granted. Other supported broadcast-specifiers are abcast for atomically ordered and ubcast for unordered broadcast.

The following code fragment demonstrates how a replicated service is created. Here we suppose that the service is not running already. ELECTRA allows for remote service instantiation by a special ELECTRA service called the Mushroomer [9], which is running on each ELECTRA host.

```
Hosts hosts;
...
Nameserver ns( "ai.mit.edu:/pub/ns2", MUSHROOM, hosts );    // instantiate replicas.
ns.install( name, address, SYNC );                          // use the service.
...
```

The hosts object contains the network addresses of the hosts on which a replica is to be instantiated. The service name "ai.mit.edu:/pub/ns2" is used to register the newly instantiated service. Other applications can now simply BIND to this service.

Code needed to make RMCs and broadcasts transparent is automatically generated by the ELECTRA stub generator.

5 Selective Transparency

The ELECTRA toolkit does *not* enforce full transparency. As seen in the code fragment above, the programmer can specify where instances of a service shall be placed in the network. If the programmer is not concerned about where a service is instantiated, he or she tells the toolkit to decide itself about the location of the service by submitting the special object anywhere which is of type Hosts. Only in this case, the toolkit decides mainly along the local load on the hosts, where to instantiate a service.

Selective transparency is well suited for building large-scale distributed systems. The programmer can decide, along performance, locality of access and security considerations where services are to be instantiated, unless the anywhere specifier is used.

6 Virtual Services Space

ELECTRA hides details of the underlying operating system and of the low-level communication- and broadcast-mechanisms by providing a *virtual services space* [8]. A virtual services space contains a collection of services that cooperate to solve a problem. The virtual services space can extend over several LANs and MANs. ELECTRA partitions service names into administrative domains (Section 3). A *master-trader* manages the part of the virtual services space inside its administrative domain, *e.g.* a LAN. Besides the services names, a master-trader also holds references to master-traders in other domains. Analyzing the <Domain> component of a service name a trader can easily find out whether the name refers to a local service or whether the name has to be forwarded to a trader inside another administrative domain.

7 Virtual Synchrony and Linearizability

Virtual synchrony [1, 12] uses the *happened before* relationship described in the classic paper of Leslie Lamport [6] to synchronize messages in a distributed system. Only when necessary, *i.e.* when a *causal relationship* between such messages exists, messages are reordered or deliberately delayed at the receiving site to match the *happened before* condition. ELECTRA exploits the low-level virtual synchrony capabilities offered by HORUS and encapsulates them into the easier to use replicated service abstraction (Section 4) and Group abstraction.

Linearizability [4] is a correctness condition for concurrent objects that exploits the semantics of abstract data types, thus leading to a higher degree of parallelism. Intuitively, this corresponds to the requirement that "concurrent objects behave in a sane, predictable manner". Since the virtual synchrony model respects causality, operations on ELECTRA services groups would normally be linearizable [1]. Thus, based on virtual synchrony and linearizability we can develop a consistency model which comprises individual ELECTRA services and services-groups.

References

[1] BIRMAN, K. P. Integrating Runtime Consistency Models for Distributed Computing. Tech. Rep. 91-1240, Cornell University Dept. of Computer Science, July 1993. To appear in Journal of Parallel and Distributed Computing.

[2] BIRMAN, K. P., SCHIPER, A., AND STEPHENSON, P. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems 9*, 3 (Aug. 1991).

[3] EPPINGER, J. L., MUMMERT, L. B., AND SPECTOR, A. Z. *Camelot and Avalon.* Morgan Kaufmann Publishers, Inc., 1991.

[4] HERLIHY, M. P., AND WING, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (July 1990).

[5] ISIS DISTRIBUTED SYSTEMS INC., ITHACA, NY. A Response to the ORB 2.0 RFI, Apr. 93.

[6] LAMPORT, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (July 1978).

[7] LISKOV, B., AND SHRIRA, L. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. *ACM SIGPLAN Notices 23*, 7 (July 1988).

[8] LUCCO, S. E. Parallel Programming in a Virtual Object Space. In *OOPSLA Conference Proceedings* (Orlando, Florida, Oct. 1987).

[9] MAFFEIS, S. Electra – Making Distributed Programs Object-Oriented. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems IV (SEDMS IV)* (San Diego, CA, 1993), USENIX. (To appear).

[10] NELSON, B. *Remote Procedure Call.* PhD thesis, Carnegie Mellon University, 1981. CMU-CS-81-119.

[11] SHRIVASTAVA, S. K., DIXON, G. N., AND PARRINGTON, G. D. An Overview of the Arjuna Distributed Programming System. Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK.

[12] VAN RENESSE, R. Causal Controversy at Le Mont St.-Michel. *ACM Operating Systems Review 27*, 2 (Apr. 1993).

[13] VAN RENESSE, R., BIRMAN, K. P., COOPER, R., GLADE, B., AND STEPHENSON, P. Reliable Multicast between Microkernels. In *Proceedings of the USENIX Workshop of Micro-Kernels and Other Kernel Architectures* (Seattle, Washington, Apr. 1992).