Building Reliable Distributed Systems with CORBA

Sean Landis
Isis Distributed Systems, Inc.
Ithaca, NY 14850
scl@isis.com

Silvano Maffeis*
Dept. of Computer Science
Cornell University
maffeis@acm.org

Abstract

New classes of large-scale distributed applications will have to deal with unpredictable communication delays, with partial failures, and with networks that partition. In addition, sophisticated applications like teleconferencing, video-on-demand, and concurrent software engineering require a group communication abstraction. These paradigms are not adequately addressed by CORBA. CORBA mainly deals with point-to-point communication and offers no support for the development of reliable applications that exhibit predictable behavior in distributed systems. In this paper we present extensions to CORBA which provide group communication, reliability, and fault-tolerance. We also describe Orbix+Isis and Electra — two CORBA object request brokers that support the implementation of reliable distributed applications and groupware.

Keywords: Distributed Systems, CORBA, Object Groups, Multicast, Reliability, Fault-Tolerance, Orbix+Isis, Electra, Isis, Horus

1 Introduction

The object paradigm has successfully been applied to the design and implementation of graphical user interfaces, application frameworks, device simulators, and object-oriented databases. Object-oriented programming of distributed applications is the next logical step.

Object-oriented distributed programming (OODP) is a variation of the client-server model. In OODP, 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.

Object-oriented design and implementation of distributed systems provides many benefits. The service guaranteed by an object is clearly separated from the technology implementing the service. The programming language which was used to implement an object, the programming libraries, or the underlying hardware can be exchanged as long as the object supports its old interface. Because object-oriented distributed systems allow the interchange of parts, they tend to evolve more easily.

The OMG CORBA standard [13] permits the design of open distributed systems in an object-oriented fashion by providing an infrastructure that allows objects to communicate independent of the specific programming languages and techniques used to implement the objects [16]. Open systems avoid excessive dependence on a single manufacturer, or on a certain operating system, hardware, or programming language, and thus offer customers freedom of choice. Development risks are reduced, and enterprises can choose the components of a distributed system according to price/performance criteria.

1.1 Limitations of Present ORB Technology

The absence of an abstraction which provides multicast requests to groups of CORBA objects in an efficient way greatly complicates the design and implementation of applications like teleconferencing, work flow management, and concurrent software engineering. The present version of CORBA [13] is based mainly on a point-to-point communication paradigm.

Moreover, CORBA does not adequately support the implementation of *reliable* distributed applications. The behavior of a reliable distributed application would be predictable in spite of partial failures and of partitioned networks. Many real-world applications have to deal with problems related to partial failures, but the development of reliable applications with current ORB technology requires a large amount of additional work.

*Supported by the Swiss National Science Foundation

1.2 Enhancements for ORB Reliability

During the past several years we have been designing and implementing CORBA environments for reliable distributed applications. The results of our work are a set of new abstractions as well as Orbix+Isis [7] and Electra [10] — two object request brokers for reliable distributed systems. We enhanced the OMA Model [16] with *object groups* and *virtually synchronous* program execution.

The rest of the paper is structured as follows. Section 2 stipulates requirements for a reliable CORBA environment. Section 3 deals with the low-level system support necessary to implement the reliable CORBA. In Section 4 and 5 we give a brief overview of Orbix+Isis and Electra. Examples of reliable applications which can be built with this kind of ORB will be given in Section 6. In Section 7 we compare our approach with approaches taken in other projects. Finally, Section 8 summarizes and concludes the paper.

2 Requirements for a Reliable CORBA

This section describes the features we feel are necessary to support a reliable Common Object Request Broker Architecture. These features can be provided by the communication system and can be transparent to the programmer.

2.1 Object Groups

The reliability of CORBA objects could be increased by developing an abstraction that manages groups of distributed objects. This abstraction would shield programmers from much of the complexity inherent in distributed systems. A researched and tested method for providing such an abstraction is associating CORBA objects together in object groups.

An object group is a group of replicated or associated CORBA objects implementing the same interface. Client communication with an object group is by reliable multicast. The client program invokes operations via a single object group reference without knowing the references of the individual objects in the group.

In CORBA, a client binds to and sends requests to a single object. With object groups, a client binds to the group and requests are sent to all members. Object group behavior can be transparent to the client:

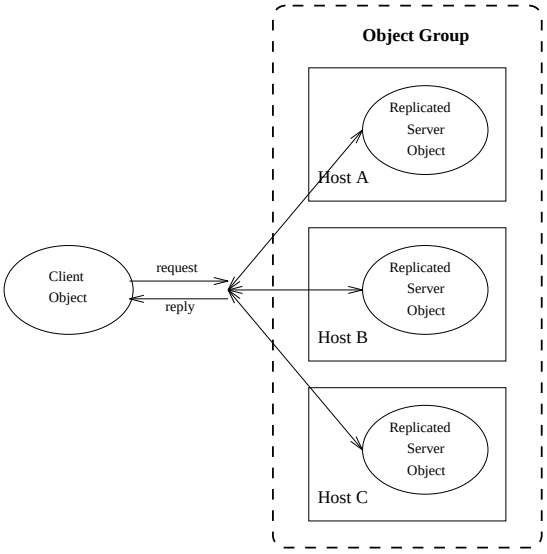

Figure 1: Server Implemented as an Object Group

- The application code used to bind to an object group is the same as for binding to a single object.
- The client sends a single request and receives a single reply, as it would with a single object.
- The object group handles all reliability protocols automatically so they are invisible to the client.
- By default, the runtime returns only one of multiple server replies to the client.

If desired, the application programmer can easily gain access to all server replies. Thus, transparency is optional.

As will be explained later, object groups provide fault-tolerance, load sharing, efficient data distribution, and object migration. Figure 1 shows an object group with three member objects.

2.2 Failure Detection

Reliable group computing requires that requests are acknowledged by all group members. Failure detection can prevent a client from blocking waiting indefinitely for responses from failed group members and clients.

The runtime performs automatic failure detection without disrupting communication among active group members. Unresponsive group members are identified using a time out failure detector which, after a specified amount of time, will declare an unresponsive member failed. The runtime forwards failure

notifications to ensure that operational objects have a consistent opinion on which objects have failed.

With failure detection and propagation of failure notifications, the runtime can transparently maintain consistency among object group members and can guarantee multicast request completion.

2.2.1 Active Replication

Active replication is a style of distributed group programming which assumes every member of the group is actively maintaining replicated state. Each object group member shares the same interface and equivalent implementation semantics to ensure that each member responds to identical requests in the same way.

Servers implemented with actively replicated objects tolerate object, host, and network failures; as long as one replica is accessible, the service remains available. Active replication can support load balanced servers and object migration.

Active replica object groups manage group membership and provide a way for objects joining the group to be updated and become exact replicas. Group members use a *View* to maintain a group-wide agreement of group membership. *State transfer* provides a way for new members to become replicas. *Monitoring* enables group members to receive notifications of group membership changes.

2.2.2 Views

A view is a dynamic list of object references representing the object group membership. Changes in group membership trigger the automatic installation of a new view at each group member. Consistent group views provide a mechanism for performing reliable multicasts to the object group.

When a view is stable, each object (client and server) sees an identically ordered list of group members. Request delivery occurs only when the view is stable.

Figure 2 illustrates one way that request delivery to an unstable group can cause application problems. A client requests a distributed database search. To speed up the operation, each of the objects in the group is to search a different portion of the database. The client sends the request to a group consisting of objects A and B, but before B receives the request, C is admitted to the group. A, having received the request when two objects were in the group, searches 1/2 of the database. B receives the request when three objects are in the group, and searches 1/3 of the database. Object C does not receive the request.

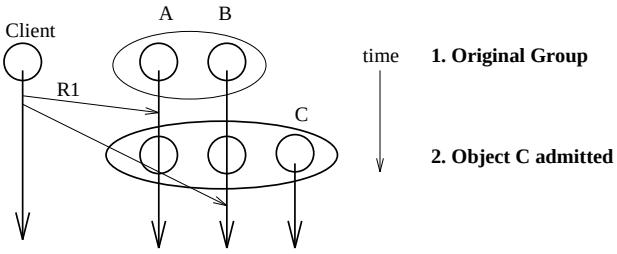

Figure 2: Disagreement in the View

When results are returned to the client, only 5/6 of the database has been searched.

View consistency is maintained by controlling when an object can gain admission to a group. Pending requests are delivered before a new object is admitted. While an object is being admitted, requests are delayed until the view is stable. Figure 3 shows request delivery to a stable view. R2 is delivered after the view is stable. View management is performed by the runtime and is transparent to the programmer.

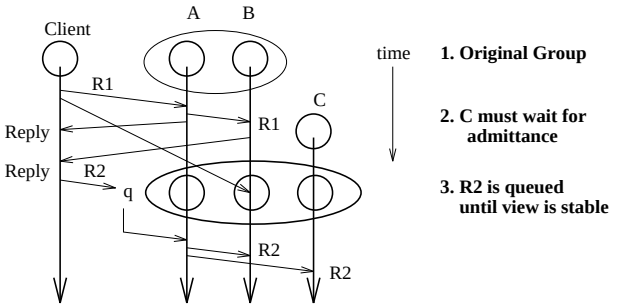

Figure 3: Delivery to a Stable View

2.2.3 State Transfer

State transfer is a feature that invokes object member functions to transfer object state from existing object group members to a new member being admitted to the group.

State transfer is a key component of active replication because it allows new objects to join the group and become exact replicas. Objects join the group to increase fault-tolerance or when they recover from failure.

When a new object is admitted to the group, the runtime system calls the state send member function on one or more members of the object group. This function packages state information for the runtime system which transfers it over the network to the new member. Upon arrival, the runtime calls the state receive member function of the new object which uses the state information to update its state.

Group membership changes are not complete until state transfer has successfully completed. At that point, the group view becomes stable and pending requests can be delivered to the group.

2.2.4 Object Monitoring

Monitoring is a feature that allows objects to be notified of changes in object group membership. The runtime transparently invokes monitoring functions on an exist group member object when a new member object is admitted to the group, or when an existing member fails or leaves the group. Monitoring gives group members the option of taking action in response to group membership changes.

2.3 Communication

For reliability, the communication system must provide guarantees beyond those provided by traditional network protocols. These guarantees can be built upon existing protocols such as TCP/IP.

2.3.1 Request Atomicity

Request atomicity guarantees that requests sent to object groups will be received by all members or by none.

Without atomic delivery, replicated objects may become inconsistent. Figure 4 illustrates how a problem can arise when a client fails after sending a request to an object group. Object A receives the request, but the client fails before its runtime can send the request to objects B and C. Without request atomicity, a programmer must find a way to recover from incomplete request delivery.

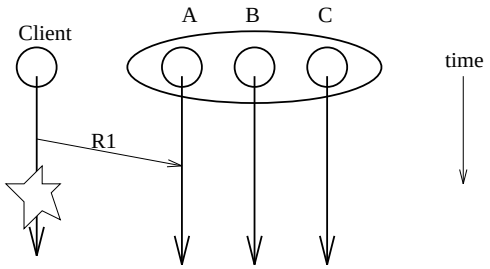

Figure 4: Incomplete Request Delivery Due to Failure

Atomic request delivery can be implemented within the communication system by having an object that received the request forward the request to the rest of the group in case of a client failure (Figure 5). Thus, all objects receive the request.

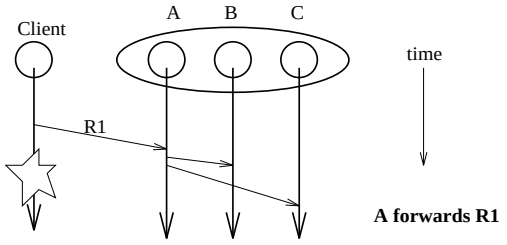

Figure 5: Request Delivery in Spite of Failure

2.3.2 Request Ordering

Most applications using group computing require some type of request ordering guarantee. Two useful types of ordering are total ordering and causal ordering.

Total Order The order in which requests are delivered to distributed objects can be compromised by a variety of delays such as network congestion, operating system latency, or resource contention. Total ordering of requests ensures that multiple requests concurrently sent to an object group by different clients will be delivered to every group member in the same order.

Figure 6 illustrates a potential request ordering problem. R1 and R2, two update requests to the same record, are sent by two clients to a group of replicated objects. Objects A and B receive R1 first and then R2. Because of network latency, object C receives R2 before R1. In subsequent queries, objects A and B may behave differently than object C.

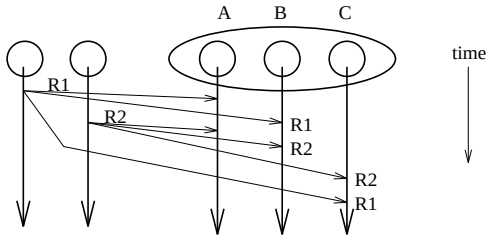

Figure 6: Unordered Request Delivery

By providing total ordering of requests, consistency can be maintained for all object group members. Figure 7 illustrates how requests are delivered in the same order to all group members.

The runtime delays delivery of R2 to object C until after R1 is delivered. Total ordering of requests greatly simplifies the development of replicated servers because the programmer can safely assume that every object in the group receives requests in the same order.

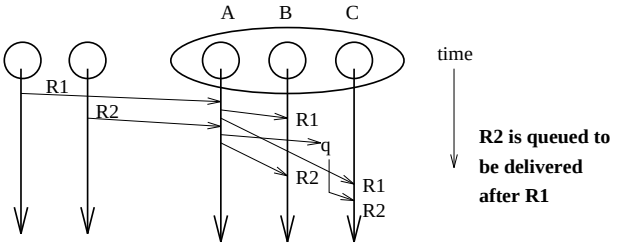

Figure 7: Totally Ordered Request Delivery

Causal Order Causal ordering ensures that if a request is potentially dependent upon a prior request, the previous request is delivered to an object first [8]. Causal order is weaker than total order, but incurs less communication overhead. Causal ordering provides an important alternative since it is sufficient for many applications and is less expensive to guarantee than total ordering. Total ordering requires a delay for every request while the runtime determines the ordering of messages. With causal ordering, the delay is only required for requests which are potentially causally related; unrelated requests can be delivered immediately.

If causal ordering is not present, a problem can occur when a causal dependency exists in a chain of asynchronous requests. In Figure 8, a client sends R1, requesting a file update, to object A. The client next notifies object B of the update with R2. Object B sends R3 to access what it believes is the current file. But R1 was delayed and has not yet arrived at object A! Object B is mistakenly accessing the wrong data.

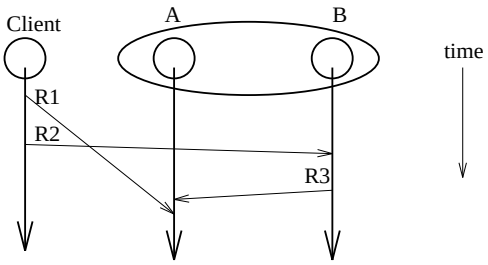

Figure 8: Causal Ordering Problem

By maintaining causal ordering, potentially dependent asynchronous requests are processed in the correct causal order as shown in Figure 9. With causal ordering, object A is assured of accessing the correct data because the delivery of R3 is delayed by the runtime system until after R1 is delivered.

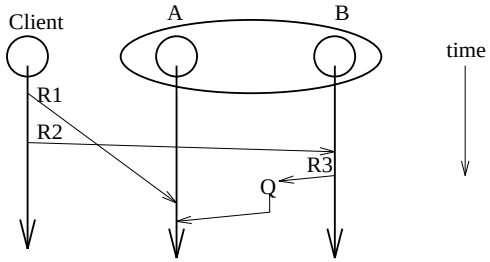

Figure 9: Causal Request Delivery

2.4 Virtual Synchrony

In virtually synchronous systems all significant events: the delivery of requests, failure notifications, and group membership changes, appear to occur at the same time in all processes. Owing to virtual synchrony, the behavior of a distributed application is predictable in spite of partial failures, asynchronous communication, and dynamic view changes. Virtual synchrony relieves programmers of problems such as handshaking and distributed consensus. The virtual synchrony model was originally developed in the context of the Isis project [4].

Atomic request delivery, view management, consistent failure detection, and ordering of events are the building blocks of the virtual synchrony model. Virtual synchrony simplifies developing distributed servers because code can be written as if events occur within each server at the same time. A request delivery or a group view change appears to be simultaneous at all object group members, even though more than one action occurs in more than one process, and at different times. Our model for reliable CORBA provides virtual synchrony.

3 System Support

Reliable CORBA needs sophisticated support from the underlying communication system. The communication system should provide multicast, group membership management, ordering of events, and virtual synchrony. Isis [4] and Horus [18] offer this kind of support with a C programming interface. Neither are CORBA compliant, but both can provide a foundation for a reliable ORB as shown in Figure 10.

3.1 Isis

The Isis Reliable™ Software Developer Kit is the first commercially available environment to provide virtual synchrony and high performance. Isis enables the implementation of fault-tolerant software

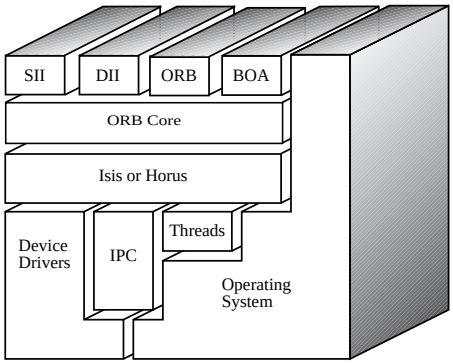

Figure 10: Reliable CORBA.

on loosely coupled hardware through a library of C functions. Core services include process groups, reliable multicast, ordering of events, and failure monitoring. Higher level services include a message spooling facility, a distributed resource manager, and a message publication-subscription layer. Isis runs on various operating systems including UNIX, Microsoft Windows™, Windows NT™, and VMS. Orbix+Isis, presented in the next section, uses Isis as its communication system.

3.2 Horus

The Horus toolkit, a research project in development at Cornell University, offers flexible group communication support by providing extensively layered and highly configurable protocol objects. Horus allows applications to pay only for services they use. Groups with different communication needs can coexist in a single system.

Horus runs on UNIX and supports communication protocols such as UDP, Deering-IP, ATM, and Mach messages. The Electra ORB, presented in Section 5, runs on both Horus and Isis.

4 Orbix+Isis

Orbix+Isis is the first commercially available system that supports the creation of fault-tolerant CORBA-compliant applications. Orbix+Isis integrates the Orbix™ CORBA-compliant C++ development environment from IONA Technologies, Ltd., with the Isis Reliable runtime technology from Isis Distributed Systems, Inc.

4.1 System Design

Orbix+Isis builds upon the strengths of its component products. Orbix provides the advantages of a CORBA environment: a standard object oriented programming environment, abstract interface definition, and distributed objects. Isis provides process groups, view management, state transfer, request ordering, and virtual synchrony.

Fault-tolerance and performance aspects of Orbix+Isis applications are specified in the Isis Repository (IsR). The IsR externalizes this information to allow changes in application behavior without code modification or recompilation.

Orbix+Isis provides a gentle migration path for Orbix applications that need a higher degree of fault-tolerance. Orbix+Isis features are transparent to the client program: fault-tolerant object group behavior does not require modification of client code.

A server implementation class gains fault-tolerant object group behavior by inheriting from a base class. Orbix+Isis provides base classes for two styles of object groups: Active Replica and Event Stream.

4.2 Active Replica Execution Style

Active Replica object groups provide fault-tolerance and load balancing through active replication. State transfer and monitoring are implemented in the base `ActiveReplica` class as do-nothing virtual functions.

To support state transfer, the server programmer overrides a send state and a receive state function. When a new object joins the object group, the send state function is called on one or more members to stream state information onto an Orbix `Request` object which is passed to the function. The Orbix+Isis runtime delivers the request to the new member and calls the receive state function which streams information off of the `Request` object and updates its state.

The programmer can control which objects send state by overriding a virtual function which returns a non-zero value if the object should send state. The default function always selects the oldest object group member.

Monitoring is supported in a similar fashion by overriding two virtual functions. Orbix+Isis calls the `_newMember()` function when an object joins the group, and `_memberLeft()` when a member leaves.

The Active Replica execution style object group offers three communication styles: Multicast, Client's Choice, and Coordinator/Cohort. These styles are specified on a per operation basis in the IsR. A single Active Replica object group can simultaneously support operations using each communication style.

4.2.1 Multicast

The Multicast communication style provides highly fault-tolerant operations. Every member of the object group receives client requests. If the operation interface specifies a return value, each member replies. To maintain client transparency, only the first reply is returned to the caller.

The programmer can gain access to all return values by writing an Orbix smart proxy (a client side proxy object provides communication support to the object group). The smart proxy inherits from the default proxy and adds behavior to process all return values. Smart proxies can be written by the server programmer and transparently installed on the client program.

For each IDL operation there are two member functions on the proxy: one with a standard C++ mapping signature, and another with an extended signature. The extended signature version consists of CORBA sequences for the return values, and for all `out` and `inout` parameters.

The default proxy simply returns the first value of each outbound sequence. By writing a smart proxy, the programmer can use the extended version to examine the sequences and craft an application specific reply to the caller.

Smart proxies allow a computation to be divided among group members. The partial results can be combined upon receipt, and a single answer can be presented to the client. This type of operation fully exploits the multiprocessing capabilities available in a distributed system.

4.2.2 Client's Choice

The Client's Choice communication style is an optimization of the Multicast style for read-only operations. It invokes the request on only one object in the group. The Orbix+Isis runtime automatically chooses a member to receive the request by calling a client side chooser function. If the chosen member fails, the chooser function is automatically called to choose a new member. The default chooser selects members round robin, but the server programmer can register an alternative chooser function.

A query to an actively replicated database would be more efficient as a Client's Choice operation since it is usually faster to send the query to only one member.

4.2.3 Coordinator/Cohort

The Coordinator/Cohort style load-balances computations which are expensive relative to communication cost. A two-phase protocol chooses a single member as coordinator. The coordinator performs the operation and then sends replies to the client and all other members (called cohorts). Cohorts can use the reply to update local state.

If the coordinator fails, a chooser function is automatically called to select a cohort which will become the new coordinator. By default, the chooser function chooses the oldest member of the group as coordinator, but the server programmer can register an alternative chooser function.

4.3 Event Stream Execution Style

Event Stream execution style object groups support asynchronous requests using a publication/subscription paradigm. In this paradigm, a CORBA implementation is registered with an Event Stream, which represents a clearing house for publication and subscription. All interface operations are defined in IDL as CORBA `oneway` operations, and are used by the client to send asynchronous events to the Event Stream. The Event Stream holds events and forwards them to subscriber objects, which are called Event Receivers, as shown in Figure 11.

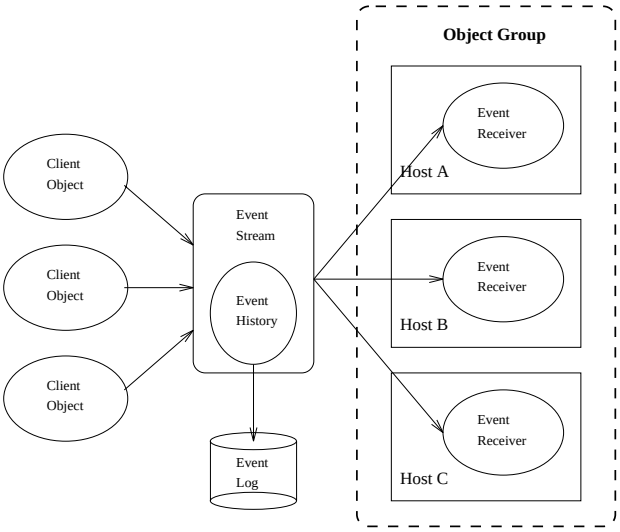

Figure 11: Orbix+Isis Event Stream

Event Receivers can join or leave groups at will, and can be configured to receive a user-defined number of events from the backlog upon joining the object group. After the initial receipt of backlog, Event Receivers will continue to receive events as they arrive on the Event Stream.

The Event Stream execution style decouples clients from servers and can provide persistence for events. Clients typically send events to the Event Stream

without knowing or caring if any Event Receivers are listening. The client may then cease communication, but the events can be kept by the Event Stream. The Event Stream can be configured to keep an event log on disk to provide recovery from a total failure of the stream.

Each member of an Event Stream receives events in the same order. An Event Stream is managed by a replicated group whose fault-tolerance parameters are configurable on a per Stream basis in the IsR.

An Event Receiver can checkpoint its position in the Event Stream and use it as a backlog starting point for the next join. In this way, an Event Receiver that has been inactive for a period of time will obtain all events.

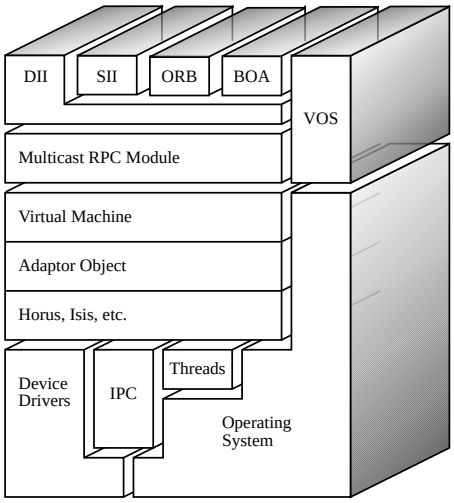

Figure 12: Electra Architecture

5 Electra

Like Orbix+Isis, Electra [10, 11] is an implementation of the reliable CORBA presented in Section 2. Electra is not a commercial product but is being used to research problems related to the migration of CORBA objects, state-reconciliation after the repair of a partitioned network, communication over ATM, and so forth. Electra differs from Orbix+Isis mainly in respect to adaptability of the ORB to various communication systems, and in the way client transparency is provided.

5.1 System Design

Electra can run on various toolkits and operating systems; the current version supports Horus, Isis, and MUTS [17]. We believe that Electra can be ported to Amoeba, Chorus, Consul, Transis, and to other platforms providing multicast and threads, without much effort.

Electra is layered as depicted in Figure 12. 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.

It would have been relatively easy to build the multicast RPC module directly on Horus. Nevertheless, 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 specifies operations for defining communication endpoints, for aggregating endpoints to groups, for asynchronous message passing, and for creating lightweight processes. The RPC module and the Virtual Machine communicate by means of downcalls and upcalls. A Virtual Machine interface suitable for Horus, Isis, and MUTS, is described in [11]. VOS is a Virtual Operating System layer used by applications to interact with the underlying operating system in a portable and thread-safe way. The VOS mainly provides operations for file and memory management.

A toolkit-dependent *Adaptor Object* maps the Virtual Machine interface onto the proprietary API provided by the underlying toolkit. An Adaptor Object encapsulates all of the code which is specific to a toolkit and necessary to support the multicast RPC module. We call this system-design principle the *Adaptor Model* [9]. 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.

5.2 Invocation Styles

Electra supports transparent and non-transparent multicast, as well as synchronous, asynchronous, and deferred-synchronous requests. All invocation styles are available both through the static and dynamic invocation interface.

In transparent mode, an object group appears to the client to be a highly available singleton ob-

ject. In contrast, non-transparent communication permits programmers to access the individual group member's results to an invocation. Each CORBA operation is thus mapped into two different SII operations, one for transparent and one for non-transparent multicast. The non-transparent signature employs CORBA sequences for the arguments which are passed back to the client, i.e., for the `out` and `inout` arguments, the return value, and for the CORBA environment object.

The synchronous, asynchronous, or deferred-synchronous invocation style can be selected on a per-request basis through the CORBA environment object which is passed along with every invocation in Electra. Consider the following IDL interface:

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

The following client code demonstrates how the operation can be called in a synchronous, asynchronous, and deferred-synchronous way:

```
// 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 the object reference `ref` is bound to an object group, the call is suspended only until the first member-reply has arrived. This default behavior can be changed by the `Environment::num_replies` member function.

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` procedure 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 much like the asynchronous one. However, by issuing the `wait` member 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 in that it permits the caller to synchronize with the remote-invocation at a later point, and thus to overlap communication with computation.

The next code fragment demonstrates transparent and non-transparent multicast. By using CORBA sequences in place of an operation's `out`, `inout`, and `Environment` arguments, programmers gain access to all replies from individual group members. Non-transparent multicast allows group members to perform different tasks.

```
// C++
void proc2(example_var& ref){
    Float outF; FloatSeq outFSeq;
    Environment sync; EnvironmentSeq defer;
    sync.call_type(SyncCall);
    defer.length(1);
    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]
    }
}
```

5.3 The Electra BOA

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) interface:

```
// 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` member function creates a new object group and binds the object-reference `group` to it. The `policy` argument is used to tell the underlying toolkit what kind of multicast protocol to employ, e.g., for total ordering or causal ordering. On Horus, the programmer can specify ATM as transport layer and pick from a variety of ordering protocols to be placed atop the ATM layer.

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

When an object joins a non-empty group, Electra obtains the internal state of some group member by invoking its `get_state` member. Subsequently, Electra transfers the state to the newcomer and invokes the newcomer's `set_state` member. A large state can be transferred in fragments. For this purpose, Electra continues to invoke the state transfer functions until the `done` return argument of `get_state` becomes `TRUE`. The `Environment` object is used to signal an interrupted state transfer due to a complete failure of the group from which the state was being received. An object's state is represented as a sequence of CORBA `Any` objects. The programmer writes application-specific `get_state` and `set_state` member functions.

The `view_change` member function 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.

6 Examples

This section describes two applications whose development is considerably simplified by reliable CORBA. The first example shows how a replicated directory server can be implemented. The second example deals with a reliable stock exchange ticker.

6.1 A Fault-Tolerant Directory Service

Consider a replicated directory service specified by the following CORBA IDL declaration:

```
interface directory {
   // Register an entry under a key.
   void insert(in string key, in any entry)
      raises(ENTRY_EXISTS);
   // Retrieve an entry by key.
   void lookup(in string key, out any entry)
      raises(NO_SUCH_ENTRY);
   // Remove the entry with key.
   void remove(in string key, out any entry)
      raises(NO_SUCH_ENTRY);
};
```

This interface declares a `directory` object which maintains entries consisting of a `string` and an `any` object. To provide a fault-tolerant replicated service, two criteria must be met:

1. All `directory` objects must be updated identically. This is accomplished by atomic request delivery and totally ordered request multicast.

2. When a new `directory` object joins a group it must be able to replicate the state of the objects already in the group. This is accomplished with view change detection and state transfer.

Without reliable extensions to CORBA, this type of application is extremely difficult to write. Although CORBA does support a kind of multicast capability[1], it does not address the vital issues of atomicity, ordering, state transfer, or view management.

Without atomicity and ordering there is no way to ensure consistent replicated state across all `directory` objects. Without state transfer new objects have no way to become replicated. Finally, without view management, none of the other mechanisms are possible.

[1] through the `send_multiple_requests` DII operation.

Given the above interface declaration, the IDL compilers of Orbix+Isis and Electra generate a set of C++ files containing the static invocation interface of the `directory`, the invocation stubs, and a file containing a skeleton of the service with one C++ member function per operation declared in the interface. This file also provides a skeleton for the state transfer member functions of the `directory` object which can be completed by the programmer as follows (in Electra syntax):

```
// C++
void _im_directory::get_state(AnySeq& state, ...){
    // Pack all directory entries into the "state" object:
    for(ULong i =0; i < 2 * keys.length(); i += 2){
        state[i] <<= keys[i/2];
        state[i+1] <<= entries[i/2];
    };
};

void _im_directory::set_state(const AnySeq& state, ...){
    // Unpack the received directory entries.
    // First we must clear "keys" and "entries":
    keys.length(0); entries.length(0);
    for(ULong i =0; i < state.length(); i += 2){
        state[i] >>= keys[i/2];
        state[i+1] >>= entries[i/2];
    };
};
```

To increase the degree of fault-tolerance of a `directory` service, a `directory` object implementation is created and joined to the respective `directory` object group. The `get_state` member function of a group member is automatically invoked by the ORB, the `state` object is marshaled, transferred to the newcomer, unmarshaled, and passed to the newcomer's `set_state` member function. Owing to totally ordered multicast and to virtual synchrony, the internal states of the group members remain consistent in spite of membership changes and crashes, and despite client applications creating and removing entries from the service while membership changes are occurring.

To migrate a `directory` object, a new group is created and the object joined to it. Then, a new `directory` object is created on the destination host and joined to the group. The state of the obsolete object is automatically copied to the new object and the two `directory` objects will run synchronized. Now, the obsolete object can simply be destroyed.

6.2 A Reliable Stock Exchange Ticker Application

The next example deals with a reliable stock exchange ticker service that supplies timely stock market information to an unlimited set of receivers. The ticker service obtains stock quotes from one or more external data feeds and multicasts them to a group of receivers that have subscribed to the service. A receiver can be represented by following CORBA IDL interface:

```
// IDL
interface ticker {
    oneway void firstQuote(in string symbol,
        in string date, in long time, in float price);
    oneway void nextQuote(in string symbol,
        in string date, in long time, in float price);
    oneway void lastQuote(in string symbol,
        in string date, in long time, in float high,
        in float low, in float close, in float volume,
        in float priceVol);
    ...
};
```

The `firstQuote` operation serves to submit the first quote of the day for a given stock. `symbol` represents the ticker symbol, e.g., `"IBM"`, `"MSFT"`, or `"AAPL"`. To submit an individual quote, `nextQuote` is invoked. The `lastQuote` operation is invoked in order to submit the last quote of the day, where `high` contains the highest quote of that day, `low` the lowest, and `close` the last quote. `volume` provides the number of shares traded during the day, `priceVol` the price per volume product. As depicted in Figure 13, an Event Stream object group (Section 4.3) is used to transport the quotes. This ensures that:

- the communication infrastructure does not become a single point of failure. For the sake of fault-tolerance, the Event Stream service is implemented by a replicated group.
- receivers can access a predefined backlog of past quotes. When a trader starts his ticker, he will be presented with the backlog. This mechanism also supports the disconnected operation of mobile laptop computers.
- all receivers obtain the same quotes in exactly the same order, which is made possible by reliable, totally ordered group communication.
- information feeds as well as receivers can join and leave the system dynamically without affecting the other receivers or the consistency of the data.
- the quotes are sent to the receivers by object group invocations, which can be transmitted by a protocol such as IP-Multicast [2] to ensure scalability and good utilization of the available network bandwidth.
- the programming effort necessary to ensure reliability of the ticker service is kept at a minimum.

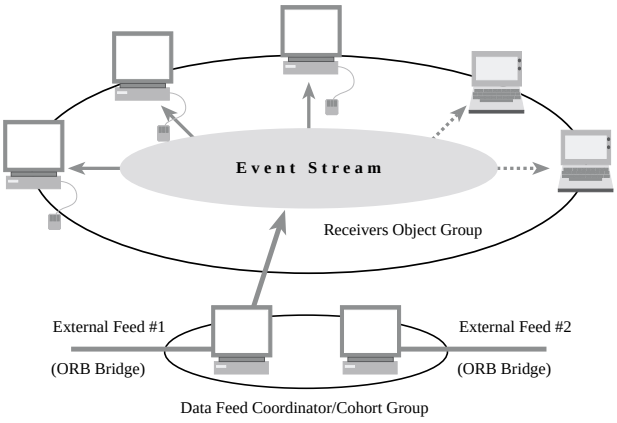

Figure 13: Reliable Stock Exchange Ticker

The data feed is provided through a coordinator/cohort object group (Section 4.2.3). Each member of the coordinator/cohort group has access to a different feed providing the same quotes, but only the coordinator of the group will transmit the quotes through the Event Stream. When the coordinator fails, a new coordinator is automatically elected and the transmission of the quotes is promptly resumed. Using a coordinator/cohort group ensures fault-tolerance of the feed and that each quote is obtained and transmitted only once.

The coordinator object injects quotes into the stream by simply invoking the Event Stream operations declared in `interface ticker`. For instance:

```
firstQuote("IBM", "950613", 765101, 91.250);
nextQuote("IBM", "950613", 765102, 91.250);
nextQuote("IBM", "950613", 765103, 91.125);
nextQuote("IBM", "950613", 765104, 91.750);
...
lastQuote("IBM", "950613", 870233, 92.250,
    90.875, 91.750, 2905.0, 266.5337);
```

Logging of events, multicast, and fault-tolerance are provided by the underlying Event Stream and Coordinator/Cohort frameworks. CORBA object request brokers like Orbix+Isis and Electra offer the fundamental system support for this kind of application, namely object groups, reliable multicast, and virtual synchrony.

7 Related Work

In this section we give a brief comparison of our approach with approaches taken in other object-oriented systems.

7.1 ANSA Interface Groups

An interface group abstraction was added to the Advanced Network Systems Architecture (ANSA) platform [14]. The interface group abstraction permits multicast of ANSA operations and group membership management. Similar to Electra, ANSA supports a transparent and a non-transparent group invocation style. Programmers can select between FIFO and total ordering of the multicasts delivered to group members.

The current implementation of the facility is very straightforward. In order to maintain total ordering and to perform group management, multicasts are channeled through a group member acting as sequencer of the group. A sequencer can become a performance bottleneck and represents a single point of failure. Multicast is realized by the repeated use of single messages and not by exploiting a hardware or software multicast facility where available. In contrast, Orbix+Isis and Electra employ fault-tolerant, distributed protocols to provide ordering and to perform group management[2]. Other differences are that ANSA is not CORBA compliant and does not ensure virtually synchronous program execution.

7.2 Arjuna, Avalon

Many of today's OODP environments offer atomic transactions in order to ensure consistency between distributed data objects. Two well known examples are Arjuna [15] and Avalon-C++ [5]. Such systems are well suited for applications which need to maintain consistency for long-lived shared data objects. When a failure occurs, transactions are aborted and partially completed operations are rolled back. Often, aborted transactions can be restarted only when the defect has been repaired, which can cause client applications to be suspended for a long time.

In contrast, Orbix+Isis and Electra are well suited for applications that require high availability and cannot tolerate long delays, for example fault-tolerant client-server applications and groupware. For this kind of application, active replication, non-blocking communication, and virtual synchrony can lead to increased performance. For a comparison of the virtual synchrony and the transactional model refer to [3, 6].

7.3 COSS Server Group

It has recently been proposed that group communication and fault-tolerance be added to CORBA by incorporating a *Server Group* abstraction into the

[2] see [4] for details.

OMA [1]. An extension of the OMG Common Object Services [12] has been suggested for that purpose. The proposed group management API is similar to the Electra BOA operations described in Section 5.3. Similar to Orbix+Isis, conglomeration functions can be provided which combine results from all group members into a single response for the client.

A Server Group service works as follows: Upon receiving a multicast request, the client's ORB dispatches it to the Server Group object. The Server Group object invokes a user-defined decomposition function to produce a set of subservice requests, and the group members are subsequently invoked. The Server Group object awaits the replies, combines them via a user-defined result combination function, and returns the result to the client.

Compared to our work, the advantage of the Server Group abstraction is that it can be added to an existing ORB without having to modify it. The disadvantages are that a Server Group object can become a performance bottleneck and a single point of failure, and that virtual synchrony is at best guaranteed between the members of a group but not in the whole application. Moreover, coherent membership management and state transfer are not addressed in the proposal, which makes the development of fault-tolerant objects difficult. Nevertheless, the Server Group abstraction is valuable for systems which do not require full virtual synchrony, e.g., certain groupware applications or parallel queries among independent databases.

Our approach is based on the realization that reliability is guaranteed only when the ORB is built on a communication system that implements the virtual synchrony model, since *all* communication must pass through the system. This means that reliability and fault-tolerance cannot simply be "added" to an ORB by means of a Common Object Service.

8 Conclusions

The present version of CORBA does not specify abstractions for the implementation of distributed applications whose behavior must be predictable in spite of partial failures, partitioned networks, failed communication links, and asynchronous communication. Process groups and virtual synchrony, on the other hand, provide the basis to build reliable and fault-tolerant distributed systems. Toolkits like Isis and Horus implement the process group and virtual synchrony paradigm, however, their APIs are proprietary and rather low-level.

To combine the benefits of CORBA and of the virtual synchrony paradigm, we suggest that a CORBA object request broker should be based on a toolkit like Isis or Horus. We believe that the combination of the group communication model with the CORBA model will lead to a compelling programming paradigm for future distributed systems.

We described how to enhance CORBA in order to achieve reliability and to support efficient one-to-many interaction with the abstraction of an object group. An object group permits the programmer to treat a collection of CORBA objects as if they were a single entity, and clients invoke operations on object groups without needing to know the exact membership of the group. Members of an object group have a consistent view of which objects are in the group. Coherent failure notification as well as state transfer are provided.

We also presented Orbix+Isis and Electra, the first two CORBA object request brokers to support the enhanced model. Application areas of our technology are the replicated directory service and the reliable stock exchange ticker outlined in Section 6, video-on-demand, groupware, distributed multimedia, and various kinds of fault-tolerant client/server applications.

References

[1] Adler, R. M. Group-Oriented Coordination Extensions to OMG's OMA/CORBA. Object Management Group presentation, San Jose, CA, June 26–29, 1995.

[2] Baker, S. Multicasting for Sound and Video. *Unix Review* (Feb. 1994).

[3] 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.

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

[5] Eppinger, J. L., Mummert, L. B., and Spector, A. Z. *Camelot and Avalon.* Morgan Kaufmann Publishers, Inc., 1991.

[6] Guerraoui, R., and Schiper, A. Transaction Model vs. Virtual Synchrony Model: Bridging the Gap. In *Distributed Systems: From Theory to Practice*, Lecture Notes in Computer Science. Springer-Verlag, 1994.

[7] Isis Distributed Systems, Inc., Iona Technologies, Ltd. *Orbix+Isis Programmer's Guide*, 1995. Document D071-00.

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

[9] 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.

[10] Maffeis, S. Adding Group Communication and Fault-Tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies* (Monterey, CA, June 1995), USENIX.

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

[12] Object Management Group. *Common Object Services Specification Volume I.* OMG Document 94-1-1.

[13] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1995. Revision 2.0.

[14] Oskiewicz, E., and Edwards, N. A Model for Interface Groups. Tech. Rep. AR.002.01, ANSA, Architecture Projects Management Limited, Cambridge UK, 1993.

[15] 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.

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

[17] van Renesse, R. A MUTS Tutorial. MUTS Documentation, Cornell University, 1993.

[18] van Renesse, R., Birman, K. P., and Maffeis, S. Horus: A Flexible Group Communication System. *Communications of the ACM* (1996). (to appear).