# A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming* **

Silvano Maffeis

Department of Computer Science
University of Zurich, Switzerland
maffeis@ifi.unizh.ch

**Abstract.** Under many circumstances, the development of distributed applications greatly benefits from mechanisms like process groups, reliable ordered multicast, and message passing. However, toolkits offering these capabilities are often low-level and therefore difficult to program. To ease the development of distributed applications, in this paper we propose to hide these low-level functions behind object-oriented abstractions such as *object-groups*, *Remote Method Calling*, and *Smart Proxies*. Furthermore, we describe how the ELECTRA toolkit provides such object-oriented abstractions in a portable and highly machine-independent way.

## 1 Introduction

Distributed systems like failure tolerant client-server applications, distributed filesystems, nameservers, groupware systems, and distributed scientific computations usually require a fine degree of synchronization between distributed, concurrent activities. These applications often benefit from the presence of process-groups [5] as a structuring mechanism, and of reliable, ordered multicast [11] to perform one-to-many communication. We shall use the term *directly distributed system* [6] to denote these sorts of applications. More specific, in directly distributed systems, distributed threads of execution interact *directly* with one another while continuously respecting constraints on their joint behavior. On the other side, in data-oriented systems like distributed data management applications, distributed threads of execution typically interact *indirectly* with one another by issuing operations on logically shared data objects. Our work primarily aims at supporting directly distributed systems, though tools for supporting purely data-oriented applications, for instance transactions and atomic objects [22, 9], could be realized on the abstractions we treat in this paper.

An intent of this paper is to propose *object-groups*, *Remote Method Calling*, and *Smart Proxies* as a widely usable and powerful abstraction mechanism for building directly distributed systems. We exemplify how these abstractions are being realized in an object-oriented toolkit, called ELECTRA [16], which is currently being developed at the University of Zurich.

This paper goes beyond previous work on object-oriented, distributed programming, for example [2, 17, 22, 9], in that it proposes a system design to support object-groups in a highly portable fashion. We have been influenced by the work carried out in the HORUS [25], ISIS [5, 4], and in the PANDA project [3], and by ideas described in [12].

The rest of the paper is structured as follows. In the next section we will give a short introduction to the ELECTRA toolkit. Throughout the paper, the toolkit serves to exemplify abstractions and mechanisms we propose. Section 3 introduces Smart Proxies and object-groups. It also addresses

* Supported by Siemens AG ZFE, Germany, by the Union Bank of Switzerland, and by the *Eidgenössisches Volkswirtschaftsdepartement, Kommission zur Förderung der wissenschaftlichen Forschung (KWF)*, Switzerland, Grants No. 2255.1, 2554.1, 2704.1

** This paper appeared in the Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming, Lecture Notes in Computer Science 791, Springer-Verlag 1994

problems inherent to object-group communication. The way ELECTRA supports object-groups and object-oriented, distributed programming independently from the underlying communication platform is described in Sect. 4. Finally, Sect. 5 summarizes and concludes the paper.

## 2 Overview of the ELECTRA Toolkit

Currently, several distributed control technologies (DCT) exist, mostly in the form of `C` programming libraries, which aid programmers in building reliable, directly distributed systems. Examples are AMOEBA [23], DELTA-4 [26], HORUS [25], ISIS [5], PSYNC [18], and TRANSIS [1].

However, the methods and tools offered by such DCT are often difficult to use by people lacking many years of experience in building distributed systems, since the provided functions are mostly low-level and not expressive enough for modelling real world problems. In addition, it normally takes a great effort to port applications written for a specific DCT to another one, since each DCT goes its own ways and provides its own programming paradigms and interfaces.

ELECTRA is a toolkit for object-oriented, distributed programming in C++. It consists of a run-time environment which allows to create, find, and access objects in a computer network, of a CORBA-IDL [8] stub generator, of class libraries, and of simple programming tools to monitor and control distributed applications. For a more detailed description refer to [16].

ELECTRA objects may have an *active* or a *passive* role. Applications can issue operations on remote, active objects. The invocation of an operation on a remote object is carried out using a syntax very similar to conventional C++ method invocations. We use the term *Remote Method Calling (RMC)* to designate this communication paradigm. RMCs are issued by invoking the methods of a local stub-object, a so-called *proxy* [21]. The proxy object exports a logically centralized and logically local service, even when the accessed object is remote and replicated. Although ELECTRA is implemented in C++ and although C++ is the only language ELECTRA provides a binding for at the moment, the proposed abstractions can be realized in other languages as well.

ELECTRA is conceived to run on top of different state-of-the-art DCTs such as the ones mentioned at the beginning of this section. The mechanisms we adopt therefore will be described later in Sect. 4.

### 2.1 Remote Method Calling

As was stated above, RMC means that the methods of a remote object or object-group can be invoked through a local proxy object. The proxy provides the same interface as the remote object it represents. Parameters of remote method invocations may include the standard C++ base types and C++ objects, and typechecking of the RMC parameters is performed at compile-time.

Human-readable names of the form `<Domain>:<Path>` are used to identify ELECTRA objects. The name `"ifi.unizh.ch:/nameserver"`, for example, refers to the object `"/nameserver"` running within the domain `"ifi.unizh.ch"`. An identification name can refer to a single object or to an object-group.

A special ELECTRA object known as the *master-trader* [16] manages the identification names for a domain. In our model such domain normally comprises one LAN.

ELECTRA supports three kinds of RMC: asynchronous mode, synchronous mode, and *promises* [15]. In asynchronous mode, a thread issuing an RMC proceeds without awaiting the reply to the remote operation. In this case an upcall method can be specified, which will be invoked by the sender's run-time system as soon as the reply (containing the RMC's return values) arrives. The upcall obtains an own thread of execution. In synchronous mode, the sender is blocked until the reply has arrived. Promises allow for an intermediate form of synchronization: the RMC is issued asynchronously, but invoking the method `Promise::wait()` blocks the sender until the reply has arrived. The following example demonstrates how an object providing a domain name service can be accessed by different kinds of RMCs:

```
DomainNameServer dns;                                   // Create a proxy.
dns.bind( "ifi.unizh.ch:/nameserver" );                 // Bind to the remote object.

Promise promise;
Name name; Address address;
...
dns.install( name, address, SYNC );                     // Synchronous RMC.
dns.install( name, address, ASYNC );                    // Asynchronous RMC.
dns.install( name, address, ASYNC, method );            // With upcall method.
dns.lookup( name, address, promise );                   // With promise object.
...
promise.wait();                                         // Explicit synchronization.
                                                        // Return value is
                                                        // available afterwards.
```

## 3 Object-Groups

To achieve high availability despite host and network failures, an active Electra object can be replicated over several hosts. Each replica may fail independently. The replicas belong to the same *object-group*, an object-group being a non-empty collection of object instances which cooperate to achieve some common purpose. Internally, communication with an object-group is by means of reliable multicast.

A *homogeneous* object-group consists of objects which all perform the same task, whereas the members of a *heterogeneous* object-group perform different tasks.

If a proxy object is bound to an object-group, the method calls issued on the proxy are transparently multicasted to all of the objects in the group. As depicted in Fig. 1, the programmer always deals with one proxy, which may represent a single object or an object-group.

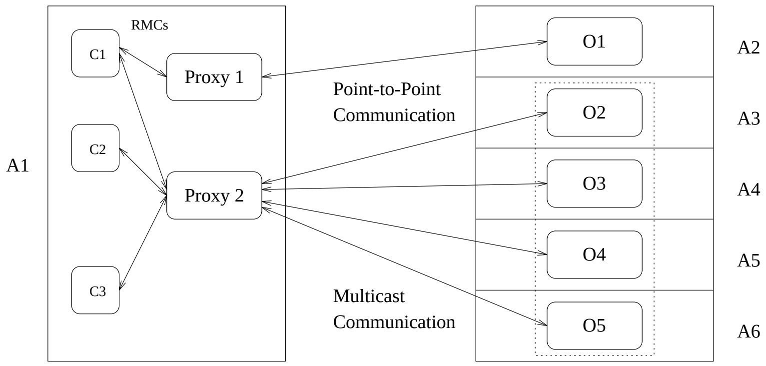


**Fig. 1.** Proxy 1 interfaces to a single object whereas Proxy 2 interfaces to an object-group. C$n$ denotes a client object, O$n$ a remote server object, A$n$ an address space.

### 3.1 Smart Proxies

We propose the term *Smart Proxy* to denote a proxy object which provides more than just a local access point to a remote object. In addition, a Smart Proxy accomplishes at least one of the following tasks:

- *Replication transparency.* A logically non-replicated interface is provided for accessing an object-group. This ensures that single objects can be replaced by object-groups without having to modify the client applications.
- *Replication control.* The degree of replication an object-group provides can dynamically be controlled through the proxy. The proxy provides run-time replication control by closely cooperating with the group management service (see Sect. 3.4) and the naming service.
- *Caching.* To increase the performance of an application, caching can be performed within Smart Proxies. For example, a Smart Proxy interfacing to a nameserver can maintain a local list holding the most frequently accessed nameserver entries.
- *Migration transparency.* A Smart Proxy can be informed by the migration service when the object associated with the proxy was migrated. Thus, object-migration can be hidden from client applications.
- *Concurrency control.* Above tasks are normally performed by a Smart Proxy at the client site. At the server site, a Smart Proxy may allow to specify *encapsulator* objects to define concurrency control schemes [10], message filters, etc. on a per-object basis.

### 3.2 Benefits of Object-Groups

Electra proxies provide for replication transparency, meaning that client applications need not know whether they communicate with a single object or with an object-group. Providing an object-group abstraction is interesting for the following reasons [13]:

- *Load sharing.* A non-replicated object can be replaced by an object-group whenever the object becomes overloaded, and without having to modify applications which use the object. Parallelism and load sharing can thus be increased step by step, provided that the toolkit offers abstraction mechanisms like Remote Method Calling and Smart Proxies, which allow communication with an object-group as if it was a non-replicated object.
- *Fault tolerance.* Availability and fault tolerance are increased by using a homogeneous object-group to implement a service. Each object can fail independently from the other objects in the group. The service then remains operational as long as at least one of the group members is operational.
- *Addressing construct.* Object-groups can be used for addressing a set of objects which share some common characteristics. For example, some objects might want to be informed whenever a certain event occurs. A heterogeneous object-group holding these objects can be created and a multicast be sent to the group whenever the event occurs.
- *Data publication.* Many applications, for example in the financial or flight reservation domain, need a mechanism for transmitting the same information to several recipients. This can be done efficiently by means of object-group multicast.
- *Distributed monitoring and management.* Object-groups provide a convenient way to distribute and gather monitoring and management information in distributed applications.
- *Encapsulation.* Object-groups encapsulate an internal, possibly replicated state and make it accessible through a set of well-defined methods.
- *Reusable components.* We see object-groups as reusable components for complex, failure-resilient distributed systems. New object-groups can be realized based on existing object-groups by means of inheritance.

Object-groups do present problems as well. The issues we consider next are the ordering of events within an object-group, group membership management, and the implementation of object-groups.

### 3.3 Ordering of Events

Within a homogeneous group, consistency of the replicated state can be guaranteed by forcing operations to occur in the same order at all the replicas, which is known as *atomically (or totally) ordered multicast* [11]. However, some applications can preserve consistency with a weaker, less expensive *causal ordering* using *causal multicast* [14, 24]. An advantage of causal multicast compared to more restrictive protocols, is that the protocol works without the introduction of extra messages.

**Causal Multicast.** Causal communication uses the *happened before* relationship described in the classic paper of Leslie Lamport [14] to order messages in a distributed system. Causal communication is a fundamental concept of asynchronous distributed systems and can be applied to settings where senders communicate with several receivers and where messages are delivered indirectly through intermediary processes.

Informally, causal communication generalizes the notion of a message depending on another one, and ensures that a message is artificially delayed by the receiver's run-time system if the message depends on one which the receiver has not seen yet.

Formally, causal dependency is defined as follows [14]:

1. If $a$ and $b$ are events in the *same* process, and $a$ happened before $b$, then $b$ causally depends on $a$.
2. If $a$ is the sending of a message by one process and $b$ is the receipt of this message by another process, then $b$ causally depends on $a$.
3. If $b$ causally depends on $a$, and $c$ causally depends on $b$, then $c$ also causally depends on $a$.

Thus, a message depends on another one if there exists a *data flow* between the two messages. In Fig. 2, message $c$ depends on message $a$ and message $d$ depends on message $b$. In the left scenario, message $d$ surpasses message $c$, and $d$ is thus delivered before $c$. In this situation, causality is violated. The right scenario demonstrates the case where a causal communication protocol is implemented in the distributed system. Now, the run-time system of process C artificially delays message $d$ until $c$ arrives, thus ensuring causality.

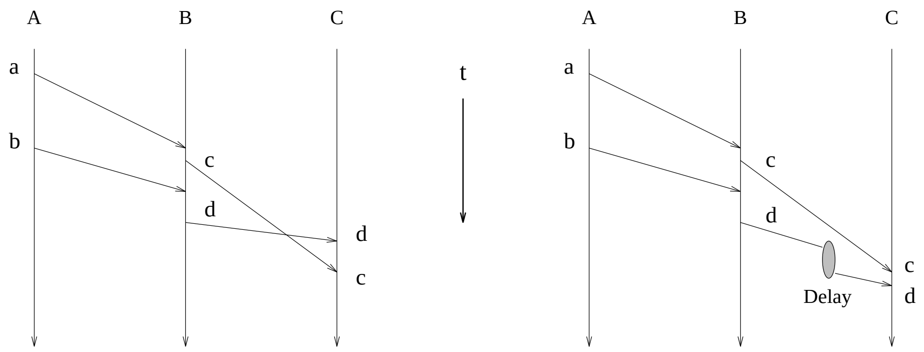


**Fig. 2.** Non-causal and causal communication. A, B and C are processes in a distributed system. An arrow denotes a message passed from one process to another.

### 3.4 Group Membership Management

The problem of group membership management consists in maintaining consistent views on group membership among the objects in an object-group. At all times, an object-group member has a certain idea of which other objects belong to its group. This is called a *view*. Since objects may join or leave (terminate or fail) a group, a view is subject to *view-changes*, and these changes must be propagated to all of the group members. Therefore, a system-wide service providing consistent information on group membership must be realized. To guarantee consistency, the service delivers view changes to all of the members in the same order. Without such a service, object-group based applications may behave incorrectly and in unexpected ways.

Basically, a view change protocol consists of two phases. In the first phase, the group membership service ensures that every member which is still alive receives every message that was sent in the current view (*flush* phase). In the second phase, the view is updated and transmitted to the members (*view update* phase). Similar group membership protocols are implemented in Isis and Horus. The group management problem in asynchronous, distributed systems is treated in [19, 20].

### 3.5 Implementing Object-Groups

In this section we address the problem of implementing object-groups on distributed control technologies (DCT) such as Isis and Horus. We chose these two systems since they are available today and provide important mechanisms like causal communication, process groups, and reliable multicast in an efficient and stable fashion.

In the current version of Isis, the unit of distribution is a process. On the other hand, the object-group model is of finer granularity. A process can instantiate many objects, and situations can thus arise in which a process tries to become member of a group several times. Since in Isis the same process cannot be member of a group multiple times, one must implement a multiplexing mechanism, such that a process containing several objects forwards incoming messages to the appropriate one. This is achieved by providing an Isis multicast-message with addressing information on the destination objects, *e.g.*, with *object-identifiers*. Multiplexing code within each receiver-process forwards the incoming messages to the appropriate objects in the process. No modifications to the Isis system need to be made for implementing this mechanism. We believe that the performance penalty of this multiplexing mechanism is minor and that multiplexing can be realized in a way which is transparent to the programmer.

The Horus toolkit provides light-weight processes (threads) as the unit of distribution. Threads can be grouped together and do communicate by point-to-point messages and reliable multicast. In Horus it is thus possible to implement multithreaded objects, and a thread acting as communication endpoint can easily be created per object. In this situation, object-group communication can be implemented without having to provide a multiplexing layer. In the following section we describe how object-groups are implemented in Electra independently from the underlying DCT.

## 4 The Electra Architecture

An important goal of Electra is to allow for object-group oriented, distributed programming on different state-of-the-art DCTs, and to ease the integration of new DCTs into Electra.

Therefore, Electra applications are based on a *virtual machine (VM) layer* which completely hides the implementation-dependent aspects of the underlying DCT (Fig. 3). The VM layer is implemented in the form of an abstract C++ class, which defines the low-level "instruction set" Electra needs to support object-groups and the other abstractions we mentioned before. Comprised in the instruction set are primitives for handling threads, for message passing, reliable multicast, thread synchronization, and group management. Choosing an adequate instruction set is crucial: a small and simple instruction set requires that much more functionality must be implemented into Electra. A large and complicated instruction set is difficult to map onto existing DCTs. In Sect. 4.1 we will therefore give a detailed overview of the instruction set we chose for the Electra VM.

To have Electra running on a concrete DCT, a so-called *concrete machine*, a specific *adaptor object* must be realized therefore (Fig. 3). The adaptor object's task is to map the VM's operations onto the operations provided by the concrete machine, and to cleanly encapsulate the program-code which is specific to the concrete machine.

Concrete machines which closely meet the requirements of Electra are, for example, Amoeba, Delta-4, Horus, Isis, Psync, and Transis. If required, there would be a possibility of implementing an adaptor object for having Electra to run on the bare UNIX[3] IPC mechanisms, but this would mean re-implementing most of the functionality provided by already existing concrete machines. Such undertaking is beyond the scope of our work. Currently, an adaptor object is provided for the Horus toolkit only.

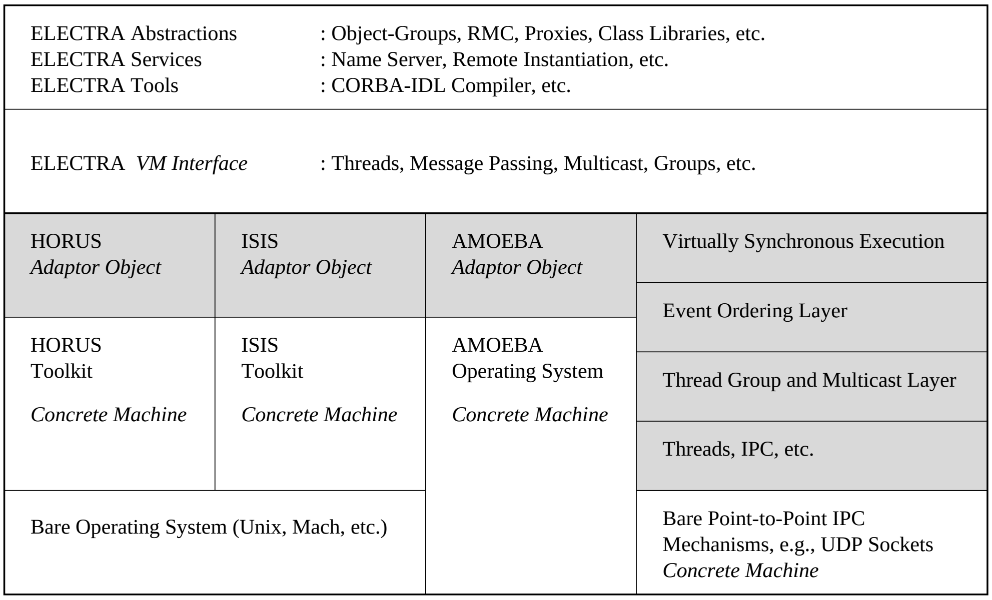


**Fig. 3.** Sample Electra architecture supporting Horus, Isis, Amoeba, and conventional point-to-point IPC mechanisms. The functionality in the shaded areas is provided by adaptor objects.

To summarize, above system design allows to execute the same object-oriented application on top of a variety of more low-level concrete machines. This is achieved by building Electra on a portable VM interface. Adaptor objects are used to map the VM's primitives onto the operations provided by the underlying, concrete machine.

### 4.1 Virtual Machine Interface

We now describe the important parts of the current Electra VM interface. As stated above, the VM is encapsulated into an abstract C++ class. Adaptor objects are instances of subclasses of the VM. The VM provides interfaces for message passing, threads, thread synchronization, multicast, and group management. These are the low-level primitives we have found necessary for building a toolkit which supports object-groups and object-oriented, distributed programming.

The Electra programmer is normally not confronted with the VM primitives we describe in the next sections. Rather, the primitives are called from the code produced by the Corba-IDL compiler and from within the Electra run-time system.

[3] Unix is a registered trademark of Unix Systems Laboratories, Inc.

**Message Passing.** ELECTRA requires methods for a reliable, non-blocking transmission of unstructured data-blocks of arbitrary length. These primitives are called by the RMC module for the low-level delivery of RMCs. If the message passing mechanisms of the underlying concrete machine fail to meet a requirement, reliability for instance, the specific adaptor object must implement the requirement. This is transparent to the layers above the VM. Following the VM's message passing interface:

```
virtual Status msgSend(Entity destination, Message msg, Thread done);
virtual Status msgReceive(Entity listenOn, Thread receive);
```

A `Status` object reports about success or failure of an operation. To deliver a message to a specific destination, `msgSend` is invoked. The opaque object `destination` encapsulates the addressing information needed by the concrete machine for the delivery. When the message has been delivered, thread `done` is invoked. `msgReceive` registers a thread which is invoked whenever a message arrives for entity `listenOn`.

**Threads.** Our VM interface for handling threads is as follows:

```
virtual Status threadDeclare(Thread& t);
virtual Status threadCreate(Thread t, ThreadParam param);
virtual Status threadDestroy(Thread t);
```

Threads are declared to the concrete machine using `threadDeclare`, and are started using `threadCreate`. The creator of the thread can specify a parameter `param` which is to be passed to the thread once created. `threadDestroy` frees the internal resources associated with a thread.

The thread interface is held very simple to allow existing thread implementations to easily slide under this interface. ELECTRA uses the thread interface mainly for creating RMC upcall methods.

**Thread Synchronization.** ELECTRA's synchronization primitives, promises for example (see Sect. 2.1), all build on the simple semaphore operations provided by the VM:

```
virtual Status semaCreate(Sema& s, int value);
virtual Status semaDec(Sema s);
virtual Status semaInc(Sema s);
virtual Status semaDestroy(Sema s);
```

A semaphore is created and initially set to `value` using the `semaCreate` operation. `semaDec` decrements a semaphore, and blocks the issuing thread if the semaphore's value was `zero`. `semaInc` increments a semaphore. If the semaphore's value was `zero`, and some thread is blocked trying to decrement the semaphore, the thread is resumed.

**Multicast.** To realize object-group communication, ELECTRA requires operations for reliable, order preserving multicast:

```
virtual Status mcastSend(GroupEntity group, Message msg, Thread done);
virtual Status mcastReceive(Entity listenOn, Thread receive);
```

To low-level deliver an RMC to an object-group, `mcastSend` is used. The addresses of the receivers are encapsulated in the group object. In analogy to `msgSend`, the `done` thread is invoked when resources associated with the message can be released. `mcastReceive` is used by a group member to register a thread which is to be invoked when a multicast arrives for it.

**Group Management.** Concrete machines like AMOEBA, DELTA-4, HORUS, ISIS, and TRANSIS provide protocols for holding the view on group membership (see Sect. 3.4) each member of a group has at a certain time consistent. This is possible despite join and leave requests, and despite failures. Next, the low-level interface ELECTRA uses to manage object-groups is provided.

```
virtual Status groupCreate(GroupEntity& group, GroupPolicy policy);
virtual Status groupJoin(GroupEntity group, Entity e);
virtual Status groupLeave(GroupEntity group, Entity e);
virtual Status groupDestroy(GroupEntity group);
```

The method `groupCreate` is used to generate a group. A `GroupPolicy` object specifies parameters like the required ordering of the messages delivered to the members, *e.g.*, unordered, causally ordered, or totally ordered multicast.

Objects are inserted into and deleted from groups by the `groupJoin` and `groupLeave` methods. Finally, a group is removed using `groupDestroy`.

**Declaring an Application's Requirements.** Being able to run object-group based applications unmodified on a variety of communication platforms is enticing, but bears following problem: different concrete machines offer different services and features. AMOEBA for instance, supports totally ordered multicast, but does not provide for causal ordering of point-to-point messages in respect to multicasted messages. HORUS and ISIS support totally ordered and causally ordered multicast, and are able to enforce causality in a system which employs point-to-point and multicast communication. This can take place even in the presence of overlapping groups and failures. To cope with this incompatibility problem, we see several solutions:

- Finding the "least common denominator" of the concrete machines we want to support. We could say that all group-communication in ELECTRA is totally ordered, and that point-to-point communication will not be ordered in respect to group-communication. This solution leads to a simple programming model, but is restrictive in the sense that the ELECTRA programmer is prevented from adequately exploiting the underlying concrete machine. Furthermore, the programmer will be seduced to circumvent the VM abstraction and to directly issue operations on the underlying concrete machine, thereby limiting the portability of applications.
- Finding the superset of the capabilities provided by the concrete machines. We could say that the ELECTRA programmer can always select from a variety of ordering rules [11], such as unordered, FIFO, causal, atomic, and causal atomic multicast. Furthermore, the programmer can specify how point-to-point messages are to be ordered in respect to multicasted messages and in respect to other point-to-point messages. Clearly, developing an adaptor object would become difficult and tedious for concrete machines which do not provide such plenty of ordering rules.
- A clean solution would consist in realizing the ordering of events completely in an ELECTRA layer residing above the adaptor objects. The underlying concrete machine would then provide for unordered, reliable group-communication only. ELECTRA could offer different ordering rules using a flexible approach like PSYNC [18] or protocols like the ones implemented in ISIS [7]. The realization of such protocols as future work is under consideration.
- The solution we actually provide consists in having the programmer specifying the requirements her or his application embodies, for example the kind of multicast needed, and "asking" the underlying adaptor object whether it can fulfill the requirements.
  If an important application-requirement is not fulfilled, the programmer now may enhance the adaptor object to provide the functionality which is missing in the concrete machine, or configure the application to use another machine. Querying the capabilities of the underlying adaptor is achieved by filling out and passing an `AppPolicy` object to the adaptor.

The next code fragment describes the part of the VM interface which realizes the policy mechanism. Method `isSupported` checks whether the application's requirements, specified in the `policy` object, are supported by the concrete machine. The method `whatsSupported` returns the characteristics of the concrete machine.

```
virtual Status isSupported(AppPolicy policy, boolean& supported);
virtual Status whatsSupported(AppPolicy& policy);
```

### 4.2 Multiple Virtual Machines

Another advantage of the proposed design is that an object-group based application can use different concrete machines simultaneously. Since the VM class uses virtual C++ methods, different concrete machines can be accessed from within the same application using the same VM interface.

For example, consider a banking application where some customer data shall be retrieved from a mainframe running an exotic operating system and be evaluated in parallel on a set of workstations. In Electra, the parallel evaluation application could be run using Isis as concrete machine. Most likely, Isis will not run on the mainframe, but probably the data could be retrieved from the mainframe using a public data network protocol like X.25. Now, an Electra adaptor object would be implemented which maps the message passing primitives (see Sect. 4.1) onto calls to operations of an X.25 programming library. The rest of the adaptor's methods would be left unimplemented. On the host side, a program would be realized[4] which interprets the incoming RMCs, issues queries on the customer database, and returns the requested data. Now, a well structured, object-oriented Electra application can be realized which uses RMCs to retrieve data from the bank's mainframe, and Isis to perform a parallel evaluation on the workstations.

The Electra programmer has the illusion of communicating with a remote object, by invoking RMCs such as `getCustomerData`, though an exotic mainframe resides behind this abstract interface. Similarly, adaptor objects could be implemented for interconnecting different platforms like Amoeba, Delta-4, Horus, Isis, Psync, and Transis, and practicing object-oriented, distributed programming on the resulting, virtual platform.

## 5 Summary

We suggested object-groups as a general structuring mechanism for object-oriented distributed systems. With the help of object-groups, programmers are less concerned about important but low-level issues like reliable multicast, ordering of events, and replication.

The Remote Method Calling (RMC) paradigm provides expressive, object-oriented communication and allows to access both single objects and object-groups with the help of Smart Proxies. With this approach, a single system component can be replaced by an object-group without having to modify applications which use the component. Thus, fault-tolerance and load-sharing can be increased step by step and complex applications become easier to realize, to modify, and to maintain.

Problems related with object-groups are the consistent ordering of messages delivered to the group-members, group view management in confrontation of joining, exiting and failing objects, and the implementation of object-groups on today's state-of-the-art platforms. Advantages of object-groups are that they allow for load sharing, fault tolerance, group addressing, data publication, distributed management, encapsulation, and reuse.

We further presented a flexible system design which we consider suited to support object-groups and object-oriented, distributed programming on various state-of-the-art platforms. This flexibility

[4] Though the VM and the Electra layers on top of it could be compiled and installed even on a mainframe, if it provides a C++ compiler. These modules are written in an operating system independent way.

is achieved by ELECTRA's *virtual machine* abstraction. The virtual machine defines a small set of simple methods, mainly for handling threads, for message passing, multicast, thread synchronization, and group management. *Adaptor objects* are used to map the virtual machine's primitives onto the operations provided by the underlying, *concrete machine*, thus cleanly encapsulating the program-code which is specific to the concrete machine.


## Acknowledgements

Henri Bal, Martin Bichsel, Ken Birman, Richard Golding, Andrew Hutchison, Robbert van Renesse, and the anonymous reviewers provided useful suggestions and comments on this work.


## References


1. AMIR, Y., DOLEV, D., KRAMER, S., AND MALKI, D. Transis: A Communication Sub-System for High Availability. In *22nd International Symposium on Fault-Tolerant Computing* (July 1992), IEEE.
2. ARCHITECTURE PROJECTS MANAGEMENT LTD. *ANSAware Version 4.1 Manual Set.* Castle Park, Cambridge UK, Mar. 1993.
3. BHOEDJANG, R., RUHL, T., HOFMAN, R., LANGENDOEN, K., BAL, H., AND KAASHOEK, F. Panda: A Portable Platform to Support Parallel Programming Languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems IV* (San Diego, CA, Sept. 1993), USENIX.
4. BIRMAN, K. P. Integrating Runtime Consistency Models for Distributed Computing. Tech. Rep. 91-1240, Department of Computer Science, Cornell University, July 1993. To appear in Journal of Parallel and Distributed Computing.
5. BIRMAN, K. P. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM 36*, 12 (Dec. 1993).
6. BIRMAN, K. P., AND JOSEPH, T. A. Exploiting Replication in Distributed Systems. In *Distributed Systems*, S. Mullender, Ed. ACM Press, 1989.
7. BIRMAN, K. P., SCHIPER, A., AND STEPHENSON, P. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems 9*, 3 (Aug. 1991).
8. DIGITAL EQUIPMENT CORP., HEWLETT-PACKARD CO., HYPERDESK CORP., NCR CORP., OBJECT DESIGN INC., SUNSOFT INC. *The Common Object Request Broker: Architecture and Specification*, Dec. 1991. Revision 1.1, OMG Document Number 91.12.1.
9. EPPINGER, J. L., MUMMERT, L. B., AND SPECTOR, A. Z. *Camelot and Avalon.* Morgan Kaufmann Publishers, Inc., 1991.
10. GUERRAOUI, R. Towards Modular Concurrency Control for Distributed Object Oriented Systems. In *IEEE Proceedings of the International Workshop on Future Trends in Distributed Computing Systems* (Sept. 1993).
11. HADZILACOS, V., AND TOUEG, S. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, S. Mullender, Ed., second ed. Addison Wesley, 1993, ch. 5.
12. HAGSAND, O., HERZOG, H., BIRMAN, K., AND COOPER, R. Object-Oriented Reliable Distributed Programming. In *Proceedings of the IEEE Workshop on Object Orientation in Operating Systems* (Paris, France, Sept. 1992).
13. ISIS DISTRIBUTED SYSTEMS INC., ITHACA, NY. *Object Groups: A Response to the ORB 2.0 RFI*, Apr. 1993.
14. LAMPORT, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (July 1978).
15. LISKOV, B., AND SHRIRA, L. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. *ACM SIGPLAN Notices 23*, 7 (July 1988).
16. MAFFEIS, S. Electra – Making Distributed Programs Object-Oriented. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems IV* (San Diego, CA, 1993), USENIX.
17. MARQUES, J. A., AND GUEDES, P. Extending the Operating System to Support an Object-Oriented Environment. In *OOPSLA Conference Proceedings* (Oct. 1989).
18. PETERSON, L., BUCHHOLZ, N., AND SCHLICHTING, R. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems 7*, 3 (Aug. 1989).

19. Ricciardi, A. M. *The Group Membership Problem in Asynchronous Systems.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, Nov. 1992. No. 92-1313.
20. Schiper, A., and Ricciardi, A. Virtually-Synchronous Communication Based on Weak Failure Suspectors. Tech. Rep. 93-1339, Department of Computer Science, Cornell University, Ithaca, NY, Apr. 1993.
21. Shapiro, M., et al. SOS: An Object-oriented Operating System – Assessment and Perspectives. *Computing Systems 2*, 4 (Dec. 1989).
22. 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.
23. Tanenbaum, A. S. *Modern Operating Systems.* Prentice Hall, 1992.
24. van Renesse, R. Causal Controversy at Le Mont St.-Michel. *ACM SIGOPS Operating Systems Review 27*, 2 (Apr. 1993).
25. 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).
26. Veríssimo, P., and Marques, J. A. Reliable Broadcast for Fault-Tolerance on Local Computer Networks. In *9th Symposium on Reliable Distributed Systems* (1990), IEEE.

This article was processed using the LaTeX macro package with LLNCS style