Distributed Programming using Object-Groups*

Silvano Maffeis

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

IFI TR 93.38

September 1993

Abstract

Many of today's object-oriented distributed toolkits focus on transactions to synchronize distributed applications. Transaction mechanisms are well suited for synchronizing *data-oriented* applications. In contrast *directly distributed systems*, like various kinds of failure tolerant client-server applications, often require a much finer degree of synchronization. Such a fine degree of synchronization is realizable by means of reliable process-group multicast, causal communication, and message passing mechanisms. However, toolkits offering these capabilities, for example ISIS or HORUS, are often low-level and therefore difficult to program. Therefore, we propose three object-oriented abstraction mechanisms, namely *object groups*, *Remote Method Calling*, and *Smart Proxies*. We argue that the abstractions are powerful enough to encapsulate process-groups, reliable multicast, and asynchronous message passing. Examples of how the abstractions are being realized in a prototype toolkit called ELECTRA illustrate this paper.

Keywords and Phrases: Replicated Objects, Object-Group Communication, Smart Proxy, Object-Oriented Distributed Programming, Remote Method Calling (RMC), Directly Distributed Systems

1 Introduction

Many of today's object-oriented systems, for example Arjuna [19] or Avalon-C++ [6], focus on transaction mechanisms to synchronize distributed, concurrent applications. Such mechanisms are powerful and expressive for data-oriented applications, but not adequate enough for providing a fine degree of synchronization [3]. A fine degree of synchronization is needed by various kinds of failure tolerant client-server applications, such as failure tolerant filesystems, nameservers, CSCW applications and so forth. These applications often require process-groups [2] as a structuring mechanism. We shall use the term *directly distributed system* [4] to denote this sort of application. More specific, in directly distributed systems,

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

distributed threads of execution interact *directly* with one another while continuously respecting constraints on their joint behavior. On the other side, in data-oriented systems distributed threads of execution interact *indirectly* with one another by issuing operations on shared (data) objects. This is the case in distributed database systems, for example.

The intent of this paper is to propose *object-groups*, *Remote Method Calling*, and *Smart Proxies* as a widely usable and powerful abstraction mechanism for building directly distributed systems. We exemplify how these abstractions are being realized in an object-oriented toolkit, called ELECTRA [14], which is in development at the University of Zurich.

This paper goes beyond previous work in object-based distributed programming, for example [1, 15, 19, 6], in that it addresses group related concepts in an object-oriented setting. We have been influenced by the work carried out in the ISIS [2, 3] and HORUS [22] project and by the ideas described in [9].

The rest of the paper is structured as follows. In the next section we give a short introduction to the ELECTRA toolkit. This toolkit serves to exemplify the proposed abstractions. Section 3 introduces object-groups and addresses some problems related with object-group communication. An example of how replicated services can be specified using an interface definition language is shown in Section 4. In Section 5 we propose Remote Method Calling as a mechanism for communicating both with single objects and object-groups, and we define the term Smart Proxy. Section 6 summarizes and concludes the paper.

2 Short Overview of the ELECTRA Toolkit

Currently, a small number of programming toolkits exist, mostly in the form of `C` programming libraries, which aid programmers in building directly distributed systems. Examples are ISIS [2, 3] and HORUS [22]. However, the methods and tools offered by such toolkits are often difficult to use by people lacking many years of experience in building distributed systems, since most of the provided abstractions are low-level and not expressive enough for modeling real world problems. Furthermore, it is mostly impossible to realize reusable software components for distributed systems directly with such programming libraries.

ELECTRA is an experimental C++ toolkit consisting of a class library for multi-threaded, active C++ objects (services), a class library for passive objects which can be communicated between services, a *Service Definition Language* (SDL) for declaring multi-threaded services using a syntax very similar to C++, a distributed runtime environment, and runtime monitoring tools. In ELECTRA, the communication with a remote object is basically asynchronous and takes place mainly by invoking the methods of a local stub-object, a so-called *proxy* [18]. The proxy object exports a logically centralized and logically local service to the client application, even when the accessed service is remote and replicated.

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

ELECTRA is built on top of the HORUS toolkit developed at Cornell University, Ithaca, New York. HORUS provides primitives for managing light-weight processes, causal communication, multicast, consistent views on failures, and process-group membership management. The primitives are provided in the form of a portable and efficient `C` library, but the abstraction level is low. For example, the communication in HORUS takes place mainly by communicating buffers of unstructured data using `send` and `receive` primitives. Abstract data types must manually be marshaled by the programmer before they can be communicated. HORUS offers no direct support for object-groups and the other abstractions described in the following sections.

3 Object-Groups

To achieve high availability despite host and network failures, an ELECTRA service can be replicated over several hosts. Each replica may fail independently. The replicas belong to the same *object-group*, an object-group being a non-empty collection of object instances which cooperate to achieve some common purpose. Inside the toolkit, communication with an object-group is by means of reliable multicast [8].

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

If a proxy object is bound to an object-group, then the method calls to the proxy are transparently multicasted to all of the objects in the group. The programmer always works with one single proxy, which may represent a single object or an object-group (Figure 1).

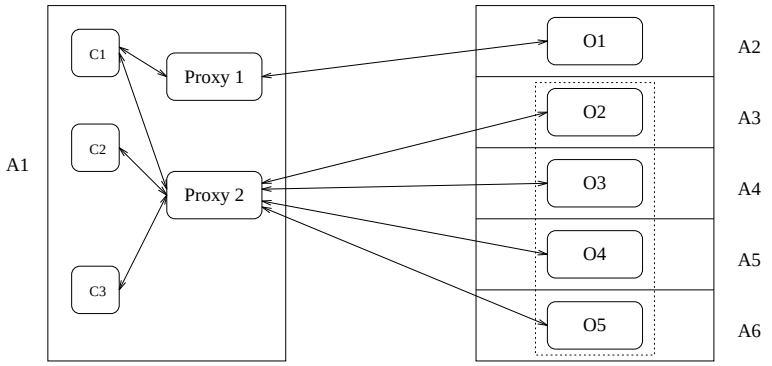

Figure 1: Proxy 1 interfaces to a single object whereas Proxy 2 interfaces to an object-group. C*n* denotes a client object, O*n* a remote server object, A*n* an address space.

3.1 Benefits of Object-Groups

Object-groups provide for replication transparency, meaning that client applications need not know whether they communicate with a single object or with an object-group [11]. Further advantages of object-groups are:

- *Load sharing.* A non-replicated object can be replaced by an object-group whenever the object becomes overloaded, without having to modify applications which use the object. Parallelism and load sharing can thus be increased step by step, provided that the toolkit offers abstraction mechanisms like Remote Method Calling and Smart Proxies (Section 5), which allow communication with an object-group as if it was a non-replicated object.

- *Fault tolerance.* Availability and fault tolerance are increased by using a homogeneous object-group to implement a service. Each object can fail independently from the other objects in the group. The service then remains operational as long as at least one of the group members is operational.

- *Addressing construct.* Object-groups can be used for addressing a set of objects which share some common characteristics. For example, some objects might want to be informed whenever a certain event occurs. It is easy to group these objects together into a heterogeneous group and to inform its members with a multicast whenever the event occurs.

- *Data publication.* Many applications, for example in the financial or flight reservation domain, need a mechanism for transmitting the same information to several recipients. This can be done efficiently by means of object-group multicast.
- *Distributed monitoring and management.* Object-groups provide a convenient way to distribute and gathering monitoring and management information in distributed applications.
- *Encapsulation.* Object-groups encapsulate an internal state and make it accessible through a set of well-defined methods. State encapsulation eases the realization of state-checkpointing and state-migration. Both issues are important for building scalable, failure tolerant applications.
- *Reusable components.* Object-groups can be used as reusable components for complex distributed systems. New (replicated) services can be realized by deriving them from existing services by means of class inheritance.

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

3.2 Ordering of Events

3.2.1 Causal Communication

Within a group of equivalent objects, consistency of the replicated state can be guaranteed by forcing operations to occur in the same order at all the replicas, which is known as *atomically (or totally) ordered multicast* [8]. However, some applications can preserve consistency with a weaker, less expensive *causal ordering* using *causal multicast*. Causal communication uses the *happened before* relationship described in the classic paper of Leslie Lamport [12] to synchronize messages in a distributed system. Only when necessary, *i.e.* when a causal relationship between messages exists, messages are reordered or deliberately delayed by the receiver's runtime system to match the happened before condition.

Causal communication is a fundamental concept of distributed systems. It is the extension of FIFO ordering in a setting with more than just one sender and one receiver [21]. Causal communication guarantees that messages which are causally related will be delivered in the same order at all processes which receive the messages (Figure 2). Unrelated messages may be delivered in different order at different processes. Causal delivery can be implemented using a simple time-vector mechanism [8, 5] and without introducing extra messages. All communication within an object-group must respect causality to obtain correct functioning, distributed applications.

3.2.2 Linearizability

Linearizability [10] is a correctness condition for concurrent objects that exploits the semantics of abstract data types, thus leading to a higher degree of parallelism. Intuitively, this corresponds to the requirement that "concurrent objects behave in a sane, predictable manner". In causal communication, operations on object-groups would normally be linearizable [3]. Thus, based on causal communication and linearizability, we can develop a consistency model which comprises individual objects and object-groups.

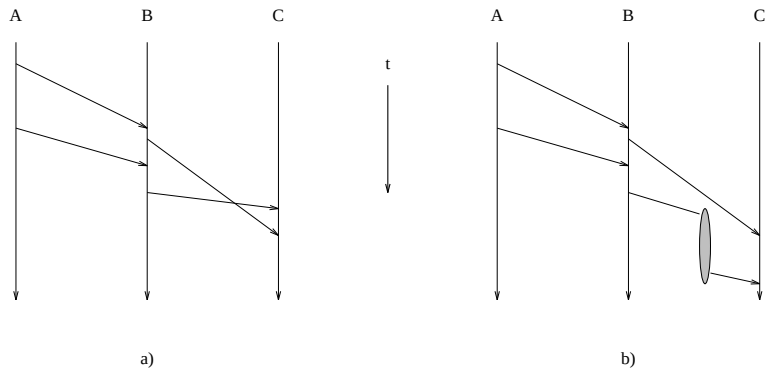

Figure 2: A, B and C are processes in a distributed system. An arrow denotes a message passed from one process to another. a) shows a situation where causal delivery is violated. b) shows causal delivery. The shaded area demonstrates that the message had to be deliberately delayed by C's runtime system to ensure causal delivery.

3.3 Group Membership Management

The problem of group membership management consists of maintaining consistent views on group membership among the objects in an object-group [16, 17]. At all times, an object-group member has a certain idea of which other objects belong to the group. This is called a *view*. Since objects may join or leave (terminate or fail) a group at any time, a view is subject to *view-changes*, and these changes must be propagated to all of the members. Therefore, a system-wide service providing consistent information on group membership must be provided. To guarantee consistency, the service delivers view changes to all of the members in the same order. Without such a mechanism, applications may behave incorrectly and in unexpected ways.

Typically, a view change protocol consists of two phases. In the first phase, the group membership service ensures that every member which is still alive receives every message that was sent in the current (old) view (*flush* phase). In the second phase, the view is updated and transmitted to the members (*view update* phase).

Group view management is not yet implemented in ELECTRA. Our goal is to make group view management as easy as possible for the programmer. Basically, we see two different approaches for realizing the group membership service:

1. We could provide a *central* ELECTRA group service. An object interested on the changes occurring in a certain group view registers with the service by specifying the group name. Whenever the group's view changes, the central membership service notifies the registered objects by means of totally ordered multicast. Failure tolerance is achieved by instantiating the service through an object-group.

   The advantage of this scheme is that it fits well into the ELECTRA philosophy: the group management service is a conventional ELECTRA service which can be replaced by the programmer, if required. The disadvantage is that the availability of the membership service depends on the degree of replication of the service itself and does not depend on the size or importance of a group view managed by the service.

2. An alternative to the previous scheme consists in having a member of each object-group to perform group membership management for its group, which is the scheme adopted in the HORUS toolkit [20]. This member is called the *coordinator* of the group. When the coordinator fails, a new coordinator is elected. For example, the oldest member of the object-group can be taken.

The advantage of this scheme is that a management service is provided per object-group and that the service remains operational as long as at least one group member is operational. The disadvantages are that it is harder for the programmer to replace this service, since more functionality must be "wired" into the toolkit than in the previous approach, and that an election mechanism must be provided for the case the coordinator's process fails or terminates.

3.4 Implementing Object-Groups

In this section we address the problem of implementing object-groups on top of distributed control technologies such as ISIS and HORUS. We chose these two platforms since they are available already and provide important mechanisms like causal communication, process groups and reliable multicast in an efficient and stable fashion.

In the current version of ISIS, the unit of distribution is a process. On the other hand, the object-group model is based on a finer degree of granularity. A process can instantiate many objects, and situations can thus arise in which a process is related to a certain group multiple times. Since the same process cannot be member of a group multiple times, a special addressing construct, an *object-identifier*, must be provided in addition to the address of the process in which the object resides [9]. ISIS messages are then tagged with the object-identifiers of the receiving objects and demultiplexing code within each process forwards incoming messages to the right object inside the process. We believe that the performance penalty of this demultiplexing mechanism is minor and that demultiplexing can be realized in a way which is transparent to the programmer.

The HORUS toolkit provides light-weight processes (threads) as the unit of distribution. Threads can be grouped together and do communicate by point-to-point messages and reliable multicast. In HORUS it is possible to implement multithreaded objects, and a thread acting as communication endpoint can thus easily be defined per object. It is therefore possible to implement efficient object-group communication, without having to provide a demultiplexing layer. The actual version of ELECTRA is built on top of the HORUS toolkit.

3.5 Result Collation

An operation which generates a return value will produce multiple return values when issued on an object-group. If the results are expected to be all the same, then the first result which arrives at the sender's runtime system can be selected and the other ones discarded.

If the results are expected to be different, for example when a group of time servers is used to read the local time on a set of workstations, then the individual results must be aggregated and passed to the issuing process.

ELECTRA-SDL (Service Definition Language) allows to specify result-collation. In the following example, `getTime()` returns the system time of each workstation a member of the object-group is instantiated on. The first version of `setAll()` sets each system clock to the same value, specified in `t`. The second version obtains a sequence of time values and sets the system clocks accordingly.

```
replicated service TimeServer: public ElectraService {
      cbcast      getTime( OUT SEQUENCE Time t );
      cbcast      setAll( IN Time t );
      cbcast      setAll( IN SEQUENCE Time t );
}
```

3.5.1 Result Collation in ISIS

The ISIS toolkit yet provides no interface definition language for declaring collation schemes and the like. Nevertheless, handling multiple replies is possible but becomes more complicated.

If the ISIS programmer requests multiple replies from a multicast, then *vectors* of answers are returned back by the multicast. The programmer specifies the type of the values returned by a multicast using a syntax similar to the `C printf()` conversion notation. For example, if a multicast returns n replies and the format was specified as `"%f"` (`float`), then the programmer needs to supply a vector with space for at least n elements of format `"%f"` as an argument to the multicast. When the reply itself is a vector, or when complex abstract datatypes are involved in a multicast, then things become troublesome. The following is an example of a simple ISIS multicast:

```
bcast(service, REQUEST, "{%f, %f, %f}[]", name, ALL,
      "%f %f", address, answer);
```

4 Declaring Replicated Services

In this section we demonstrate how replicated services can be defined in ELECTRA-SDL. The following SDL code fragment declares a replicated `NameServer` which is derived from `ElectraService`:

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

The application programmer supplies a non-replicated implementation of the service, ignoring complications due to distribution and replication, and defines the *category* of each operation. The category `cbcast` indicates that `install()` and `remove()` operations (see the example above) shall be multicasted to the members of the object-group using causal multicast. The `method` specifier indicates that `lookup()` requests are not to be multicasted since they do not alter the service's state. A `lookup()` can be issued on any replica, preferably on one where fast access is granted. Other supported multicast-categories are `abcast` for atomically ordered and `ubcast` for unordered multicast.

The following code fragment demonstrates how a replicated service is created. Here we suppose that the service is not running already. ELECTRA allows for remote object instantiation by providing a special service called the `Mushroomer` [14], which is running on each ELECTRA host.

```
Hosts hosts;
...                                          // fill the hosts object.
NameServer ns( "cs.cornell.edu:/nameserver" ); // create a local proxy.
ns.mushroom( hosts );                        // instantiate the object-group.
...
```

The `hosts` object contains the network addresses of the hosts on which a replica is to be instantiated. The service name `"cs.cornell.edu:/nameserver"` is used to register

the newly instantiated service into the system-wide trader service (see Section 5). Other applications can now simply bind to the newly created `NameServer`.

5 Remote Method Calling

In ELECTRA, Remote Method Calling (RMC) means that the methods of a remote object or object-group can be invoked thru a local proxy object. The proxy object exports a logically centralized and logically local service to the client application, even when the accessed service is remote and replicated. The parameters of these method invocations may include the standard C++ base types and C++ objects and may return standard C++ base types and C++ objects.

The parameter to a proxy instantiation is the identification name of the service to which the proxy shall be bound. This name obeys to the format <Domain>:<Path>. The name "cs.cornell.edu:/nameserver" for example, refers to the service "/nameserver" running within the domain "cs.cornell.edu". An identification name can refer to a single object or to an object-group. Different names can refer to the same object or object-group.

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

ELECTRA supports three kinds of RMC: asynchronous mode, synchronous mode and *promises* [13]. In asynchronous mode, an *upcall* method can be specified. This method will be invoked as an own light-weight process when the RMC returns. The following code fragment demonstrates ELECTRA RMCs. Whether the accessed service is an object-group or not does not influence the syntax of the RMCs:

```
NameServer ns( "cs.cornell.edu:/nameserver" );   // create a proxy.
ns.bind();                                       // bind to the remote object.
Promise promise;                                 // a "promise" object. Used later.
Name name; Address address; ...                  // parameters.

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

5.1 Smart Proxies

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

- A logically non-replicated interface is provided for accessing an object-group. As was stated above, this ensures that single objects can be replaced by object-groups without having to change the client applications.
- Caching can be performed within the Smart Proxy to increase the performance of an application. A Smart Proxy interfacing to a nameserver, for example, can maintain a local list holding the most frequently accessed nameserver entries.

- Using a protocol similar to the group membership protocol described in Section 3.3, a Smart Proxy can be informed by the migration service when an object associated with the proxy was migrated. Thus, object-migration can be hidden from client applications.
- The degree of replication an object-group provides can dynamically be controlled thru the proxy. The proxy provides runtime replication control by closely cooperating with the group management service (Section 3.3) and the naming service.
- Above tasks are normally performed by a Smart Proxy at the client site. At the server site, a Smart Proxy may allow to specify *encapsulator* objects to define concurrency control schemes [7], message filters etc. on a per-object basis.

Caching, migration, runtime replication control and encapsulators are not yet implemented in ELECTRA.

6 Conclusions

We suggest object-groups as a general structuring mechanism for object-oriented distributed systems. With the help of object-groups, programmers are less concerned about important but complicated issues like causal communication, reliable multicast, and replication.

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

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

References

[1] ARCHITECTURE PROJECTS MANAGEMENT LTD. *ANSAware Version 4.1 Manual Set.* Castle Park, Cambridge UK, Mar. 1993.

[2] BIRMAN, K. P. The Process Group Approach to Reliable Distributed Computing. Tech. Rep. 91-1216, Cornell University, July 1991. To appear in Communications of the ACM.

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

[4] BIRMAN, K. P., AND JOSEPH, T. A. Exploiting Replication in Distributed Systems. In *Distributed Systems*, S. Mullender, Ed. ACM Press, 1989.

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

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

[7] Guerraoui, R. Towards Modular Concurrency Control for Distributed Object Oriented Systems. In *IEEE Proceedings of the International Workshop on Future Trends in Distributed Computing Systems* (Sept. 1993).

[8] Hadzilacos, V., and Toueg, S. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, S. Mullender, Ed., second ed. Addison Wesley, 1993, ch. 5.

[9] Hagsand, O., Herzog, H., Birman, K., and Cooper, R. Object-Oriented Reliable Distributed Programming. *IEEE Workshop on Object-Orientation in Operating Systems* (Sept. 1992).

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

[11] Isis Distributed Systems Inc., Ithaca, NY. A Response to the ORB 2.0 RFI, Apr. 93.

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

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

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

[15] Marques, J. A., and Guedes, P. Extending the Operating System to Support an Object-Oriented Environment. In *OOPSLA Conference Proceedings* (Oct. 1989).

[16] Ricciardi, A. M. *The Group Membership Problem in Asynchronous Systems.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, Nov. 1992. No. 92-1313.

[17] Schiper, A., and Ricciardi, A. Virtually-Synchronous Communication Based on a Weak Failure Suspector. Tech. Rep. 93-1339, Cornell University, Apr. 1993.

[18] Shapiro, M., et al. SOS: An Object-oriented Operating System – Assessment and Perspectives. *Computing Systems 2*, 4 (Dec. 1989).

[19] 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.

[20] van Renesse, R. The Horus Uniform Group Interface. Horus Documentation.

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

[22] van Renesse, R., Birman, K. P., Cooper, R., Glade, B., and Stephenson, P. Reliable Multicast between Microkernels. In *Proceedings of the USENIX Workshop of Micro-Kernels and Other Kernel Architectures* (Seattle, Washington, Apr. 1992).