# Adding Group Communication and Fault-Tolerance to CORBA

Silvano Maffeis
Department of Computer Science
Cornell University, Ithaca, NY

# Adding Group Communication and Fault-Tolerance to CORBA*†

Silvano Maffeis
maffeis@acm.org
*Department of Computer Science*
*Cornell University, Ithaca, NY*

## Abstract

Groupware and fault-tolerant distributed systems stimulate the need for structuring activities around object-groups and reliable multicast communication. The object-group abstraction permits to treat a collection of network-objects as if they were a single object; clients can invoke operations on object-groups without needing to know the exact membership of the group. Object-groups mainly serve to increase reliability through replication, performance through parallelism, or to distribute data from one sender to a large number of receivers efficiently. This paper describes how object-groups and reliable multicast communication can be added to a CORBA compliant Object Request Broker. It also presents ELECTRA — a CORBA Object Request Broker whose architecture is pervaded by the group concept.

**Keywords:** Object-Groups, Multicast, Replication, CORBA, Electra, Horus, Isis

## 1 Statement of Problem

### 1.1 One World: CORBA

Object-oriented programming is believed to be one of today's best programming models to cope with complex systems while providing maintainability, extensibility, and reusability [14]. A model claiming such attributes is particularly interesting for distributed systems, as these tend to become very complex.

In industry, the client-server model is finding increasing consideration for interconnectivity. In this model, servers provide clients with access to services such as file storage or authentication by IPC mechanisms like message passing or RPC. Object-oriented, *distributed* programming (OODP) is a generalization of the client-server model, in that objects encapsulate an internal state and make it accessible through a well-defined interface. Client applications may import an interface, bind to a remote instance of it, and issue remote object invocations [6]. This use of objects naturally accommodates heterogeneity and autonomy: heterogeneity since messages sent to objects depend only on their interfaces and not on their internals, autonomy because object implementations can change transparently, provided they maintain their interfaces [16].

In 1989, the Object Management Group (OMG) started elaborating an open standard for OODP, called the Common Object Request Broker Architecture (CORBA) [18]. At the time of this writing, OMG counted more than 500 members worldwide, including enterprises like Apple, DEC, HP, IBM, NCR, Novell, and SUN. Many of the large software producing firms have committed to make their products comply with CORBA withing the next few years.

People living in what we could call the "CORBA world" emphasize aspects such as reusability, portability, interoperability, and integration of legacy information systems. Unfortunately, the current version of CORBA as well as the Object Request Brokers (ORBs) in use today do not adequately treat two of the most fundamental problems which occur in real-world distributed applications, namely partial failures and consistent ordering of distributed events [9]. In face of partial failures, large distributed applications implemented with current ORB technology may behave in an unpredictable way, leading to inconsistent data or to other malfunctions. Moreover, reliable asynchronous communication is not adequately supported in the CORBA standard, though asynchronous communication is important for building scalable distributed applications, as it permits to overlap computation with communication and to hide network latencies.

*Research supported by grants from the Swiss National Science Foundation, Siemens-Nixdorf, Union Bank of Switzerland, and KWF/CERS

†In: Proceedings of the USENIX Conference on Object-Oriented Technologies, Monterey, CA, 1995

### 1.2 Another World: Horus, Isis, etc.

In current distributed systems research, there is a growing community working on problems related to partial failures and on system models to ensure predictable behavior of distributed applications. Examples of such models are "Virtual Synchrony" [3], "Pseudo Synchrony" [19], and "View-Synchronous Communication" [1, 20]. The models are similar; they mainly ensure that related actions taken by different processes of a distributed system are mutually consistent, and that actions are timely correct even when multiple components communicate asynchronously to perform some task. Hence, by following these models the behavior of a distributed application is predictable despite crashes, asynchronous communication, and in spite of processes joining and leaving the system dynamically.

Process-groups and fault-tolerant multicast are at the heart of the models. The models mainly differ in how membership-changes are propagated to the members of a process group, in how partitioned networks are treated, and in how context information associated with messages is represented.

In contrast to CORBA, these models address issues related to partial failures, process replication, reliable multicast, asynchronous communication, and ordering of events. Implementation of robust distributed systems on conventional hardware is enabled by toolkits that follow these models, e.g., by Horus [25], Isis [3], Transis [1], and Consul [15]. Unfortunately, the programming interface provided by these toolkits is proprietary and rather low-level; it mainly consists of a rich set of C-procedures presenting access to light-weight processes, unstructured messages, process groups, message passing primitives, and so forth. Moreover, it is hard to port an application from one toolkit to another.

### 1.3 Electra: Bridging the Gap

Indeed, the two worlds provide complementary functionality and are both very important for future distributed systems. The goal of our work has been to design and implement Electra — a novel programming environment combining the benefits of CORBA with the strengths of systems like Horus and Isis. Electra is a CORBA-compliant ORB which, in addition to the functionality provided by today's ORBs, permits the grouping of object-implementations, reliable multicast communication, and object-replication. Electra is conceived to run on platforms such as Horus and Isis. We believe that an ORB should be based on primitives as provided by Horus or by one of the other aforementioned toolkits, and not directly on the primitives of contemporary operating systems, e.g., RPC, UNIX sockets, or Windows NT pipes. In addition, a flexible system design has been devised allowing developers to customize Electra for various toolkits. Hence, Electra can be thought of as a *generic* Object Request Broker.

Details of the Electra object model are provided in [13], whereas in [12] the performance of Electra is assessed. The rest of this paper is structured as follows. In the next section we describe the object model underlying our ORB. Group-communication is addressed in Section 3, whereas Section 4 describes enhancements we made to the standard CORBA interfaces in order to support group-communication and fault-tolerance. Section 5 deals with the mapping of CORBA interfaces onto object-references that support both unicast and group-communication. Portability of Electra is addressed in Section 6. Finally, Section 7 compares Electra with conventional ORBs and concludes the paper.

## 2 The Electra Object Model

In our work we consider *asynchronous distributed systems* consisting of objects which run on a collection of machines, and which interact by message passing or remote method invocation. In asynchronous systems, there are no constraints on the speed at which objects make progress and on the message transmission delays. Furthermore, neither an exact synchronization of the local clocks nor a reasoning based on "global time" [9] is possible. In this situation, interprocess communication remains the only feasible means of synchronization. We further assume that failures respect the *fail-stop* model [21], which means that objects fail by crashing without the emission of spurious messages.

In analogy to the OMG Object Management Architecture (OMA) [23], the Electra object model consists of objects implemented in various programming languages scattered over an arbitrary number of machines. The ORB is the communication heart in the model. It provides an infrastructure allowing objects to communicate, independent of the specific programming languages and techniques used to implement the objects. Client applications use object-references to send messages to remote object-implementations. References are valid across node boundaries and can thus be passed from one node to another. Furthermore, objects are not tied to a client or server role; a client acting as server at a certain moment can become a client at a later point and vice versa.

Electra enhances OMA in that objects can be aggregated to so-called *object-groups* [11]. An object-group is a means of combining network-objects and naming them as a unit (Figure 1). Communication is by *reliable multicast*, which means that a CORBA operation issued through a group-reference is received by all implementations which are members of the group.

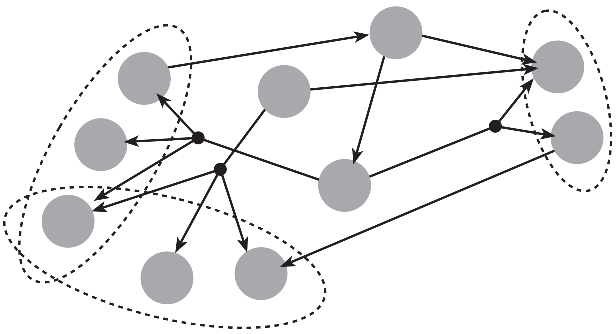

Figure 1: Electra objects communicating by point-to-point and multicast invocations. Circles represent objects, dotted ovals define object-groups.

Programmers can bind object-references to both singleton objects and object-groups using the same programmatical expressions, and multicasts can be issued both through the CORBA static and dynamic invocation interface. Object-group communication can be performed in a transparent or in a non-transparent way. In transparent mode, an object-group appears as if it was a highly available singleton object. In contrast, non-transparent communication permits programmers to access the results of an invocation which were produced by the individual group members.

Another difference to CORBA is that object invocations can be performed synchronously, asynchronously, or deferred-synchronously, through both the static and dynamic invocation interface, and both for singleton and group destinations. Within an object-implementation, each invocation obtains its own thread of execution. The Electra model is thus inherently asynchronous and multi-threaded.

Applications fitting our model are such requiring efficient multicast, asynchronous communication, fault-tolerance, or a combination thereof. Prospective application areas of Electra are video-on-demand servers, video conferencing systems, groupware, distributed parallel computing, and different kinds of fault-tolerant client-server applications.

# 3 Object-Groups

## 3.1 Motivation

In distributed systems, communication can take two different forms depending on the number of participants: point-to-point and multicast. The first form is trivial and is provided by conventional communication mechanisms like message passing or RPC. The second form is more powerful and more complicated than point-to-point communication, it has recently received much attention [3, 7, 27, 4, 8]. *We believe that the conjunction of the group communication model with the object model will lead to a compelling programming paradigm for future distributed systems.*

If the underlying communication hardware or software provides a multicast facility, as it is the case in the Ethernet or in an extended version of the IP protocol (so-called IP-Multicast or MBONE [2]), Electra can take advantage of it to transmit group invocations efficiently. In the worst case, a multicast is mapped into one point-to-point message per group member. This affects the performance of an Electra application but not the programming model.

## 3.2 Reliability

If Electra is configured to run on one of the toolkits we mentioned in Section 1.2, object-group communication is reliable. This means that when an operation is issued through a group-reference, all operational group members will dispatch the same set of operations (Agreement), that this set will include all operations multicast by operational objects to the group (Validity), and that no spurious operations will ever be dispatched (Integrity) [5]. Low-level reliable multicast is realized by the underlying toolkit and not by Electra. Electra's contribution is in providing an easy-to-use, CORBA-compliant interface to group communication.

## 3.3 Ordering of Events

Creating object-groups as well as joining and removing objects from groups is accomplished by special Electra operations which were included into the CORBA Basic Object Adapter (BOA). When creating an object-group, programmers specify ordering requirements for the invocations dispatched by the group members. The ordering protocols available depend on the underlying toolkit. For instance, if Electra is configured for Isis, programmers can specify that all group members dispatch all operations in exactly the same order (Isis *abcast* protocol), in a causal order [9] (Isis *cbcast* order), or by Isis *gbcast* protocol [3]. If the programmer specifies an ordering-protocol that is not available in the underlying toolkit, an exception is thrown.

### 3.4 Application Areas

Object-groups have many interesting application areas:

- **Fault-Tolerance**: Availability and fault-tolerance of an object are increased by using active replication, passive replication, or multi-versioning. Each approach can be mapped onto object-groups. In Electra, an object fails independently of the other members[1], and the service remains available as long as at least one of the members is operational.
- **Load Sharing**: A singleton object can be replaced by an object-group when the object becomes overloaded, and, in many cases, without having to modify applications which use the object. For instance, the group members may share the available work-load to increase throughput. Parallelism and load sharing can thus be increased step by step.
- **Caching**: In certain situations, the response time of a service is decreased if provided by an object-group, since member-objects can be placed at the sites where the service is frequently accessed.
- **Efficient Data Distribution**: Object-group multicast can be mapped onto hardware (e.g., Ethernet) or software (e.g., MBONE [2]) multicast facilities. This allows the same network-message to be received by all members of an object-group.
- **Network Management**: Object-groups offer a convenient solution to the problem of propagating monitoring and management information in distributed applications.
- **Mobility**: In Electra, multicasts are issued through opaque CORBA object references and senders do not need to know the network addresses of the members. Consequently, an object can leave a group, move to another place, then rejoin the group and continue working. Object-groups hence offer support for mobility and for system reconfiguration.

[1] provided that the group members were instantiated on different machines.

## 4 The BOA and Environment Class

Electra is implemented in the C++ programming language and C++ is the only target-language supported by the present version[2]. Our IDL-to-C++ mapping follows the specification in OMG Document 94-9-14 [17]. To add group-communication and fault-tolerance to CORBA, only two C++ interfaces had to be enhanced with a few special operations: operations for managing object-groups were included in the `BOA` class, while operations for selecting an invocation style were added to the `Environment` class. Note that both classes still comply with the CORBA standard, since new methods were added to them without altering signature or semantics of the standard methods.

### 4.1 Enhanced BOA Interface

In Electra, an object-implementation is an instance of a subclass of the `BOA` class below. Thus, `BOA` operations can be issued on any object-implementation.

```
// C++
class BOA {
public:
    // Standard BOA interface. See OMG doc. 94-9-14:
    //
    Object_ptr create(const ReferenceData&,
        InterfaceDef_ptr, ImplementationDef_ptr);
    void dispose(Object_ptr);
    ...

    // Electra-specific operations:
    //
    static void create_group(Object_ptr group,
        const ProtocolPolicy& policy
        =default_protocol_policy,
        Environment_ptr =0);
    void join(Object_ptr group, Environment_ptr =0);
    void leave(Object_ptr group, Environment_ptr =0);
    static void destroy_group(Object_ptr group,
        Environment_ptr =0);

    virtual void get_state(AnySeq& state,
        Boolean& done, Environment_ptr env);
    virtual void set_state(const AnySeq& state,
        Boolean done, Environment_ptr env);
    virtual void view_change(const View& newView);
};
```

The `BOA::create_group` method creates a new object-group and binds the object-reference `group` to it. The reference can be installed in a name server or converted to a human-readable string by the `ORB::object_to_string` operation. The `policy`

[2] language bindings for Smalltalk, Lisp, Fortran, SML etc. can be provided by writing language-specific backends for the IDL compiler.

argument is used to tell the underlying toolkit what kind of multicast protocol to employ, e.g., for total ordering or causal ordering [9, 5]. If Electra is configured to run on Isis, the `policy` object will select either the Isis *abcast*, *cbcast*, or *gbcast* protocol to transmit a multicast. In the Horus configuration, the `policy` object serves to compose a Horus protocol stack [26]. For instance, the programmer can specify ATM as transport layer and pick from a variety of ordering protocols to be placed atop of the ATM layer. The policy-object mechanism could be extended to cover quality of service guarantees. For example, the minimum bandwidth necessary to sustain a certain service could be defined through a policy object.

Objects in the network join or leave a group simply by retrieving its reference from the name server and by issuing the `join` or `leave` operation with the reference as parameter. `destroy_group` irrevocably destroys an object-group. Note that the group-members themselves are not destroyed.

When an object joins a non-empty group, Electra will obtain the internal state (i.e., the values of all instance variables) of some group-member by invoking its `get_state` method. Subsequently, Electra transfers the state to the newcomer and invokes the newcomer's `set_state` method. A large state can be transferred in fragments. For this purpose, Electra will continue to invoke the state transfer methods until `TRUE` is assigned to the `done` return argument of `get_state`. The `Environment` object is used to signal an interrupted state transfer due to a failure of the member from which the state was being received. An object-state is represented as a sequence of CORBA `any` objects.

State transfer is necessary for redundant computations to permit the replication-degree of an object to be increased at run-time. It is the programmer's task to write application-specific `get_state` and `set_state` methods. An example of how to write such methods will be provided in Section 5.2.4. These methods can also be used to checkpoint the state of an object to non-volatile storage or to perform object migration.

The `view_change` method of an object is invoked whenever another object joins or leaves the group. The `newView` object contains information on the new cardinality of the group as well as the object-references of the group-members.

In Electra, object-group members need to be of the same type, i.e., instances of the same interface, or they must at least have one ancestor interface in common. In the latter case, only the operations inherited from a common ancestor can be multicast to the group.

## 4.2 Enhanced Environment Interface

When clients invoke an operation on a remote object, a CORBA `Environment` object can be passed as additional parameter. In CORBA, `Environment` objects are used mainly to pass exceptions from the server to the client. Electra enhances the standard `Environment` class with three special operations. By the `call_type` method below, clients specify whether an invocation will be performed synchronously (blocking), asynchronously (non-blocking) or by an intermediary form using a "promise" abstraction [10] to synchronize with the invocation at a later point.

```
// C++
class Environment {
public:
    // Standard Environment interface.
    // See OMG doc. 94-9-14:
    //
    void exception(Exception*);
    Exception *exception() const;
    void clear();

    static Environment_ptr _duplicate(Environment_ptr);
    static Environment_ptr _nil();

    // Electra-specific operations:
    //
    typedef enum {SyncCall, AsyncCall, DeferCall} tCall;
    void call_type(tCall);
    void num_replies(Long);
};
```

The `num_replies` method permits to specify how many member-replies the client's ORB will collect during an invocation. The constant `ALL` demands that the replies of all operational group-members be collected, whereas `MAJORITY` means that the call is active only until a majority of the members have replied. If the constant `COMPARE` is passed to `num_replies`, the ORB collects a reply from each operational member. The replies are then compared and the most frequent one is chosen. If some replies disagree, an exception is raised. If the selection is equivocal on differing replies, another exception is raised. In analogy to comparator mechanisms in fault-tolerant hardware [22], this "poor man's" approach permits to detect faulty system components and to act accordingly. Finally, an arbitrary integral number ranging from one to the cardinality of the group can be specified to collect an exact number of replies. If a majority or a certain exact number of replies cannot be collected due to a failure, an exception is thrown.

# 5 IDL-Mapping of Group Operations

## 5.1 Operation Signatures

Electra's C++ mapping goes beyond the OMG specification in that IDL operations are mapped on an additional signature which is needed for non-transparent group communication.

```
// IDL
interface example {
    void op1(in float i, out float o);
    float op2(in float i, out float o);
    void op3(in float i);
};
```

The mapping of CORBA interfaces onto Electra invocation-stubs can be summarized as follows:

- Every operation is mapped into two C++ signatures: one for performing unicast and *transparent* multicast, another one for performing *non-transparent* multicast. In the non-transparent case, environment, return, `out`, and `inout` arguments are mapped into CORBA sequences of the arguments' data types. Using the non-transparent invocation form, programmers gain access to environment objects, return values, `out` and `inout` arguments generated by the individual object-group members. Transparent group invocations, on the other hand, fill the first arriving reply into the arguments and convey the illusion of communicating with a non-replicated, highly available object. Note that a singleton object can be treated like a group with only one member.

- Operations with `void` return-type, for instance `op1` and `op3`, are mapped into Electra operations which can be issued synchronously, asynchronously, and deferred-synchronously. The programmer selects a call type through an `Environment` object acting as additional parameter for the invocation.

- An upcall method can be specified for asynchronous and deferred-synchronous operations. The upcall is automatically invoked with an own thread of execution when the reply to an operation has arrived at the client. The signature of the upcall is determined by omitting all `in` parameters from the signature of the associated interface operation.

- Operations with non-void return-type, `op2` for instance, are mapped into Electra operations which can be issued only synchronously.

By following these rules, the Electra stub generator translates `interface example` into the following C++ class definition. This class serves as static invocation interface to objects of type `example`:

```
// C++
class example: public Object {
    // typedefs for upcalls:
    //
    typedef void (*op1_upcall)(Float o,
        const Environment&);
    typedef void (*op1_upcall_mc)(const FloatSeq& o,
        const EnvironmentSeq&);
    typedef void (*op3_upcall)(const Environment&);
    typedef void (*op3_upcall_mc)(
        const EnvironmentSeq&);

    // signatures for unicast and transparent multicast:
    //
    void op1(Float i, Float& o, Environment&,
        op1_upcall =0);
    float op2(Float i, Float& o, Environment&);
    void op3(Float i, Environment&, op3_upcall =0);

    // signatures for non-transparent multicast:
    //
    void op1(Float i, FloatSeq*& o,
        EnvironmentSeq&, op1_upcall_mc = 0);
    FloatSeq op2(Float i, FloatSeq*& o,
        EnvironmentSeq&);
    void op3(Float i, EnvironmentSeq&,
        op3_upcall_mc =0);
    ...
};
```

In the above example, the first version of the operations permit transparent communication with singleton objects and object-groups. Issued on an object-group, per default the first arriving member reply is assigned to the `out` arguments, `inout` arguments, to the environment, and to the operation's return value. Replies arriving later will be discarded. The second version of each operation has a sequence type in place of its return arguments, and serves for non-transparent multicast. `op2` can be issued only synchronously as it is a non-void operation.

Since the non-transparent form employs sequences for the `inout` arguments, the question arises how an `inout` parameter is passed from the client to the server. Our solution is to store the parameter in the first position of the sequence. In contrast, `out` parameters are unproblematic, since data is passed from the server to the client only.

Note that the multicast version of an operation requires a sequence of `Environment` objects because such objects contain information on exceptions, and each group-member can report an exception by itself. Analogously to inout arguments, the first element of an Environment sequence informs the run-time of the operation type and of the requested number of replies.

## 5.2 Examples

### 5.2.1 Object-Groups and Multicast

In the following code fragment we demonstrate how two object-implementations, impl1 and impl2, are inserted into an object-group. For the sake of simplicity we create both implementations in the same process. For fault-tolerance, impl1 and impl2 would be instantiated on two different machines. After having created impl1 and impl2 in the server process, the BOA::create_group operation is issued to create an empty object-group and to bind the object-reference group to it. Subsequently, impl1 and impl2 are inserted into the group, and the group-reference is registered with a system-wide name server.

```
// C++
// Server site:
//
// Create two implementations of interface example:
_im_example impl1, impl2;
// declare a group-reference:
example_var group;
// Create an object group, insert impl1 and impl2:
BOA::create_group(group);
impl1.join(group);
impl2.join(group);
// Register the group reference with the name server:
NameServer.bind("my object-group", group);
```

```
// Client site:
// Bind to the group and multicast op3:
//
Environment env;
// Obtain the group reference from the name server:
example_var ref =NameServer.resolve("my object-group");
// transparent, blocking multicast (default):
ref->op3(7.3, env);
```

On the client site, an object-reference is bound to the group and operations issued through the reference will be delivered to both impl1 and impl2. Object-implementations may join or leave the system dynamically and operations can be issued as long as there exists at least one operational group-member.

### 5.2.2 Synchronous and Asynchronous Communication

The next example demonstrates synchronous, asynchronous, and deferred-synchronous object invocation. Note that the example works independently of whether ref is bound to a singleton object or to a group. In the latter case, transparent group-communication is performed.

```
// C++

// upcall procedure for asynchronous invocation:
void op1_upcall(Float outF, const Environment& env){
    // outF holds the result of the asynchronous
    // invocation below.
}

void proc1(example_var& ref){
    Float outF;
    Environment sync, async, defer;
    sync.call_type(SyncCall);
    async.call_type(AsyncCall);
    defer.call_type(DeferCall);

    // Synchronous (blocking) invocation:
    ref->op1(7.3, outF, sync);
    // at this point outF holds the result.
    ...
    // Asynchronous (non-blocking) invocation:
    ref->op1(7.3, outF, async, op1_upcall);
    // outF is undefined. The result will be passed
    // to the upcall.
    ...
    // Deferred-synchronous invocation:
    ref->op1(7.3, outF, defer);
    // outF is undefined.
    // perform local computations ...
    defer.wait(); // suspends the caller only if necessary.
    // at this point outF holds the result.
}
```

The synchronous call suspends the issuing thread until the reply has arrived. After the call, outF contains the result returned by the server. In case that ref is bound to an object-group, the call is suspended only until the first member-reply has arrived, unless this default behavior is changed by the Environment::num_replies method.

In asynchronous mode, the issuing thread is not suspended and outF remains undefined. As soon as the reply is received by the caller's ORB, the op1_upcall method is started with its own thread of execution, and with outF as parameter. If the server has returned an exception it will be assigned to the Environment parameter of op1_upcall.

The deferred-synchronous call works like the asynchronous one, however, by issuing the wait method on the Environment object, the caller is suspended until a reply is received from the server. When wait returns, the outF argument is defined. Thus, the Environment parameter acts like a "promise" object [10] in that it permits the caller to synchronize with the invocation at a later point, and thus to overlap communication with computation.

#### 5.2.3 Non-Transparent Group Invocation

The next example shows the difference between transparent and non-transparent multicast. By using CORBA sequences in place of an operation's return arguments, programmers gain access to the replies of the individual group-members. Non-transparent multicast is useful when group-members perform different tasks.

```
// C++
void proc2(example_var& ref){
    Float outF; FloatSeq outFSeq;
    Environment sync; EnvironmentSeq defer;
    sync.call_type(SyncCall);
    defer[0].call_type(DeferCall);
    defer[0].num_replies(MAJORITY);

    // transparent multicast (as in proc1):
    ref->op1(7.3, outF, sync);
    ...

    // non-transparent, deferred-synchronous multicast:
    ref->op1(7.3, outFSeq, defer);
    // local computations...
    defer[0].wait();
    // outFSeq now contains the replies of
    // a majority of the members:

    for(ULong i =0; i < outFSeq.length(); i++){
        // do something with outFSeq[i]
    }
}
```

#### 5.2.4 Replication, Migration, State Transfer

The next example addresses the implementation of a fault-tolerant domain name server (DNS) whose replication degree can be dynamically varied. Moreover, the DNS can be migrated from one machine to another while clients are constantly using the service.

```
// IDL
interface DNS {
    void install(in string host, in string address)
        raises (ENTRY_EXISTS);
    void host_to_addr(in string host, out string address)
        raises (NO_SUCH_HOST);
    void addr_to_host(in string address, out string host)
        raises (NO_SUCH_ADDRESS);
    void remove(in string host, in string address)
        raises (NO_SUCH_HOST, NO_SUCH_ADDRESS);
};
```

Fed with the above interface declaration, the Electra IDL compiler generates a set of C++ files containing the static invocation interface of the DNS, the server stubs, as well as a file containing a skeleton of the service with one C++ method per operation declared in the interface. This file also provides a skeleton for the state transfer methods of the DNS object which can be completed by the programmer as follows:

```
// C++
void _im_DNS::get_state(AnySeq& state, Boolean& done,
    Environment_ptr){
    // Pack all DNS entries into the "state" object:
    //
    for(ULong i =0; i < 2 * hosts.length(); i += 2){
        state[i] <<= hosts[i/2];
        state[i+1] <<= addresses[i/2];
    };
    // The whole state was read:
    done = TRUE;
};

void _im_DNS::set_state(const AnySeq& state,
    Boolean done, Environment_ptr){
    // Unpack the received DNS entries.
    // First we must clear "hosts" and "addresses":
    //
    hosts.length(0); addresses.length(0);
    for(ULong i =0; i < state.length(); i += 2){
        state[i] >>= hosts[i/2];
        state[i+1] >>= addresses[i/2];
    };
    // We are not interested in "done" since
    // the whole state was transmitted at once.
};
```

To increase the replication degree of a DNS service, a DNS object implementation is created and joined to the respective DNS object group. Electra will invoke the `get_state` object of a group member, marshal the `state` object, transfer it to the newcomer, unmarshal it and invoke the newcomer's `set_state` method. Owing to totally ordered multicast and to Virtual Synchrony, the internal states of the group-members remain consistent in spite of DNS objects joining and leaving the group, and in spite of client applications creating and removing entries from the service while membership changes are occurring.

In order to migrate a DNS object which is a member of a certain group, one just has to instantiate a new DNS object on the destination machine and to join the object to the group. Electra will automatically transfer the state of the obsolete object to the newcomer object and the two DNS objects will run synchronized. Now, the obsolete object can simply be destroyed.

# 6 Configurability

Electra can run on various toolkits and operating systems, the current version supports Horus, Isis, and MUTS [24]. We believe that Electra can be ported to Amoeba, Chorus, Consul, Transis, and to other platforms providing multicast and threads, without much effort. Although Electra could be configured to run directly atop of contemporary operating systems such as UNIX or WINDOWS NT, we prefer a solution where a platform like Horus or Isis is employed, since we regard reliable group-communication and Virtual Synchrony as a fundamental part of an ORB.

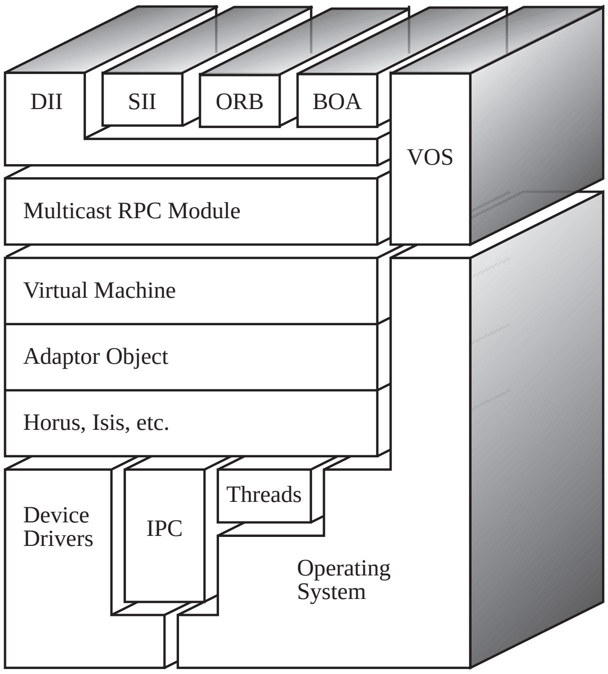


Figure 2: Electra Architecture.

Electra is layered as depicted in Figure 2. The CORBA Static Invocation Interface (SII), Object Request Broker Interface (ORB), and Basic Object Adapter (BOA) are based on the Dynamic Invocation Interface (DII) which can be seen as the core of the ORB. The core itself is built atop of a multicast RPC module supporting asynchronous RPC to both singleton and group destinations. A straightforward approach would consist in building the multicast RPC module directly on Horus, but for the sake of flexibility and portability we decided to base the module on a toolkit-independent veneer, called the *Virtual Machine*.

The Virtual Machine interface exports operations for creating communication endpoints, for aggregating endpoints to groups, for asynchronous message passing, and for lightweight-processes. RPC module and Virtual Machine communicate by means of downcalls und upcalls. The Virtual Machine can be thought of as representing the "least common denominator" of the toolkits to be supported. Determining the "instruction set" of the Virtual Machine was challenging, the resulting Virtual Machine suitable for Horus, Isis, and MUTS is described in [13]. VOS is a Virtual Operating System layer used by applications to interact with the underlying operating system in a portable and thread-safe way.

To map the Virtual Machine interface onto the proprietary API provided by the underlying toolkit, a toolkit-dependent *Adaptor Object* is implemented. An Adaptor Object cleanly encapsulates all of the program code which is specific to a toolkit and necessary to support the multicast RPC module. We call this system-design principle the *Adaptor Model* [11]. To port Electra to a new toolkit, programmers only have to develop an appropriate Adaptor Object. Our adaptors for Horus, Isis, and MUTS comprise less than 1000 lines of C++ code each.

Electra-applications can be reconfigured to run on another toolkit by simply relinking them with the appropriate Adaptor Object. Recompilation of applications is not necessary, therefore applications delivered in binary form can be reconfigured as well.

# 7 Discussion

## 7.1 The Direct Approach Would Not Work

One might ask where Electra will give better results than an ORB built on conventional RPC or message passing mechanisms. In fact, conventional CORBA ORBs also provide multicast object invocation, namely through the `send_multiple_requests` dynamic invocation interface operation. This approach works at best when only one client multicasts to an object-group, or with multiple clients and static group membership. However, if two or more clients issue operations on a group, and given that some of the operations alter the state of the objects, the internal states will eventually become inconsistent since multicasts sent by different clients might arrive in different order at the members (Figure 3). Furthermore, if objects need to join and leave the group dynamically, a group membership protocol is necessary to maintain the view[3] that each of the client applications has on the current group membership consistent and for enabling state transfer to newcomer objects. Also, the operations multicast by the clients must be synchronized with view changes. Such ordering and membership protocols are not part of conventional ORB technology but are the basis of Horus and Isis.

Another problem can occur when a causal dependency [9] exists in a chain of asynchronous object

[3] i.e., the array of CORBA `Request` objects passed to `send_multiple_requests`.

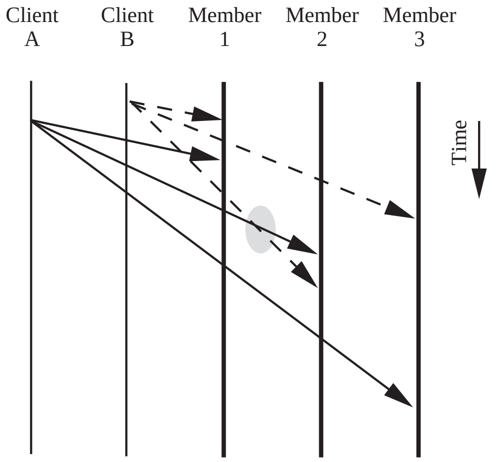


Figure 3: Group operations may be dispatched in inconsistent order if the multicast protocol does not provide total ordering.

invocations. In Figure 4, a client invokes an operation on Object 1 to deposit money on a bank account. Later, the client notifies Object 2 of the deposit. Now, Object 2 tries to withdraw what it believes is on the account, but the deposit operation is delayed due to an overloaded network. Thus, the account is mistakenly overdrawn. Horus and Isis avoid such problems by maintaining causal ordering of messages across multiple processes. Owing to context information appended to messages, the run time system of Object 1 would recognize that a causally preceding message is missing and would delay the withdraw operation until the deposit operation could be dispatched. Conventional ORBs do not guarantee causal delivery.

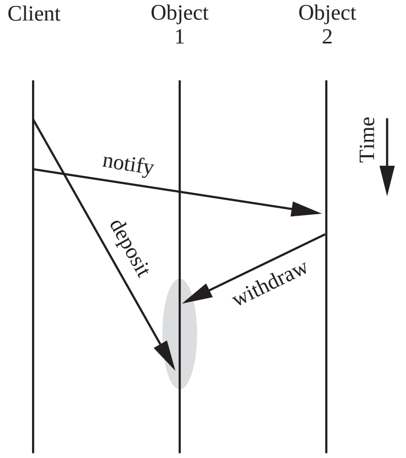


Figure 4: Causality may be violated if the communication subsystem does not guarantee causal delivery.

A further weakness of ORBs that straightly follow the CORBA specification is that `oneway` operations have only best-effort semantics. This means that a `oneway` operation can get lost even when no failure occurs, although reliable, asynchronous communication is often necessary for building efficient distributed applications. In contrast, Electra provides reliable, asynchronous communication, and reliable, order-preserving multicast both through the static and dynamic invocation interface.

Last but not least, client applications built on conventional ORB technology may have incongruous opinions on which objects in the system have failed, which can lead to unpredictable behavior of applications. In contrast, systems like Horus and Isis provide failure detection and consistent propagation of failure beliefs to solve this problem.

## 7.2 The Best of Both Worlds

The thesis underlying our work is that an ORB based on group communication primitives and on the Virtual Synchrony model will considerably simplify the development of robust, scalable distributed systems, and that communication primitives provided by operating systems in widespread use today are not the adequate foundation of an ORB. We regard platforms like Horus, Isis, Transis, and Consul as providing the necessary system support for an ORB, including for instance, fault detection, reliable group communication, and consistent ordering of events. Coming from this position, we described the design and implementation of Electra — a novel CORBA Object Request Broker combining the benefits of OMG CORBA with the strengths of systems like Isis. Electra aims to support applications that require high availability, efficient diffusion of data to large groups of recipients, parallelism, or a combination thereof. Prospective application areas of Electra are video-on-demand servers, video conferencing systems, groupware, distributed parallel computing, and different kinds of fault-tolerant client-server applications.

In addition to the functionality provided by conventional ORBs, Electra permits to aggregate object-implementations to logical groups and to name them as a single unit. Per default, such object-groups are transparent to the programmer and appear like highly available singleton objects. An operation on an object-group will succeed as long as at least one member survives the operation, and the replication degree can be varied at run-time. If required, programmers can break this transparency and gain access to the results generated by the individual members of a group.

Object-groups can be employed for fault-tolerance, load sharing, caching, efficient data distribution, network management, and object-migration. To hide communication latencies, Electra operations can be performed synchronously, asynchronously, or deferred-synchronously, both through the static and dynamic invocation interface. We also have shown how group communication and Virtual Synchrony can be added to a CORBA ORB, namely by enhancing the `BOA` and `Environment` interfaces with a few easy-to-use methods, and by employing Horus or Isis as communication subsystem.

### Acknowledgements

The author would like to thank Ken Birman, Roy Friedman, Sophia Georgiakaki, Sean Landis, Robbert van Renesse, Scott Swarts, and Alexey Vaysburd for their support and for their suggestions.

### Availability

Please contact `maffeis@acm.org` if you are interested in using Electra. Presently, Electra is a working prototype consisting of a CORBA IDL compiler, a Dynamic Invocation Interface, an ORB Interface, a Basic Object Adapter, and Adaptor Objects for Horus, Isis, and MUTS.

## 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] BAKER, S. Multicasting for Sound and Video. *Unix Review* (Feb. 1994).

[3] BIRMAN, K. P., AND VAN RENESSE, R., Eds. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[4] GOLDING, R. A. Weak Consistency Group Communication for Wide-Area Systems. In *Proceedings of the 2nd IEEE Workshop on the Management of Replicated Data* (Nov. 1992), IEEE.

[5] HADZILACOS, V., AND TOUEG, S. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, S. Mullender, Ed., second ed. Addison Wesley, 1993.

[6] HERBERT, A. Distributing Objects. In *Distributed Open Systems*, F. Brazier and D. Johansen, Eds. IEEE Computer Society Press, 1994.

[7] JAHANIAN, F., FAKHOURI, S., AND RAJKUMAR, R. Processor Group Membership Protocols: Specification, Design and Implementation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems* (Princeton, New Jersey, Oct. 1993), IEEE.

[8] KAASHOEK, M. F. *Group Communication in Distributed Computer Systems.* PhD thesis, Vrije Universiteit, Amsterdam, 1992.

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

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

[11] MAFFEIS, S. A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming. In *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming* (1994), Lecture Notes in Computer Science 791, Springer-Verlag.

[12] MAFFEIS, S. System Support for Distributed Computing. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe 1994* (1994), Lecture Notes in Computer Science 797, Springer-Verlag.

[13] MAFFEIS, S. *Run-Time Support for Object-Oriented Distributed Programming.* PhD thesis, University of Zurich, Department of Computer Science, 1995.

[14] MEYER, B. *Object Oriented Software Constrution.* Prentice Hall, 1988.

[15] MISHRA, S., PETERSON, L. L., AND SCHLICHTING, R. D. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. *Distributed Systems Engineering Journal 1*, 2 (Dec. 1993).

[16] NICOL, J. R., WILKES, C. T., AND MANOLA, F. A. Object Orientation in Heterogeneous Distributed Computing Systems. *IEEE Computer 26*, 6 (June 1993).

[17] OBJECT MANAGEMENT GROUP. *IDL C++ Language Mapping Specification*, 1994. OMG Document 94-9-14.

[18] OBJECT MANAGEMENT GROUP. *The Common Object Request Broker: Architecture and Specification*, 1995. Revision 2.0.

[19] PETERSON, L., BUCHHOLZ, N., AND SCHLICHTING, R. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems 7*, 3 (Aug. 1989).

[20] SCHIPER, A., AND RICCIARDI, A. Virtually-Synchronous Communication Based on Weak Failure Suspectors. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing* (Toulouse, June 1993), IEEE.

[21] SCHLICHTING, R. D., AND SCHNEIDER, F. B. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems 1*, 3 (Aug. 1983).

[22] SIEWIOREK, D. P., AND SWARZ, R. W. *Reliable Computer Systems: Design and Evaluation.* Digital Press, Bedford, Mass., 1992.

[23] SOLEY, R. M. *Object Management Architecture Guide.* Object Management Group. OMG Document 92-11-1.

[24] VAN RENESSE, R. A MUTS Tutorial. MUTS Documentation, Cornell University, 1993.

[25] VAN RENESSE, R., AND BIRMAN, K. P. Fault-Tolerant Programming using Process Groups. In *Distributed Open Systems*, F. Brazier and D. Johansen, Eds. IEEE Computer Society Press, 1994.

[26] VAN RENESSE, R., AND BIRMAN, K. P. Protocol Composition in Horus. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing* (Ottawa, Ontario Canada, Aug. 1995).

[27] VERÍSSIMO, P., AND RODRIGUES, L. Group Orientation: A Paradigm for Distributed Systems of the Nineties. In *Proceedings of the Third Workshop on Future Trends of Distributed Computing Systems* (Apr. 1992), IEEE Computer Society.