Electra – Making Distributed Programs Object-Oriented*

Silvano Maffeis
maffeis@ifi.unizh.ch
University of Zurich Dept. of Computer Science
Winterthurerstr. 190, CH–8057 Zurich, Switzerland

IFI TR 93.17†

July 1993

Abstract

Building failure-resilient, distributed software is difficult. In this paper we describe abstractions which help the programmer in developing such software systems by means of object-oriented programming. An object-oriented toolkit called Electra is presented, which provides abstractions for *Remote Method Calling* (RMC), object-groups, object-group communication with type checking, object-location trading and so forth. Electra allows the building of failure-resilient, *directly distributed systems* by reusing software components. With a simple example we demonstrate how a distributed, fault-tolerant client-server application can be realized in Electra.

Keywords: Failure-Tolerance, Object-Groups, Object-Oriented Distributed Programming, Replication, Reusability

1 Introduction

There are two important requirements which toolkits for the development of reliable distributed systems should fulfill: one is to provide powerful abstractions which help programmers in building reliable distributed systems, the other is to increase the extent to which software components can be reused.

Currently, a small number of programming toolkits exist, mostly in the form of C programming libraries, which aid programmers in building reliable distributed software over non-distributed operating systems. Examples are Isis [3], Ansa [1], Psync [20] and Horus [23]. 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

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

†In: Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems IV, USENIX, San Diego, CA, Sept. 1993

the provided abstractions are low-level and not expressive enough for modeling real world problems. Therefore, it is difficult to realize reusable software components for distributed systems directly with such programming libraries.

Object-oriented programming is known to be one of today's best programming concepts to cope with complex systems while providing maintainability, extensibility and reusability [17]. A programming concept claiming such attributes is especially interesting for failure-resilient distributed systems, as they tend to become very complex. In this paper we introduce an object-oriented approach for building distributed software systems, and demonstrate how the proposed abstractions have been realized in a prototype, called the Electra toolkit, which is developed at the University of Zurich.

The rest of this paper is organized as follows: in Section 2 we introduce the Electra toolkit and its underlying architecture. Section 3 proposes reusable classes for building distributed systems. Asynchronous *Remote Method Calling* and object-group communication is addressed in Section 4. Service location-trading and remote service instantiation is described in Section 5. We compare our work with other approaches in Section 6. Section 7 motivates further work and concludes the paper.

2 The Electra Toolkit

2.1 Short Overview

Electra [14, 15] is an object-oriented toolkit providing a set of new abstractions helping to build reliable distributed systems out of reusable components. The current prototype implementation of Electra is written in C++, but the concepts described in this paper apply to other object-oriented languages as well. The toolkit allows one to instantiate *active* objects, called *services* in this context, on remote computers in a heterogeneous network. A service is an abstract definition of what a server is prepared to do for its clients. It can be seen as an object encapsulating an internal state and is accessible through a set of well-defined methods. To achieve high availability despite host and communication failures, services can be *replicated* over a set of hosts using object-groups [10]. Applications can access the service as long as at least one replica of it is operational. Front-end code running in the client applications hides this replication. The communication with such services is basically asynchronous and mainly takes place by invoking the methods of a local stub-object, a so-called *proxy*. The local invocations are transmitted to the remote service by code the Electra stub generator produced. The process of transparently communicating with remote services by invoking methods of a local stub object is called *Remote Method Calling (RMC)* in Electra.

Electra services are multi-threaded, which means that several methods of a service can be invoked concurrently and that several services can be active in the same program at the same time. Whenever a RMC arrives at a service, a new thread of execution is created for handling the request. Electra provides a set of thread-safe classes (described in Section 3.2) which are used to implement reentrant resources inside a service.

Typical applications developed with Electra comprise so-called *directly distributed systems* [4], for example distributed file-systems or replicated nameservers. In directly distributed systems, multiple processes interact *directly* with one another while continuously respecting constraints on their joint behavior. *Data-oriented systems*, such as distributed databases and transaction processing systems are not the main application area of ELECTRA. In data-oriented applications, processes share data but are independent. Nevertheless, data-oriented applications can be accommodated in ELECTRA by realizing appropriate synchronization mechanisms on top of the ELECTRA reliable broadcast and RMC facility. ELECTRA and object-oriented databases do not compete, they can coexist.

The software development process in ELECTRA consists of identifying, defining and exploiting a set of services a distributed system should offer. The *identification* of services can be achieved by means of object-oriented analysis [5] of the requirements the system must fulfill. Services can then be *defined* using the ELECTRA service definition language (SDL) called ELECTRA-SDL. Finally, the programmer *exploits* services mainly by communicating with them using asynchronous or synchronous RMC and reliable object-group broadcast.

Thinking about abstract services and instantiating services through objects and object-groups is a powerful paradigm for building complex distributed systems. Inheritance and polymorphism apply also to ELECTRA services and thus provide reusability of existing services and higher productivity of the programmer.

In this paper we demonstrate several abstractions using a simple example of a stateless[1], multithreaded network file-server which is accessible by remote client applications. The definition of the file-server in ELECTRA-SDL is the following[2]:

Example 1:

```
service FileServer: public ElectraService {
      method    read( IN File f, OUT Buffer b );
      method    write( IN File f, IN Buffer b );
      method    unlink( IN File f );
      ...
}
```

FileServer is derived from the base-service ElectraService. ElectraService provides general methods for querying the version number of a service, for testing if a service is still operational and so forth.

A File object contains the name of the file to be accessed as well as an index specifying the seek position within the file. A Buffer object contains data read or to be written. The method File::at_eof checks whether the end of the file has been reached during a read operation.

Service definitions are fed to the ELECTRA stub generator which in turn generates the code needed to transparently access services using RMC and reliable broadcast.

[1]Meaning that no *state* information, for example seek positions related with individual files, are maintained by the server.

[2]The interface of the file-server is held simple for didactical reasons. Note that no open and close method is needed by a stateless file-server. Using a File and a Buffer object the read and write requests specify the area of the file to be read or written.

2.2 Architecture

We chose the Horus toolkit [22, 23] to form the basic transport layer of Electra (see Figure 1). Horus is the new Isis [3] implementation developed at Cornell University, Ithaca, New York. It comprises a C library for managing causality domains, consistent views over process-group membership, failure detection and so forth. Horus itself is based on the *Multicast Transport Service* (Muts) [23].

Muts offers low-level primitives mainly for managing threads (lightweight-processes), sending and receiving messages and performing reliable broadcasts with groups of threads in heterogeneous distributed systems. Electra's support for different operating-systems and communication protocols is ensured by the Muts layer.

Currently, Muts allows the handling and communication of unstructured data buffers only. The marshaling and communication of abstract datatypes is left to the programmer. In contrast, Electra allows programmers to transparently handle and communicate complex C++ objects.

The layers Electra is based upon could be replaced by another package offering low-level primitives for multi-threading, reliable broadcast and so forth.

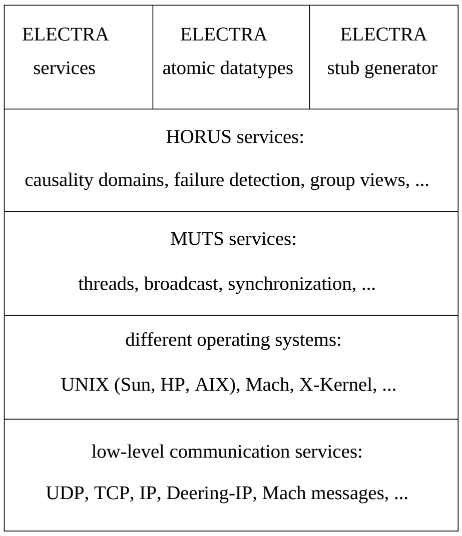

Figure 1: The architecture of the Electra toolkit.

3 Reusable Components for Distributed Systems

3.1 Electra Services

The Electra toolkit provides a set of reusable base-services useful for building distributed applications. Services are encapsulated into multi-threaded C++ objects accessible by RMC and reliable broadcast. Figure 2 depicts some basic services which are part of the Electra toolkit. Programmers can reuse these services to build their own, more sophisticated ones. Electra-SDL allows for multiple inheritance within service definitions.

The Trader and Mushroomer service will be addressed in Sections 5.1 and 5.2. The WindowServer allows a user program to open windows on workstation screens and to draw graphics on them. The GroupServer maintains group membership information applications need to perform broadcasts with service-groups. An ObjectServer is used to store and retrieve objects from storage using numeric object identifiers. The subclass OS_Mapped implements an easy to realize but slow object storage which maps objects onto files of the underlying operating system. LFS_Mapped maps single objects or object-clusters onto a fast log-structured file-system which is also part of the ELECTRA toolkit. Since numeric object identifiers impose a user-unfriendly naming mechanism, a DirectoryServer can be used to manage an ObjectServer. Basically, a DirectoryServer maps human-chosen ASCII names to numeric object identifiers. Hierarchic_DS implements a hierarchic name space, whereas Flat_DS realizes a flat, unstructured name space. Programmers can provide more sophisticated directory servers on their own by reusing the DirectoryServer class or one of its subclasses. The same ObjectServer can be associated with several DirectoryServers providing different naming schemes. Storage services and directory services are separated for flexibility.

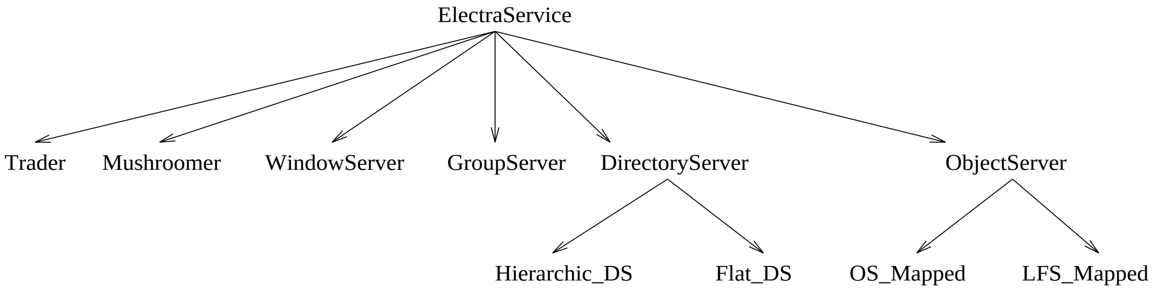

Figure 2: Basic ELECTRA services.

3.2 Passive ELECTRA Objects

Instances of the class ElectraService are *active* in the sense that they can receive RMCs and broadcasts. The toolkit also provides a thread-safe class library for *passive* objects which can be communicated by remote method calling or broadcasting. Passive ELECTRA objects are part of the ElectraLibrary class hierarchy as depicted in Figure 3. Next, we explain how

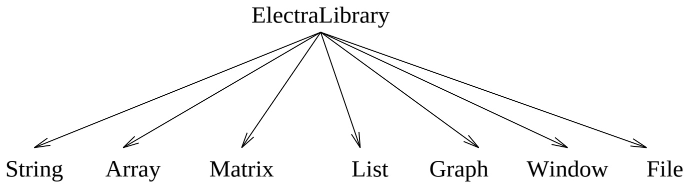

Figure 3: Passive ELECTRA objects.

instances of ElectraLibrary subclasses can be transferred to remote services and how they can be made persistent.

The abstract base-class ElectraLibrary dictates that each of its subclasses must provide the methods

```
dump(char *buffer);
undump(char *buffer);
```

dump stores the state of an object into the memory region buffer points at. Its complementary method undump changes the state of an object to the state stored in buffer. ElectraLibrary objects are communicated by transferring their state over the network. The client-code generated by the stub generator stores an object's state into a buffer by calling dump. The buffer data is then transferred over the network, broadcast or point-to-point, using primitives from the HORUS layer. On the receiving site, server-code also produced by the stub generator creates an "empty" object of the appropriate type and invokes undump with the just received buffer-data to initialize this empty object. This all happens inside the toolkit and is transparent to the programmer.

The ELECTRA ObjectServer relies on the dump and undump methods to store and read, respectively the content of objects. Objects can explicitly be made persistent by communicating them to an ObjectServer.

Using the C++ stream operators[3] an ElectraLibrary object can be stored on and read from a file of the underlying operating system. This allows to checkpoint an object's state and to initialize an object out of such a checkpoint. The stream operators themselves work by calling dump and undump.

4 Remote Method Calling and Object-Groups

As mentioned before, applications built with the ELECTRA toolkit make extensive use of Remote Method Calling (RMC) to communicate with remote services. RMC is an object-oriented variant of the well-known RPC [19] paradigm. RMC allows an application to invoke *methods* of a remote service using ordinary C++ syntax. The remote method call may include all the C++ base datatypes and also objects as arguments and it may return base datatypes and objects.

Communication with remote services is basically asynchronous and takes place by invoking the methods of a local stub-object, a so-called *proxy*. These local invocations are transferred to the remote service by code the ELECTRA stub generator produces. Clearly, the local application must first instantiate a stub-object for the remote service before it can interact with it. Remote accessible services are registered with a special service called the ELECTRA trader, described in Section 5.1.

A user program wishing to interact with our FileServer uses the C++ *client class definition* of the FileServer to instantiate a local stub-object. This client class definition is automatically generated out of the service definition in Example 1 by the ELECTRA stub generator, and is shown in the following code fragment:

Example 2:

[3] operator>> and operator<<

```
class FileServer {
      error_t      read( File f, Buffer& b, calltype c, Answer a = NO_ANSWER);
      error_t      write( File f, Buffer b, calltype c, Answer a = NO_ANSWER);
      error_t      unlink( File f, calltype c, Answer a = NO_ANSWER);
}
```

Each method of a client stub holds two more parameters than the associated method in the service definition: the calltype parameter indicates whether an asynchronous (async) or synchronous (sync) RMC shall be performed. In asynchronous calltype, the thread invoking an RMC is not blocked and an Answer parameter can be provided to the RMC indicating what action shall be raised when the RMC has terminated. Answer defines a method to be invoked with an own thread of execution when the response to the RMC arrives. Answers can be omitted from the call. RMC invocations always return an object of type error_t containing an error description in case a remote operation failed.

To summarize, the following C++ code fragment demonstrates how asynchronous and synchronous RMCs are performed. The example code contacts a remote file-server to copy a file from there to another remote file-server by successively transferring data blocks of size BUF_SIZE. In this particular case, read-operations from the file-server must be performed synchronously. Write-operations into the file-server can be performed asynchronously, since in this case only IN parameters are transferred. Answers to asynchronous RMCs are guaranteed to arrive in the same order as the related RMCs were invoked.

Example 3:

```
Buffer buf( BUF_SIZE );
File from( "/tmp/from" );
File to( "/tmp/to" );

FileServer eterna( "/machines/eterna/FileServer", bind );
FileServer sirius( "/machines/sirius/FileServer", bind );

while( ! from.at_eof() ){
      eterna.read( from, buf, sync);
      from.position += buf.size;

      sirius.write( to, buf, async);
      to.position += buf.size;
}
```

A third calltype exists which allows for an intermediate form of synchronization. When the Promise [13] calltype is specified, ELECTRA returns a Promise object which allows for synchronization at a later stage in the program:

Example 4:

```
Promise p;
remote.read( from, buf, promise, p );
...

p.wait();
```

The read operation is issued asynchronously, but p.wait() blocks the calling thread until the RMC is finished, if this is not already the case. After this explicit synchronization, the file data read is available in the buf object.

The first parameter to the FileServer constructor ("/machines/eterna/FileServer") is needed by ELECTRA to find the physical service-address of the remote FileServer. A physical service-address provides information such as the network address and the port number of the service. The mapping between keys and service-addresses is performed by a special service known as the *location-trader*. The trader is an important part of the ELECTRA system and will be described in Section 5.1.

4.1 Reliable Object-Group Communication

In many situations, it is necessary to have a facility which allows one to send messages to a *group* of services by means of reliable broadcast [2]. For example, operations on replicated services are transparently broadcasted to all of their replicas to keep their internal state consistent. Broadcasts are reliable in the sense that all group members or none of them, in the case a failure occurred during the operation, will receive a broadcast. A broadcast can be performed *atomically ordered*, thus ensuring that subsequent broadcasts arrive in the same order at all replicas. However, some applications can preserve consistency with a weaker, *causal* ordering [12] leading to better performance.

In ELECTRA-SDL the ordering of events can be defined on a per-method basis by specifying *abcast* for atomically ordered broadcast, *cbcast* for causal broadcast or *ubcast* for unordered broadcast.

As an example, imagine a replicated FileServer where the write and unlink operations are reliably broadcasted to the replicas:

Example 5:

```
replicated service FileServer: public ElectraService {
      method    read( IN File f, OUT Buffer b );

      abcast    write( IN File f, IN Buffer b);
      abcast    unlink( IN File f );
}
```

In this example, read operations are not broadcasted since the file is not altered by a read. write and unlink operations are broadcasted using *atomically ordered* broadcast, thus ensuring that subsequent operations always arrive in the same order at all replicas. For a more detailed discussion of operation-orderings on replicated services refer to [11].

The HORUS layer is capable of delivering broadcasts on networks providing no hardware or software support for broadcast, such as the *Internet* without adequate extensions [6]. For such networks, a broadcast is automatically delivered using a point-to-point message for each member of the group. If support for broadcasting is provided, which is the case in LANs like the *Ethernet*, MUTS takes advantage of it. This optimization is transparent to the programmer.

ELECTRA enhances the basic HORUS broadcast facilities by encapsulating them into the service abstraction, by providing strong type checking for the parameters of a broadcast at compile time and by allowing compound objects to be broadcasted. All these enhancements have been found to considerably ease the realization of distributed systems involving replicated resources.

4.2 Benefits of Object-Groups

Object-groups provide for replication transparency, meaning that client applications need not to know whether they communicate with a single instance of a service or with an object-group [10]. Further advantages of object-groups are:

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

5 Special ELECTRA Services

5.1 Location Trading

Remote services are located using the ELECTRA location-trader. The location trader is a special server known to all ELECTRA applications. Whenever a client stub-object is instantiated by a user program, the first parameter to the constructor (see Example 3) contains the human-chosen name of the service which is transmitted to the trader by the generated stub code. The trader returns the physical address of the service. The second parameter to a service instantiation can be set to `bind`. This indicates that an existing service address should be bound to a local stub-object. The whole process is performed inside the toolkit and is transparent to the programmer.

When programmers wish to register their own services, this second parameter will be set to install instead of bind, resulting in a new service that other user programs can bind and exploit as well.

The format of the human-chosen service name is <Domain>:<Path>. For example, the service name "cs.cornell.edu:/campus/nameserver" refers to a service registered at the master trader in the "cs.cornell.edu" domain. When the <Domain> specifier is omitted, then the local domain is chosen per default.

The ELECTRA *master* trader, which has a special service address known to all applications within its administrative domain, a LAN for example, also manages references to master traders in other domains. A master trader per LAN can be defined holding information about the services running in its LAN. Analogously to nameserving in the *Internet* [18], one can imagine a huge network of traders each serving its own administrative domain and holding "pointers" to other traders in other domains. To achieve failure-tolerance, master traders are replicated over several hosts in their administrative domain.

From time to time the trader sends the are_you_alive RMC to each registered service. Services failing to respond to the signal are discarded from the trader's database. The programmer can specify that a service failing to respond be automatically restarted by the trader. The are_you_alive method is realized in the ElectraService base-service (Figure 2) and thus inherited by every ELECTRA service.

5.2 The Mushrooming Mechanism

Remote services can be dynamically created on demand. The programmer can thus have his services "pop up" elsewhere in the network. We call this concept *mushrooming.* On each host being part of the ELECTRA system a local service called the Mushroomer is instantiated at boot-time of the host. Mushrooming of a remote service is performed through the following steps:

- The mushroomer at the target host must determine whether an instance of the requested service has previously been created on this host. It does this by consulting an internal table containing the names of the services already running on the host.
- If no instance of the requested service is running, then the mushroomer obtains the *path* from which the executable binary for the service shall be loaded. The mushroomer loads the appropriate executable which is time-expensive. Finally, the new service is instantiated and registered with the master trader. The mushrooming process ends here.
- Otherwise, the executable binary for the requested service has been loaded previously. In this case a new instance of the service is quickly created. The new service is then registered with the master trader.

This process is also transparent to the programmer. Instantiating a FileServer on a remote host is achieved as follows:

Example 6:

FileServer fs("/machines/eterna/FileServer", "eterna.ifi.unizh.ch", mushroom);

The service is instantiated on the remote host "eterna.ifi.unizh.ch" and registered into the master trader of domain "ifi.unizh.ch" using the searchkey "/machines/eterna/FileServer".

6 Related Projects

In this section we briefly refer to some of the object-oriented systems which motivate and influence our work.

Arjuna [21] is an object-oriented programming system which makes use of nested atomic transactions to grant integrity between persistent objects.

Avalon/C++ is a superset of C++ and relies on the *Camelot* Carnegie-Mellon Low Overhead Transaction Facility [8]. Avalon/C++ provides transaction semantics for atomic objects.

Arjuna, Avalon/C++ and most of today's object-oriented, distributed toolkits are conceived for *data-oriented* applications, where processes share data-objects but are independent. Atomic (nested) transactions are the natural consistency model for such systems. Most of the object-oriented environments of today focus on transactions. Such systems will surely prove powerful for database applications, but awkward where the goal is to support high availability client-server applications auch as cooperative computing applications and other object-group based applications [3]. ELECTRA is explicitly conceived for such object-group based applications, where processes interact *directly* with one another while continuously respecting constraints on their joint behavior. Transaction mechanisms are not adequate enough here. We propose the causal object-group broadcast and asynchronous RMC abstractions for efficiently synchronizing such applications and realizing replicated, highly available services.

The *Comandos* (Construction and Management of Distributed Operational Systems) project aims at defining and implementing an integrated platform supporting transparent distribution and persistent programming [16]. It introduces the concept that an object is member of one or several domains. Every Comandos domain can span multiple machines and accessing a remote object causes the expansion of a domain to include the remote object.

In contrast, ELECTRA does explicitly *not* provide full distribution transparency, although layers providing full distribution transparency can be built on top of ELECTRA. ELECTRA provides the notion of *selective* transparency [15], meaning that the programmer can explicitly place services and objects on selected hosts, if required, and thus break the location-transparency. If not required, the choice of where a new service is to be placed can be left to ELECTRA. Full transparency can lead to severe performance problems in large-scale distributed systems. ELECTRA's concept of trading domains allows the efficient interconnection of applications running in different networks.

CORBA (Common Object Request Broker Architecture and Specification) [7] provides mechanisms by which objects transparently make requests and receive responses. The *Common Object Request Broker* provides interoperability between applications on different machines in heterogeneous networks. System independence is mainly achieved by having an Object Request Broker (ORB) Architecture offering specific functions independent from the underlying operating-system. Language independence is mainly achieved by specifying service interfaces in CORBA-IDL (Interface Definition Language).

CORBA and ELECTRA have several aims and features in common. In contrast to CORBA, we address the aspect of object-group broadcasting, ordering of events, failure reporting and replication within the toolkit. The actual version of CORBA-IDL allows for a simple form of public inheritance between interfaces. In analogy to C++, ELECTRA-SDL supports public, private and virtual derivation for service specifications, which is useful for structuring complex, object-oriented distributed systems. Furthermore, inheritance and dynamic binding are also supported for ELECTRA objects which act as parameters to RMCs and broadcasts.

Unfortunately, the current CORBA specification [7] does neither address object-groups nor consistent view on failures for the processes in a distributed system. We feel that both concepts are important for reliable distributed systems. The extension of the CORBA object request broker with object-groups has been proposed in [10].

7 Conclusions and Future Work

In this paper we have described object-oriented concepts for easing the realization of directly distributed systems and how these concepts are realized in a prototype called the ELECTRA toolkit. For small distributed applications, the object-oriented paradigm can lead to more readable programs already, but in complex projects it may mean the difference between success and failure.

We have shown that it is possible to enhance an existing, procedural programming library for distributed systems, HORUS in our example, with powerful object-oriented abstractions. These abstractions considerably ease the development of failure-resilient distributed systems and allow for reusability of software components.

The reusable components have been successfully used over the past months to build a distributed nameserver, a simple distributed file-server, a parallel pattern-matching program and other applications. First experiences have been very motivating for us from the viewpoint of abstraction level and performance. Nevertheless, the toolkit is just at the beginning of its evolution and exploitation.

Whereas ELECTRA is primarily conceived for directly distributed applications, it would be useful to enhance the toolkit with constructs for atomic actions based on the causal broadcast facility. This would ensure better support for purely data-oriented applications.

Currently, only the migration of passive ELECTRA objects (data-objects acting as parameter to RMCs and broadcasts) is supported. Future work will also consist in realizing appropriate mechanisms to "freeze" active, multi-threaded services, transferring their state using the proposed dump/undump mechanism, and resuming their threads on remote hosts. This would give the programmer an abstraction for migrating threads in heterogeneous systems by only migrating the *state* of multi-threaded objects and not their program code.

The implementation of metaobjects [9] is also being considered. This type-information available at runtime would give better support for the marshaling of complex object-structures such as graphs, trees and so on. Type-information in form of metaobjects is also helpful for debugging distributed applications by allowing the inspection of the state of active and passive objects at runtime.

Acknowledgements

We would like to thank Robbert van Renesse for easing the integration of Horus in Electra and for his motivating comments. Ken Birman, Clemens Cap, Andrew Hutchison and Markus Kiser carefully reread earlier versions of this paper and provided helpful suggestions. A special thank goes to Markus Kolland and to Lutz Richter for making this work possible. This work is financed by Siemens AG, ZFE, Germany and the Schweizer Bundesamt für Konjunkturfragen, Grants No. 2255.1 and 2554.1.

Availability

The primary goal of the Electra project is to research the design and implementation of object-group based, distributed systems. The described research-prototype is not available outside the University of Zurich at this time. Further technical reports about the matter will be placed on the *anonymous ftp* server of the University of Zurich, network address `ftp.ifi.unizh.ch`. Requests and comments regarding Electra can be directed to `electra@ifi.unizh.ch`.

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] Coad, P., and Yourdon, E. *Object-Oriented Analysis*, second ed. Prentice-Hall, 1991.

[6] Deering, S. Host Extensions for IP Multicasting. RFC 1112, Stanford University, Aug. 1989.

[7] Digital Equipment Corp., Hewlett-Packard Co., HyperDesk Corp., NCR Corp., Object Design Inc., SunSoft Inc. *The Common Object Request Broker: Architecture and Specification*, Revision 1.1 ed., Dec. 1991. OMG Document Number 91.12.1.

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

[9] Goldberg, A., and Robson, D. *Smalltalk-80: The Language.* Addison Wesley, 1989.

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

[11] Ladin, R., Liskov, B., and Ghemawat, S. Replication. *ACM Transaction on Computer Systems 10*, 4 (Nov. 1992).

[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. The Electra Programming Environment. Tech. Rep. 93.16, CS Dept. University of Zurich, Switzerland, Dec. 1992. (In preparation).

[15] Maffeis, S. Remote Method Calling and Object Group Communication. ECOOP Workshop on Object-based Distributed Programming, Kaiserslautern, Germany, July 1993.

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

[17] Meyer, B. *Object Oriented Software Constrution.* Prentice-Hall, 1988.

[18] Mockapetris, P. V. Domain Names – Concepts and Facilities. RFC 1034, Network Working Group, Nov. 1987.

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

[20] Peterson, L., Buchholz, N., and Schlichting, R. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems 7*, 3 (Aug. 1989).

[21] 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.

[22] van Renesse, R. A MUTS Tutorial. MUTS documentation, Cornell University, 1993.

[23] 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).