Run-Time Support for Object-Oriented Distributed Programming

Dissertation
der Wirtschaftswissenschaftlichen
Fakultät
der Universität Zürich

zur Erlangung der Würde
eines Doktors der Wirtschaftswissenschaft

vorgelegt von

Silvano Maffeis
von Männedorf ZH

genehmigt auf Antrag von

Prof. Dr. Lutz H. Richter
Prof. Dr. Kenneth P. Birman

Die Wirtschaftswissenschaftliche Fakultät gestattet hierdurch die Drucklegung der vorliegenden Dissertation, ohne damit zu den darin ausgesprochenen Anschauungen Stellung zu nehmen.

Zürich, den 22. Februar 1995

Der Dekan: Prof. Dr. H. Garbers

"Is it a fact—or have I dreamt it—that, by means of electricity, the world of matter has become a great nerve, vibrating thousands of miles in a breathless point of time? Rather, the round globe is a vast head, a brain, instinct with intelligence! Or, shall we say, it is itself a thought, nothing but thought, and no longer the substance which we deemed it!"

Nathaniel Hawthorne
The House of the Seven Gables, 1851

Contents

List of Figures

List of Tables

Acknowledgements

My thanks go to all the people who supported and encouraged me when working on this thesis. Faced with the nontrivial problem of whom to thank first, I finally decided to list the acknowledgements in alphabetical order. My sincerest gratitude goes to:

Claudine Ackermann for the excellent administration of our library; *Philipp Ackermann* for his encouragement; *Nick Almássy* for getting me to use Emacs and for his support; *Henri Bal* for the valuable exchange of ideas and for his encouragement; *Bela Ban* for the valuable discussions and for his encouragement; *Kurt Bauknecht* for building up our Computer Science Department and for providing the research environment which made this work possible; *Peter Baumann* for a fruitful exchange of ideas; *Raoul Bhoedjang* for the valuable discussions, and for a good time in San Diego; *Ken Birman* for acting as referee of this thesis, for many hints and contacts, for all the invaluable research carried out by himself and by his group, for inventing Isis, for providing me with useful comments on my papers, on this thesis, and on Electra; *Walter "Bischi" Bischofberger* for the interesting discussions we had, for his fruitful collaboration, and for allowing me to write Beyond-Sniff's multicast transport service; *Mike Brodie* for a fruitful exchange of ideas; *Clemens Cap* for his encouragement and for the valuable discussions we had; *Werner Dreyer* for his encouragement, and for the exciting discussions at the Kowloon restaurant; *Martin Dürst* for a fruitful exchange of ideas; *Maja Ebner* for solving administrative problems; *Jürg Fässler* for his encouragement; *Antonio Gatti* for his encouragement; *Anton Gilg* for his support; *Richard Golding* for the valuable discussions on wide area networking, for his motivating comments on my work, and for inviting me to Amsterdam; *Thomas Grotehen* for providing me with papers, contacts, and for the valuable discussions we had; *Rachid Guerraoui*

for his proofreading and for his valuable suggestions; *Katharina Gutmann* for the proofreading of this thesis; *Thomas Haas* for carrying out programming work for me; *Eric Hamilton* for his encouraging comments on my work; *Ralf Hauser* for his encouragement and for getting me to use Gopher and World Wide Web; *Holger Herzog* for his support; *Andrew Hutchison* for his encouragement and for his valuable suggestions; *Thomas Jell* for his encouragement and for supporting Beyond-Sniff; *Dag Johansen* for his encouragement; *Markus Kiser* for his encouragement, and for his excellent administration of our Usenet service; *Markus Kolland* for his valuable suggestions and for his encouragement; *Lotti Kündig* for solving administrative problems; *Koen Langendoen* for a private demonstration of Amoeba and Orca; *Edgar Lederer* for his encouragement; *Jacob Levy* for his valuable comments on Electra; *Apostolos Lytras* for carrying out programming work for me; *Kai-Uwe Mätzel* for a fruitful exchange of ideas and for his indispensable contributions to the Beyond-Sniff project; *Corinne Maurer* for solving administrative problems; *Daniel Meier* for his encouragement; *Philip Mötteli* for the valuable discussions we had; *Thomas Nadig*, *Zafer Öztürk*, and *Markus Pilz* for their encouragement; *Beat Rageth* for providing me with all the privileges a Unix user can imagine and for his excellent and liberal administration of our network; *Kerstin Reiher* for the excellent administration of our library; *Robbert van Renesse* for his patience, for answering all of my 1000 e-mails, for inventing Horus, and for making my work on Electra possible; my supervisor *Lutz Richter* for giving me a job as research assistant, for allowing me to work on this exciting subject, for his supervision and support, for his proofreading, and for providing me with the liberty I need; *Reinhard Riedl* for the interesting discussions we had, for useful suggestions, and for having been an agreeable office-mate; *Xavier Rousset de Pina* for his valuable suggestions, his fatherly encouragement, the discussions we had, and for inviting me to Grenoble; *René Schaad* for his encouragement; *Bruno Schäffer* for the valuable discussions we had and for his indispensable contributions to the Beyond-Sniff project; *André Schiper* for his encouragement; *Beat Schmid* for his encouragement; *Petra Schmidli* for solving administrative problems; *Walter Schnell* for providing me with information on Bankers Trust; *René Schwarb* for his encouragement; *Marc Shapiro* for a fruitful exchange of ideas; *Enrico Solcà* for his excellent maintenance of our computers and for his readiness to help when my PC broke down; *Kurt Stadler* for encouraging and supporting my work on Electra; *Adrian "Ramo" Steinmann* for the proofreading of this thesis, for lots of valuable suggestions, and for having been my first Unix guru;

Volker Strumpen for his critical remarks; *Takashi Suezava* for carrying out experiments for me; *Kornél Szabó* for the valuable discussions we had; *Paul Verschure* for his encouragement, for a good time in La Jolla, and for lending me his mountain bike; *Nigel Walder* for some precious information on Bankers Trust; *Thomas Wilkes* for his encouraging comments on Electra; ...and to all those I forgot to mention.

My special thanks go to my parents, *Giulio* and *Giovanna Maffeis*, and to my brothers *Luciano* and *Fabio*, for their ever continuing support. Most of all I would like to thank *Patrizia* for marrying me, believing in me, and for her loving support.

This work was funded by Siemens-Nixdorf AG, by the Union Bank of Switzerland, and by Eidgenössisches Volkswirtschaftsdepartement, Kommission zur Förderung der wissenschaftlichen Forschung (KWF/CERS), Switzerland, Grants No. 2255.1, 2554.1, 2704.1.

Abstract

This thesis presents the *Electra Object Model* for object-oriented programming of reliable distributed systems. The Electra Object Model is an execution model for object-oriented distributed applications, which is based on several theories and system services stemming from contemporary distributed systems research. It provides asynchronous one-to-many communication and virtually synchronous execution, which makes it ideal for programming failure-resilient, efficient, and robust applications out of reusable software components. Groups of objects which can be dealt with in a uniform and nearly transparent way are the key component of the model.

In this thesis, we further discuss design and implementation of a novel CORBA (Common Object Request Broker Architecture) programming environment, called the *Electra toolkit*, which is based on the aforementioned execution model. The Electra toolkit is unique in that it permits object-oriented programming of reliable distributed software with a CORBA-compliant interface by exploiting the powerful primitive operations provided by communication subsystems such as CONSUL, DELTA-4, HORUS, ISIS, and TRANSIS. Because of its *flexible system design*, Electra can be configured to fit various communication subsystems.

Keywords and Phrases:

Distributed Systems, Run-Time Systems, Object-Oriented Distributed Programming, Object-Groups, Multicast, Software Implemented Fault-Tolerance, System Design, CORBA, Electra

Zusammenfassung

Die Begriffe *verteiltes System* und *Rechnernetz* werden oft miteinander verwechselt. Unter einem Rechnernetz versteht man eine Anzahl autonomer PCs, Arbeitsplatzstationen oder Grossrechner, welche über Kommunikationskanäle miteinander verbunden sind. Die Benutzerinnen und Benutzer von Rechnernetzen können mittels Datei-Transferprotokollen Daten austauschen oder über virtuelle Terminals mit anderen Rechnern Verbindungen aufnehmen. Im besten Fall sorgt ein Netzwerk-Dateisystem für eine systemweite Sicht auf die vorhandenen Benutzerdaten. Die Dezentralisierung wird aber spätestens dann, wenn ein Rechner ausfällt, auf schmerzliche Weise ersichtlich.

Einem verteilten System im Sinne dieser Arbeit liegt zwar immer ein Rechnernetz zugrunde, ausgefeilte System Software spiegelt jedoch eine weitgehend transparente Rechenumgebung vor. Benutzerinnen und Benutzer müssen sich nicht mehr um die physikalischen Adressen von Rechnern, Peripheriegeräten oder Daten kümmern, und mittels Replikation wird eine hohe Verfügbarkeit wichtiger Systemdienste gewährleistet.

In Betrieben werden immer mehr Rechnernetze und selten auch schon verteilte Systeme eingesetzt, weil diese oft mehr Flexibilität und ein besseres Preis-Leistungs-Verhältnis bieten als teure Grossrechner. Gerne spricht man heute von "Downsizing", von "offenen Systemen" und vom "Client-Server" Modell. Die objektorientierte, *verteilte* Programmierung stellt eine Verallgemeinerung des "Client-Server" Modells dar und nimmt für sich in Anspruch, die Entwicklung qualitativ hochstehender, verteilter Systeme gut zu unterstützen.

Das Ziel dieser Dissertation ist es aufzuzeigen, wie skalierbare, ausfalltolerante, verteilte Systeme mittels eines objektorientierten Ansatzes implementiert werden können. Zu diesem Zweck wurden im Rahmen dieser Dis-

sertation ein *Ausführungsmodell* (Kapitel 3), eine *flexible Systemarchitektur* (Kapitel 4) und eine *Programmierumgebung* (Kapitel 5) entworfen, realisiert und beschrieben.

Ausführungsmodell

Das Ausführungsmodell beinhaltet Protokolle und Systemdienste, welche die Implementierung von stabiler Software auf asynchronen, verteilten Systemen unterstützen. Beispielsweise sind dies Protokolle, um eine konsistente Ordnung verteilter Ereignisse anhand einer *logischen* Systemzeit zu garantieren, oder ein Systemdienst, welcher Ausfälle erkennt und die Objekte in einer verteilten Anwendung davon benachrichtigt. Da asynchrone Systeme durch das Fehlen einer zentralen Systemuhr gekennzeichnet sind, ist Erkennung und konsistente Behandlung von Ausfällen wichtig für wohlfunktionierende Anwendungen.

Um ausfalltolerante Systemdienste zu realisieren, bietet unser Modell *Objektgruppen* an. Unter einer Objektgruppe verstehen wir eine Anzahl kooperierender Objekte, wobei sich im Extremfall jedes Objekt auf einem anderen Rechner des verteilten Systems befindet. Aus der Sicht der Klienten verläuft die Kommunikation mit einer Objektgruppe analog zur Kommunikation mit einem Einzelobjekt, und das Laufzeitsystem stellt sicher, dass die Nachrichten der Klienten an alle Gruppenmitgliedern zugestellt werden, was als *reliable multicast* Kommunikation bezeichnet wird. Der von der Gruppe gemeinsam realisierte Systemdienst ist für Klienten solange verfügbar, wie mindestens ein Gruppenmitglied verfügbar ist. Objektgruppen ermöglichen aber nicht nur eine Ausfalltoleranz, sondern unterstützen unter anderem auch parallele Systemdienste, effiziente Datenreplikation, mobile Komponenten, CSCW Anwendungen und Netzwerk Management.

Das Ziel des Ausführungsmodells ist, dem Programmierer die Illusion eines enggekoppelten, objektorientierten Systems zu vermitteln, obwohl in Wirklichkeit eine lose gekoppelte Rechenumgebung vorliegt. Trotz des dynamischen Erscheinens oder Austretens von Objekten und trotz Ausfällen bleibt in unserem Modell das Verhalten einer verteilten Anwendung voraussagbar. Man spricht in diesem Zusammenhang von *virtueller Synchronie*.

Flexible Systemarchitektur

Unterschiedliche Betriebssysteme, inkompatible Standards und sich laufend ändernde Programmierschnittstellen sind typisch für die Informatik heute. Der zweite Beitrag dieser Dissertation besteht darin, ein flexibles Systemdesign vorzuschlagen, welches es ermöglicht, systemnahe Software portabel und wiederverwendbar zu gestalten. Die Grundidee ist, systemnahe Software basierend auf den Operationen einer generischen Portabilitätsschicht zu realisieren. Diese Schicht wird dann mittels eines sogenannten *Adaptor Objekts* an die darunterliegende, reale Systemschnittstelle angepasst. Um verschiedene Systeme zu unterstützen, werden verschiedene Adaptor Objekte realisiert. Die auf dieser Schicht basierende Software wird dadurch portabel und kann auch besser strukturiert werden, da der systemabhängige Programmcode sauber eingekapselt ist. Adaptor Objekte können oft wiederverwendet werden und liegen in Form einer Klassenbibliothek vor.

Programmierumgebung

Der dritte Beitrag dieser Dissertation besteht in einer unter Anwendung des Ausführungsmodells und der flexiblen Systemarchitektur realisierten Programmierumgebung für robuste verteile Systeme, welche *Electra Toolkit* genannt wird. Das Electra Toolkit folgt dem *Common Object Request Broker Architecture (CORBA)* Standard. Es stellt insofern eine Verbesserung heute erhältlicher CORBA Produkte dar, als dass virtuelle Synchronie und Objektgruppen unterstützt werden. Andererseits stellt Electra aber auch eine Erweiterung von Plattformen wie HORUS oder ISIS dar, da eine objektorientierte Programmierung ermöglicht wird.

Electra wurde nicht direkt mit den Kommunikationsprimitiven heutiger Betriebssysteme entwickelt. Stattdessen basiert Electra auf Plattformen wie HORUS oder ISIS, welche Operationen für *reliable multicast* und virtuelle Synchronie anbieten. Adaptor Objekte machen es möglich, dass Electra für diverse Plattformen konfiguriert werden kann.

Erreichtes Ziel

Diese Dissertation zeigt, dass objektorientierte Konzepte die Realisierung von skalierbaren, stabilen und wiederverwendbaren verteilten Systemen

markant erleichtern können. Nach Ansicht des Autors verkörpern das vorgeschlagene Ausführungsmodell, das Architekturmodell und die Programmierumgebung einen vielversprechenden Ansatz für die Entwicklung von weiträumig vernetzten Anwendungen der Informations- und Kommunikationstechnik.

Chapter 1

Introduction

1.1 What is a Distributed System?

Leslie Lamport once jokingly defined a distributed system as "one that stops you from getting any work done when a machine you've never even heard of crashes" [Mul89]. Today, many applications running on loosely-coupled hardware are marketed as "distributed systems". The author prefers to think of a distributed system as a "virtual unicomputer", consisting of workstations, PCs, laptop computers, servers, and peripherals as depicted in Figure 1.1. Sophisticated system software is used to turn the underlying computer network into a distributed system, by providing a single system image. From this point of view, the definition of Tanenbaum and van Renesse [TvR85] is well put:

> *A distributed [operating] system is one that looks to its users like an ordinary centralized [operating] system but runs on multiple, independent CPUs. The key concept here is transparency. In other words, the use of multiple processors should be invisible (transparent) to the user. [...] the user views the system as a "virtual uniprocessor".*

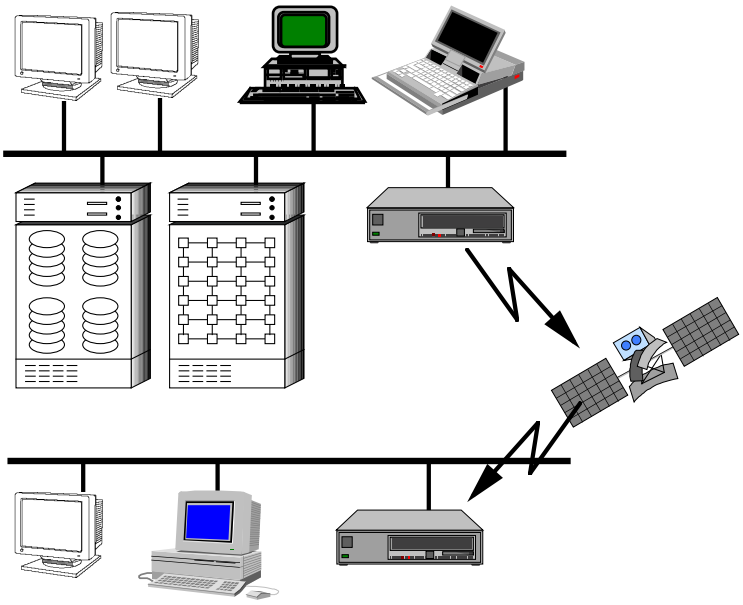

Figure 1.1: Virtual unicomputer: A distributed system may consist of workstations, PCs, mobile computers, CPU servers, disk servers, and LAN bridges.

Sape Mullender [Mul89] completes the definition:

> *Tanenbaum and van Renesse's definition gives a necessary condition for a distributed [operating] system, but I believe it is not a sufficient one. A distributed [operating] system also must not have any single points of failure — no single part failing should bring the whole system down.*

We will refer to the term "distributed system" quite often in this thesis, "single system image" and "no single point of failure" being distinctive features of such a system.

1.2 Why Distributed Systems?

Roughly speaking, there are four major justifications for distributed systems, namely *sharing of resources*, *fault-tolerance*, *cooperative work*, and *performance*.

- **Sharing of Resources.** Many users need to share data. For example, a bank customer may want to withdraw money from her account at different subsidiaries of the bank, or a banking clerk may want to check the balance of all the accounts owned by a multinational company, using a computerized information system. It is clear that shared data and single system image are very important for this kind of banking application. In a similar way, distributed systems permit users in different locations to share expensive devices such as supercomputers or photo-typesetters, leading to a better exploitation of such resources.

- **Fault-Tolerance.** Distributed systems permit the replication of important services on different nodes of the system and to perform important computations redundantly. Thereby, applications can protect the user from hardware or software failures, and the availability of the overall system is increased.

- **Cooperative Work.** Globalization, wide area networks, workstations, and graphical user interfaces influence the way people interact. The combination of these trends and distributed systems leads to what is called the *virtual corporation* model [DM92, KGR94]. A virtual corporation consists of an electronic, organization wide, virtual

workplace in which people interact by multimedia e-mail, bulletin boards, multiuser editors, or computer conferencing. Distributed systems providing single system image and failure resilience form the core of the virtual corporation model.

- **Parallelism.** Due to technological improvements of computing circuits and memory devices, the speed at which computers operate has undergone a rapid and continual improvement during the last years. However, since the rate of increase in single CPU performance is diminishing, distributed computing is gaining attention. Distributed systems allow the aggregation of a vast number of low-cost CPUs into a cost-effective, powerful system, thus offering normal performance at low cost or extremely high performance at moderate cost.

1.3 Intrinsic Problems

Distributed systems also bear inherent problems, such as *software complexity*, *network saturation*, *failures*, and *inconsistencies*.

- **Software Complexity.** The design and implementation of software which provides single system image and fault-tolerance is challenging, but the required mechanisms and policies are not yet adequately understood. It is still being debated which programming model, operating system concept, or toolkit is best-suited for distributed systems.

- **Network Saturation.** Harnessing thousands of workstations into one powerful distributed system is an enticing idea. However, one must bear in mind that communication in distributed systems is achieved by communication paths which provide a low throughput and long delays compared to the links employed in tightly-coupled supercomputers, for instance. Therefore, distributed systems may lead to unacceptable performance when applied on fine-grained parallel applications.

- **Failures.** In distributed systems, failures of system components or communication links are not exceptional occurrences. A failure may be caused by a hardware error, by a software *bug*, or by something as simple as a user switching off his workstation. If a distributed application is not prepared to deal with failures, a single component

failing could stop the whole system. Thus, a programming model for distributed systems should treat failures as a common occurrence, and should permit the realization of systems which can cope with partial failures.

- **Inconsistencies.** A possible way to implement fault-tolerant services is to use multiple servers that fail independently. The state of the service is replicated among these servers. Replication can be used to solve other types of problems as well. For example, in order to cut down on execution time, the same piece of data can be replicated and distributed over a set of processes that subdivide a task.

  Unfortunately, a consistency problem arises when a client updates the state of such a server. A centralized "coordinator" process, to which clients direct their requests, would solve the consistency problem, however, it would present a single point of failure and a potential performance bottleneck. In order to ensure consistency and fault-tolerance, a server can be designated as the *primary* and all others as the *backups* [BMST93]. Clients send requests only to the primary, and when the primary fails, one of the backups takes over. Another approach is to present the client requests in the same order to all replicas, a concept which is called *active replication* [Sch93a].

1.4 Building Reliable Distributed Applications

Contemporary research is busy devising mechanisms to help programmers build *reliable distributed systems*. In the context of this thesis, a reliable distributed system is one that shows predictable behavior, despite processes joining and leaving the system dynamically, despite failures, and despite asynchronous interprocess communication. Moreover, a reliable distributed system allows the replication of critical software objects on different nodes of the system in order to achieve fault-tolerance.

Based on the work carried out in the context of projects such as Isis [BvR94] and in our own experience, we believe that *reliable unicast communication*, *reliable multicast communication*, a *group abstraction*, *consistent ordering of events*, *failure monitoring*, and *Virtual Synchrony* [SS93] are essential for the building of well-functioning distributed systems.

- **Reliable Unicast Communication**: The processes in a distributed system communicate by non-generating, lossless, point-to-point FIFO channels. Unicast communication will be addressed in Section 3.4.

- **Reliable Multicast Communication**: Reliable multicast enables the sending of the same message to a group of processes by one single communication operation. All correct processes deliver the same set of messages, this set includes all messages multicast by correct processes, and no spurious messages [HT93]. Reliable multicast is a prerequisite for realizing replicated and parallel system services, and is important for distributed application development in general.

- **Group Abstraction and Group Management**: The recipients of a multicast form a *group*. The members of a group are automatically notified of group-related events such as processes joining or leaving the group by a group membership service [JFR93, BCG91]. Multicast and groups are treated in Section 3.5.

- **Failure Monitoring**: For the correct functioning of a distributed system, a failure monitoring service [RSB93] is also important. The failure monitoring service collects information on failed (or suspicious) processes, and propagates this information to the operational ones. This kind of service guarantees that processes have a consistent and accurate view on which other processes are operational and which are not. Failure monitoring is addressed in Section 3.6.

- **Ordering of Events**: Reliable multicast imposes no restriction on the order in which messages are delivered to the members of a group, but for many applications a consistent ordering of events is crucial. For example, events in a distributed system can be brought to a total order or to a causal order [HT93], this will be explained in Section 3.8.

- **Virtual Synchrony**: The mentioned services can be aggregated to provide a *virtually synchronous* execution environment [Bir93a, SS93]. In Virtual Synchrony, all significant events, i.e., the delivery of unicasts, multicasts, view-changes, and failures, appear as if each event had occurred at the same *logical time* [Lam78] in all processes. Owing to Virtual Synchrony, the behavior of a distributed system is always predictable, despite failures and dynamic reconfiguration of the system. Moreover, Virtual Synchrony can help in implementing consis-

tency models such as Serializability or Linearizability [HW90, PS93] of events. Virtual Synchrony will be treated in Section 3.9.

A few programming environments for reliable distributed systems already exist. Since they are relevant for the programming model and the toolkit presented in this thesis, we shall give a brief description of CONSUL, DELTA-4, HORUS, ISIS, and TRANSIS. As the hardware base, these environments assume a loosely coupled system, e.g., a workstation network. Basically, the environments offer a procedural programming interface and interprocess communication is by message-passing or RPC.

1.4.1 Consul

CONSUL is a highly modular communication substrate which supports fault-tolerant distributed programs based on a state machine approach [MPS93a]. CONSUL is implemented on the *x*-kernel [PHOH90] and is in development at the Department of Computer Science, University of Arizona. The fundamental services in CONSUL include a *reliable multicast service*, a *membership service* to maintain system-wide consistent group membership information, and a *recovery service* to recover failed processes. The PSYNC [PBS89] protocol, which allows causally ordered and reliable multicast, is at the heart of CONSUL.

1.4.2 Delta-4

The goal of the DELTA-4 ESPRIT-project is to provide fault-tolerance by employing off-the-shelf host computers, standard LAN technology, and a minimum of specialized hardware [PBB+94]. Distributed applications are described and programmed using an object-oriented notation, and extensive use of multicast is made to support replicated, fault-tolerant processing. In contrast to the other toolkits described in this section, DELTA-4 aims at supporting *real-time* applications on *homogeneous* nodes. A synchronized-clocks service and a specialized network attachment controller [Pow94], to guarantee that the network fails silently, are at the heart of the DELTA-4 architecture.

1.4.3 Isis

The ISIS toolkit provides mechanisms such as process groups, reliable multicast, ordering of events, failure monitoring, and thus Virtual Synchrony. ISIS enables the implementation of fault-tolerant software on loosely coupled systems [Bir93a, BvR94]. ISIS is commercially available and marketed by Isis Inc., a wholly owned subsidiary of Stratus Computer, Inc. ISIS was the first environment to provide Virtual Synchrony and high performance. Besides the mentioned low-level fault-tolerance mechanisms, ISIS provides higher-level services like a message spooling facility, a distributed resources manager, a fault-tolerant message publication-subscription system, a reliable network file system, and a CORBA [Dig93] compliant programming environment.

1.4.4 Horus

The development of ISIS began about ten years ago. Based on lessons learned with ISIS, the Computer Science Department at the Cornell University is working on a redesign of ISIS, called the HORUS toolkit [vRB94]. HORUS' architecture is influenced by modern microkernel design principles, in which a small and lean core system provides base functionality such as IPC and memory management, and more complicated services, for example fault detection, are based on the core. To allow the implementation of efficient and scalable applications involving a possibly huge number of groups, a *light-weight group* model has been devised. In this model, *threads* can be grouped and act as recipients for multicasts. HORUS supports several operating systems and many communication protocols, for instance UDP and ATM.

1.4.5 Transis

TRANSIS [ADKM92b] is a communication sub-system for high availability in development at The Hebrew University of Jerusalem, Israel. A TRANSIS system configuration comprises machines that can dynamically crash and networks that might partition and merge. An important difference between TRANSIS and ISIS is that TRANSIS allows *partitionable operation*: if, due to a failure, a process group is partitioned, then each partition continues observing the Virtual Synchrony model separatedly. When the partitions

merge, the system will be *virtually synchronized* [MADK94]. Similar partition handling mechanisms have been implemented in the HORUS toolkit recently [vRHB94]. TRANSIS supports a variety of reliable message-passing services within arbitrary topology networks. The transport layer services provide group membership management, reliable multicast, and various event ordering protocols.

1.5 Object-Oriented Distributed Programming

Object-oriented programming is known to be one of today's best programming models to cope with complex systems while providing maintainability, extensibility, and reusability [Mey88]. A model claiming such attributes is particularly interesting for distributed systems, as these tend to become very complex. The *client-server* model finds increasing consideration for interconnectivity. In this model, servers provide clients with access to services such as file storage or authentication by IPC mechanisms like message-passing or RPC.

Object-oriented, *distributed* programming (OODP) is a generalization of the client-server model, in that objects encapsulate an internal state and make it accessible through a well-defined *interface*. The interface definition in Example 1 specifies a file server object with operations to read and write a fixed-size block of a file. Client applications may *import* the interface, *bind* to a remote instance of the interface, and issue remote object invocations (Figure 1.2). This use of objects naturally accommodates heterogeneity and autonomy: heterogeneity since messages sent to objects depend only on their interface and not on their internals, autonomy because object implementations can change transparently, provided they maintain their interfaces [NWM93].

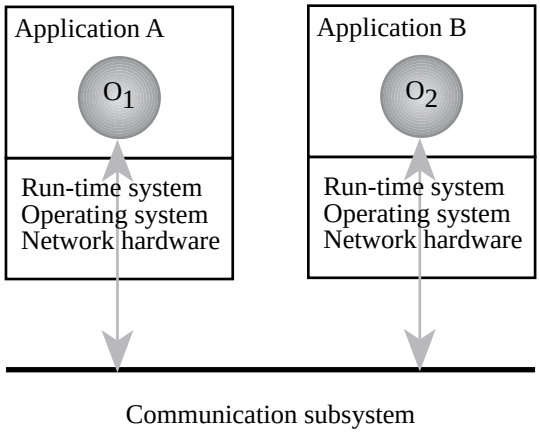

Figure 1.2: Remote object invocation. O_i denotes a network object.

Example 1:

```
interface FileServer {
   const long BLOCK_SIZE = 8192;
   typedef sequence<octet,BLOCK_SIZE> Buffer;

   long read(in string fileName, in long pos, out Buffer b);
   long write(in string fileName, in long pos, in Buffer b);
};
```

Object-oriented programming in its purest sense is defined as "*programming implemented by sending messages to objects*" [PW88], an object being an instance of a class, and a class an implementation of an abstract data type. Steven Jobs was once heard to say: "an object is a living, breathing blob of intelligence that knows how to act in a given situation".

In pure object-oriented programming, in Smalltalk for example, the instance data of an object can be accessed only through the object's methods. This approach is well-suited for distributed systems, since complete encapsulation is ensured and since method calls are a convenient place to insert the communication required by distributed systems [BNOW93]. To be considered object-oriented, a programming language must provide *abstraction*, *encapsulation*, *inheritance*, and *polymorphism* [PW88]. In distributed pro-

gramming, each attribute has interesting implications.

1.5.1 Abstraction

Abstraction in the most general sense is the process of identifying the characteristics that distinguish a collection of similar objects. It is the concise representation of a more complicated idea. Abstraction enables a higher level of human functioning than when one is confronted with many details.

A fault-tolerant file server, for example, can be modeled by a group of cooperating `FileServer` objects (Example 1) and by reliable, totally ordered object-group communication. Thus, the internal structure of the file server object or the realization of group communication is abstract.

1.5.2 Encapsulation

Encapsulation means that an object contains specific internal data, which is accessible only through a set of well-defined methods. Encapsulation is the language mechanism by which external aspects, for example the interface of an object, are separated from the implementation of the object.

In heterogeneous distributed systems, encapsulation permits to devise several implementations of an interface, one per different hardware architecture or operating system, for instance. This permits the tailoring and optimization of an implementation for a given environment, whereas the client applications are confronted with one interface declaration, no matter which specific implementation provides the service. Furthermore, clients do not notice that a new version of an object has been installed, as long as the new version supports the old interface.

1.5.3 Inheritance

Inheritance is the mechanism by which an object interface can include the attributes and methods defined in another, more general interface. It is thus possible to reuse the code of an existing interface implementation. The resulting implementation can then be tailored by method overwriting.

Since distributed systems tend to become large and complex, inheritance is valuable since it allows to structure them in the form of an interface hierarchy. Refined components for distributed applications can be derived from

more general ones. In addition, inheritance can be used to alleviate the revision control problem. For example, the SPRING operating system [HK93] uses interface inheritance to address problems in system software evolution [HR94]. When an operating system service must be modified a new SPRING interface is created. For minor revisions, for instance when adding an operation to a service, the new interface inherits directly from the old one. For major revisions, for example when an argument is added to an operation of a service, the new interface directly inherits from the topmost base interface of the service (Figure 1.3).

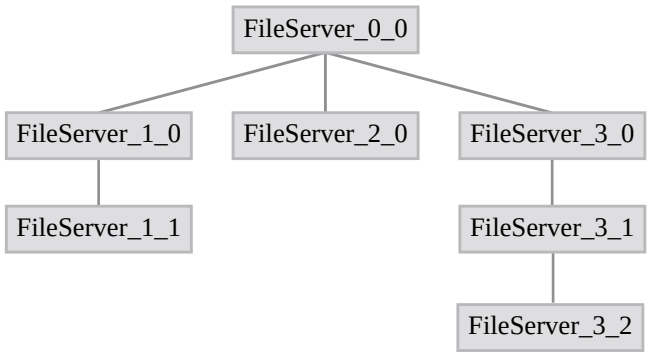

Figure 1.3: A SPRING revision hierarchy.

1.5.4 Polymorphism

Polymorphism is a request handling mechanism that selects a method based on the type of the target object. A request can thus be sent to a set of objects and each object will react according to its type. For example, the message `drawYourself` can be sent to objects of type `circle`, `triangle`, and `line`, whereby each object will perform the type-specific drawing operations. The ability to use the same message for a similar operation on different kinds of objects is consistent with the way humans think about solving problems [PW88].

Also this attribute has interesting implications for distributed systems. Using *mixin inheritance* [Sem93][1] functionality can be added to a set of

[1] in *mixin inheritance* a superclass is used to specify a common functionality or in-

interfaces as depicted in Figure 1.4. Here, the interface `ManagedObject` is used to add system management and monitoring operations to services of a distributed system. For instance, the `ManagedObject` interface can define operations such as `resetState`, `suspendOperation`, and `resumeOperation`, plus local attributes such as `numberOfErrors` and `numberOfRequests`. Objects will react in a type-specific manner to the management operations invoked.

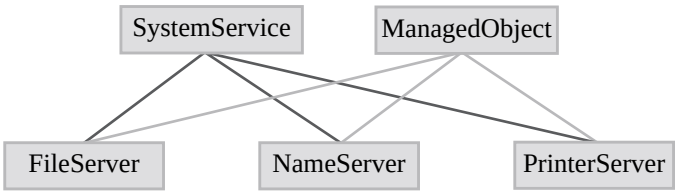

Figure 1.4: Mixin inheritance and polymorphism.

1.6 Why OODP?

Having described the main attributes of OODP, the question arises what the advantages of OODP are as compared to conventional distributed programming by message-passing or RPC. This section addresses the benefits of designing and implementing a distributed system in an object-oriented fashion.

1.6.1 Service Guarantee

In a distributed system, an object encapsulates an abstraction which is meaningful to its clients. The object is more than just instance data, it provides a *service* to its clients [Her94]. As mentioned before, a distinctive feature of OODP is that the service guaranteed by an object is clearly separated from the technology implementing the service. Benefits:

- **Heterogeneity**: Objects are effective means for encapsulating the peculiarities of the underlying hardware or system software, in that

terface for a number of subclasses. This functionality is then *mixed* into the subclasses using multiple inheritance.

the exported interface is independent of the implementation of the object, and in that an interface can have several implementations.

- **System Evolution**: Systems structured in terms of objects evolve easily by parts being replaced [Kra93]. The programming language of an object, the underlying system software or hardware can be changed as long as the object supports its old interface. This leads to systems which are more resilient to changes, and it reduces development risks.

1.6.2 Component Based Development

OODP permits to view systems *outside-in* from the standpoint of a system designer. By examining the interfaces of the components of a distributed system, an engineer will gain an understanding of the functioning of the system quickly without bothering about internal details. Benefits:

- **Composition**: Component based development is an expressive principle by which reusable software objects are combined into complex systems [Weg93]. Smaller components can be aggregated to larger ones and so forth, similar to integrated circuit design.
- **Management**: The maintenance of large system components is simplified and can be performed by different communities, using their own terminology and goals [WWC92].

1.6.3 Wrapping Legacy Applications

Global banks and other multinational companies own expensive, mission-critical systems, frequently written in archaic programming languages like COBOL or Fortran. Economic restrictions and the fact that mission-critical systems must be available all the time keep the organizations from moving to newer technology. Software that already exists and has proven value for the organization is often called *legacy software* [Bro92, CI94, Zuc94]. Object technology permits to interface state-of-the-art distributed technology to legacy information systems:

- **Wrapper Objects**: Often, an interface and a wrapper object can be devised to an existing, geriatric system component, after which the component can be accessed from within an object-oriented distributed application.

- **Relevance**: Legacy systems can become major liabilities to large organizations, and OODP can help in migrating to new technology.

1.6.4 Location Independence

Objects in a distributed system are normally assigned unique, location-independent names and clients may thus access objects without knowing about their location in the network. Benefits:

- **Flexibility**: The location of an object can be changed as long as the object maintains its original name.
- **Consistent Invocation Scheme**: Remote object invocations always take the same form independent of the location of an object and independent of the underlying communication software.

1.6.5 Scalability and Fault-Tolerance

Overloaded, singleton objects can be replaced transparently by so-called *object-groups* (to be addressed in Section 3.5), which parallelize the singleton's task. This provided that the object-group is accessible by the same interface as the singleton. Benefits:

- **Scalability** is thus facilitated without having to modify client applications which used the singleton object.
- **Replication**: An object-group may also be used to manage replicas of an object. Safety-critical objects can thus be replaced transparently by object-groups to increase fault-tolerance.

1.6.6 Performance

By separating an interface from its implementation, programmers can devise several implementations for the same object, and tailor them to specific target systems. Benefits:

- **Efficiency**: The specific characteristics of the underlying hardware (high performance I/O interfaces, co-processors, accelerator boards, etc.) can be exploited.

- **Tuning**: Distributed applications can be fine-tuned by maintaining a well-structured system design.

1.6.7 Open Distributed Systems

An open system is one that can cooperate with other systems by using standards that govern the interprocess communication and such [Tan92]. Open systems avoid excessive dependence on a single manufacturer, on a certain operating system, hardware, or programming language, and thus offer customers a maximum freedom of choice. Development risks are reduced, and enterprises can choose the components of a distributed system according to price-performance criteria. OODP is well-suited for the building of open systems:

- **Object interfaces** are publicly accessible for interaction, for modification, and for reuse. Distributed systems can thus be constructed by interconnecting software components developed by a wide variety of vendors as well as in-house developers.
- **Open standards** for OODP, such as OMG CORBA [Dig93, Vin93] or the ISO/CCITT object model [Ash93], appeared recently and a growing number of organizations have agreed to comply with them.

1.6.8 Object-Oriented Programming

It is known that object-oriented programming encourages the development of maintainable and reusable software [Boo91]. The following advantages ascribed to the non-distributed object model hold for OODP as well:

- Expressive power of the object model
- Software reuse
- Enables programming-by-difference
- Helps programmers in coping with complexity, by the systematic use of abstraction, encapsulation, inheritance, and polymorphism
- Support for incremental solutions to complex problems
- Reduced development costs, development time, and development risks

- Extensions can be made to a class while leaving the original code intact
- Owing to code reuse and enhanced expressivity, source code size is often reduced [Boo91, Sch86]

1.7 Risks of OODP

Nevertheless, OODP is not a panacea. Performance loss and acquaintance costs, for instance, are two of the major problems.

1.7.1 Performance Loss

There are performance risks inherent in OODP and those which are ascribed to object-oriented programming in general. The main performance risk of OODP is that even though the programmer has the impression of accessing a local object, (a local representative of the remote object with the same interface as the remote object), method invocations take a long time compared to purely local invocations. This happens because the call induces the sending of a request and of a reply packet over the network. Of course, any programming model for distributed systems is subject to this problem. In comparison with low-level message-passing using `send()` and `receive()` operations, communication in OODP is more costly since operation parameters need be marshaled and unmarshaled, and the call must be dispatched by the receiver. Furthermore, a message header identifying the target object and the target method must be included. Nevertheless, when programming a complex distributed application using low-level message-passing primitives, similar administrative code would have to be provided by the programmer, whereas in OODP this code is most likely to be generated automatically.

General performance risks of object-oriented programming are that intensive dynamic allocation and destruction of objects takes place at runtime, that dynamic method invocations take more time than static ones, and that good object-oriented programming practice may end up in a large amount of method invocations. Fortunately, compilers for object-oriented languages are continuously being optimized [Höl94]. Moreover, *inlining* of frequently used methods and memory pool-allocators may speed up object-oriented applications considerably.

1.7.2 Acquaintance Costs

As with conventional object-oriented programming, the acquaintance costs of learning OODP might be high. Such costs comprise the training of employees at object-oriented design and -programming, purchasing software development tools, devising corporate programming guidelines, reimplementing old libraries and applications, and so forth. As a rule of thumb we can say that newcomers will face several months of training and frustration until they start profiting from the OODP paradigm and become more productive than they were before learning OODP. This development is sketched in Figure 1.5. It is important to note that OODP does not just mean the learning of a new programming language, but the adoption of a new way of solving problems.

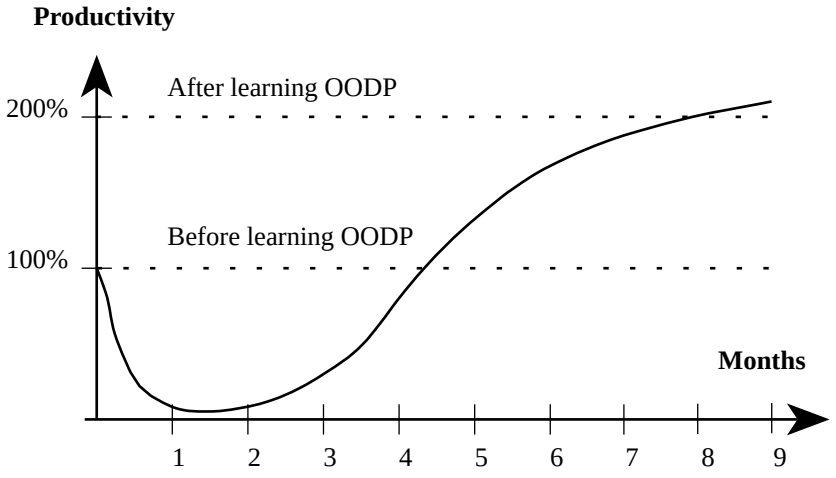

Figure 1.5: Sample OODP learning-curve.

1.8 Summary

"Single system image" and "no single point of failure" are two important requirements for distributed systems. We believe that tools such as *reliable unicast communication*, *reliable multicast communication*, a *group abstraction*, consistent *ordering of events*, *failure monitoring*, and thus *Virtual*

Synchrony are essential for the development of robust distributed systems. Programming environments providing this sort of system support are, for example, CONSUL, DELTA-4, HORUS, ISIS, and TRANSIS.

Today, the use of object-oriented programming is advocated because of its ability to cope with complex systems, while providing maintainability, extensibility, and reusability. From this point of view, object-oriented *distributed* programming (OODP) is gaining attention both in academic institutions and in the industry, a fact which is demonstrated by the recent boost in related publications and by the ongoing standardization efforts. Advantages of OODP are that heterogeneity is advocated, that object-oriented systems evolve gracefully, that the maintenance of large applications is simplified, that software can be reused, that open distributed programming is promoted, and that flexibility and scalability are enhanced. Potential disadvantages are computational overhead and acquaintance costs. In this thesis we will apply the object-oriented model to reliable distributed systems.

Chapter 2

Contributions and Related Work

2.1 Contributions

This thesis presents the ELECTRA *Object Model* for object-oriented programming of reliable distributed systems. Based on this model, it then proceeds to discuss the design and implementation of the ELECTRA toolkit, a novel CORBA programming environment. This toolkit is unique in that it permits object-oriented programming of reliable distributed software through a CORBA-compliant interface, and by exploiting the powerful primitive operations provided by communication subsystems such as CONSUL, DELTA-4, HORUS, ISIS, and TRANSIS. ELECTRA can be configured for various communication subsystems, which is made possible by its flexible system design. ELECTRA is a running prototype system developed as part of this dissertation. At the time of writing, the ELECTRA toolkit supported the MUTS platform [The94b], the HORUS as well as the ISIS system. So this thesis makes three important contributions:

- **The Electra Object Model**: The ELECTRA Object Model (EOM) presented in Chapter 3, is an execution model for object-oriented distributed applications. It is based on several theories and system services stemming from contemporary distributed systems research. It provides asynchronous one-to-many communication and virtually

synchronous execution making it ideal for programming failure-resilient, efficient, and robust applications from reusable software components. Groups of objects which can be dealt with in a uniform and nearly transparent way are the key component of EOM. Our own contribution consists in identifying the important components of a programming model for robust object-oriented distributed systems, pointing out design alternatives, and collecting those components into a well-rounded execution model.

- **The Adaptor Model**: The second contribution consists of a software architecture which allows for design and implementation of flexible and portable system software. Flexible system support will be discussed in Chapter 4.

- **The Electra Toolkit**: In Chapter 5 we will describe the implementation of a toolkit which is based on EOM, adheres to the Object Management Group's CORBA standard, and which is structured according to the aforementioned system design principles. Here, our contribution consists in demonstrating that an efficient and flexible implementation of EOM can be devised in about one man-year, provided that a state-of-the-art technology such as HORUS or ISIS lies at the core of the implementation.

Concepts and mechanisms described in Chapter 3, 4, and 5 make up the *run-time system* of an OODP environment, and could be provided by a future operating system. Based on this sophisticated system support, Chapter 6 describes application frameworks for distributed systems. Finally, Chapter 7 summarizes the main findings and concludes the thesis.

2.2 Related Work

While this thesis is motivated by fault-tolerance and multicast aspects, the ELECTRA object model is influenced by systems such as ARGUS [Lis88], ARJUNA [SDP], AVALON C++ [EMS91], CLOUDS [DLA88], COMANDOS [CBHRdP93], COOL [J+94], EDEN [ABLN85], EMERALD [BHJ+87], and SOS [S+89]. The main difference between EOM and other distributed programming models is that EOM supports asynchronous group communication and Virtual Synchrony. In this section we describe projects which are related to our work.

2.2.1 ANSA Interface-Groups

The ANSA (Advanced Network Systems Architecture) team already recognized the need for object-group invocations several years ago, and so the ANSA *interface-group* model [CSB92] was developed. An ANSA interface-group can be seen as an object with two interfaces. The first interface is the one by which service operations are multicast to the members of a group. The second interface is the *group control interface* by which the behavior of a group is managed. It provides operations for joining and leaving a group, operations to lock and to unlock a group, operations for configuring a group, and operations which allow clients to register an interest in join- and leave-events.

Two group invocation styles are supported, a transparent and a non-transparent one. In transparent invocation style, clients need not be aware that a group-invocation takes place, and replies are automatically collated to a single termination. In non-transparent invocation style, programmers obtain several results from an invocation, and the result arguments are specified as ANSA `SEQUENCE OF` data types.

Programmers can select between a FIFO and a total ordering of the multicasts delivered to the members of a group. The current implementation scheme centralizes the group state into a single member object. The adopted group communication protocol is similar to the one in AMOEBA, where all group communication is channeled through a central member object, which might become a performance bottleneck or a single point of failure. Moreover, multicast is realized by the repeated use of single unicast messages, not by exploiting the multicast capabilities of the underlying network hardware or system software.

Although the ANSA interface-group model and the ELECTRA model share several aims and features, our work differs from the ANSA interface-group model in that we map group operations on low-level hardware or software multicast where available. Further, EOM is not based on centralized group information and is independent of the underlying transport platform. Finally, the implementation of EOM described in Chapter 5 follows the CORBA specification, whereas ANSA tries to establish its own standard.

2.2.2 Arjuna, Avalon

Most of today's OODP environments focus on transactional mechanisms. Two examples are ARJUNA [SDP] and AVALON C++ [EMS91]. EOM is influenced by these environments, although they are mainly concerned with programs that access long-lived shared data. The transactional model guarantees that the stored data is always consistent with some logical model of the data in spite of failures. In the case of a failure, transactions are aborted and can be restarted only when the defect has been repaired, e.g., a damaged disk has been replaced.

Many applications cannot tolerate such delay. In this case, *active replication* and asynchronous multicast communication can lead to substantially increased availability and performance. On the other hand, active replication requires reliable multicast, object-groups, and consistent ordering of the invocations delivered to the objects which make up a distributed application. Our work is mainly concerned with such mechanisms and with applications requiring continuous processing. Although transactional mechanisms could indeed be added to EOM, we do not deal with them in this thesis.

2.2.3 COMANDOS

The COMANDOS (Construction and Management of Distributed Operational Systems) project aims at defining and implementing an integrated platform supporting transparent distribution and manipulation of long-lived, persistent data [CBHRdP93]. Objects are passive entities, they are accessed by processes in a uniform manner, regardless of whether objects are local or remote, persistent or volatile. Thus, the key features of the COMANDOS model are transparent distribution, uniform treatment of volatile and persistent objects, consistent sharing of data, and intrinsic fault-tolerance mechanisms. COMANDOS mainly focuses on cooperative applications running within one LAN.

In EOM, objects can have an active or a passive role. The ELECTRA approach does *not* enforce full transparency so that programmers can decide whether an object is remote-accessible or not. We further aim at supporting scalable applications which may spawn several LANs and at minimizing the cost of remote operations. Our fault-tolerance mechanism, mainly object-groups, are not intrinsic although in many situations programmers can

treat and access object-groups as if they were single entities, providing a high degree of transparency in accessing remote objects. EOM does not address persistency.

2.2.4 Flame

FLAME [DPJ90] is an experimental language for distributed programming based on the C++ programming language [ES91]. Similar to EOM, the two main features of FLAME are object-orientation and object-groups, and any FLAME object can invoke remote operations or reply to incoming requests. FLAME uses the ISIS toolkit as run-time system. The current implementation of FLAME employs coarse-grained objects, which means that a FLAME object encapsulates a whole ISIS process.

The main difference between FLAME and ELECTRA is that the former approach consists in developing a new programming language for distributed systems, by taking the C++ programming language as a starting point. Special expressions for declaring processes and process-groups, and for binding to remote objects were added to C++. In contrast, ELECTRA leaves the target language, which is not restricted to C++, untouched and uses a separate interface definition language for defining object interfaces. In ELECTRA, it is thus possible for objects implemented in different target languages to interact. Furthermore, ELECTRA supports fine-grained objects, which means that a process may contain many ELECTRA objects.

2.2.5 GARF

GARF (Génération Automatique d'Applications Résistantes aux Fautes) is an object-oriented system supporting the design and implementation of reliable distributed systems [GGM94]. The present version of GARF is based on ISIS and implemented in Smalltalk-80. GARF supports an incremental programming methodology, where an application is first developed in a centralized environment, and then distributed over a network. The second step does not require a modification of the centralized application. To achieve fault-tolerance, critical objects can be replicated.

Another distinction is that behavioral features which concern concurrency, distribution, and fault-tolerance are clearly separated from functional ones. To achieve this, GARF objects do not interact directly (Figure 2.1). Object invocations issued by a client C are channeled through the client's

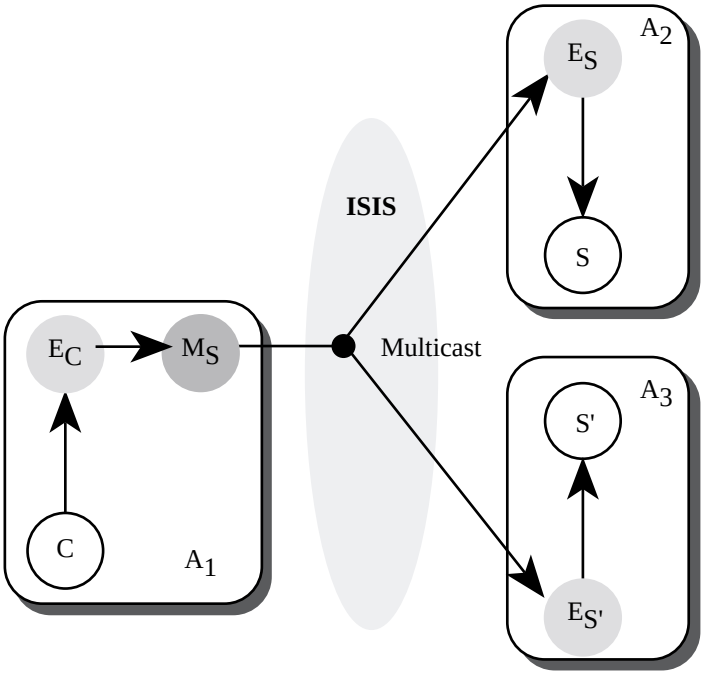

Figure 2.1: GARF execution model.

encapsulator object (E_C) residing in the client's address space A_1. The task of the encapsulator is to wrap the client object and to control its communication. GARF provides several kinds of encapsulators, namely such for serializing invocations, for communicating with replicated server objects, and so forth. Remote communication is performed by a *mailer* object (M_S), also residing in the client's address space, and supporting various forms of multicast and unicast communication. Finally, a remote invocation is received by the server encapsulators (E_S and $E_{S'}$) and dispatched to the associated server objects.

Although both ELECTRA and GARF aim at supporting reliable distributed systems, the systems provide complementary functionality. ELECTRA lacks such clear distinction between behavioral and functional aspects, while language independence, platform independence, and CORBA compliance are not addressed in the current version of GARF. Combining both

systems would surely lead to a powerful and flexible programming environment.

2.2.6 ISIS/RDO

ISIS/RDO (ISIS Reliable Distributed Objects) is a commercially available, CORBA-compliant programming environment which supports object-groups. It is based on the ISIS toolkit and is similar to ELECTRA. Modules for realizing replicated servers, master-slave computations, group notification, and group management [Isi93a] are part of the environment. The object interfaces of a distributed application are specified in CORBA-IDL, they can be implemented in Smalltalk, Objective-C, C, C++, or any combination of these. ISIS/RDO also supports most of the EOM requirements we shall stipulate in Chapter 3.

Our work differs from ISIS/RDO mainly in that we employ a flexible and modular system design (Chapter 4), and in that ELECTRA can run on several communication platforms, including ISIS. Moreover, in ELECTRA objects residing in the same process can join groups individually and be members of the same group (light-weight group model).

2.2.7 OMG CORBA

CORBA [Dig93] (Common Object Request Broker Architecture and Specification) is a vendor-independent standard which aims at interoperability and portability of distributed, object-oriented applications. The standard is being developed by the Object Management Group (OMG)[1]. Although the ELECTRA *Object Model* is independent of any existing standard, CORBA is important for the ELECTRA *toolkit* described in Chapter 5. The main components of CORBA are:

- **IDL**: Before an application can access a remote object, it must know what services the object exports. In CORBA, object interfaces are described in CORBA-IDL (Interface Definition Language), a purely declarative language resembling C++. CORBA-IDL provides basic data types (such as `short`, `float`, `octet`, and `char`), constructed

[1] Presently, more than 400 enterprises and academic institutions all over the world are members of the OMG.

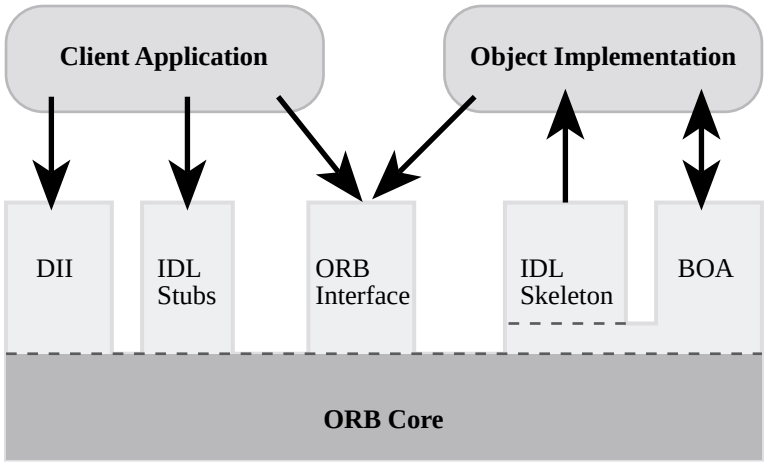

Figure 2.2: The architecture of a CORBA object request broker.

types (such as `struct`, discriminated `union`, and `enum`), and template types (such as `sequence` and `string`). Furthermore, object references can be sent over the network and used by the receiver to access the associated object. The fact that CORBA-IDL is purely declarative makes for a clear distinction between interfaces and implementations. The main purpose of the IDL compiler is to produce the communication stubs by which applications issue remote object invocations.

- **ORB**: The purpose of the ORB (Object Request Broker) is to find the object implementation for a request, to prepare the implementation to receive the request, to transmit the request from the client to the object implementation, and to return output arguments back to the client. Owing to the ORB, the object interface a client application sees is independent of the location of the object implementation and of the target programming language.
- **BOA**: The BOA (Basic Object Adapter) is the primary interface an object implementation uses to access ORB functionality. The BOA exports operations to create object references, register and activate object implementations, and authenticate requests.

- **Dynamic Invocation Interface**: Sometimes, programmers need to access object implementations whose interface stubs do not exist at the time the programming takes place. The Dynamic Invocation Interface (DII) defines functions for creating request messages and for delivering them in a message-passing communication style. To construct a request at run-time, a client application obtains a data structure describing the interface of the target object from an interface repository or from another information source. A client who uses the DII to send a request to an object obtains the same semantics as a client using the interface stub generated by the IDL compiler.

- **The Repositories**: CORBA also specifies an interface and an implementation repository. The interface repository contains information on the object interfaces available in the system. This information can be used for dynamic invocation, for revision control, for run-time object inspection, and so forth. The implementation repository provides information which allows the ORB to locate and activate implementations of an object. This information mainly comprises the executable code of an object implementation, and implementations are activated by means of process creation or dynamic object-code loading.

2.3 Summary

This thesis focuses on run-time support for object-oriented distributed programming of reliable distributed systems. From this point of view our contribution is threefold. First, the ELECTRA *Object Model* deals with services and protocols which permit the implementation of well-functioning and robust distributed applications. Second, a modular and flexible run-time system design is devised. The third contribution is the description of the design and implementation of a novel toolkit for object-oriented distributed programming which follows the ELECTRA Object Model, the CORBA specification, as well as the mentioned run-time system design.

The ELECTRA Object Model is influenced by systems like ARGUS, ARJUNA, AVALON C++, CLOUDS, COMANDOS, COOL, EDEN, EMERALD, and SOS. Systems such as ANSA Interface-Groups, FLAME, GARF, and ISIS/RDO influenced the way ELECTRA supports object-groups and group invocations.

Chapter 3

The Electra Object Model

Programming distributed systems is difficult. In Section 1.4 we presented some development tools for dependable distributed applications. On the one hand, we saw that most of today's state-of-the-art toolkits for reliable distributed systems, for example CONSUL, DELTA-4, HORUS, ISIS, and TRANSIS, provide a rather low-level, procedural application programmer interface. On the other hand, we believe that the powerful mechanisms included in such toolkits should be the basis of higher-level programming environments for well-functioning distributed systems. This thesis contributes the abstraction mechanisms and metaphors to hide the low-level interfaces of such toolkits, whereas this chapter presents *The Electra Object Model (EOM)* as a first step in that direction. EOM is the abstract representation of the basic mechanisms we believe necessary for constructing reliable, distributed applications in asynchronous environments. Furthermore, EOM aims at supporting object-oriented, distributed programming, which allows programmers to compose high quality distributed applications out of reusable software components.

In EOM, all first-class entities are *objects*, an object being an instance of a class, and a class an implementation of an abstract data type. EOM objects can be accessed only by using their methods, complete encapsulation is thus ensured. Distributed applications are structured as collections of objects that cooperate by remote method invocation to solve certain

problems. The overall effect is the illusion of a *virtually synchronous* program execution in a *virtual object space*. The virtual object space might encompass a huge number of computers and networks.

EOM itself is independent of any underlying operating system, hardware platform, or programming language. It comprises a *minimal set of requirements* which a toolkit for object-oriented programming of reliable, distributed systems should fulfill. More sophisticated models, for instance those providing full distribution transparence, persistence, atomic transactions, or those fitting a particular application area can be based on EOM.

Next, we describe the system model EOM is based on. Then, a survey of the components of EOM will be made and each individual component will be explained.

3.1 Underlying System Model

In this thesis we consider *asynchronous distributed systems* [BM93] consisting of processes with disjoint memory spaces, which run on a collection of machines, and interact by message-passing. In asynchronous systems, there are no constraints on the speed at which processes make progress and on the message transmission delays. Furthermore, neither an exact synchronization of the local clocks [Cri89] nor a reasoning based on "global time" [Lam78] is possible. In this situation, interprocess communication remains the only feasible means of synchronization.

We assume that failures respect the *fail-stop* model [SS83], which means that processes fail by crashing without the emission of spurious messages. This means that we do not consider *Byzantine failure modes* [LSP82]. The handling of Byzantine failures requires complicated techniques and protocols which are beyond the scope of this work and in fact beyond the scope of many real-world systems.

We assume that the message-passing system fails by omitting or delaying messages for a long period of time. No assumption is made about the network topology or about the protocols making up the interprocess communication service. However, we assume that each process can send messages to any other process by a lossless and non-generating FIFO channel.

The advantages of the asynchronous system model are that it is simple to describe and realistic in the sense that many real-world distributed sys-

tems fit this model. In real-world distributed systems, the message delay depends on many random factors such as the actual load situation, the unpredictable delay occurring in the software layers which implement reliable communication and such. Basically, an application implemented for an asynchronous distributed system is more portable than an application written for a synchronous one, in that the former kind of application is not based on assumptions like synchronized clocks or maximum message delays.

The drawback of asynchronous systems is that failures here are difficult to handle. In [FLP85], Fischer *et al.* demonstrate the impossibility to distinguish a crashed process from one that is very slow, and the infeasibility to reach a consensus in asynchronous systems with faulty processes. To cope with this problem, Section 3.6 presents a system service which permits the detection of failures in asynchronous distributed systems and to solve the consensus problem.

3.2 Components

In EOM, objects may reside in different address spaces, and communication is mainly by remote object invocation, called *Remote Method Calling (RMC)* in this context. A local object issues an RMC to a remote object by invoking the methods of a local representative, a so-called *proxy* object [Sha86]. The proxy marshals the RMC's parameters into a message structure and submits the message using a low-level communication primitive. On the opposite side, the receiver's run-time system unmarshals the message and determines the object it is addressed to. Finally, it invokes the appropriate method of the target object. If required by the operation's signature, a result message will be returned to the sender with a similar procedure.

Interaction with remote objects is carried out in a uniform manner, independent of their physical location, and often without the knowledge of it. Nevertheless, full transparency is not enforced by our model, which means that the programmer can find out whether an operation is carried out locally, on another host in the same LAN, or across a WAN connection. We assume that programmers wish to take advantage of their knowledge of the network structure in order to achieve maximum performance and availability, which leads to a programming model in which the cost of an

operation is predictable.

The main differences between EOM and other object models, for instance the ANSA [Her94], COMANDOS [CBHRdP93], ARJUNA [SDP], or the AVALON C++ [EMS91] one, are that EOM aims at specifying a minimal set of requirements for object-oriented programming of dependable systems. Furthermore, group-communication (Section 3.5) and Virtual Synchrony (Section 3.9) are fundamental in our model.

Many of today's object-oriented distributed programming environments are built around transaction mechanisms. In contrast, EOM aims at assisting applications that need object replication, efficient one-to-many communication, and overlapping computation and communication phases. Often, such applications cannot be accomodated in programming environments based on traditional transaction mechanisms. Thus, the key benefit of EOM is that it allows for efficient non-blocking communication in an object-oriented environment, while guaranteeing a consistent ordering of distributed events.

The following elements constitute EOM: objects and interfaces, remote method calling, object-groups, a failure detection service, a group membership service, ordering of events, and virtually synchronous execution (Figure 3.1). The following sections introduce these components and describe various design alternatives with their advantages and disadvantages.

3.3 Objects and Interfaces

3.3.1 Active and Passive Objects

EOM distinguishes between active and passive objects. Active objects can buffer and process incoming RMCs by creating a new thread of control to execute the method. An active ELECTRA object thus behaves in a way similar to an *actor object* [Agh86, KL91]. Multithreading allows for a maximization of throughput by overlapping computation with communication and for concurrency inside an object. This comes at the price that programmers have to synchronize accesses to the instance data of an object. *Object-references* are used to identify and access active objects in a uniform manner. Object-references can be seen as virtual addresses which are valid on any node, and which can be transmitted across node boundaries.

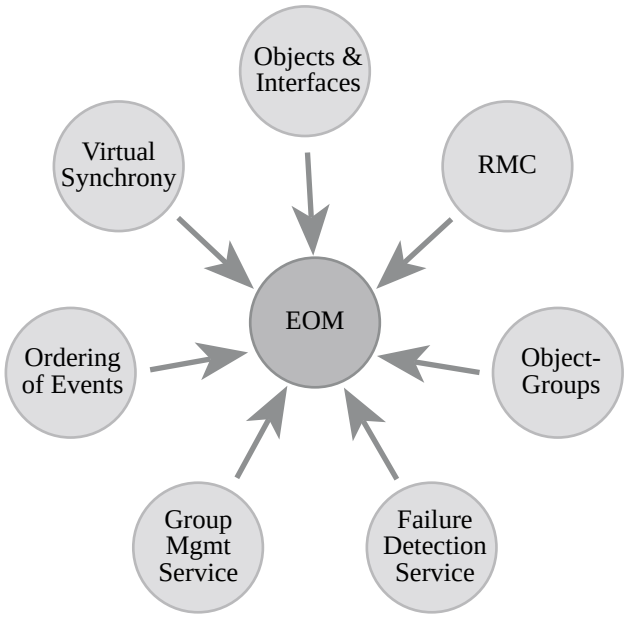

Figure 3.1: The elements of the Electra Object Model.

Passive objects, on the other hand, act as data-holders and are mainly used by active objects to exchange information. Thus, passive objects do not buffer messages. They provide two special methods, one for marshaling and one for unmarshaling their state. These methods can either be manually written by the programmer, they can rely on available run-time type information, or be generated by an IDL compiler as is shown in Section 5.6.

Since objects invoke operations on each other in an unrestricted fashion, it is not useful to designate some objects as clients or servers. In EOM, client or server are roles that components play only at particular moments, and objects are not tied to a client or server role. Thus, an object acting as server at a certain moment can become a client at a later point. Differentiating between passive and active objects has both advantages and disadvantages:

Advantages

- **Efficiency**: By explicitly declaring passive objects as such a space- and computation-efficient implementation can be devised for them.
- **Simplicity**: It is easier to implement a programming system where the programmer gives a hint about remote-accessible objects than to implement a uniform object model.
- **Adequacy**: Thinking about active entities which interchange passive entities is a natural abstraction, which is adopted in many distributed system projects [BNOW93, DSC92, HHBC92, Her94, Dig93].

Disadvantages

- **Passivation**: Strict adherence to the scheme would prevent active objects from migrating from one address space to another, since per definition only passive objects can be interchanged between different address spaces. An implementation of EOM would thus have to provide operations which permit an active object to become passive (by "freezing" activities inside the object) and *vice versa*.
- **Classification**: An object cannot always easily be classified passive or active. For example, imagine a `Mailbox` object which serves to store message for a specific user of an electronic mail system,

and which also provides methods for sending and receiving electronic mails. On the one side, the `Mailbox` object has to be passive since it should be possible to forward a complete mailbox from one user to another. On the other side, it has to be active since it provides methods for receiving messages. In this case, the problem can easily be solved by introducing a new object, called `MailServer` for instance, which is an active entity, and by redefining the `Mailbox` object to be a passive entity.

- **Lack of Transparency**: One could argue that a uniform model consisting of objects scattered over many address spaces would lead to a simpler and more elegant programming model. Further, such a uniform model could be extended to accomodate persistence and fully transparent communication [CBHRdP93, CFH+93]. However, it is yet not clear whether a transparent and uniform model is better suited to implement large-scale distributed systems.

Requirement 1:

The ELECTRA Object Model consists of active and passive objects. Active objects communicate by interchanging passive objects. Passive objects provide methods for marshaling their state into a sequence of bytes, and for unmarshaling their state out of a sequence of bytes.
Active objects are not tied to a client or server role: any active object can invoke operations on other active objects and handle incoming requests. EOM objects are not fragmented, but completely contained in one address space. The granularity of EOM objects varies between very small and very large. Object-references can be transmitted across node boundaries, allowing applications to operate on distributed objects in a uniform way.

Nice-to-have:

Operations to convert active objects into passive ones and *vice versa* may be provided.

3.3.2 Interface Declarations

Unless otherwise stated, the term *object* denotes an active entity for the rest of the work. Objects are specified in an *Interface Definition Language*

(IDL). An IDL is a purely declarative programming language with the main purpose of describing the operations an object performs. Interface declarations serve as formal contract between objects that use a service and objects that provide a service.

The IDL is independent from the *target language*, which is the programming language in which the behavior of an object is implemented. To support EOM, a target language is not necessarily object-oriented, although the key abstractions of EOM are easier to deal with using an object-oriented target language. The basic features that an IDL must provide to be compliant with EOM are:

- Interface declarations
- Support for common basic data types (integer, floats, strings, etc.)
- Support for constructed data types (structures, unions, etc.)
- Support for user-defined, passive objects
- Interface inheritance

CORBA-IDL [Dig93], for example, fulfills most of these requirements and is a good starting point for an implementation of EOM.

Specification and Validation

An IDL specification describes the operations which can be issued on an object along with the type of their arguments. In addition, it would be useful if the IDL allowed to specify legal sequences of operation invocations, so-called *legal behaviors* [Wei93] of the object. An implementation would then be *correct* if every behavior that it currently produces is in the set of legal behaviors. There are different approaches to behavioral specification and verification of implementations. For example, the specification could simply list all of the legal sequences of operation invocations, or use *representation invariants* and *abstract functions* [Win89, Wei93] to specify and verify the implementation of an object. Examples of such specification languages are the CORBA-IDL derivate ACL [SH94] and Z [Spi88].

Verification serves the purpose of proving object implementations *correct*, whereas mechanisms like object-groups and Virtual Synchrony (to be

covered later in this chapter) serve the purpose of having distributed applications function *well* [MCWB94]. Both aspects are important for the building of highly dependable software, and neither of the two precludes the other.

Advantages

Differentiating between IDL and target language has the following advantages:

- **Separation**: Specification and implementation of an object are clearly separated. This allows programmers to abstract from the underlying system software and hardware when a distributed application is designed. Heterogeneity can be accomodated by providing different implementations of the same interface, even using different target languages. Objects implemented with different target languages can interact with each other if their interface is specified using a common IDL, and if the IDL compiler can produce invocation stubs for the target languages.
- **Synergism**: There is no agreement on whether new languages should be created for programming distributed systems or whether it is better to devise distributed programming libraries for existing languages [LKBT92, SM91, DPJ90, Den92]. In our experience, an IDL can help in combining the advantages of both approaches, and indispensable constructs which are not part of a conventional target language can be provided by the IDL. Wide-spread, non-distributed programming languages can then be used to implement distributed applications, and existing code can be reused to a certain extent.
- **Reusability**: Interchangeable, reusable components for distributed systems are now becoming part of reality. An IDL can help in breaking objects free from the ties of a specific programming language or operating system [Int93], since separating interfaces from implementations means that applications access objects regardless of which programming languages created them and on what platforms the objects are instantiated.

Disadvantages

An explicit separation, on the other hand, might cause several problems:

- **Stylistic Excrescence**: The programmer is confronted with two different programming languages, which could embody incompatible programming paradigms. For example, the IDL might dictate an object-oriented programming style, whereas the target language might be a functional one. Furthermore, the IDL might not support important features of the target language, passing complex user defined objects as parameters to remote object invocations, for example.
- **Lack of Expressivity**: The semantics of the resulting, compound programming language might not be powerful enough for implementing certain aspects of a complex distributed system.
- **Language Design Restrictions**: Most of the programming languages in use today were not designed with distribution in mind. Therefore, programmers have to be aware that the semantics of remote operations are different from those of local ones. For example, passing pointers as parameters to remote operations is complicated [TvR88]. Such problems can be addressed and solved in a proper manner as new languages are designed [LKBT92].

Requirement 2:

In EOM, the operations of an active object are specified with an Interface Definition Language (IDL). The IDL supports interface declarations, common basic data-types, constructed types, user-defined objects as parameters to remote operations, and interface-inheritance. EOM is neither tied to a specific target language nor to a specific IDL.

Nice-to-have:

Legal behaviors of an object can be specified formally, and a verification tool checks whether the object implementation is correct in respect to its specification.

3.4 Remote Method Calling

As previously mentioned, RMC means that the methods of a remote object can be invoked through a local proxy object (Figure 3.2). The proxy provides the same interface as the remote object it represents. Parameters of remote method invocations may include fundamental data types as well as user-defined, passive objects, and type checking of the parameters can be performed at compile-time.

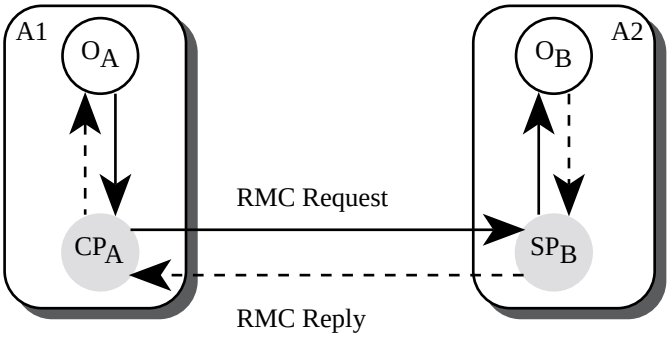

Figure 3.2: Objects communicating by Remote Method Calling. O_i denotes an object, CP_i a client proxy, SP_i a server proxy, and A_n an address space.

Communication is carried out through a lossless and non-generating FIFO channel. This ensures that requests issued by a source object are received by the destination object in the same order. RMCs provide *exactly-once* semantics under normal operation and *at-most-once* semantics in the case of failures.

EOM supports three kinds of RMC: asynchronous mode, synchronous mode, and deferred synchronous mode using *promise* objects [LS88]. In asynchronous mode, a thread issuing an RMC proceeds without awaiting the reply for the remote operation. In this case, an upcall method can be specified, which will be invoked by the sender's run-time system as soon as the reply (containing the RMC's return values) arrives. This upcall obtains an own thread of execution. In synchronous mode, the sender is blocked until the reply has arrived.

In distributed systems high throughput and performance often comes from asynchronous communication patterns [vECGS92], where a sender

launches a message into the network and continues computing, and the receiver is notified by the run-time system when the message arrives. EOM is ideal for this kind of application.

Promises allow for an intermediary form of synchronization: the thread issuing an RMC uses a special promise object to synchronize with the RMC. Issuing the method *wait* on the promise object blocks the calling thread until the reply for the RMC has arrived.

An example of effective communication using promise objects is when a client program wishes to create objects on several remote hosts to interact with them afterwards. The object creation itself is carried out by a resource manager object instantiated on each remote host. On the one hand, creating an object on a remote host takes a relatively large amount of time, and serializing the creation of many objects is hence too expensive. On the other hand, relying on purely asynchronous communication is not possible since the client object needs to know for sure that all objects have been created before it can interact with them, for example to carry out a parallel computation. Using promise objects, the problem can be solved in a both elegant and efficient way. The client application fires up all process-creation RMCs in parallel by assigning a promise object to each call. Issuing *wait* on the promise objects automatically blocks the client until all objects have been created, i.e., all resource managers have replied. Creating several objects could therefore take about the same time as to create one object.

Requirement 3:

Basically, communication is carried out by Remote Method Calling (RMC). Therefore, a client object invokes the methods of a local representative of the remote object, called the client proxy. The proxy supplies the same set of operations as the remote object. An RMC obtains an arbitrary number of input parameters and returns an arbitrary number of results. RMCs can be carried out in a synchronous, in an asynchronous, and in a deferred synchronous way. The exact mapping of the RMC types and the syntax of remote method invocations depends on the target language and is thus not specified by EOM. Communication with an object is carried out through a lossless and non-generating FIFO channel.

Nice-to-have:

The programmer can specify weaker semantics such as best-effort delivery and unordered communication. Stronger semantics such as atomicity (*zero-or-once* delivery) can be specified for applications requiring a transactional communication style.

3.5 Object-Groups

In distributed systems, communication can take two different forms depending on the number of participants: point-to-point and group communication. The first form is trivial, it is provided by conventional communication mechanisms like message passing [Tan92] or RPC [Nel81]. The second form is more powerful and more complicated than point-to-point communication, it has recently received much attention [VR92, Bir93a, Isi93b, MPS93a, vRBC+92, ADKM92b, GL93, SS93, Par92, CNL89]. We believe that the conjunction of the group communication model with the object model will lead to a compelling programming paradigm for future distributed systems.

One of the most basic mechanisms that EOM provides is a means of grouping objects and naming them as a unit. Such a construct is called an *object-group* [Bir93a, CSB92, Isi93a, HHBC92, Maf94a]. An object-group can contain just one member-object, but it will often consist of a number of objects residing on machines anywhere in the system. The main purpose of object-groups is to simplify the implementation of replicated or parallel system services, and thus to increase availability or performance. Client

applications communicate with object-groups by *reliable multicast*, using programmatical expressions which are similar to and often even identical with point-to-point RMC. The members of a group fail independently, and invocations can be performed on the group as long as at least one member is operational.

The terms *multicasting* and *broadcasting* are often confused. We use the term multicasting to refer to a message which is sent to all members of a group. We refer to broadcasting when a message is sent to all objects of an application, regardless of which group they belong to. The terms multicast and group-communication will be used interchangeably.

If a proxy object is bound to an object-group, the method calls issued on the proxy are transparently multicast to all of the objects in the group. Programming an object-group should hence be no harder than programming a singleton object. In principle, a singleton object could be replaced by an object-group without needing to modify the applications which used the singleton object. As depicted in Figure 3.3, the programmer always deals with one proxy, which may represent a single object or an object-group.

3.5.1 Tasks and Types of Group Members

Object-groups can be classified into two major categories: *homogeneous* and *heterogeneous*. A homogeneous object-group consists of objects which all perform the same task, whereas the members of a heterogeneous object-group perform different tasks. For example, a replicated file system can be realized using a homogeneous object-group. The clients then see a logical, highly available file system service which remains operational as long as at least one group-member is functioning.

A typical application area for heterogeneous groups is in controlling distributed applications: Objects in a distributed application may perform their own tasks, but they may also provide a set of management operations inherited from a controller interface. These operations allow for resetting the state of an object, for shutting down an object, and so forth, they can be applied to all objects or to a selected subset of them using one multicast operation. EOM requires that

- all object-group members are of the *same type*, i.e., instances of the same interface (Figure 3.4 a)
- or that the group members have an ancestor-interface in common.

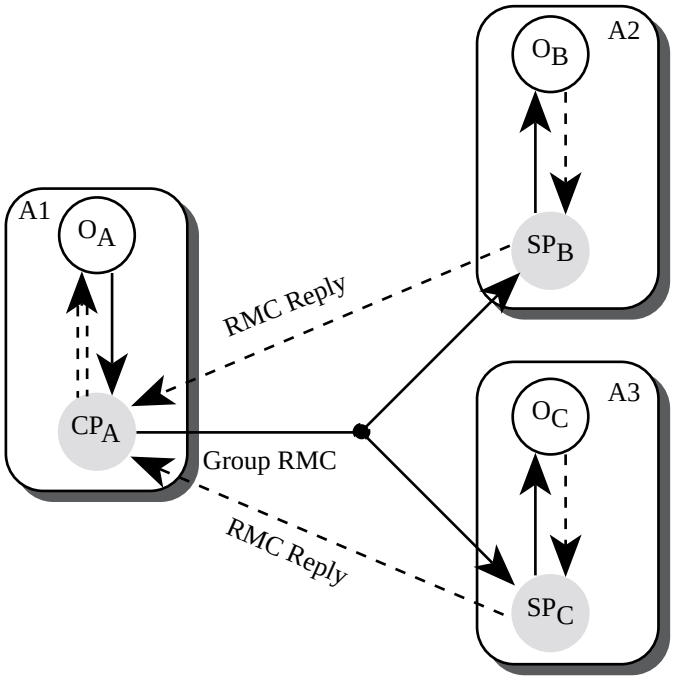

Figure 3.3: Group RMC communication. O_i denotes an object, CP_i a client proxy, SP_i a server proxy, and A_n an address space.

In this case, only the operations inherited from the ancestor can be multicast to the group (Figure 3.4 b).

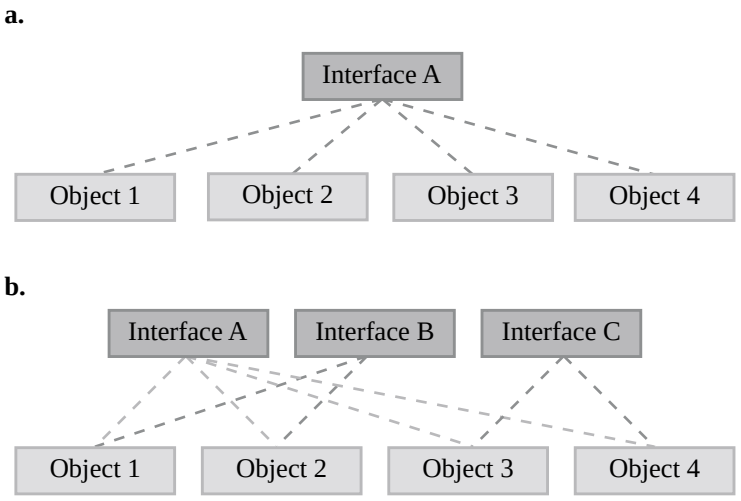

Figure 3.4: Type relationships between group members. Dotted lines denote "is instance of" relationships. Object 1, 2, 3, and 4 are in the same group. Figure a. depicts the situation where group members are of the same type. In Figure b. only the operations declared by interface A can be multicast to the group.

Without a common interface, objects cannot be grouped together. For grouping completely unrelated objects, one needs to *mixin* an artificial ancestor interface. Operations the programmer wishes to multicast to the unrelated objects are then specified by this interface. Depending on the type and task of their members, object-groups can be classified as in Table 3.1.

Active replication is when each group member performs the same computation, it mainly serves for fault-tolerance. With *multi-versioning*, group-members perform the same task, but are implemented by another person or team [Avi85]. In this case we assume that all members are of a different type but were derived from the same base-interface. Operations are multicast to this heterogeneous group, and a voting mechanism selects the result value that the majority of the members returned. A homogeneous group

	Same Interface	*Common Ancestor*
Same Task	Active replication	Multi-versioning
Different Tasks	Load balancing or primary-backup	Monitoring and control

Table 3.1: Classification of object-groups.

of objects with same type can be used for *load balancing*, in that the client requests are distributed evenly to the members, and unlike active replication, each client request is computed only once. The same configuration can be employed to carry out a *primary-backup* computation [GS91], where the primary object performs the computation and periodically sends checkpoint messages to the backup object. The backup performs no computation for the clients as long as the primary is operational. When the primary fails, the backup becomes the new primary. The primary-backup model can be extended to several backup objects, thus allowing a higher degree of resilience. This is often called the *coordinator-cohort* approach [Bir93a]. Finally, a heterogeneous group of objects performing different tasks is useful for *monitoring and control* purposes, as was mentioned before.

3.5.2 Other Design Issues

Besides the task and type of group members, there are some other design issues for object-groups, namely the ordering of messages sent to a group, the reliability of the multicast protocol, and the group structure. Ordering of events will be addressed in Section 3.8. Concerning the reliability of the multicast protocol, EOM requires *reliable multicast* [CM84, KTHB89], which means that all operational group members deliver the same set of messages (Agreement), that this set includes all messages multicast by operational objects to the group (Validity), and that no spurious messages are ever delivered (Integrity) [HT93].

Object-groups can be implemented as closed structures, where only members can issue multicasts on their group, or as open structures, where non-members can issue multicasts as well. For example, process groups in the AMOEBA [TvRvS+90] operating system are closed structures, whereas HORUS and ISIS groups are open ones. Note that in closed groups one can mimic open communication by having a group member forward messages

which originate at non-members. We request that EOM object-groups be open constructs.

Another structural issue is whether groups are static or dynamic. Dynamic groups allow objects to join or leave at any time, whereas in static groups the members are specified initially when the group is created. Members cannot leave or join static groups. For example, dynamic groups are employed in AMOEBA, HORUS, and ISIS, whereas PSYNC [PBS89] and the MPI standard [MPI94] provide static groups. Note that dynamic groups can be simulated with static groups, a join- or leave-event inducing the creation of a new group and the destruction of the old one. If an object can be member of several groups simultaneously, we say that overlapping groups are supported. In EOM, objects may join and leave groups at any time, and groups may overlap.

3.5.3 Advantages

Object-groups have the following main advantages:

- **Load Sharing**: A non-replicated object can be replaced by an object-group whenever the object becomes overloaded, and, in many cases, without having to modify applications which use the object. For instance, the group members may share the available work load to increase throughput. Parallelism and load sharing can thus be increased step by step, provided that the toolkit offers abstraction mechanisms like Remote Method Calling, which allow communication with an object-group as if it were a non-replicated object.

- **Fault-Tolerance**: Availability and fault-tolerance are increased by using active replication, passive replication, or multi-versioning. Each approach can be mapped onto object-groups. An object fails independently of the other members, and the service remains available as long as at least one of the members is operational.

- **Addressing Construct**: Object-groups can be used for addressing a set of objects which have a common characteristic. For example, some objects might want to be informed every time a certain event occurs. A heterogeneous object-group containing these objects can be created and a multicast be sent to inform the group of the event.

- **Locality of Access**: In certain situations, the response time of a service is shortened if provided by a homogeneous object-group, since member-objects can be placed at the sites where the service is frequently needed.
- **Efficiency**: Object-group multicasts can be mapped onto hardware or software multicast facilities. This ensures that a physical message is received by several objects in the same network. Multicast support may be granted by the networking hardware, as it is the case for the ETHERNET [IEE85], or by the communication software, as is the case in an extended version of the IP protocol [Dee89].
- **Effectiveness**: Applications in the financial or in the flight reservation domain, for example, need a mechanism for transmitting the same information to several recipients in a fault-tolerant fashion. Object-groups facilitate the realization of such applications.
- **Application Management**: Object-groups offer a convenient solution to the problem of propagating monitoring and management information in distributed applications.
- **Mobile Computing**: In EOM, operations are invoked on abstract groups and senders do not need to know the network addresses of the members. Consequently, an object can leave a group, move to another place, then rejoin the group and continue working [CB94]. Object-groups thus offer basic support for mobility and for system reconfiguration.
- **Encapsulation**: Object-groups encapsulate a distributed state by making it accessible through a well-defined interface.
- **Reusable Components**: We see object-groups as reusable components for failure-resilient systems. New components can be created out of existing ones by means of inheritance and operation overwriting.

3.5.4 Disadvantages

Object-groups are a powerful abstraction, although they cause problems as well:

- **System Support**: Reliable object-group multicast requires sophisticated communication primitives which most operating systems available today do not provide, and which are much more complicated to implement than conventional point-to-point communication primitives. Nevertheless, communication substrates providing reliable multicast primitives are being researched intensively and a few of them, for instance AMOEBA, HORUS, and ISIS, are widely available already.

- **Group Management**: At all times, an object-group member has a certain idea of which other objects belong to its group. Since objects may join or leave a group at any time (by terminating or by failing), the group membership is subject to changes and the members' views on group membership can become incongruous. This is known as the group-membership problem and will be addressed in Section 3.7.

- **Consistency**: When operations are issued on replicated objects, inconsistencies may arise if individual operations arriving at the members do not respect certain ordering rules. Ordering of events will be discussed in Section 3.8.

- **Result Collation**: A group RMC might return one result per group member, and programmers might be interested in the first arriving result, in all results, or in a number inbetween, which complicates the transparent handling of object-groups. Therefore, programmatical constructs must be provided which allow programmers to specify how many results they wish to obtain from a group RMC. Result collation is treated in Section 5.5.

Requirement 4:

EOM supports collections of objects, so-called object-groups. Proxy objects hide replication to a large extent and allow clients to interact with object-groups analogously to the way interaction with singleton objects occurs. All operational objects deliver the same set of messages, and this set includes all messages multicast by operational objects, and no spurious messages (reliable multicast).
Object-groups are open in the sense that also non-members may issue multicasts on the group. Object-groups are dynamic in that objects may join or leave a group at any time, and an object may be member of several groups simultaneously. Group RMCs return responses as long as at least one group member is operational, and group members fail independently from each other.
Where an RMC produces a set of results, programmers can specify that the results be collated to a single termination, or that a specific number of results, a majority for instance, be returned.

Nice-to-have:

Unreliable multicast can be selected for applications which do not require the strong guarantees provided by reliable multicast. The programmer can specify that a voting mechanism be employed to compare incoming results of a group operation and to select the most frequent one.

The issues we consider next are the detection of group-member failures, membership management, and ordering of events within an object-group based application.

3.6 Detecting Failures

In distributed computing environments, failures should be seen as a normal and not as an exceptional occurrence. Failures occur due to software errors, hardware errors, power outages, human lapses, or catastrophic events such as inundation. Inconsistent reporting of failures has been argued to be at the heart of a number of forms of incorrect behavior in contemporary distributed systems [BG93]. Consistent failure reporting in asynchronous systems is treated in [SM94a, MMSA93, CT93, BG93, SR93, MPS93b, FLP85, SS83, Sch93b].

3.6.1 Failure Suspector Service

A way around the impossibility of reaching consensus in an asynchronous distributed system with faulty processes [FLP85], is to incorporate a *Failure Suspector Service (FSS)* for suspecting failed objects and for propagating failure beliefs in a consistent manner. To ensure scalability and flexibility, we must implement failure detection without making synchrony assumptions. Basically, the FSS guarantees that if one object detects a failure, the other objects will also be informed of the failure. Timeout mechanisms and hints from the underlying operating system are used to suspect failures. The FSS is allowed to make incorrect failure assumptions, meaning that under certain conditions a correct object will be suspected to have failed. This does not cause any problem as long as this belief is propagated to the remaining objects. According to Ricciardi *et al.* [RSB93], the three important building blocks of a FSS are failure belief stability, channel disconnect, and propagation of failure beliefs.

- **Failure Belief Stability**: Once adopted, a failure belief is true forever. Of course, an object may recover and re-join the system. In this case, it appears as a new object.
- **Channel Disconnect**: Once an object P suspects an object Q to be faulty, it neither accepts any further messages from Q nor sends any further messages to Q.
- **Propagation of Failure Beliefs**: This property aims at achieving failure belief consistency. Since failure detection in an asynchronous system is always inaccurate, we need a mechanism to propagate failure beliefs consistently from one object to the others in an application. The FSS propagates failure beliefs along causal chains of events. This means that once P detects the failure of Q, any subsequent message sent by P to any process k will not be received until k has also detected the failure of Q [SM94a]. For instance, object P piggybacks its belief on outgoing messages, and so k learns that Q is believed faulty.

The Group Membership Service which we will describe in Section 3.7, the Reliable Multicast Component of Section 3.5, and the FSS cooperate closely as depicted by Figure 3.5. For instance, the Group Membership Service uses the FSS to detect membership changes due to member failures, whereas the Reliable Multicast Component detects failures occurring during a multicast by consulting the FSS.

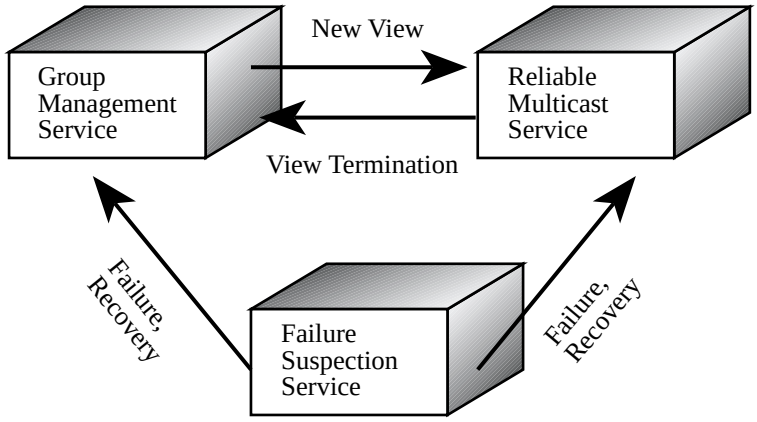

Figure 3.5: Interaction between the Group Membership Service, the Reliable Multicast Component, and the FSS.

3.6.2 Partition Model

The fail-stop model we assumed in Section 3.1 can be approximated by working in a *primary-partition* model, as it is done in the ISIS system. If a network partitions, then one primary partition at most is permitted to continue execution. The FSS forces processes outside the primary partition to leave the system.

A more sophisticated partition model has been implemented in the HORUS and in the TRANSIS system. A HORUS or a TRANSIS system configuration comprises machines that can crash dynamically and networks that may partition and merge. Both systems can be configured to allow progress during network partitions, and partitioned execution is thus permitted [vRHB94, ADKM92a]. When a subdivision disappears, non-primary partitions may merge with the primary partition. In any case, the processes will be notified that a partition occurred and in what partition they are, so that application-dependent provisions can be taken.

Requirement 5:

In EOM, a Failure Suspector Service (FSS) is used to suspect failures and to propagate failure beliefs. Failure beliefs are gathered mainly with timeout mechanisms and hints from the operating system; they cannot be completely accurate since the EOM is based on an asynchronous system model. This does not cause any problems as long as incorrect beliefs are handled and propagated in a consistent way. Failure beliefs are piggybacked onto the messages exchanged by the objects in the application and travel along causal chains of events. A close cooperation takes place between the FSS, the Group Membership Service, and the Reliable Multicast Component. In analogy to ISIS, we assume a *primary-partition* model.

Nice-to-have:

Similar to HORUS and to TRANSIS, a multiple-partition model could be supported.

3.7 Object-Group Membership Service

As mentioned before, objects may join or leave their groups at any time. Objects leave a group by explicitly issuing a leave-request or by failing. The members of an object-group always have a certain idea about which other objects belong to the group; this is called a *view*. A view of the group is a snapshot of the group membership at a certain point in the execution of an application. To hold individual views consistent, a system-wide service is necessary, which informs the group members of view changes. Without such a service, object-group based applications could behave in incorrect and unexpected ways. We call this service the *Group Membership Service (GMS)*, the term *Group Membership Protocol (GMP)* is used to refer to the protocol employed to synchronize the individual views.

The group membership problem in asynchronous systems is treated in [BJ87a, MPS93a, ADKM92b, PBB+94, CM84, Kaa92, Ric92, JFR93, GT92]. In this section we shall first give an overview of the fundamental GMS protocols providing different degrees of consistency. Then, we will present the GMP required by EOM. Basically, three different kinds of GMP can be distinguished: weak, strong, and hybrid GMP.

3.7.1 Weak GMP

The weak GMP [JFR93, GT92] guarantees that when no failures and no requests to join or to leave occur during a certain period of time, the views of all member objects eventually merge into a single and consistent view. Temporary inconsistencies may thus occur within a weak GMP. Formally, given that the last membership-change occurred at time t, the probability that the views of two distinct group members disagree at time $t+\delta$ tends to zero, as δ grows towards infinity. Weak membership can be guaranteed by a one-phase protocol: each node of the system has a GMS daemon running. A GMS daemon simply propagates the changes of view to the group members it knows about, the views of two members might thus differ. An application area of weak group membership protocols is in WAN settings, for applications which can tolerate temporary inconsistencies between group views [Gol92].

3.7.2 Strong GMP

A strong GMP guarantees that membership changes are seen in exactly the same order by all members of a group [Ric92]. Formally, given that the last membership change occurred at time t, the probability that the views of two distinct group members disagree at time $t+\delta$ is zero for every δ at which a message is submitted or delivered by the group members. The strong GMP differs from weaker forms in that membership changes are acknowledged. Usually, two or more communication phases are necessary to handle a view change, which makes this protocol more expensive than the weak one.

3.7.3 Hybrid GMP

A hybrid GMP provides a level of consistency between that of the weak and that of the strong protocol [GT92]. The main difference between the strong and the hybrid protocol is that the latter preserves a partial ordering [Lam78] of membership changes and not a total one. A partial ordering can be guaranteed without the introduction of extra messages and is thus less expensive to maintain than a total one.

A GMS obtains information when the client objects issue join- and leave-requests, and gathers data on failures by cooperating with the Failure Suspector Service, as was shown in Figure 3.5.

Requirement 6:

Object-group members register with a Group Membership Service to be notified of membership changes. The service is implemented by a Strong Group Membership Protocol, which ensures that membership changes are seen in the same order by all members. Furthermore, membership changes are synchronized with the application's RMCs (point-to-point and multicast), meaning that at the moment an RMC is sent or delivered by the run-time system, views are consistent and the latest view is installed in the run-time system.

Nice-to-have:

Weaker group membership protocols may be provided in addition to the strong one, and programmers can select a protocol which best fits their applications' requirements.

3.8 Ordering of Events

Reliable multicast guarantees that all correct objects agree on the set of RMCs they deliver, that all RMCs multicast by correct objects are delivered, and that no spurious RMCs are ever delivered. However, reliable multicast imposes no restriction on the *order* in which RMCs are delivered to the members of an object-group, which can lead to incorrect behavior of distributed applications. In this section, we shall give an overview of fundamental event-ordering protocols. Ordering of events in asynchronous, group-based systems is an important topic of the present distributed systems research, it is being studied in the context of projects such as AMOEBA [MvRT+90], CONSUL [MPS93a], DELTA-4 [PBB+94], HORUS [vRB94], ISIS [BvR94], and TRANSIS [ADKM92b].

In the following, the term *message* refers to an RMC request issued by the sender, or to an RMC reply returned by the receiver. The term *process* denotes the part of the run-time system which transports remote object-invocations. A receiver process may delay an incoming message until a certain requirement of the ordering protocol is fulfilled, after which the message is delivered to the destination object.

3.8.1 FIFO Multicast

The weakest ordering protocol which we consider simply guarantees that the RMCs sent by the *same* object are delivered in the same order. Messages sent by different objects to the same group may arrive in different orders. Specifically, if an object multicasts rmc_1 before rmc_2, then no correct process delivers rmc_2 before it has delivered rmc_1 [HT93]. This is called the FIFO (First-In-First-Out) multicast, or FMCAST for short.

Implementation

FIFO ordering is realized simply by having each object maintain one local sequence counter per object-group it sends multicasts to.

3.8.2 Atomically Ordered Multicast

An atomically (or totally) ordered multicast protocol [BSS91, HT93], AMCAST for short, ensures that RMCs are delivered in the *same order* to all group members. More specificly, if two correct objects P and Q both deliver rmc_1 and rmc_2, then P delivers rmc_1 before rmc_2 if, and only if, Q delivers rmc_1 before rmc_2. This holds even when rmc_1 and rmc_2 are issued by different senders. Note that AMCAST does not imply FMCAST in general, but for the sake of simplicity we assume that AMCAST respects FIFO ordering.

Implementation

An early AMCAST protocol is described in [BJ87b]. Basically, the protocol operates with a two-phase commit mechanism. Each multicast is assigned a sequence number, and processes deliver messages in the order of the numbers. When a process receives a new message, it stores it in the *undeliverable* queue, and sends a message containing a proposed sequence number for the multicast back to the initiator. The proposed number is higher than any other number the site has proposed or received in the past. The initiator collects all numbers, selects the highest, and sends this value back to the recipients. The value becomes the final sequence number of the multicast, and the message is marked *deliverable*. This truly distributed AMCAST protocol requires three messages per multicast. A less costly variant is proposed in [BSS91] and realized in the Isis toolkit.

The AMOEBA multicast protocol [Kaa92], on the other hand, also guarantees total ordering, but employs a different approach than ISIS. In an AMOEBA process-group, total ordering is accomplished by a special group member, which is called the *sequencer* of the group. To submit a multicast, a client sends a point-to-point message containing the data to be multicast to the sequencer of the target group (called the PB method of the protocol). The sequencer assigns the next sequence number to the message and multicasts the message. All multicasts to a specific group are thus issued by the same sequencer. Hence, no multi-phase protocol is needed to assign sequence numbers. The AMOEBA protocol provides further optimizations like *negative acknowledgements* and a multicast method which is better suited for large messages (called the BB method of the protocol).

Comparison

The advantage of the ISIS AMCAST protocol is that it is truly distributed. This ensures scalability, load distribution, and failure-tolerance. The disadvantage of ISIS AMCAST is that it is more complicated than the AMOEBA protocol. The advantages of the AMOEBA protocol are its simplicity, its elegance, and its efficiency. The disadvantage is that the sequencer may become a performance bottleneck and a single point of failure. There are situations where the ISIS protocol is the better choice and situations where the AMOEBA protocol is better suited.

3.8.3 Causal Multicast

Some applications can preserve consistency with a weaker, less expensive *causal ordering* using *causal multicast* [Lam78, SM94b] (CMCAST). An advantage of CMCAST, as compared to AMCAST, is that the protocol works without the introduction of extra messages, the associated overhead thus being considerably lower.

Causal communication uses the *happened before* relationship described in Lamport's paper [Lam78] to order messages in a distributed system. Causal communication is a fundamental concept of asynchronous distributed systems and can be applied to settings where senders communicate with several receivers in an asynchronous fashion, and where messages are delivered indirectly by means of intermediary processes.

Informally, causal communication generalizes the notion of one message

depending on another, and ensures that a message is delayed by the receiver's run-time system if the message depends on one the receiver has not seen yet. Formally, causal dependency is defined as follows:

- If a and b are events in the same process, and a happened before b, then b causally depends on a, which is denoted by $a \rightarrow b$.
- If a is the sending of a message by one process and b is the receipt of this message by another process, then $a \rightarrow b$.
- If $a \rightarrow b$, and $b \rightarrow c$, then $a \rightarrow c$ (transitivity). Two distinct events a and b are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$.

Thus, one message depends on another if it is possible that there exists an information flow between the two messages. In Figure 3.6, message c depends on message a, and message d depends on message b. In the left scenario, message d surpasses message c, and d is thus delivered before c, which violates causal delivery. In large systems, such effects can lead to application-malfunctions which are very difficult to debug. The right scenario shows a CMCAST protocol employed in the distributed system. Here, the run-time system of process R delays the delivery of message d until c arrives, thus ensuring causal order.

Implementation

CMCAST can be implemented in two different ways: using time-vectors or using message dependency graphs.

Time-Vector Protocol Each member of an object-group is assigned a *time-vector* [SM94b] similar to the ones in Figure 3.7. Time-vectors are managed by the run-time system, they are transparent to the applications. The size of the vector equals the number of group members. Vector element i relates to group member i, and the group members are numbered from 0 to $n-1$. Each time an object sends a message, its slot on the local vector is incremented and the vector is appended to the outgoing message. When a message arrives, the receiver's run-time system increments the local vector-element which is associated with the sender of the message. The receiver's run-time system then compares the local vector with the one just received and decides whether to deliver the message along the following conditions,

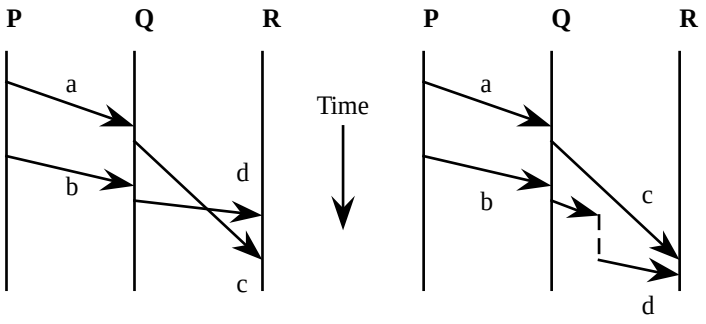

Figure 3.6: Non-causal and causal communication. P, Q, and R are processes in a distributed system, a, b, c, and d are messages. An arrow denotes a message being passed from one process to another.

where L denotes the local vector, R the received vector, and j the number of the sender:

$$R_j = L_j + 1 \tag{3.1}$$

$$R_i \leq L_i, \forall i \neq j \tag{3.2}$$

Condition 3.1 simply states that this is the next message which arrived in sequence from j. Condition 3.2 demands that the message does not causally depend on a message the receiver has not yet seen. If at least one condition proves false, the message is marked undeliverable and delayed by the runtime system, until another message arrives so both rules are proved true. Note that no extra messages are needed to implement causal delivery, since vectors are piggybacked onto outgoing messages. Moreover, vector compression [SK92] can be applied to reduce the amount of piggybacked data considerably. Figure 3.7 presents events occurring between four processes along with the associated time-vectors.

A similar algorithm is implemented in the Isis *cbcast* protocol, where provisions are taken for overlapping groups and for failure recovery. This results in a protocol which is considerably more sophisticated and complex than the one we described.

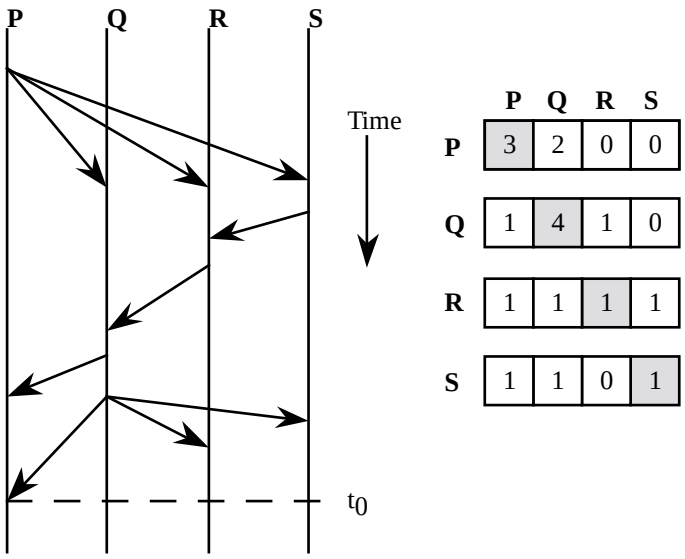

Figure 3.7: Distributed events involving the processes P, Q, R, S, along with the state of the associated vectors at time t_0. A shaded vector-element contains the number of messages sent by the associated process. The other vector-elements contain the number of messages received by the process.

Dependency Graph Another variant of the implementation of causal delivery consists in having the run-time system of each group-member maintain a directed, acyclic *dependency graph* [PBS89, DKM93]. The nodes in the graph represent messages, whereas the arcs connect messages that are directly dependent in a causal order. Figure 3.8 shows the dependency graph for the former example.

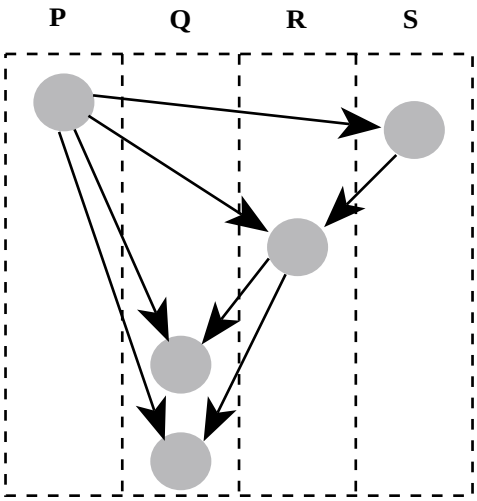

Figure 3.8: Causal order and message dependency graph for the situation in Figure 3.7.

Context information is attached to outgoing messages, so the receiver's run-time can adapt its local graph according to it. By traversing the graph, the receiver's run-time establishes whether the newly arrived message can be delivered or not, i.e., whether some of its causal predecessors are missing. A protocol of this kind is implemented in the CONSUL and in the TRANSIS system. Dependency-graphs are a flexible and general approach in that other traversal algorithms can be applied to them as well, resulting in a variety of ordering protocols, such as FMCAST or AMCAST.

Comparison

There are situations where causal ordering is too weak to satisfy the requirements of an application. An example for this is a replicated file server,

where requests have to arrive at every replica in the same order to guarantee that the file copies remain identic. This must be true even for causally unrelated requests, a condition which can be met by an AMCAST protocol. Another example is when objects communicate through a "hidden communication channel", for instance by passing information through a network file system, which goes undetected by the run-time.

Some authors, see [CS93b, APR93] for instance, are concerned about the fact that most implementations of CMCAST provide causal delivery based on low-level message flows such as $\texttt{send}(\texttt{msg}_1) \rightarrow \texttt{receive}(\texttt{msg}_1)$, and not on the semantics of the application, a controversy which is known as the *end-to-end argument* in system design [SRC84]. The authors claim that focussing on low-level message flows will lead to poor performance or to incorrect behavior, since it might impose causal ordering where application semantics do not require it, or it might not recognize certain application-level dependencies. On the other side, experiences with HORUS and ISIS have shown good scalability of the CMCAST protocol [BC94]. They also demonstrated that dependencies which are not reflected by low-level message flows can be introduced with a token-passing tool, for instance [vR94].

It is our opinion that CMCAST protocols as provided by toolkits like CONSUL, HORUS, ISIS, and TRANSIS are valuable abstractions (see also [Shr94]). After all, HORUS or ISIS programmers are not forced to use causally or totally ordered communication. If necessary, unordered communication can be specified or ordering policies of their own can be implemented. Nevertheless, it would be useful to offer a programming interface which facilitates the control of the causality mechanism where necessary. For example, an application communicating by a "hidden channel" could specify the resulting, application-level dependencies to the underlying CMCAST module.

3.8.4 Causal Atomic Multicast

AMCAST itself does not require that messages are delivered in causal order. We can therefore define a stronger multicast protocol called ACMCAST, which satisfies both causal and total delivery. The ISIS *abcast* protocol [BSS91] respects causality and orders messages correctly even in the presence of overlapping groups. In contrast, the aforementioned AMOEBA multicast protocol does not respect causality or support overlapping groups.

Implementation

An ACMCAST protocol can be realized by combining a CMCAST with a token-passing protocol. In each process group, a special message known as the *token* circulates. The process in possession of the token at a specific time is known as the *token holder*. The token holder remains in that position until it terminates or crashes, whereupon a new token holder is elected. If the token holder wants to ACMCAST a message, it simply uses the CMCAST protocol. If a process not in possession of the token wishes to issue an ACMCAST, it has to proceed step by step: it CMCASTs the message, the receivers mark it as undeliverable, and the token holder multicasts a message to the receivers indicating the delivery order to adopt. For an indepth description of this protocol see [BSS91].

3.8.5 Causality Domains

Ordering of events is crucial not only for group-based distributed applications, it plays an important role in any asynchronous distributed system [vR93b]. Point-to-point RMCs should therefore be delivered causally in respect to multicast RMCs and other point-to-point RMCs. Multicasts issued on overlapping groups should also respect causality. Constructs allowing programmers to define *causality domains* would be useful in structuring complex distributed applications. A causality domain consists of a set of communication endpoints with all communication respecting causality. Communication crossing domain boundaries does not respect causality.

Requirement 7:

In EOM, communication is inherently asynchronous, which is why we require that the run-time system provides causal delivery (CMCAST) for all kinds of object invocations. Additionally, programmers can specify that object-group multicasts originating in different senders are seen in exactly the same order by all receiver objects (ACMCAST), or they can choose a FMCAST protocol for applications which are not sensitive to ordering. Overlapping groups are supported, but objects being simultaneously member of two different groups might receive the same set of RMCs in different orders. EOM does not assume the existence of a synchronized-clocks service.

Nice-to-have:

Programmers can influence the underlying causality mechanism to give hints about "hidden" causal relationships. Furthermore, it is possible to specify that ordering semantics hold true even in presence of overlapping groups. Large applications may be structured using causality domains.

3.9 Virtually Synchronous Execution

In asynchronous systems, a reasoning based on "global time" is not possible [Lam78]. To tackle this problem, execution models like *Virtual Synchrony* [Bir93a] or *Pseudo Synchrony* [PBS89] have been proposed. The two models are very similar, so we shall use the former term for both.

Informally, Virtual Synchrony can be defined as follows [SS93]: all significant events, for instance the delivery of unicasts, multicasts, view-changes, and failures, appear as if each event had occurred at the same logical instant in all processes. Events are thus synchronous in terms of logical time [Lam78] and asynchronous in terms of global time. Virtual Synchrony can be used to synchronize distributed activities by releaving programmers of problems such as handshaking and distributed consensus.

In a virtually synchronous system, causal delivery is respected and events have to be synchronized (delayed by the underlying run-time system) only to the degree that an application is sensitive to event ordering. Although an external observer may detect situations where events occur in different orders at different processes, the processes themselves are un-

able to detect this, since such events are causally unrelated per definition. The Failure Suspection Service, the Group Management Service, and causal delivery are important components of the Virtual Synchrony model. Figure 3.9 depicts events in an asynchronous system executed without and with Virtual Synchrony.

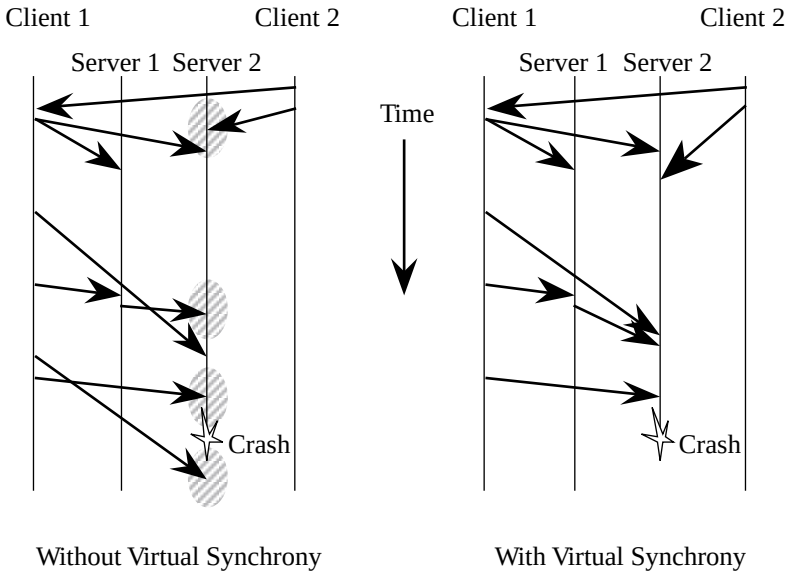

Figure 3.9: Events in an asynchronous distributed system, without and with Virtual Synchrony. Patterned areas indicate situations where Virtual Synchrony is violated.

3.9.1 Advantages

An important benefit of the Virtual Synchrony model is that it makes asynchronous systems appear as if they were synchronous. Virtual Synchrony combines the advantages of an asynchronous system with the advantages of a synchronous one, while avoiding certain disadvantages of both models. Despite changes in group membership and despite failures, distributed applications behave in a predictable way. Moreover, Virtual Synchrony allows for efficient asynchronous communication, and messages are delayed only

when necessary. Thus, the inherent parallelism of an application can be exploited while still maintaining a consistent ordering of the events.

3.9.2 Disadvantages

The Virtual Synchrony model is well-suited for so-called *directly distributed systems*, in which distributed threads of execution interact *directly* with one another while continuously respecting constraints on their joint behavior [BJ89]. Examples of directly distributed systems are distributed file systems, name servers, groupware applications, audio- and video-cast systems, and various kinds of fault-tolerant client-server applications. Without additional synchronization primitives, however, the Virtual Synchrony model is not able to support certain *data-oriented* applications. For example, applications managing long-lived data require atomic transactions and Serializability [BHG87]. Nevertheless, it is important to understand that Virtual Synchrony is a more fundamental synchronization model than the transactional one. In [GS94], Guerraoui and Schiper describe an approach to combine the Virtual Synchrony with the transaction model. The authors come to the conclusion that it should be possible to build a system offering both the transactional model and Virtual Synchrony.

3.9.3 Linearizability and R-Linearizability

Herlihy and Wing [HW90] propose a model called *Linearizability*, which is very similar to Virtual Synchrony. Linearizability is based on a distributed, object-oriented system model, in which objects are accessed through *invocations* and return *matching responses*. An execution of a concurrent object is said to be *linearizable*, if a sequential execution exists that respects the dependency relationship of the concurrent execution, so that each invocation produces the same response as in the concurrent execution. In [Bir93b] Birman suggests that the Linearizability model is adopted to link the Virtual Synchrony model with the transactional one to arrive at a flexible, generic approach to concurrency control in object-oriented, distributed systems. A modification of the Linearizability model to support replicated objects, called *R-Linearizability*, is proposed in [PS93].

Requirement 8:

Significant events like the delivery of unicasts, multicasts, view-changes, and failures appear as if each event had occurred at the same logical instant in all objects, i.e., executions are virtually synchronous. Virtual Synchrony is the low-level synchronization model of EOM.

Nice-to-have:

Other synchronization forms such as Linearizability, R-Linearizability, or Serializability can be provided as well and should be based on the Virtual Synchrony model.

3.10 Required System Support

Active objects require some form of multi-threading and mechanisms for inter-thread synchronization, like semaphores or monitors. Remote Method Calling on singleton objects asks for reliable point-to-point communication primitives, whereas object-group communication demands reliable multicast. Furthermore, group management protocols, failure suspectors and event-ordering protocols like ISIS' *abcast*, *cbcast*, and *gbcast* [BvR94] or CONSUL's PSYNC [MPS93a] must be provided to guarantee a consistent, pseudo-synchronous ordering of events in a distributed EOM application.

An interesting question is how to design a toolkit conforming to the EOM requirements. The complexity of any such toolkit requires that it should be structured in a highly modular fashion by employing object-oriented analysis and design [Boo91, CY91]. The architectural model could be similar to the one sketched in Figure 3.10.

Concerning the layers below the EOM abstractions, the low-level primitives which have to be put there have to be deduced, and the decision has to be made whether the primitives shall be developed from scratch or whether an existing programming library shall be used. The next chapter is dedicated to the system support needed by an EOM-compliant programming environment.

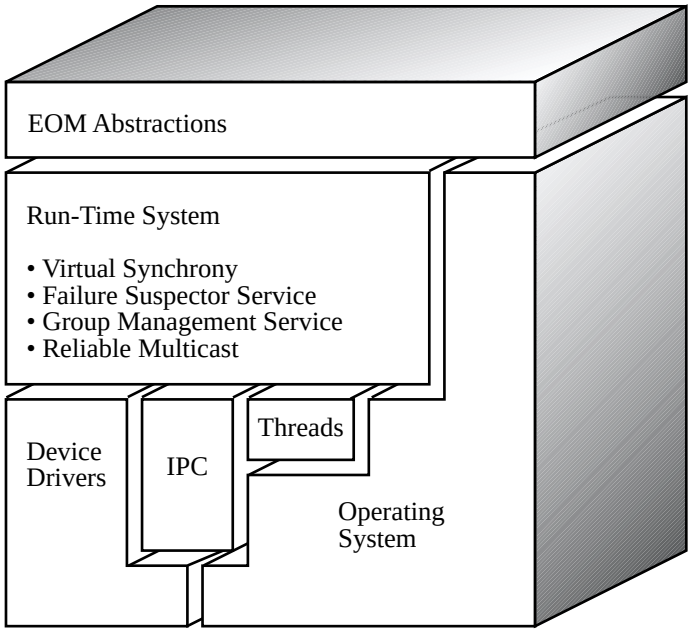

Figure 3.10: Fundamental architecture of an EOM-compliant toolkit.

3.11 Summary

In this chapter we described *The Electra Object Model (EOM)*, which comprises a minimal set of requirements we believe necessary for supporting object-oriented, distributed programming. It is based on an asynchronous system model and on the conviction that failures must be dealt with by a programming model for asynchronous distributed systems. EOM enables object-replication to cope with failures. The following elements constitute EOM: objects and interfaces, remote method calling, object-groups, a failure detection service, a group membership service, ordering of events, and virtually synchronous execution.

We believe that more sophisticated object models, for example those supporting persistence, transactions, or distribution transparency, should take EOM as a starting point. EOM itself is a conceptual model and thus largely independent of the target programming languages, of the system software, and of the networking hardware. Tables 3.2 and 3.3 provide an overview of our "need-to-have" and "nice-to-have" requirements. Appendix A summarizes the requirements.

Issue	*EOM Requirement*	*Req. No.*
Role of objects	active, passive	1
Granularity	very small to very large	1
Fragmented objects	not part of EOM	1
Interface declarations	in IDL	2
RMC	FIFO, reliable	3
Group structure	open, dynamic, overlapping	4
Reliability	reliable multicast	4
Result collation	specifiable	4
Failure detection	FSS, primary-partition model	5
Group management	strong GMP	6
Ordering	CMCAST, ACMCAST, FMCAST	7
Execution model	Virtual Synchrony	8

Table 3.2: EOM "need-to-have" requirements.

Issue	*EOM Requirement*	*Req. No.*
Interfaces	specification and validation	2
RMC	best effort, transactions	3
Reliability	best effort	4
Failure detection	multiple-partition model	5
Group management	weak and hybrid GMP	6
Ordering	causality domains	7
Execution model	Linearizability, Serializability	8

Table 3.3: EOM "nice-to-have" requirements.

Chapter 4

Flexible System Support

This chapter deals with the design of an EOM compliant toolkit, in particular with its run-time system. We will first identify the primitive operations the toolkit requires, and then propose a flexible and generic run-time system design.

Flexible system architectures are becoming more and more important as industry moves towards open computing environments, especially since portability issues as well as software reuse have become a major concern. New operating system interfaces and new programming libraries are being delivered to developers at a fast pace. This poses the developer of an EOM toolkit difficulties, and the question arises whether one should spend a lot of time to keep up with new interfaces or choose to support only the most popular ones. We propose a portable and flexible system architecture which offers an invariable API that is independent of the underlying system software. This permits the configuration of an EOM toolkit for various communication subsystems and the exploration of the kind of features future generations of operating systems will have in their kernels. The key idea is to base the run-time on a portability veneer thereby hiding the peculiarities of the underlying *real machine (RM)* (i.e., the communication subsystem), and to establish a system-specific *adaptor object*[1] to act as glue between the veneer and the real machine (Figure 4.1).

The RM consists of an operating system or of a distributed control tech-

[1] Not to be confused with adaptors in CORBA or in NEXTSTEP

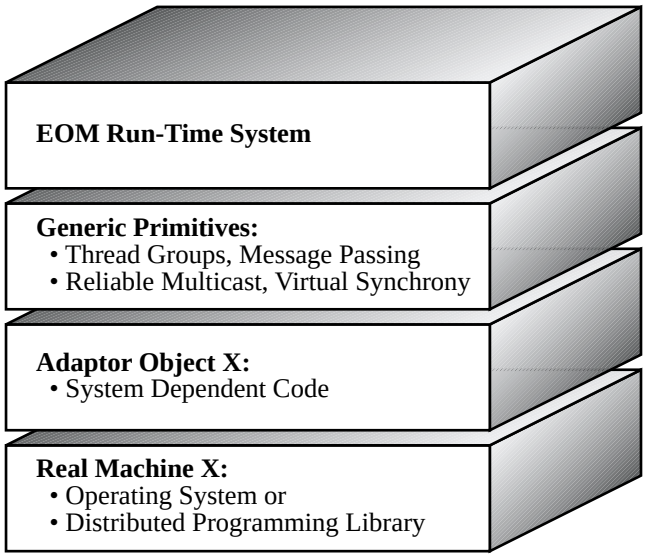

Figure 4.1: Flexible system design for an EOM Toolkit.

nology implementing the primitive operations we mentioned in Section 1.4. Examples of operating systems suited to EOM are AMOEBA [MvRT+90] and CHORUS [R+88]. Examples of suitable distributed control technologies are CONSUL, DELTA-4, HORUS, ISIS, and TRANSIS. The features common to the mentioned systems are support for low-level group communication, efficient asynchronous message-passing, and multi-threading.

4.1 Examples

A few systems offer flexible system support similar to the one we will propose later in this chapter. In this section we describe how flexible system support is granted in GTS, GUIDE-2, HORUS, PANDA, the VANILLA file system, and WINDOWS NT.

4.1.1 Generic Multicast Transport Service

The Generic Multicast Transport Service (GTS) [MBM95] is a run-time system which supports the implementation of process-group based, fault-tolerant, heterogeneous applications on wide-area networks. As the main abstraction, the GTS provides reliable multicast, reliable point-to-point communication (unicast), and process-groups on protocols such as TCP, UDP, DEERING IP, and even on e-mail and UUCP. New protocols can be incorporated into the service easily. The GTS makes use of hardware or software multicast, if available. Moreover, the GTS does not require that communicating processes are available at the same time, and temporarily unavailable processes can remain in their process-groups. To that purpose, the GTS will spool messages on nonvolatile storage and deliver them to their recipients as soon as they become available and reconnect to the GTS.

In the GTS' system model, two kinds of processes are to be distinguished: on the one side the GTS *servers* implement message spooling, reliable multicast and unicast communication. On the other side the end-user *applications* use the GTS. A GTS server along with the applications connected to it makes up what we call a *cluster* (Figure 4.2). Normally, a cluster is contained in one LAN. If an application of cluster A wants to send a message to an application of cluster B, it submits it to server S_A, which in turn sends it to server S_B. Finally, S_B delivers the message to the destination application.

The GTS uses *Uniform Resource Locators* (URLs) of the form

```
protocol://server:localAddress/entityAddress
```

to identify communication endpoints. The URL

```
gts-tcp://claude.ifi.unizh.ch:3477/1
```

for example, determines that communication to entity 1 located in cluster `claude.ifi.unizh.ch` is achieved by the TCP protocol. 3477 is the TCP port number of the GTS server in the destination cluster.

A GTS server, which is implemented in the C++ programming language, is structured in a way similar to the *x*-kernel [PHOH90]. Each GTS server consists of a collection of *adaptor objects* plugged together to form a *protocol tree* as depicted in Figure 4.3. The root object (`GTSroot`) communicates with the client applications running in its cluster. Leaf objects, called protocol adaptors, perform unreliable message-passing on specific communication protocols. The utility adaptors in the middle area carry out tasks such as reliable communication (the `Actor` object), compression, encryption, en-

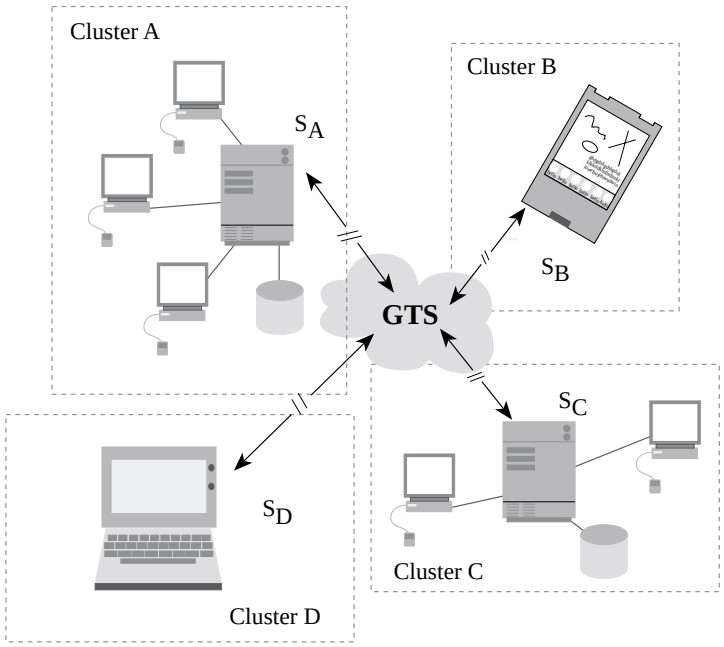

Figure 4.2: A typical GTS system configuration. S_i denotes the GTS server for cluster i. Applications run on the workstations attached to the servers.

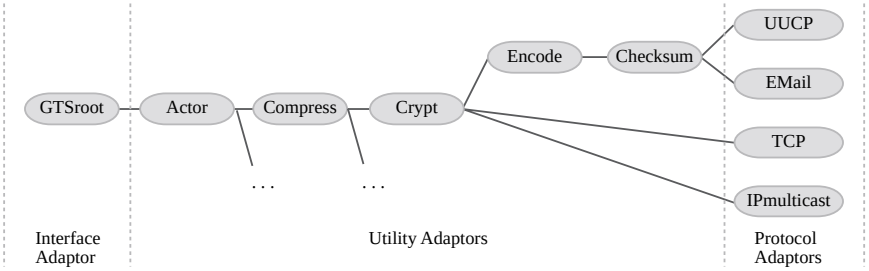

Figure 4.3: A sample protocol tree.

coding, integrity check (the `Checksum` object), and so forth in a generic and thus protocol-independent way. Each adaptor object passes the messages it receives down the tree to one of its child adaptors. A message is routed through the tree according to its destination URL until it reaches a protocol adaptor. Finally, the protocol adaptor transmits the message by using the protocol it represents.

At the destination server, if e-mail or UUCP was employed, the message is received by the corresponding protocol adaptor and passed up the tree and through the utility adaptors where it is checked, decoded, decrypted, decompressed, and unfragmented. Finally, the received message arrives at the `GTSroot` object and is transmitted to the destination application in the cluster.

Adaptor objects are organized in the form of the class hierarchy in Figure 4.4. Each adaptor object obeys the interface it inherits from the `Adaptor` base class, thus offering a common interface to dissimilar functionality and simplifying the task of configuring a GTS server.

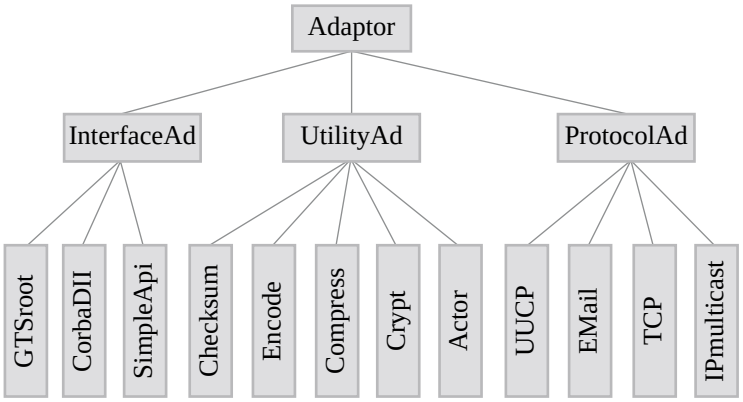

Figure 4.4: Adaptor inheritance hierarchy.

GTS was originally developed to support Beyond-Sniff [BKMS95], a novel cooperative software engineering environment in development at the Information Technology Laboratory of Union Bank of Switzerland (UBILAB). More than 95% of the GTS' program code is protocol-independent and portable. Multicast, ordering of events, and message spooling is implemented in a generic way. To incorporate a real communication protocol,

the programmer has to implement a simple protocol adaptor for it. The minimum service a protocol adaptor must provide is unreliable unicast.

4.1.2 Guide-2

GUIDE-2 (Grenoble University's Integrated Distributed Environment) is a component of the aforementioned COMANDOS environment. The main goals of GUIDE-2 are to provide generic support for object-oriented languages and to exhibit a modular and configurable system structure [Kra93]. GUIDE-2 aims at supporting cooperative applications running within one LAN. The GUIDE-2 *Virtual Machine* [CFH+93, Kra93] (Figure 4.5) shows the generic run-time system interface on which object-oriented, distributed applications are based. Due to this flexible system design, different programming languages and operating systems can be accomodated in the GUIDE-2 architecture.

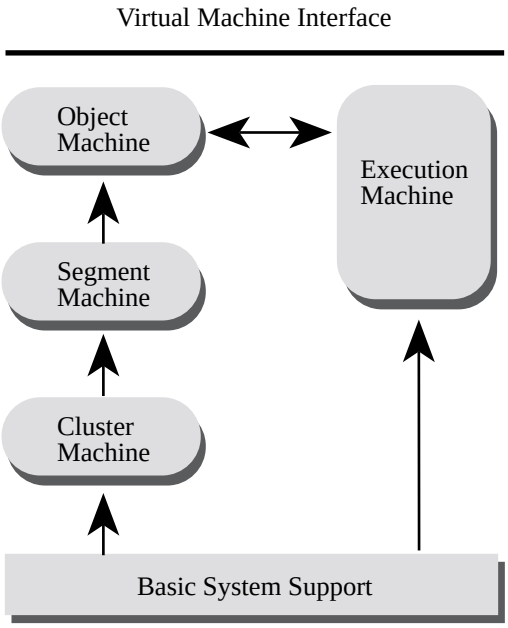

Figure 4.5: The GUIDE-2 architecture.

The *Object Machine* offers generic primitives that allow applications to instantiate objects, to perform object invocations, and so forth. The Object Machine itself is based on the *Segment Machine*, which provides the basic mechanisms for mapping memory segments to different addresses and for binding object names at execution time. Groups of segments, called clusters, are the unit of sharing. The *Cluster Machine* supplies the mechanisms for managing clusters in permanent storage. Finally, the *Execution Machine* implements threads of control and diffusion mechanisms which allow a thread to cross a context boundary in order to run on another machine.

4.1.3 Horus

As mentioned in Section 1.4.4, the architecture of HORUS is influenced by microkernel design principles, in that primitives such as threads and memory allocation are provided by a lean run-time system, and services such as name serving and fault detection are based on it.

HORUS actually runs on a variety of UNIX and non-UNIX operating systems and on communication protocols such as UDP, TCP, raw IP, IP-multicast [Dee89], MACH messages [Dra90], and AMOEBA FLIP [KvRvST93]. This flexibility is achieved by employing a layered system design, where functionality such as reliable multicast or threads are supplied by a portability layer, called the *Multicast Transport Service* (MUTS) [vR93a]. The interface of the MUTS is invariable and does not depend on the underlying operating system (Figure 4.6). Machine-dependent modules, which are part of the MUTS, are used to map the generic MUTS operations onto the interfaces of the underlying operating system and communication infrastructure.

4.1.4 Panda

The PANDA project [BRH+93] in development at Vrije Universiteit Amsterdam, aims at providing generic run-time support for parallel programming languages. Again, the key idea is to devise an invariable interface to threads, RPC, and totally-ordered group communication for a variety of operating systems and communication protocols, and to base programming languages like ORCA [Bal90] and LINDA [CGA86] on PANDA. Applications implemented on PANDA would thus run directly on all the platforms it is

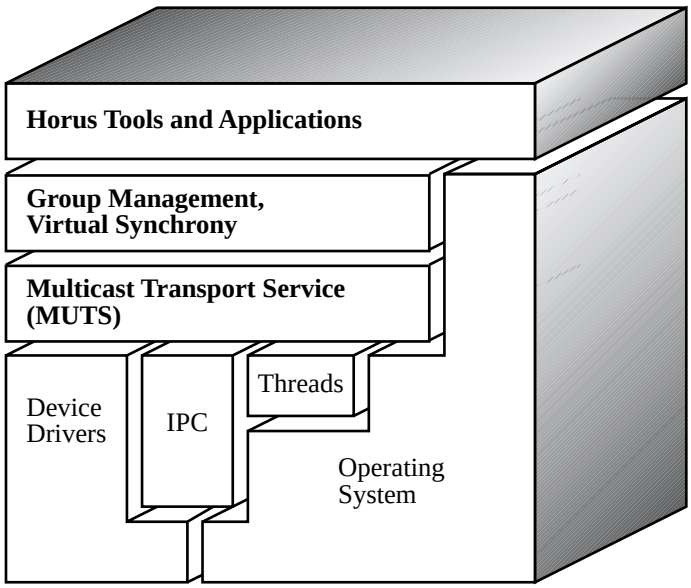

Figure 4.6: The architecture of the HORUS toolkit.

configurable for.

PANDA is structured in two layers to ensure portability. The upper layer, which is the one application programmers are confronted with, offers a generic application programming interface for the handling of messages, RPCs, threads, and multicast. The lower layer encapsulates the peculiarities of the underlying operating system and communication protocols. The PANDA programmer is thus hardly concerned whether the underlying system is a cluster of workstations, a vector supercomputer, or a transputer.

4.1.5 Vanilla FS

The VANILLA file system (VFS) [Maf94b] aims at integrating different media types (RAM, harddisks, WORM devices, CD ROM, streamer devices, etc.) into a uniform, virtual storage. Filesystems can be built by adapting and reusing the components of the VFS C++ class library. Most of the file system program code is device-independent and contained in the virtual base class `Storage` (Figure 4.7). Real, device-dependent VFS storages

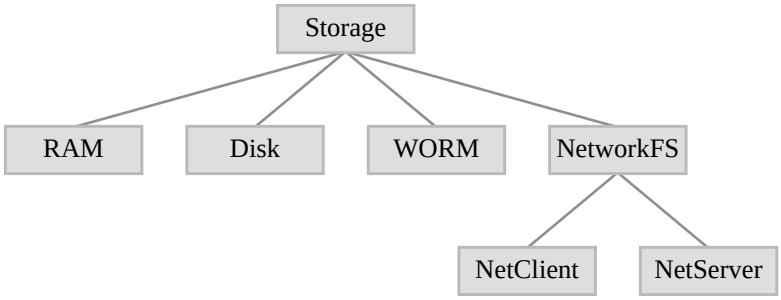

Figure 4.7: The VANILLA FS class hierarchy.

are realized as subclasses of class `Storage`, and only a small set of methods containing the device-dependent code must therefore be implemented. The methods mainly perform low-level reads and low-level writes on the underlying physical storage. `NetworkFS` objects are used to integrate remote devices into a VFS. Storage architectures like hierarchies, mirrors, and data-striping can be built by simply plugging together VFS objects. The following code fragment demonstrates how a storage hierarchy consisting of RAM cache, a harddisk, and a WORM device is configured.

Example 1:

```
RAM ram(1024 * 1024);                  // 1 mb of RAM cache
Disk disk("/dev/rsd1g");               // harddisk storage
WORM worm("/dev/worm1");               // WORM storage

ram.attach(disk).attach(worm);         // a storage hierarchy
                                       // à la Plan 9

File f = ram.open(...);                // access files on it.
...
```

4.1.6 Windows NT

Flexible system design principles are employed in commercial operating systems as well. Microsoft's WINDOWS NT [Cus93], for instance, places a *Hardware Abstraction Layer (HAL)* between the NT executive and the hardware platform on which WINDOWS NT is running. The hardware platform may comprise one or several INTEL, MIPS, or ALPHA processors. The HAL offers an invariable interface to hide hardware-dependent details such as I/O interfaces, interrupt controllers, and multiprocessor communication mechanisms. The components of the NT executive maintain portability and flexibility by avoiding direct interaction with the underlying hardware. Instead, HAL routines are called when the NT executive needs platform-dependent information.

4.2 The Adaptor Model

The system designs just described have in common that system-dependent functionality is hidden behind a portability veneer, and that machine-specific adaptor modules are employed to map the veneer onto the interfaces of the underlying system software. In this section we propose a system design model which is a generalization of the aforementioned approaches, and which makes possible the implementation of portable and yet efficient run-time systems. The suggested system design is called the *adaptor model* [Maf94a]. In the context of this thesis, the adaptor model serves to structure the run-time system of an EOM toolkit.

The adaptor model provides the illusion of a *virtual machine (VM)*, by which the upper layers of an EOM toolkit are always confronted with the same set of communication primitives, independent of the underlying RM. The programmatical interface is encapsulated in an abstract class, called the VM interface. To map these generic operations onto the operations provided by the RM, an *adaptor object* is used. The adaptor object cleanly encapsulates all of the non-portable code. As depicted in Figure 4.8, a system configuration may comprise several adaptor objects, one for each RM to be supported. The interface of the VM is always the same, regardless of the underlying RM. The toolkit communicates with the VM by invoking the operations of this interface (*downcalls*), and the VM notifies the toolkit of important events using *upcalls* [Cla85].

Example 2:

```
class VirtualMachine {
 public:
   VirtualMachine();
   VirtualMachine(const VirtualMachine&);
   VirtualMachine& operator=(const VirtualMachine&);
   virtual ~VirtualMachine();

   virtual operation1();
   virtual operation2();
   ...
};

class AdaptorX: public VirtualMachine {
   AdaptorX();
   AdaptorX(const AdaptorX&);
   AdaptorX& operator=(const AdaptorX&);
   ~AdaptorX();

   operation1();
   operation2();
   ...
};
```

Example 2 demonstrates the virtual machine C++ canonical form. The

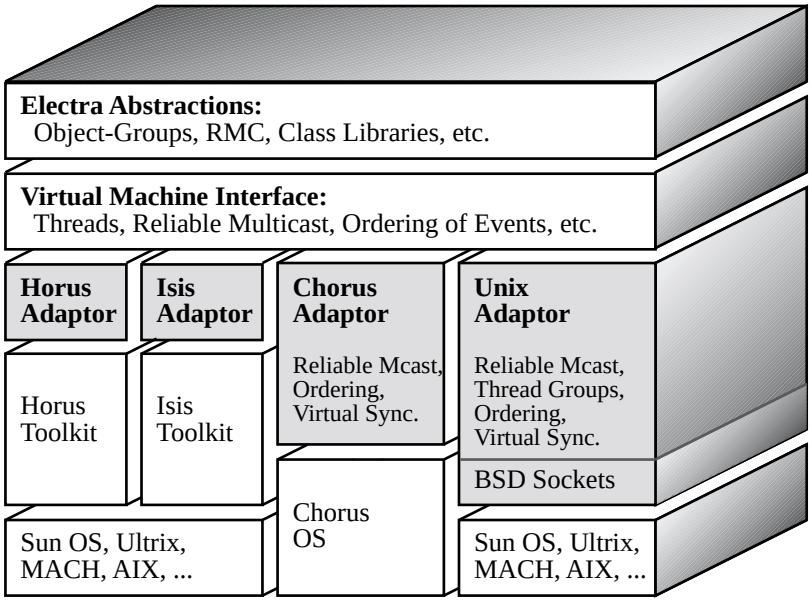

Figure 4.8: A sample ELECTRA configuration with adaptors for HORUS, ISIS, CHORUS, and UNIX.

adaptor classes are descendants of the virtual machine class. Note that the virtual machine class itself may provide generic functionality common to all adaptors. The advantages of the proposed adaptor model are:

- **Flexibility**: Run-time systems structured along the adaptor model can be reconfigured easily.
- **Portability**: Applications which are restricted to use the operations provided by the virtual machine layer will run unmodified on all RMs with an adaptor.
- **Software Quality**: System-dependent code is no longer scattered over applications but concentrated in an adaptor. This clean encapsulation facilitates the implementation and the maintenance of distributed systems considerably.
- **Code Reuse**: New adaptor objects can be created out of existing adaptors by means of class inheritance. To implement an adaptor for a specific operating system, an adaptor for a similar operating system can be taken as a starting point.
- **Cleaner System Design**: When implementing a system along the adaptor model, one has to think in terms of similarities and dissimilarities between the RMs which are to be supported. Our experience tells us that this leads to better structured and better portable software.

One might argue that the proposed model imposes an unacceptable computational overhead since encapsulation and modularization result in the loss of information important for good performance [SRC84]. Nevertheless, the application experiences we report in Section 5.2.5, and experiences with HORUS [vRB94] and with the *x*-kernel [PHOH90] indicate that high performance is achievable also by a layered and flexible system.

4.3 The Electra Run-Time System

As part of this dissertation an implementation of EOM, called the ELECTRA toolkit, was devised by following the adaptor model. The ELECTRA toolkit is an object request broker implemented in C++ by following the CORBA

specifications [Dig93]. Chapter 5 describes the toolkit in more detail. This section focusses on the employed system design. In the following, we shall use the short term ELECTRA to refer to our toolkit.

4.3.1 Architectural Model

The main objective of ELECTRA is to allow for object-oriented, distributed programming on various state-of-the-art RMs, such as CONSUL, DELTA-4, HORUS, ISIS, and TRANSIS. Although ELECTRA could be configured to run directly atop of contemporary operating systems such as UNIX and WINDOWS NT, the author prefers a solution where one of the aforementioned RMs is employed. Implementing an adaptor based on the UNIX or WINDOWS NT primitives is theoretically possible but requires man-years of effort and the implementation of functionality similar to the one provided by ISIS. Moreover, in our opinion, an object request broker should be built on sound system support and not directly on the primitives provided by contemporary operating systems, such as UNIX sockets.

4.3.2 Virtual Machine Interface

Now, we describe a VM interface which we found adequate for MUTS, HORUS, and ISIS, and which we believe adequate for other platforms as well. As previously mentioned, the VM interface is encapsulated in an abstract C++ class. Adaptor objects are instances of subclasses of the VM class. The VM class offers methods for message passing, threads, thread synchronization, multicast, group management, and so forth. Appendix B contains the VM class definition for the present version of the ELECTRA toolkit, whereas the next C++ code examples represent fragments of it. ELECTRA's run-time is implemented by calling methods of the VM class, thus without directly accessing the RM.

Startup and Shutdown

The following methods are provided to trigger initialization and shutdown activities in the underlying RM:

Example 3:

```
virtual ORBStatus machInit();
virtual ORBStatus machQuit();
virtual ORBStatus electraInit(const AppPolicy&);
virtual ORBStatus electraQuit();
```

Most methods return an `ORBStatus` object encapsulating information on the success of an operation. The `machInit` method is employed by ELECTRA when it starts up and initializes the underlying RM. The `machQuit` method invokes operations to flush message buffers and to shut down the RM.

Since different RMs also provide different ordering semantics, group management protocols, and so forth, the ELECTRA programmers must pass an `AppPolicy` object to the adaptor before they can interact with it. Application-specific requirements, such as the ordering semantics to employ, are specified by an `AppPolicy` object. For example, an application programmer might know in advance that her application requires Virtual Synchrony and specifies an appropriate `AppPolicy` object. If, at a later point, ELECTRA is reconfigured for a RM supporting unordered multicast only, the call to `electraInit` will return an error status. If an application requiring unordered multicast only is run on a RM which implements the Virtual Synchrony model, however, `electraInit` will return successfully. This way, the `electraInit` method knows about the attributes of the underlying RM and contains a set of simple rules which check whether an application policy is compatible with the RM. In general, policy objects and rules deciding on the compatibility of application policies are an important part of a generic and efficient system design, since they allow the selection of low-level primitives appropriate for a specific application in a portable and expressive way.

Finally, the `electraQuit` method is issued by the programmer when she wishes to terminate the application. `electraQuit` causes various ELECTRA specific shutdown and cleanup activities, and a call to `machQuit`. Note that normally the ELECTRA programmer is not confronted with the VM operations. Programmers access them through the CORBA programming interfaces we will describe in Chapter 5.

Entities

Communication endpoints are identified by instances of class `Entity`:

Example 4:

```
class AdaptorData {
 public:
   AdaptorData();
   AdaptorData(char *priv, int privSize);
   AdaptorData(const AdaptorData&);
   AdaptorData& operator=(const AdaptorData&);
   virtual ~AdaptorData();

   void setPriv(char *priv, int privSize);
   int getPrivSize() const;
   char *getPriv() const;

protected:
   char *priv_;
   int privSize_;
};
```

Example 5:

```
class Entity:  public AdaptorData {
 public:
   ...
   boolean isLocal();
   boolean isEndpoint();
   boolean isGroup();
   boolean isRawGroup();
   ...
};
```

Addressing information specific to the RM is referenced through the `priv_` pointer of the `AdaptorData` class. `AdaptorData` is the base-class of ELECTRA classes which maintain RM specific information. No meaning is associated with the `priv_` data by the ELECTRA layers above the VM.

Only the adaptor and the RM know about the meaning of the data. For instance, with HORUS used as the RM, the `Entity::priv_`members point to `eid_t` [vR93a] or to `sockaddr_horus` [The94a] structures, depending on which HORUS interface is employed. With ISIS as RM they point to ISIS `address` structures [Isi92]. This is how ELECTRA handles endpoint addresses independently of the low-level transport service. The VM interface defines a small set of methods for managing `Entity` objects:

Example 6:

```
virtual ORBStatus entityCreate(Entity&,
   const ProtocolPolicy&);
virtual ORBStatus entityDestroy(Entity&);
virtual boolean entityEqual(const Entity&, const Entity&);
```

The method `entityCreate` generates a low-level communication address and attaches it to the `Entity` object. By the `ProtocolPolicy` object, requirements regarding the protocol, the importance of the entity and so forth are specified. To tell the VM that an `Entity` is no longer needed ELECTRA issues the `entityDestroy` operation. Whether two `Entity` objects belong to the same communication endpoint is checked with the `entityEqual` method. Again, ELECTRA programmers do not normally deal with `Entity` objects and the related VM methods directly. Instead, CORBA object references are employed to encapsulate entities.

Light-Weight Processes

Example 7:

```
class Thread:  public AdaptorData {
 public:
   ...
   tUpcall getUpcall() const;
   const void *getEnv() const;
   tPrio getPrio() const;
   int getStack() const;
   ...
};

virtual ORBStatus threadDeclare(Thread&, tUpcall f,
   void *env, tPrio, int stack);
virtual ORBStatus threadCreate(Thread&, void *param);
virtual ORBStatus threadDestroy(Thread&);
```

Since ELECTRA supports asynchronous communication, upcalls, and several threads of execution per object, an interface to light-weight processes (threads) is part of the VM. Associated with each thread is a `tUpcall` variable[2] which identifies a C++ procedure or static method. When `threadCreate` is called, the upcall is invoked with its own thread of execution and obtains two `void`-pointers as its arguments. One argument is the `env` pointer which was specified by `threadDeclare`, the other argument is the `param` pointer specified by `threadCreate`. A scheduling priority is assigned by the `tPrio` parameter, and the `stack` parameter defines the thread's stackspace. In analogy to `Entity` objects, the adaptor attaches information specific to the underlying thread package to the `priv_`instance variable.

Here, the VM tries to provide a simple interface which allows existing thread packages to be incorporated easily. Yet, the interface is powerful enough to support the ELECTRA mechanisms which require threads. Synchronization between threads is achieved by the methods we will describe next.

[2]Defined as `typedef void (*tUpcall)(void *, void *)`.

Synchronization Primitives

Semaphores [Dij65] act as our fundamental synchronization primitive. More complicated primitives, event counters and *promises* (Section 3.4) for instance, are implemented with semaphores. Class `Sema` does not need to add functionality to class `AdaptorData`:

Example 8:

```
class Sema:  public AdaptorData {
};

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

A semaphore is created and initially set to `value` by the `semaCreate` operation. `semaDec` decrements the semaphore and blocks the issuing thread if the semaphore's value was zero. `semaInc` increments a semaphore. If the semaphore's value was zero, and some thread is blocked trying to decrement the semaphore, the thread is resumed. Finally, `semaDestroy` notifies the adaptor that a semaphore is no longer needed.

Message-Passing

RMC requires operations for reliable, non-blocking message passing. ELECTRA uses the same message-passing interface, regardless of whether an entity belongs to a singleton object or to a group. We omit the definition of class `Message` since it is simple and mainly maintains a pointer to a data buffer. Subclasses of `Message` are provided which create and maintain messages with a special layout, or to offer methods which allow for the concatenation of messages, etc.

Example 9:

```
virtual ORBStatus msgSend(Entity& destination,
   const Message&, tUpcall sendDone, void *env);
virtual ORBStatus msgReceive(Entity& listenOn,
   tUpcall receive, void *env);
virtual ORBStatus msgReceiveDone(void *);
```

To deliver an RMC to a specific singleton- or group-destination, ELECTRA invokes `msgSend`. When resources associated with the message can be released, the RM will invoke the `sendDone` upcall thread, and `env` will act as parameter for the upcall. The method `msgReceive` registers an upcall which is invoked whenever a message arrives for entity `listenOn`. ELECTRA typically calls `msgReceive` to register a thread which handles incoming RMC requests and reply messages. When a thread is finished processing a received message, it calls `msgReceiveDone`.

Group Management

RMs like HORUS, ISIS, and TRANSIS provide sophisticated group membership protocols similar to the one we described in Section 3.7.2. Consistent group membership management is an important component of EOM, it is accessible by the following VM operations:

Example 10:

```
virtual ORBStatus grpCreate(Entity& group,
   const ProtocolPolicy& policy);
virtual ORBStatus grpJoin(Entity& group, Entity& newMember,
   tUpcall monitor, void *,
   tUpcall getState, void *,
   tUpcall setState, void *);
virtual ORBStatus grpLeave(Entity& group, Entity& exMember);
virtual ORBStatus grpDestroy(Entity& group);
```

The method `grpCreate` is used to create an empty group of `Entity` objects to which a message can be multicast with the `msgSend` operation in Example 9. The `group` object identifies the group and serves as destination for the multicasts. A `ProtocolPolicy` object specifies parameters

like the required ordering of the messages delivered to the members, e.g., unordered, causally ordered, or totally ordered multicast, the kind of group membership management to employ, whether group membership changes shall be synchronized with communication, and so forth.

Objects are inserted into a group by the `grpJoin` operation. When a new view is installed for the group, the RM invokes the `monitor` upcall. State transfer to a new group member is accomplished with the help of the `getState` and `setState` upcalls. `grpLeave` is called to remove an entity from a group. Finally, a group is destroyed by the `grpDestroy` operation. Whenever a member joins a group, leaves it or fails, the new view is automatically propagated to all of the involved adaptors by the underlying RM.

Failure Monitoring

Since certain applications are interested in the information the failure suspector service gathers, an interface for this service is provided:

Example 11:

```
class Monitor:  public AdaptorData {
};

virtual ORBStatus monMonitor(Monitor&, const Entity&,
   tUpcall mon, void *env);
virtual ORBStatus monCancel(Monitor&);
```

Monitors can be set up for both singleton and group entities. In the case of a group entity, the monitor is invoked on a complete failure of the group. When a monitored entity is believed faulty, the `tUpcall` thread is invoked. The `mon` thread is invoked with a pointer to a structure holding information on the suspicious entity.

4.3.3 VM Class Library

A sophisticated ELECTRA configuration will provide a variety of adaptor objects with inheritance relationships as depicted by Figure 4.9. In our experience, the proposed VM primitives are powerful enough to support

the EOM requirements and also general enough to be realizable on most of today's state-of-the-art platforms for reliable distributed programming.

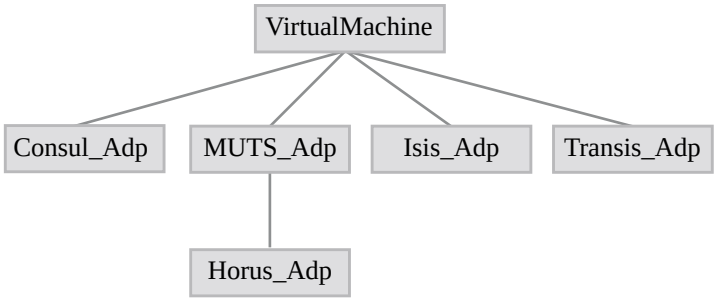

Figure 4.9: Virtual Machine inheritance graph.

4.3.4 Towards an Integration Framework

Another advantage of the adaptor model is that it lets a toolkit use several transport protocols simultaneously. For instance, an ELECTRA configuration could consist of a main adaptor residing on the ISIS toolkit, and of a simple adaptor which maps the message-passing operations onto a reliable stream protocol such as TCP by leaving the remaining VM methods unimplemented. Now, ELECTRA proxy objects could be instructed to use the TCP adaptor for the delivery of certain RMCs, and the ISIS adaptor for the remaining operations.

Let's consider a banking application where customer data must be retrieved from a mainframe running an old legacy information system and evaluated in parallel on a cluster of modern RISC workstations. In ELECTRA, the parallel evaluation application could be run using ISIS as RM. Most likely, ISIS will not run on the mainframe, but probably the data could be retrieved from there by a public data network protocol like X.25. In analogy to the former example, an adaptor object could now be elaborated to map the message-passing primitives onto calls to an X.25 programming library. The rest of the adaptor's methods would be left unimplemented. On the host's side, a program would be realized[3] which interprets the incoming

[3]Nevertheless, if the mainframe provides a C++ compiler, the VM and the layers on top of it could be installed.

RMCs, issues queries on the customer database, and returns the requested data. Now, a well structured, object-oriented ELECTRA application can be realized which transparently uses X.25 to retrieve data from the bank's mainframe, and which employs ISIS to perform a parallel evaluation on the workstations.

By invoking RMCs such as `getCustomerData`, the ELECTRA programmer here has the illusion of communicating with a remote object, although a legacy mainframe resides behind this abstract interface. This approach is well-suited for accessing legacy software [Bro92] from within a state-of-the-art distributed application.

In a similar way, adaptor objects could be implemented to interconnect different RMs, and to practice object-oriented, distributed programming on the resulting, virtual platform. Of course, the inter-platform communication flows will not respect causality, and it is not possible to have object-group members spread over different RMs. To arrive at an integration framework which transparently integrates different platforms and poses no restriction on the residence of group members, one would have to realize the event ordering protocol, group membership management, and reliable multicast in ELECTRA.

4.4 Summary

This chapter presented an object-oriented, flexible system design which we consider adequate for implementing an EOM-compliant run-time system on platforms such as AMOEBA, CHORUS, CONSUL, DELTA-4, HORUS, ISIS, and TRANSIS. To exemplify the proposed system design, the architecture of the ELECTRA toolkit was described. High flexibility is achieved by ELECTRA's *virtual machine* abstraction. The virtual machine interface defines a reduced set of simple primitives, mainly for the handling of threads, for message-passing, multicast, thread synchronization, and group management. This interface is independent of the underlying communication infrastructure. An *adaptor object* is used to map the primitives onto operations provided by the underlying *real machine*, thus cleanly encapsulating the program-code specific to the real machine. Applications interact with the virtual machine by means of downcalls and upcalls. We named this approach the *adaptor model*. Finally, we pointed out how the adaptor model can help in accessing legacy software from within a state-of-the-art

distributed application. In our experience, the adaptor model allows for design and realization of highly configurable, portable and efficient run-time systems.

Chapter 5

The Electra Toolkit

This chapter deals with the ELECTRA toolkit, an implementation of EOM carried out as part of this Ph.D. project. The ELECTRA toolkit follows the CORBA standard, it embodies the flexible system design principles we described in the previous chapter, and fulfills the EOM requirements proposed in Chapter 3.

5.1 Design Goals

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

As was previously noted, platforms such as CONSUL, DELTA-4, HORUS, ISIS, and TRANSIS provide system support for implementing dependable distributed systems but have the disadvantage of a rather low-level programming interface. These toolkits essentially supply a rich set of C procedures enabling programmers to create and to group communication endpoints, to issue reliable multicasts, and so forth. Although the provided mechanisms are powerful, people lacking years of experiences with such toolkits often find it too hard to develop distributed applications. Furthermore, software written with one of these toolkits is difficult to reuse or to port to another toolkit.

On the other side, OODP often is hailed as a panacea, though most of the OODP environments available today lack the fundamental tools needed for building well-functioning distributed software. Most manufacturers of OODP systems tend to focus their efforts on programming abstractions, class libraries, network management tools, and on supporting as many object-oriented programming languages as possible. However, the fundamental problems of asynchronous distributed systems, namely partial failures and violation of causal delivery, are not addressed by most of the present OODP environments.

We believe that OODP can help in building robust, reusable, and efficient distributed applications in heterogeneous environments. This happens on the condition that the abstractions are built on system support as provided by the aforementioned platforms, and by adhering to the EOM requirements. Therefore, the development of the ELECTRA toolkit is guided by the following design goals:

- To provide adequate system support for OODP,
- tracking current research in the area of robust, asynchronous distributed systems,
- to follow structuring paradigms like process groups and execution models such as Virtual Synchrony,
- to adhere to the EOM requirements,
- and to respect open standards for OODP.

As part of the author's dissertation two versions of the ELECTRA toolkit were developed: a first rudimentary prototype to estimate the feasibility of the project [Maf93b] and the real ELECTRA toolkit [Maf94a, Maf94c]. The prototype was completed and its performance assessed. The real toolkit is in an advanced stage of development.

5.2 The Electra Prototype

The prototype implementation consisted of the following parts:

- A run-time system residing on HORUS and written in the C++ programming language

- Simple class libraries for active and for passive objects
- An interface definition language, called the ELECTRA Service Definition Language (SDL)
- A name server and a remote instantiation facility, the latter allowing the creation of active objects on remote hosts

Synchronous, asynchronous, and deferred-synchronous point-to-point RMC was supported. A limited support for object-group communication was also provided. Only EOM requirements 1, 2, and 4 were fulfilled by the prototype.

5.2.1 Run-Time System

The run-time system only made limited use of the HORUS primitives, in that mainly the thread, the RPC, and the multicast module were used. HORUS' group-view management and Virtual Synchrony facilities were not exploited. Furthermore, the design of the run-time did not follow the adaptor model, in that HORUS primitives were directly accessed by the run-time.

5.2.2 Class Libraries

Two different class libraries were implemented. One for passive objects (data holders) such as matrices, arrays, strings, lists and so forth, and another for active objects, such as the name server, and the instantiation facility.

5.2.3 Service Declaration Language

Active objects were declared in ELECTRA SDL. The following code fragment exemplifies the SDL specification of a replicated file server:

Example 1:

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

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

In this example, `read` operations are not multicast since the file is not altered by them. `write` and `unlink` operations are multicast using the AMCAST protocol, thus making sure that subsequent operations always arrive in the same order at all replicas. The reason why we devised our own definition language instead of using an existing one, for example CORBA-IDL [Dig93], was that when the prototype was designed we regarded IDL declarators like `amcast` as indispensable. Section 5.2.6 discusses the lessons we learned in the process.

5.2.4 Services

Naming Facility

In the prototype, the name space was partitioned into *administrative domains*. A domain usually comprised one LAN, each domain had its own *location trader* object with a predefined network address known to all objects in the domain. When an object was instantiated, its human-readable name was specified by the programmer. This name was installed in the local trader by the generated stub code. Applications used this name to bind to the just created object. Names obeyed the form <Domain>:<Path>. For example, the name "ifi.unizh.ch:/pub/fileserver_9" referred to a file server object in the "ifi.unizh.ch" domain.

Lookup-requests for objects outside the local domain were also directed to the local trader. Each trader maintained references to traders in other domains, and forwarded the lookup requests accordingly. Analogously to name serving in the INTERNET [Moc87], it was possible to configure a large network of traders, each serving its own administrative domain and maintaining "pointers" to traders in other domains.

Remote Instantiation Facility

On remote hosts, objects could be dynamically created, a concept we called *mushrooming*. On each host that is part of an ELECTRA configuration, a local service called the *mushroomer* was started at boot-time. Mushrooming of a remote object was performed by proceeding along the following steps:

- By consulting an internal table containing the names of the objects running locally, the mushroomer at the target host found out whether an instance of the requested object-interface had already been created.
- If no instance was running, the mushroomer obtained the path of the object's executable file. Then, the mushroomer loaded the executable, a time-consuming process. Finally, the new object was instantiated and registered with its trader. The mushrooming process ended here.
- Otherwise, the executable binary for the requested object had been loaded previously. In this case, an object was simply instantiated. The new object was then registered with the trader as above.

5.2.5 Performance Evaluation

To get a first impression of the scalability of ELECTRA applications, a partial differential equation solver (PDES) was implemented in the form of distributed ELECTRA objects cooperating by point-to-point RMC. The PDES was originally implemented on PVM 2.4.0, SCA NETWORK LINDA[1] 2.4.7 [Pan], POSYBL-LINDA[2] 1.102 [Sch90], and on a PARSYTEC MC-2/32-2 transputer[3] as part of the research project described in [CS93a]. For our purposes, porting the PDES application to ELECTRA was sufficient.

The transputer-based implementation served as indicator of the speedup that could be achieved if communication went through dedicated links. All other experiments were performed on the same workstation network, which consisted of 38 SUN SPARCSTATION 1 computers interconnected by a 10 Mbps ETHERNET [IEE85]. The experiments were carried out on a dedicated network, which means that users were prevented from using the system, and that services such as FTP, net news, and e-mail were shut down.

[1] from Scientific Computing Associates (SCA), Inc., New Haven, CT.
[2] from the Department of Computer Science, University of Crete, Greece.
[3] containing 32 T-800 processors.

More specificly, the considered application was a PDES from the theory of heat conduction. Imagine a rectangular plate with a certain initial temperature situation at its boundaries. By applying the PDES, we simulated how the heat situation on the whole plate changes over time. The application was parallelized by partitioning the plate into vertical slices, and by assigning each slice to a worker-object on a different workstation. Each worker-object computed the heat situation on its slice and communicated the left slice-boundary to its left neighbor and the right boundary to its right neighbor (supposing the worker-object was not in an outermost position) using non-blocking communication. Then, the worker-object suspended the computation until it received the left and the right boundary from its neighbors. After that, the worker-object computed the next time iteration, and so on. The outermost worker-objects used a constant boundary condition. Figure 5.1 depicts the described situation. For more details on the application see [CS93a].

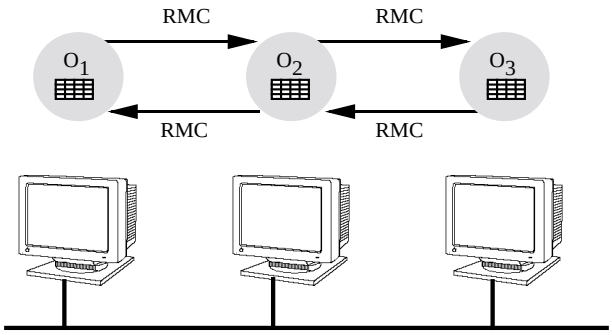

Figure 5.1: Interaction patterns between the workstations. O_i denotes a worker-object encapsulating a slice of the plate.

The ELECTRA-based implementation of the PDES used blocking and non-blocking point-to-point RMCs to communicate the boundaries. C++ classes were employed for the plate, for the slices, and for the worker-objects. Message-passing was used in the PVM-based application. Within the LINDA application, the tuple space served to exchange the boundaries. Finally, the transputer version employed the OCCAM message-passing primitives to transfer the slice boundaries. To compile the ELECTRA application, we used the GNU C++ compiler version 2.3.3. The POSYBL, SCA NETWORK LINDA, and PVM application were compiled with the SUN OS 4.1.3

C compiler.

Figure 5.2 depicts the speedup graph for the various implementations of the PDES. We chose a plate size of 3800 per 100 points and computed 500 time iterations. Each workstation was assigned a slice of the same size. The graphs associated with the transputer, with the PVM, and with the LINDA implementation are results of the experiments described in [CS93a], which were carried out on the same workstation network and under the same conditions as the ELECTRA experiment.

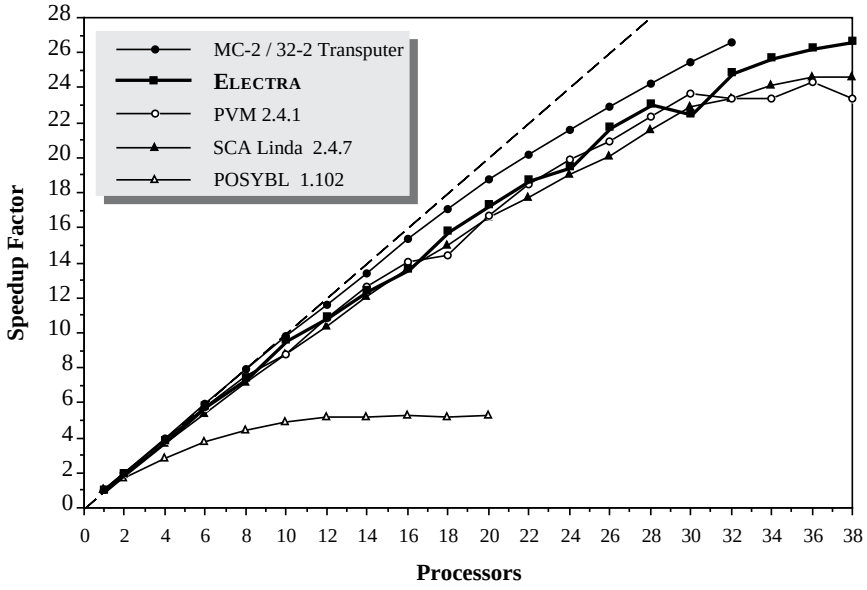

Figure 5.2: Speedup of the distributed PDES application.

To obtain a distinct speedup value, we measured the total execution time of the application for the considered number of workstations. Each experiment was carried out three times, and the average execution time was then determined. Execution times did not include the time to start up the worker-objects on the workstations. Table 5.1 contains the average execution time and the speedup depending on the number of workstations.

The speedup curves show that among the PDES application run on a workstation network, ELECTRA achieved the highest speedup with 32 to

Work-stations	POSYBL		SCA Linda		PVM		Electra	
	T	S	T	S	T	S	T	S
1	1264.3	1.0	1264.3	1.0	1264.3	1.0	1285.0	1.0
2	737.2	1.7	662.2	1.9	648.0	2.0	654.4	2.0
4	442.6	2.9	342.6	3.7	328.0	3.9	333.5	3.9
6	339.3	3.7	235.5	5.4	219.0	5.8	219.5	5.9
8	284.6	4.4	175.8	7.2	168.4	7.5	171.6	7.5
10	260.2	4.9	144.3	8.8	143.6	8.8	134.1	9.6
12	244.7	5.2	122.1	10.4	116.6	10.8	118.0	10.9
14	242.7	5.2	104.5	12.1	100.1	12.6	103.6	12.4
16	239.5	5.3	92.8	13.6	90.0	14.0	94.3	13.6
18	242.6	5.2	84.5	15.0	87.5	14.4	81.1	15.8
20	241.6	5.2	76.0	16.6	75.8	16.7	74.0	17.4
22	—	—	71.5	17.7	68.5	18.5	68.4	18.8
24	—	—	66.5	19.0	63.6	19.9	65.7	19.6
26	—	—	63.1	20.0	60.5	20.9	59.0	21.8
28	—	—	58.5	21.6	56.7	22.3	55.7	23.1
30	—	—	55.1	23.0	53.5	23.6	57.1	22.5
32	—	—	54.0	23.4	54.0	23.4	51.6	24.9
34	—	—	52.4	24.1	54.0	23.4	50.0	25.7
36	—	—	51.4	24.6	52.0	24.3	48.8	26.3
38	—	—	51.3	24.6	54.0	23.4	48.2	26.7

Table 5.1: Execution wallclock-time in seconds (T) and speedup-factor (S) in dependence of the number of workstations.

38 nodes[4]. With 38 workstations, ELECTRA achieved a speedup-factor of 26.7, SCA NETWORK LINDA of 24.6, and PVM of 23.4. This was possible despite of the fact that object-oriented communication was practiced in ELECTRA, that we employed a highly layered system design, and that ELECTRA is conceived for various kinds of directly distributed systems and not only for those aiming at high speedups.

From the results we conclude that HORUS allows for the building of scalable distributed application and that it is suited for a wide application area. Another conclusion is that it is possible to realize an EOM-compliant toolkit providing good performance, which serves to the development of various kinds of distributed applications.

5.2.6 Lessons Learned

In the process of designing and implementing the prototype we learned several lessons:

- **System Design**: As was pointed out in Section 5.2.1, the design of the prototype did not follow the adaptor model in that HORUS operations were directly accessed by the run-time system, and in that calls to HORUS primitives were spread over a large part of the prototype's program code. This led to an inflexible system which was difficult to port and to maintain.

- **Class Library**: The distinction between active and passive objects is desirable for the applications we intend to support. An efficient implementation for ELECTRA objects was possible, we did not feel restricted by the distinction.

- **Service Declaration Language**: We could not confirm our initial assumption that an IDL needs special keywords to support object-groups and group communication. Actually, we found that as much group-support as possible should be provided by employing policy objects, and not by devising special IDL expressions for it. This leads to more flexibility since the parameters of an object-group can be selected and tuned on a per-instance basis, so they can be changed at

[4] the kinks in the ELECTRA graph for 24 and 30 workstations are due to a weakness in the HORUS flow control layer. HORUS' flow control layer has since been improved.

run-time and are not wired into the object-group's interface specification. What an IDL declaration could provide are minimal requirements necessary for the correct functioning of an object-group.

- **Services**: The remote instantiation facility proved indispensable for the implementation of applications which need to create and destroy objects on a large number of workstations. The adopted naming scheme consisting of a domain- and of a local component was found adequate for building applications spawning several LANs. Our prototype implementation of the trader was done in the form of a singleton object with a predefined address known to all applications in its domain. As expected, this centralized service became a single point of failure for the domain. We will consider replicated traders administering a partitioned name space [Lam86] in our future work.

All in all, we were satisfied with the results and experiences we gathered from the prototype. The performance of the prototype was better than the author and his colleagues had expected, and first application experiences proved EOM abstractions such as RMC and object-groups to be profitable. The lessons we learned with the prototype led us to the "real ELECTRA toolkit" we will describe in the next section.

5.3 The Real Electra Toolkit

Based on our experiences with the prototype, the real ELECTRA toolkit is now under development. In addition to the design goals stated in Section 5.1, the following aims are being pursued:

- to provide a flexible system design which allows an easy customization of the toolkit for several platforms and operating systems,
- to structure the run-time system in a proper and object-oriented fashion,
- to make it possible for certain ELECTRA applications to run unmodified on different platforms,
- to replace the ELECTRA SDL compiler with the OMG IDL compiler,
- to adhere to the CORBA standard,

- and to devise an OMG Object Request Broker for the implementation of robust distributed systems.

Like its prototype, ELECTRA is being developed in the C++ programming language [ES91], since this language is in widespread use, since it offers reasonable support of the object-oriented paradigm, and since it permits to write programs with an efficiency close to the one of programs written in C. Moreover, a C++ standard is being developed by the ANSI committee. Presently, ELECTRA applications are implemented in C++ as well, although target languages such as C, Lisp, Objective-C, Self, Smalltalk, or SML could also be supported by enhancing our IDL compiler to generate adequate communication stubs.

5.4 Architecture

Several layers form the ELECTRA architecture (Figure 5.3). The ELECTRA programmer is mainly confronted with IDL specifications, with the Dynamic Invocation Interface (DII), the Static Invocation Interface (SII), the Object Request Broker (ORB), and the Basic Object Adapter (BOA) [Dig93]. These components have been equipped with operations to support group communication.

The Dynamic Invocation Interface is founded on a generic multicast RPC module, which itself is built on the virtual machine (VM) layer described in Section 4.3. Operations of the underlying operating system, which are inherently non-portable or which must be provided by a thread-safe library as part of the real machine, are accessed through the Virtual Operating System (VOS). In the following, we will explain the main architectural components.

5.4.1 Multicast RPC Module

There is a generic RPC module at the heart of the ELECTRA architecture carrying out the low-level delivery of remote object invocations. The main function of the module is to render possible asynchronous RPC to both singleton and group destinations. Synchronous and deferred-synchronous invocation mechanisms are provided by the Dynamic Invocation Interface (Section 5.4.2) residing on top of the RPC module. The main difference

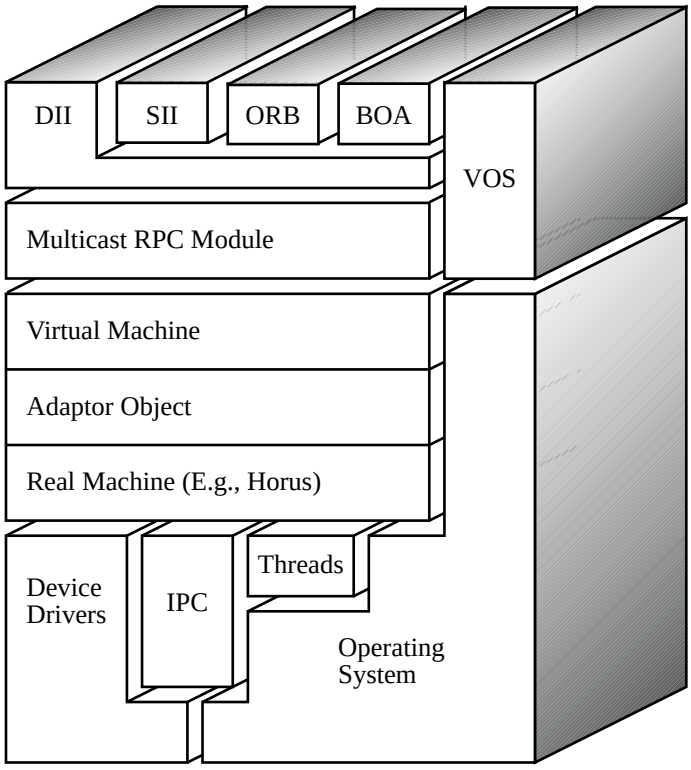

Figure 5.3: The ELECTRA layers.

between an RPC and a remote object invocation is that the former communication paradigm embodies a lower level of abstraction, since only raw messages (unstructured data buffers) are transmitted between communication endpoints. In contrast, a remote object invocation has a signature which obeys the related interface declaration, typechecking of the operation's arguments is performed at compile time, giving the illusion that communication is taking place between objects.

The RPC module handles group membership changes and reply-collation in a way independent of the underlying communication subsystem. The RPC module is based solely on the VM interface. Using upcalls, the VM informs the RPC module of arriving messages, of group membership changes, and of failures. Using downcalls, the RPC allocates communication endpoints, registers receiver-threads, and sends messages. In the present version of ELECTRA, there is only one instance of the RPC module per process, and all objects in a process interact with the same RPC module.

Interface of the RPC Module

The following code fragment contains the interface of our RPC module. Before a sender can submit an RPC, it must obtain an `RpcHandle` object which is used to identify the connection to the destination entity. To represent RPC communication endpoints, the module employs instances of the `Entity` class described in Section 3. When an RPC arrives for a certain destination entity, the module allocates an `RpcReceiveData` object, fills it with the received message and with sender-information, and passes it on to the registered receiver-thread.

Example 2:

```
class RpcLayer {
public:
   RpcLayer();
   RpcLayer(const RpcLayer&);
   RpcLayer& operator=(const RpcLayer&);
   ~RpcLayer();

   Status getHandle(Entity& src, Entity& dst, RpcHandle *&);
   Status releaseHandle(RpcHandle *);
   Status rpcEntryPoint(Entity&, tUpcall);

   Status rpcSend(RpcHandle *, Message *, int numRep,
      tUpcall sendDone, tUpcall gotReply);
   Status rpcReply(RpcReceiveData *, Message *,
      tUpcall sendDone);
   Status rpcDone(RpcReceiveData *);

   static void viewChange(RpcHandle *, int members);
};
```

The `getHandle` method creates an `RpcHandle` for the given source-destination pair, whereas `releaseHandle` deallocates internal resources associated with a handle and closes the underlying connection. To make possible the receipt of RPCs on a certain entity, `rpcEntryPoint` is issued with the entity as one argument. The `tUpcall` argument specifies a procedure to be invoked when an RPC arrives for the entity.

The method `rpcSend` performs an asynchronous RPC to the destination represented by the `RpcHandle`. As soon as resources associated with the message can be released, the `sendDone` upcall is invoked by the VM. When a reply for the RPC arrives, the `gotReply` upcall is invoked. The `numRep` parameter will be explained in the next subsection.

To reply to an RPC, the receiver issues `rpcReply` by specifying a pointer to the `RpcReceiveData` object it obtained from the module after the associated RPC request arrived. When a receiver is finished with an RPC message or when the sender has finished processing a reply, `rpcDone` must be issued so that the module can deallocate the internal resources associated with the call.

Multicast RPC

The `numRep` parameter of the `rpcSend` operation contains one of the following values:

- **1**: The RPC returns exactly one reply, granted that at least one group member remains operational during the call. If the last member fails during the call, an exception is thrown. The RPC module selects the first reply and discards replies which arrive later.
- **MAJORITY**: The module collects replies until the majority of the group members have responded. Membership changes taking place during the call are considered and the value representing the majority is adapted.
- **ALL**: Replies are collected until all operational members have answered. Again, membership changes taking place during the RPC are considered.
- **N**: Any exact number can be specified. If the specified number of replies cannot be met, an exception is thrown.
- **COMPARE**: The RPC module awaits a reply from each operational member. Thereafter, the replies are compared and the most frequent one is returned. If replies disagree, an exception is raised. If the selection is equivocal on differing replies, another exception is raised. In analogy to comparator mechanisms in fault-tolerant hardware [SS92], this "poor man's" approach permits to detect faulty system components and to act accordingly.

As was mentioned before, low-level group management is carried out in the VM. Nevertheless, the RPC module needs to be informed when a membership change occurs. For instance, the number of outstanding replies to await in the case of a `MAJORITY` call needs adaptation after a membership change took place. Therefore, the VM is instructed to invoke the `viewChange` method of the RPC module when a view change occurs.

Interaction with the Virtual Machine

The RPC module and the VM cooperate as described in Table 5.2. The `getHandle` method creates an entity object by issuing `entityCreate`, and

registers an RPC receiver-thread for the entity by calling `msgReceive`. The method `releaseHandle` invokes `entityDestroy` to instruct the VM not to listen on the entity any more. In analogy to the `getHandle` operation, `rpcEntryPoint` also creates an entity and issues a call to `msgReceive`. The `rpcSend` and the `rpcReply` operation both give rise to a call to `msgSend`. Finally, `rpcDone` is mapped onto a call to `msgReceiveDone`.

RpcLayer operation	*VirtualMachine operation*
`getHandle`	`entityCreate`, `msgReceive`
`releaseHandle`	`entityDestroy`
`rpcEntryPoint`	`entityCreate`, `msgReceive`
`rpcSend`	`msgSend`
`rpcReply`	`msgSend`
`rpcDone`	`msgReceiveDone`

Table 5.2: The mapping of `RpcLayer` operations onto `VirtualMachine` operations.

5.4.2 Dynamic Invocation Interface

CORBA offers two means for performing remote object invocations, namely the IDL stubs described in Section 5.4.3, and the Dynamic Invocation Interface (DII). To support EOM, no modification of the CORBA DII is required. Example 3 contains the main component of the DII, which is a class representing a request to be sent to a CORBA object.

Example 3:

```
class Request {
public:
   ...
   Request(const Object& obj, const Identifier& oper,
      Environment& env, const Context& ctx,
      const Typecode& resultT, void *result);
   ...

   void add_arg(const Identifier& name,
      const Typecode& type,
      Arg::Mode mode, void *value);
   void send();
   void send_oneway();
   Boolean poll_response();
   void pend_response();
   void invoke();
   ...
}
```

The parameters of the constructor are used to specify the destination of the invocation (`Object`), the operation to be invoked (`Identifier`), an `Environment` object to hold an exception resulting from the operation, the CORBA context of the operation, and a memory region where the result of the operation will be stored. The `add_arg` operation is used to marshal the arguments of an invocation one by one. `send` performs an asynchronous invocation. To determine whether a remote invocation has been completed, `poll_response` must be invoked. Invoking `pend_response` suspends the calling thread until the `send` operation has been completed. In ELECTRA, the `send_oneway` operation performs the same type of remote invocation as `send`. In contrast to the CORBA specification, ELECTRA `send_oneway` operations are reliable. Finally, `invoke` performs a synchronous invocation during which the caller is suspended until the invocation has been completed.

Interaction with the RPC Module

An ELECTRA object-reference (an instance of class `Object`) encapsulates an `RpcHandle`. The `send`, `send_oneway`, and `invoke` operations are mapped onto a call to `rpcSend`. Object-implementations use an instance of the same `Request` class for submitting a reply. In this case, the `send` operation is mapped onto `rpcReply` instead of `rpcSend`. `invoke` and `pend_response` use semaphores to suspend the caller until the invocation is terminated.

DII operation	*RpcLayer operation*
`send`	`rpcSend` or `rpcReply`
`send_oneway`	`rpcSend`
`invoke`	`rpcSend`

Table 5.3: The mapping of DII operations onto `RpcLayer` operations.

Note:

ELECTRA required no modification to the standard DII interface. The functioning of the ELECTRA DII differs from the CORBA specification in that ELECTRA `send_oneway` operations have *exactly once* semantics when no failure occurs, whereas CORBA `send_oneway` operations only have *best effort* semantics. Thus, CORBA `send_oneway` messages may get lost even when no failures occur, whereas ELECTRA operations can not.

5.4.3 Static Invocation Interface

The ELECTRA IDL compiler translates CORBA-IDL language constructs into C++ code. At the time of writing, the mapping of CORBA IDL specifications onto C++ was not part of the CORBA standard. ELECTRA follows the C++ language mapping proposed by the HYPERDESK Corporation [Hyp93], but the mapping will be changed to accord with the CORBA one as soon as the OMG will have agreed on it.

The C++ client stubs generated by the ELECTRA IDL compiler present access to the IDL-defined operations on an object, they are the primary interface by which programmers access CORBA objects. These stubs are often referred to as the *Static Invocation Interface (SII)*. In ELECTRA, the SII is based on the DII, i.e., the stub code allocates objects of type `Request` to

perform invocations. In Section 5.5 we will address the mapping of CORBA-IDL interface declarations onto object-group invocations. In Section 5.7 we will give examples of how to use the SII.

5.4.4 ORB Interface

The CORBA ORB interface is used mainly to convert object-references to and from string format and to create default-context objects. To support EOM, no modification of the standard ORB interface was necessary. Example 4 contains the C++ declaration of class `ORB`.

Example 4:

```
class ORB {
public:
   String<0> object_to_string(const Object&) const;
   Object string_to_object(const String<0>&) const;
   Context default_context(Environment& env) const;
   ...
};
```

A CORBA object-reference is converted to a string by the `object_to_string` operation, whereas `string_to_object` does the opposite. The functions are used to bring object-references in a form that allows them to be passed through a name server or a file system. `default_context` returns the default context of an operation.

Note:

No modification of the standard ORB interface was required to support the ELECTRA object model.

5.4.5 BOA Interface

The BOA is the primary interface an object-implementation uses to access ORB functionality. In ELECTRA, the BOA interface contains special operations to create groups of CORBA objects, to join, to leave, and to destroy groups.

Example 5:

```
class BOA {
public:
   Object create(const ReferenceData&, const InterfaceDef&,
      const ImplementationDef&);
   void dispose(const Object&);
   ReferenceData id(const Object&);
   void change_implementation(const Object&,
      const ImplementationDef&);
   Principal principal(const Object&,
      const MethodEnvironment&);
   void impl_is_ready(const ImplementationDef&
      = default_impl);
   void deactivate_impl(const ImplementationDef&);
   void obj_is_ready(const Object&,
      const ImplementationDef&);
   void deactivate_obj(const Object&);

   // Electra specific operations:
   //
   void create_group(Object *,
      const GroupPolicy& = default_grouppolicy);
   void create_group(String name,
      const GroupPolicy& = default_grouppolicy);
   void join(const Object *);
   void join(String name);
   void leave(const Object *);
   void leave(String name);
   void destroy_group(Object *);
   void destroy_group(String name);

   virtual void get_state(Sequence <Any>& state);
   virtual void set_state(const Sequence <Any>& newState);
   virtual void view_change(const View&);
}
```

ELECTRA object-implementations are descendants of class BOA, and all of the above operations can thus be issued on any object-implementation.

To create an object-group, the `create_group` operation is issued with an (undefined) object-reference as parameter. To insert an object-implementation into a group, the `join` operation is issued. An object-implementation is extracted from a group by the `leave` operation. Finally, a group can be destroyed by the `destroy_group` operation. For example, a group-reference could be created, converted to a string by the `ORB::object_to_string` operation, and stored in a network file system. Remote processes could retrieve the string, convert it to an object-reference, and issue join operations with the reference as parameter. The overloaded signatures taking a `String` instead of an object-reference use the ELECTRA name server to look up the object-reference registered under `name`, whereas the `String`-version of `create_group` produces a new entry in the name server.

State Transfer

An object joining a group often needs to be initialized with an application-dependent *state*. In active replication, for instance, all members of a replica group must have a synchronized internal state, and newcomers need to obtain the current state before they can participate in the redundant computation. In ELECTRA, state transfer is accomplished with the help of the `BOA::get_state` and `BOA::set_state` methods every object-implementation inherits and thus can overwrite. When an object joins a group, the VM invokes the `get_state` method of an arbitrary group member to obtain its internal state. In collaboration with the underlying group membership protocol this data is transmitted to the newcomer, and its `set_state` operation is invoked. An object-state is represented by a sequence of CORBA `any` types. Furthermore, when a view change occurs, each member's `view_change` operation is automatically invoked. The `View` object contains information on the actual number of members and on which object left or joined the group.

Note:

In addition to the BOA functionality defined in the CORBA standard, ELECTRA provides operations for creating, joining, leaving, and destroying groups of CORBA objects. Further special BOA operations are needed by the virtual machine to obtain and to set the internal state of a group member, and to notify group membership changes.

5.5 C++ Mapping of Object-Group Operations

In this section we will deal with the C++ stubs the ELECTRA IDL compiler produces and explain the mapping of CORBA interfaces onto client stub definitions. C++ code fragments presented assume support for C++ templates [Str88]. The following simple interface serves to demonstrate our mapping scheme:

Example 6:

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

The mapping of CORBA interfaces onto ELECTRA invocation-stubs is summarized as follows:

- Operations with `void` return-type, for instance `op1` and `op3`, are mapped on ELECTRA operations which can be issued synchronously, asynchronously, and deferred-synchronously. The programmer selects an invocation type by a `CORBA::Environment` object. If no `Environment` object is passed on to an operation, a synchronous call is performed.
- CORBA `oneway` operations are issued asynchronously.
- A void-operation containing `out` or `inout` arguments is mapped onto two C++ signatures: one for performing unicasts and *transparent* multicasts, another one for performing *non-transparent* multicasts. In the non-transparent case, `out` and `inout` arguments are mapped onto `CORBA::Sequence` data types. Using the non-transparent invocation form, programmers gain access to result parameters generated by the individual object-group members. Transparent group invocations, on the other hand, fill the first arriving reply into the `out` and `inout` arguments and convey the illusion of communicating with a non-replicated, highly available object.

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

- CORBA operations with a non-void return-type, `op2` for instance, are mapped onto ELECTRA operations which can be issued only synchronously. Applied to an object-group, such operations can be issued in the transparent mode only, where they always return the first arriving response.

By applying these rules, the ELECTRA stub generator translates the interface in Example 6 into the following C++ class definition:

Example 7:

```
class example:  public Object {
public:
   // for declaring upcalls:
   //
   typedef void (*op1_upcall)(float, Environment&);
   typedef void (*op1_upcall_mc)(Sequence<float>&,
      Sequence<Environment>&);
   typedef void (*op3_upcall)(Environment&);

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

   // non-transparent multicast operation signatures:
   //
   void op1(float i, Sequence <float, 0>& o,
      Sequence <Environment, 0>&, op1_upcall_mc = 0);

   // to bind to an object-implementation:
   static example *bind(String,
      Environment& = default_env);

   // Constructors etc.
   ...
};
```

In Example 7, `op1`, `op2`, and `op3` enable transparent communication with singleton objects *and* with object-groups. Issued on an object-group, the first arriving member-reply is assigned to the `out` arguments, `inout` arguments, and to the operation's return value. Replies arriving later will be discarded. The second version of `op1` has a sequence type in the place of its `out` argument, and serves for non-transparent multicast. `op2` can be issued only synchronously and in the non-transparent mode, as it is a

non-void operation. Since op3 contains neither out nor inout arguments, only one signature is generated which can be used for unicast, and for both transparent and non-transparent multicast. To bind to an object-implementation which is registered under a certain string with the name server, the bind operation is invoked.

The Environment::set_numreplies method is needed to specify how many responses the programmer wants to obtain from a non-transparent multicast. The value 1, MAJORITY, ALL, N, or COMPARE acts as an argument therefore. The client stub automatically adapts the length of the out and inout sequences to the set_numreplies value and the current cardinality of the object-group.

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

Note that the multicast version of an operation requires a sequence of CORBA::Environment objects. This is so because such objects contain information on exceptions [Hyp93], and each group-member might want to report an exception by itself. Analogously to inout arguments, the first element of an Environment sequence informs the run-time of the operation type and of the requested number of replies.

Note:

Our C++ language mapping differs from the HYPERDESK-mapping in that Environment objects specify an asynchronous, synchronous, or deferred-synchronous invocation type, in that upcall threads can be declared, and in that certain operations are mapped onto two different C++ client stub signatures, one serving for unicasts and transparent multicasts, the other for non-transparent multicasts. In the non-transparent invocation form, CORBA sequences are employed for the in and inout arguments to gain access to the replies of the individual group members.

5.6 Marshaling

Argument values (passive ELECTRA objects) and results must be marshaled by the DII module before they can be transmitted over the network. We will not address the marshaling of IDL data types as this is obvious. Rather, we are concerned about how a user-defined, complex C++ object, a graph-object for instance, can be transmitted with a CORBA operation. Since the current version of CORBA-IDL [Dig93] does not permit a specification of CORBA operations taking complex objects as arguments, the marshaling of such arguments must be performed by the programmers themselves. For instance, `sequence<octet>` types could be employed to hold the marshaled data. Next, we describe how marshaling of complex objects is accomplished in ARJUNA, in the Modula-3 network objects system, and in ET++. Then, we will describe the approach taken by ELECTRA.

5.6.1 Arjuna

In ARJUNA [SDP], two methods to each user-defined, recoverable object, namely `save_state` and `restore_state`, must be implemented by the programmer. If a user-defined, complex object consists of ARJUNA library objects, the programmer can rely on the marshaling methods which are part of the library. When an object is transmitted over the network or written to non-volatile storage, the ARJUNA run-time invokes the object's `save_state` method to obtain its internal state in form of a byte sequence. The complementary `restore_state` method assigns a state to an ARJUNA object.

5.6.2 Modula-3 Network Objects

At the Digital Systems Research Center a network objects system has recently been implemented for Modula-3 [BNOW93, BNOW94]. Within this system, objects are marshaled by a generic mechanism which relies on the run-time type information maintained by the Modula-3 environment. This approach is called *pickling*. Since the default behavior of the pickle package might not be adequate for all data types, programmers can tailor the behavior by providing their own marshaling procedures. Imagine the situation where an object contains a file descriptor as a data member: The default behavior of the pickle package is to transfer the file descriptor it-

self, which is meaningless to the remote receiver. In the Modula-3 network objects system programmers can modify the default behavior by providing a marshaling procedure which transfers the contents of the file represented by the descriptor instead of the descriptor itself.

5.6.3 ET++

ET++ [WGM88] is a C++ class library for integrating user interface building blocks. ET++ requires run-time type information for its run-time object inspector, for debugging purposes, and for object input/output. Since most of today's C++ compilers do not provide run-time type information [Str93], ET++ makes available a set of preprocessor-macros for conserving type information until run-time. The basic idea here is to call a special macro, called `MetaImpl`, in the implementation of a class. The macro obtains enough information for a *meta class* [PW88, Sem93] holding the type information for the class to be generated at compilation time. The syntax of such macro call is
`MetaImpl(class_name, superclass_name, list_of_symbols)`,
where the list of symbols identifies the type of each data member of the class by entries such as

- **T**(`member`) for any built-in type or ET++ object
- **TP**(`member`) for a pointer to a built-in type or ET++ object
- **TA**(`member`) for a fixed sized array
- **TAP**(`member`) for a fixed sized array of pointers
- **TV**(`member`) or **TVP**(`member`) for a dynamically growing array of values or pointers, respectively

The ET++ library contains a default marshaling mechanism to marshal and to unmarshal objects by following their type information. Programmers change the default behavior by overwriting methods.

5.6.4 Electra

In analogy to the ARJUNA approach, the present version of ELECTRA requires the programmer to write marshaling procedures. For future work

we think of using an approach similar to the one employed in ET++ and Modula-3. Three procedures must be implemented[5] to a user defined ELECTRA class T. Procedure ::dump(const T&, char *buffer) stores the object-state into the buffer. The complementary procedure ::undump(T&, const char *buffer) takes an empty object as argument and initializes it with the state stored in the buffer. Finally, ::dumpSize(const T&) calculates the buffer space necessary to hold the state of an object of type T. For a class called Window, the programmer would have to provide the following procedures:

Example 8:

```
u_int dumpSize(const Window& win) {
   // Return the needed buffer space in bytes.
};

u_int dump(const Window& win, char *buf) {
   // Write the object's state to buf.
   // Return the amount of bytes written.
};

u_int undump(Window win&, const char *buf) {
   // Set the object's state to the state in buf.
   // Return the amount of bytes read.
};
```

We decided to use marshaling procedures instead of class methods to ensure that user-defined and built-in types could be processed in a similar way, which facilitated the implementation of the stub generator considerably. For example, the dump method to a CORBA::Sequence object issues dump for each element in the sequence, regardless of whether it is of a built-in or of a constructed type. In the case of a constructed type T, dump(T, buffer) is automatically issued recursively until a built-in type is met. Built-in types are marshaled by procedures which are part of the ELECTRA library. Canonical data representation must be addressed only by the marshaling procedures which belong to built-in types.

[5] this is not necessary for built-in and for CORBA data types.

5.7 Writing Electra Applications

5.7.1 Creating and Accessing Object-Groups

The code fragment below demonstrates how an object-group is created and how a multicast invocation is issued on the group:

Example 9:

```
example *create_a_group(example& ex1, example& ex2){
   // create a NULL object reference:
   example *group = example::bind();

   // create an empty object-group.
   // ``electra'' is the default BOA:
   electra->create_group(group);

   // let ex1 and ex2 join the group:
   ex1.join(group); ex2.join(group);

   //return the reference to the group:
   return group;
};

void invoke_a_group(example& group){
   // multicast the value 7 to the group:
   // (synchronous operation)
   group.op3(7);
};
```

The procedure `create_a_group`, which is part of a server program, creates an empty group of `example` objects (see Section 5.5). After that, the CORBA objects `ex1` and `ex2` are inserted into the group. The object-reference returned by the procedure can be installed into a name server. Procedure `invoke_a_group`, which is part of a client program, obtains a reference to a group of `example` objects as a parameter. This reference was retrieved from a name server, for instance. Subsequently, operation `op3` is multicast to the group.

5.7.2 Invocation Types and Upcalls

The below version of `invoke_a_group` issues operation `op1` and causes the `upcallOp1` thread to be invoked when the operation has completed:

Example 10:

```
void upcallOp1(float o, Environment&){
   // do something with o
};

void invoke_a_group(example& group){
   float o;
   Environment promise(ePromise), async(eAsync);
   promise.set_numreplies(1);
   async.set_numreplies(1);

   // a deferred-synchronous multicast:
   group.op1(7, o, promise, upcallOp1);
   // ...
   // block until the first reply arrives:
   promise.wait();
   // do something with o.
   // (At this point the upcall-thread has been
   // invoked with a copy of o)

   // a synchronous multicast (the default):
   group.op1(7, o);
   // do something with o

   // an asynchronous multicast:
   group.op1(7, o, async, upcallOp1);
   // o remains undefined, but o in the upcall
   // contains the reply.
};
```

The operation is invoked three times to exemplify the three different invocation types ELECTRA supports, namely deferred-synchronous, synchronous, and asynchronous.

5.7.3 Transparent and Non-Transparent Multicast

In the next code fragment, the difference between transparent and non-transparent multicast is shown:

Example 11:

```
// upcall for transparent multicast:
void upcallOp1(float o, Environment& env){
   // o and env contain the first arriving result
};

// upcall for non-transparent multicast:
void upcallOp1Seq(Sequence<float>& o,
   Sequence<Environment>& env){
   // o and env contain the results of ALL group members
};

void invoke_a_group(example& group){
   Environment promise(ePromise);
   Sequence<Environment> promiseSeq;
   promise.set_numreplies(ALL);
   promiseSeq[0].set_calltype(ePromise);
   promiseSeq[0].set_numreplies(ALL);

   // transparent multicast:
   group.op1(5, o, promise, upcallOp1);
   // ...
   // wait only for the first result:
   promise.wait();
   // o contains the first reply.

   // non-transparent multicast:
   group.op1(5, oSeq, promiseSeq, upcallOp1Seq);
   // ...
   // wait until all members have replied:
   promiseSeq[0].wait();
   // oSeq contains the replies of all group members.
};
```

5.7.4 A Fault-Tolerant Directory Service

To demonstrate active replication and state transfer, we sketch a fault-tolerant directory service. The following interface declares a `directory` object which maintains an ordered collection of `String`/`Any` pairs:

Example 12:

```
interface directory {
   void insert(in string key, in any entry)
      raises(ENTRY_EXISTS);
   void lookup(in string key, out any entry)
      raises(NO_SUCH_ENTRY);
   void remove(in string key, out any entry)
      raises(NO_SUCH_ENTRY);
};
```

To register an entry under a certain search key, the `insert` operation is issued. The `lookup` operation searches the directory for a certain entry, whereas an entry can be deleted by the `remove` operation.

By feeding this interface declaration to the ELECTRA IDL compiler, a *skeleton header file* containing the class specification in Example 13 is produced. The skeleton file mainly serves as a template for the programmer and helps in implementing the `directory` service.

Example 13:

```
class directory_sk :  public virtual directory_im {
public:
   // operations:
   //
   void insert( const String <0ul>& key, const Any& entry,
      Environment& );
   void lookup( const String <0ul>& key, Any& entry,
      Environment& );
   void remove( const String <0ul>& key, Any& entry,
      Environment& );

   // state transfer:
   //
   virtual void get_state(Sequence <Any>& state);
   virtual void set_state(const Sequence <Any>& newState);

   // Constructors, etc.
   ...
};
```

Since we intend to instantiate a homogeneous group of `directory_sk` objects, the `get_state` and `set_state` operations must accomplish state transfer (see Section 5.4.5). Specifically, the `get_state` method will copy all `String`/`Any` pairs the object maintains to the `state` argument. The `set_state` operation first removes all entries from the object's internal data structure, and then inserts the `String`/`Any` pairs it obtains from the `newState` argument into the data structure. Both operations are automatically invoked by ELECTRA when a state transfer is required.

After having implemented the skeleton's `insert`, `lookup`, and `remove` operations, our fault-tolerant directory service is ready. When a `directory_sk` object joins a group, ELECTRA obtains the internal state of a group-member by invoking its `get_state` operation, then it transfers the state over the network, and assigns it to the newcomer by calling its `set_state` operation.

5.7.5 An Audiocast Facility

In this section we present a simple but effective ELECTRA application, good for multicasting audio data to a group of recipients. Example 14 demonstrates the CORBA-IDL specification of an object which receives and handles audio data. The idea is to instantiate a `radio` object on each workstation to which audio data is sent. A transmitter object multicasts audio data to the objects by invoking their `receive_audio` operation. `radio` objects can be grouped at will to arrive at several sessions and audio channels on the same local area network:

Example 14:

```
const long BLOCK_SIZE = 1024;

interface radio {
   oneway void receive_audio(
      in sequence <char, BLOCK_SIZE> data);
};
```

The next code fragment contains the implementation of above `receive_audio` operation. Our utility simply writes the data to *standard output*:

Example 15:

```
// This thread is created by Electra when a receive_audio
// multicast arrives.  The data is written to stdout:
void radio_sk::receive_audio(
   const Sequence<char, BLOCK_SIZE>& data,
   Environment& env
){
   cout << data;
};
```

Example 16 exemplifies a multi-threaded main program which multicasts the data it reads from *standard input* and writes the multicast it receives to *standard output*. Thus, the program acts both as client and server. It can be compiled and linked with the `radio_sk` implementation to obtain an executable file called `radio`:

Example 16:

```
main(int argc, char **argv){
   // pass the application policy to the adaptor:
   electra->appInit(raw_app_policy);

   // create the audiogroup, if requested:
   if(argc == 2)
      electra->create_group("audiogroup", raw_multicast);

   // SERVER PART:
   // instantiate the implementation which
   // will receive multicasts:
   radio_sk myReceiver;
   // start receiving multicasts:
   myReceiver.join("audiogroup");
   electra->impl_is_ready();

   // CLIENT PART:
   // bind to the "audiogroup", read data from stdin
   // and multicasts it to the "audiogroup":
   Sequence<char, BLOCK_SIZE> s;
   radio *receivers = radio::bind("audiogroup");

   while(cin >> s){
      receivers->receive_audio(s);
   };

   // release the object-reference:
   receivers->release();
};
```

This `radio` utility was used to multicast a radio program on the LAN at the author's research department. On the UNIX workstation which had a radio connected to its audio device, the utility was started with the command `radio < /dev/audio` to multicast the audio data on the LAN. To be able to listen to the radio program, clients had to execute the command `radio > /dev/audio`.

The utility was also employed to practice voice-conferences on our LAN.

After the participants had executed the command `radio > /dev/audio < /dev/audio`, conversation was possible with a microphone and a pair of headphones, both attached to the participant's workstation. The attentive reader will have noted that the utility consists of only about twenty lines of C++ code.

5.7.6 Switching Adaptor

We were able to run the `radio` utility *unmodified* on MUTS, HORUS, and ISIS, since only unordered, raw multicast is required by the application[6], which is supported by each of the platforms. Example 17 shows how the application is reconfigured for the aforementioned platforms. Note that no recompilation of the application is required, which makes it possible for ELECTRA applications delivered in binary form to be reconfigured as well:

Example 17:

```
CC -o radio radio_sk.o ...-lmutsAd -lmuts -lelectra
   (use the MUTS version of radio)
CC -o radio radio_sk.o ...-lhorusAd -lhorus -lelectra
   (use the Horus version of radio)
CC -o radio radio_sk.o ...-lisisAd -lisis1 -lisis2 -lisism \
   -lelectra
   (use the Isis version of radio)
```

All of the platform-independent ELECTRA code is contained in the `electra` library. The `mutsAd`, `horusAd`, and `isisAd` library contains the adaptor object for the respective platform. The remaining libraries are part of the platforms.

5.8 Lessons Learned

We learned the following lessons in the process of designing and implementing the real ELECTRA toolkit:

- **Adaptor Model**: Although ELECTRA was still under development at the time this thesis was written, the toolkit was advanced enough to

[6] an audio packet lost from time to time will hardly be noticed by the listeners.

permit the implementation of object-group based, fault-tolerant applications. The employed flexible system design permitted to break ELECTRA free from the ties of the underlying communication substrate and operating system. Adaptors were written for MUTS, for the HORUS group socket interface, and for ISIS version 3.1. Appendix C contains the C++ code to the present version of our ISIS adaptor. Developing an adaptor for MUTS, HORUS and for ISIS was a matter of less than one man-week of work for each adaptor. Such an adaptor typically comprises less than 1000 lines of C++ code. The fact that the adaptor model facilitates a clear encapsulation of platform-dependent code helped design a well-structured and maintainable product. We believe that the proposed system design is flexible enough to allow ELECTRA to run on AMOEBA, CHORUS, TRANSIS, as well as on other systems.

- **Configurability**: The idea of a portable object request broker for reliable distributed systems is enticing. The disadvantage of our approach is that the special features of a real machine cannot be exploited by ELECTRA, as the upper ELECTRA layers must rely on a portable VM interface. The VM interface offers only the base-functionality which is common to the platforms we want to support. So more sophisticated functionality must be implemented in ELECTRA.

- **Development Time**: A simple EOM-compliant toolkit can be realized in about one man-year, provided that a technology such as HORUS or ISIS is the basis of the implementation.

- **CORBA Compliance**: Our commitment to the CORBA standard resulted in the industry taking an interest in the ELECTRA project and in the design of a well-defined programming interface. By employing the CORBA-IDL compiler, which is available from SUN Microsystems for free, we were spared tedious development work. Unfortunately, the present version of the CORBA specification does not comprise requirements for well-functioning distributed systems. For example, neither the system's behavior in the face of failures nor replicated objects are addressed by CORBA. In contrast, EOM embodies the prerequisites for building reliable, distributed software in an object-oriented fashion. EOM is an enhancement, not a replacement, for CORBA.

- **CORBA-IDL**: We found CORBA-IDL well-suited for our EOM-compliant toolkit. In the ELECTRA approach, most parameters related with object-groups, i.e., the number of members or the event ordering protocols, are specified in the target programming language using policy objects and not in the IDL specification, as they are a concern of the object-implementation and not of its interface.

- **Complex Object Passing**: It would be a good feature if CORBA-IDL allowed complex objects with a layout unknown to the IDL compiler to act as arguments to object invocations. User-defined methods or run-time type information are necessary to marshal such objects.

- **Readonly Operation Attribute**: Another good idea is a `readonly` IDL operation attribute for tagging operations which do not change the internal state of an object. Owing to such operation declaration syntax, a `readonly` group invocation would be performed on one group member only, e.g., on a local or nearby one, whereby operations altering an object's state would be multicast to all members. The modified CORBA IDL syntax for operation attributes (see Section 4.9.1 of the CORBA specification, revision 1.2 [Dig93]) looks as follows:

```
<op_attribute> ::=   "oneway"
                   |   "readonly"
```

5.9 Summary

This chapter dealt with the implementation of the ELECTRA toolkit. ELECTRA comprises a CORBA object request broker, a CORBA-IDL compiler, classes for object-oriented, distributed programming in C++, and other utilities. ELECTRA is unique in that it combines a CORBA programming environment with the powerful group-operations provided by systems such as HORUS and ISIS, and in that a flexible and modular system design is employed. Our toolkit is thus suited for object-oriented programming of *reliable* distributed applications, in that ELECTRA applications can tolerate partial failures, and in that distributed events appear to occur in the same order at all objects, despite failures and despite objects joining and leaving the system dynamically. ELECTRA improves the CORBA specification in that

- ELECTRA objects can be grouped
- object invocations can be multicast to object-groups
- programs execute in a virtually synchronous environment, provided that the underlying platform supports Virtual Synchrony, and granted that an appropriate application policy is employed
- the Static- and the Dynamic Invocation Interface makes possible asynchronous, synchronous, and deferred-synchronous invocations
- all communication is reliable
- various communication protocols, event ordering rules, and so forth can be selected by policy objects
- results generated by individual object-group members can be concatenated with CORBA sequences
- the `CORBA::BOA` class supplies operations for object-group management
- an ELECTRA application can be reconfigured to run on another platform, even when the source code of the application is not available.

To support EOM, modifications of the interfaces listed in the CORBA specification were not necessary. The CORBA::BOA and `CORBA::Environment` interfaces had to be enhanced with operations to support object-groups. Presently, ELECTRA provides adaptors for MUTS, for the HORUS group socket interface, and for ISIS version 3.1 (Appendix C). We believe that our system design is general enough for quick realization of adaptors for other communication substrates.

Chapter 6

Distributed Frameworks

As this thesis deals with system support for OODP, the statements were focussed on the run-time of an EOM toolkit. Although the proposed mechanisms are powerful, more than a run-time system, an interface definition language, and Virtual Synchrony is needed to build real-world distributed systems. This chapter deals with the design and realization of *application frameworks* for distributed systems using EOM as a starting point.

6.1 The Role of Frameworks

An application framework consists of a set of classes that provide a general solution to a problem [JF88, Sch86, And94]. Thus, a framework represents a *generic application*, which can be tailored to specific needs, and which serves as a solution to a family of related problems. From the business perspective, a framework encapsulates expertise about a problem domain and can reduce development and maintenance cost. The main difference between a framework and a conventional programming library is that the methods defined by the user will often be called by the framework itself and not by the user's application code. The framework frequently plays the role of the "main program" by coordinating and orchestrating application activity. A toolkit is based on one or several frameworks.

A well-known application domain of frameworks is in the design and implementation of graphical user interfaces. The Model View Controller

triad [PW88] in Smalltalk or the ET++ [WGM88] application framework are good examples for this. Object programming requires proper frameworks, and in the future, programmers will want more than frameworks for graphical user interfaces. We believe that this concept can be applied to OODP as well, and we shall use the term *distributed framework* to denote a set of related interfaces and classes that provide the basic functionality of a working distributed system or of a component thereof, but which can easily be tailored to individual needs.

The main goal of distributed frameworks is to facilitate the realization of complex distributed system, and to make possible the reuse of system components and good design. To encourage research on distributed frameworks, we will describe a coordinator-cohort, a network management, and an information space framework. Distributed frameworks are to be realized on the ELECTRA layers described so far. Figure 6.1 presents the resulting, now complete ELECTRA architecture.

6.2 Coordinator-Cohort Framework

The coordinator-cohort computation scheme is popular in the ISIS toolkit since it provides inexpensive fault-tolerance for computations which can be performed by all members of a process-group [Isi92]. In this scheme, the coordinator is a special group member that carries out the computations associated with the client-requests. The other group members, known as the cohorts, are passive unless the coordinator fails, whereupon a cohort will become the new coordinator. The coordinator is further able to periodically inform the cohorts of its internal state for backup purposes.

We can define a simple coordinator-cohort framework, consisting of CORBA-IDL declarations and library code which can be tailored to specific purposes. To participate in a coordinator-cohort computation, an object must be compatible with the interface in Example 1.

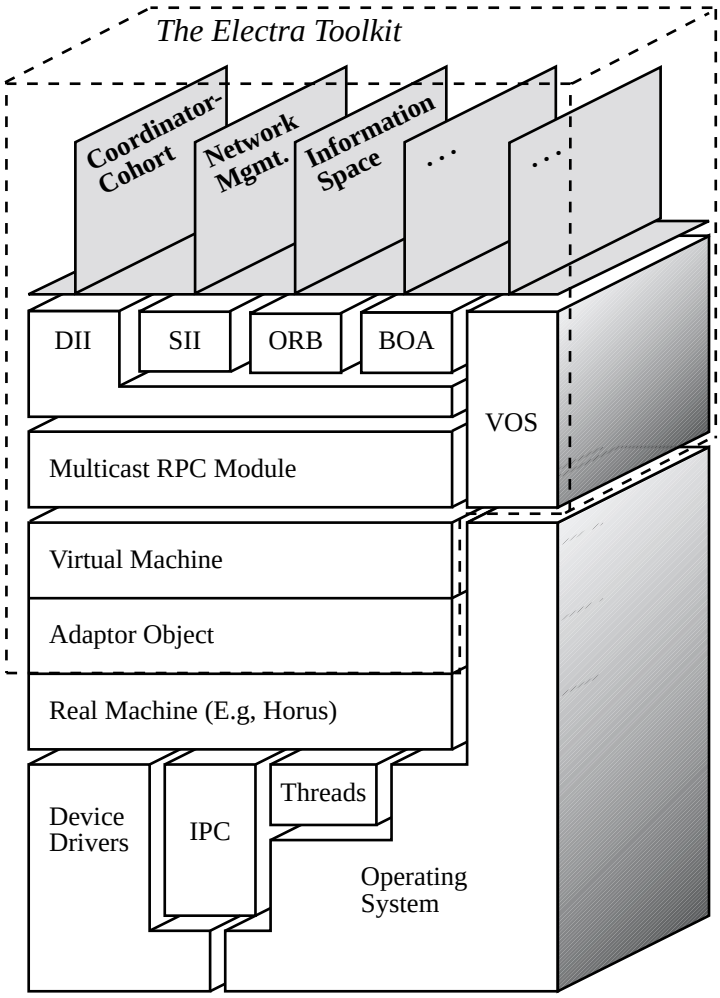

Figure 6.1: The architecture of the complete ELECTRA toolkit.

Example 1:

```
interface CoordinatorCohort {
   enum role {coordinator, cohort};

   void ccEnter();
   void ccLeave();
   void transferState(in sequence <any> state)
      raises (NOT_COORDINATOR);
   void setState(in sequence <any> state);

   readonly attribute role myRole;
};
```

When a `CoordinatorCohort` object wishes to become part of the ongoing coordinator-cohort computation, it calls the framework's `ccEnter` operation. To leave the computation `ccLeave` must be called. To transfer an application-dependent state, the coordinator issues the `transferState` operation. The framework will transfer the state and invoke the cohorts' `setState` operation. Finally, by inquiring the `myRole` attribute objects find out what role they are playing.

The advantages of the coordinator-cohort framework are that it is simple to use, that fault-tolerance is ensured, and that each individual client-request must be computed only once. Furthermore, load balancing can be achieved by having the framework distribute the coordinator-role among the group members. A single member could be coordinator for one or for a few requests, whereupon the coordinator role is assigned to the next member, and so forth.

6.3 Application Management Framework

To manage complex distributed applications, system administrators need tools for monitoring an object, for reinitializing it, for shutting it down, or for defining certain monitoring intervals and alarm conditions. Network management is becoming an important research area, it has created network management protocols such as SNMP [CMRW93] and OSF/DME [Aut92]. In this section, we propose a framework that turns a normal ELECTRA object into a *managed object* by means of mixin inheritance. An application

can issue managing operations on such an object (or object-group of course) through an inherited management interface. Programmers may rely on the management operations which are part of the framework, or tailor the framework to their specific needs.

Analogously to SNMP V.2 [CMRW93], the management framework could provide operations for handling alarms, events, and notifications. An *alarm* is an exceptional condition detected by monitoring the value of a certain object attribute periodically, whereby the sampling interval is configurable. For instance, an alarm may occur when an object attribute falls out of a configured range. Each alarm triggers an internal *event*, and an event can cause one or more *notifications* to be sent to objects interested in the event. Our management interface thus defines operations for setting up alarm conditions, events, and notifications. It also provides an alarm, an event, and a notification table, for which we will give the CORBA-IDL specification now.

Example 2:

```
enum SampleType {
   absoluteValue, deltaValue
};

struct AlarmEntry {
   short alarmVariable;
   long alarmInterval;
   SampleType alarmSampleType;
   long alarmValue;
   long alarmRisingThreshold;
   long alarmFallingThreshold;
};
```

An alarm table consists of a sequence of `AlarmEntry` elements. Each managed object provides such a table, and program code which is part of the framework maintains the table. `alarmVariable` stores a key which identifies the object attribute to be monitored, and `alarmInterval` defines the sampling interval in seconds. Whether the sampled value itself or the difference between the present and the old sample will be compared with the thresholds is determined by `alarmSampleType`. The value of the monitored object attribute which was obtained during the last sample is stored in

alarmValue. When the present sample returns a value greater (smaller) than or equal to the alarmRisingThreshold (alarmFallingThreshold), and the last sample was less than (greater than) the threshold, the related event is triggered.

Example 3:

```
// forward declaration:
interface Receiver;

enum NotificationType {
   rising, falling
};

struct Notification {
   NotificationType alarmType;
   long alarmTime;
   Receiver alarmReceiver;
};
```

A structure of type Notification defines a notification object which shall be sent to an object interested in a certain alarm. Whether the notification was triggered by the crossing of an upper or lower threshold is specified by the alarmType field. alarmTime contains the time at which the alarm condition was detected. alarmReceiver represents the reference to the CORBA object or object-group to be notified.

Example 4:

```
struct EventEntry {
   short alarmVariable;
   string eventDescription;
   long eventLastTimeSent;
   sequence <Notification> eventNotifications;
};
```

Finally, the event table consists of EventEntry objects and defines notifications to be delivered when an alarm condition is met. alarmVariable has the same meaning as in the alarm table and identifies the object at-

tribute the notification is related to. `eventDescription` contains a comment describing the event. `eventLastTimeSent` holds the time value at which this entry last generated a notification. Finally, the `eventNotifications` sequence contains receivers to which the notification will be sent.

The monitoring process works as follows: When an attribute falls out of the configured thresholds, the event table of the affected object is searched for an entry whose `alarmVariable` matches the `alarmVariable` value associated with the attribute. If such an entry is found, a notification is sent to all receivers listed in the `eventNotification` sequence of the element. In order to act as a managed object, an object must inherit the operations defined in interface `ManagedObject`:

Example 5:

```
// forward declaration:
interface NotificationReceiver;

interface ManagedObject {
   // general operations:
   void suspendOperation();
   void resumeOperation();
   void resetState();

   // alarm handling operations:
   void registerNotification(
      in AlarmEntry alarm,
      in NotificationReceiver rcv,
      out short notificationID
   );
   void unregisterNotification(in short notificationID);

   readonly attribute short numAlarms;
   readonly attribute short numReceivers;
};
```

The methods `suspendOperation` and `resumeOperation` serve to freeze and to resume, respectively, activity inside an object. The method `resetState` reinitializes an object. A client monitors an object by invoking the `registerNotification` operation of the object it is interested in. Here,

an `AlarmEntry` structure is passed to the managed object to inform it of the internal attribute, the value, the sampling interval, the thresholds, and so forth. `rcv` specifies the object reference of the object to be notified when the alarm occurs. A `notificationID` is returned to allow the client to alter certain `AlarmEntry` parameters at a later time, or to unregister from the object with the `unregisterNotification` operation. Finally, the attributes contain information on the number of alarms the object has risen so far and the total number of receivers registered.

Example 6:

```
interface NotificationReceiver {
   // an event occurred:
   void notification(in AlarmEntry alarm,
      in short notificationID);
   ...
};
```

Objects which shall act as receivers of notifications must be derived from the above `NotificationReceiver` interface. When an event occurs in which the client object has registered an interest using the `registerNotification` method, the framework will invoke the client's `notification` method.

6.4 Information Space Framework

The objective of the next framework is to make it unnecessary to think about the coupling between singleton objects and object-groups. Therefore, we provide a shared *Information Space* abstraction, which is motivated by the LINDA Tuple Space Model [CGA86], the ISIS Distributed News software [Isi92], and the Information Bus [OPSS93]. The Information Space framework can be thought of as a virtually shared, fault-tolerant bulletin board system. Applications can "post" objects on the board by labeling them with an individual "subject" field. Applications register an interest in certain subjects, upon which the framework will forward all objects labeled with the specified subjects to the interested applications (Figure 6.2). Internally, this forwarding takes the form of object-group multicasts.

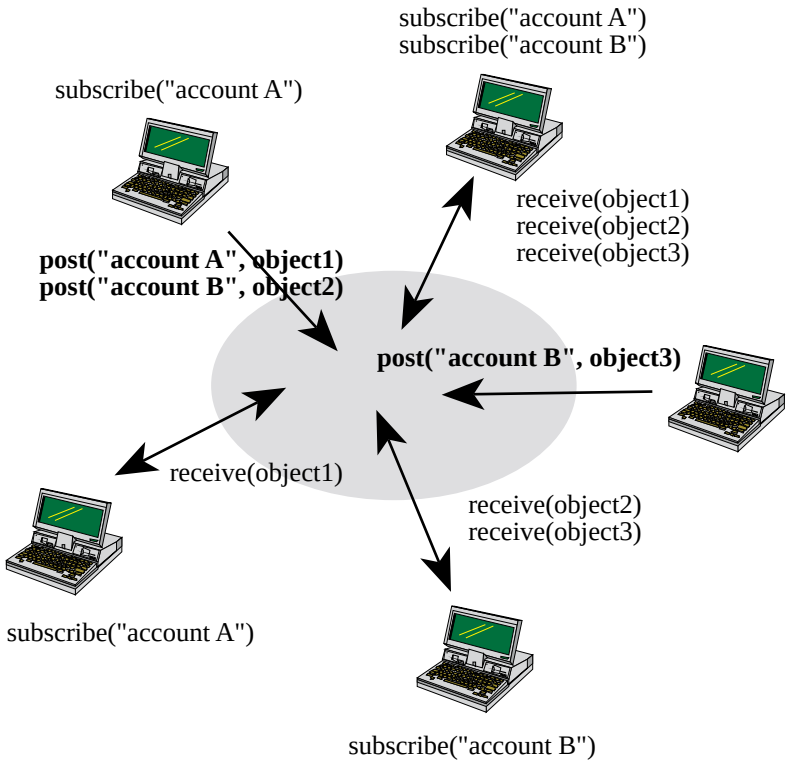

Figure 6.2: The Information Space model.

The framework's benefits are that powerful and fault-tolerant computing is possible with a very simple API, that communication is anonymous, and that it uses a popular "publisher/subscriber" communication style, which is similar to USENET [Keh92]. Unlike USENET but similar to ISIS Distributed News, the ELECTRA Information Space framework supports reliable and ordered group communication. All applications subscribed to a certain subject receive all postings sent to the subject in the same order. Moreover, conventional ELECTRA remote object invocation and Information Space communication can be used from within the same application.

Anonymous communication facilitates dynamic system reconfiguration and evolution. For example, an object consuming or producing postings can be replaced dynamically, as long as the posting-type and the subjects received or generated by the new object conform to the old one. This replacement will not affect the other producers and consumers of the postings. Below CORBA-IDL fragment contains the interface definitions for the framework:

Example 7:

```
struct Posting {
   string subject;
   any object;
};

interface Subscriber {
   void receive(in Posting p);
};

interface InformationSpace {
   void post(in Posting p);
   void subscribe(in Subscriber s, in string subject);
   void unsubscribe(in Subscriber s, in string subject)
         raises (NOT_SUBSCRIBED);
};
```

Objects passed through the Information Space are of the CORBA any type. An object interested in using the framework creates an `InformationSpace` object reference. Postings are submitted by the `post` operation,

and subscription to a certain subject is accomplished by the `subscribe` operation, which takes a reference to a `Subscriber` interface as a paremeter. When a posting to a certain subject is submitted, the framework invokes the `receive` operation of all objects which registered an interest in the subject by a reliable and order preserving group RMC.

Programmers can modify the framework by subclassing and method overwriting. A more elaborate version of the framework could support hierarchically structured subjects, subscription using regular expressions, mobile clients, persistent postings, and so forth. The model could further be expanded to an "object trading place", where a supplier offers an object at a certain price, and a consumer informs the system of her willingness to acquire an object at a certain price. A tailorable demand-matching and price-setting mechanism governs the exchange of objects. This "object trading place" can be seen as a general platform for implementing electronic markets and other kinds of business information systems.

6.5 Summary

EOM specifies the components of a run-time system to support object-oriented, distributed applications. For the building of complex distributed applications, however, the interfaces and abstractions offered by an EOM run-time might still be too low-level, that is why this chapter proposed *distributed frameworks* based on the EOM run-time. A distributed framework consist of reusable interfaces and classes which make up a generic component of a distributed system, they can be tailored to specific needs. Thus, frameworks allow programmers to reuse code and good design. To encourage research and discussion on distributed frameworks, we presented a coordinator-cohort, an application management, and an Information Space framework.

Chapter 7

Conclusions

A computer network consists of workstations, PCs, portable computers, servers, and peripheral devices interconnected with a communication subsystem. A sophisticated run-time environment can turn a computer network into a *distributed system* offering a single system image. In a distributed system, shared resources, replication, and failures are hidden to a large degree by the run-time environment. The realization of distributed systems is difficult and requires sound low-level system support and powerful application frameworks which today are not generally available in the operating systems and programming environments.

As they seek competitive advantages, the time required to develop and market a new product is a major concern of many enterprises. Fast access to relevant business information, service availability, transparency, and scalability thus become important issues. For many business and industrial applications, networks of workstations and high-end PCs have greater performance potential than mainframes at significantly lower cost and with greater flexibility. In order to take more advantage of their computer networks, many companies are now moving towards open distributed processing and client-server applications. We believe that object-oriented distributed programming (OODP) is the next major step in the evolution of client-server computing.

It has been the objective of our research to facilitate the implementation of open distributed systems and to help reuse their software components. To be more specific, we have devised an *execution model*, an *architectural*

model, and a *toolkit* for object-oriented distributed programming. In this context, the failure of system components is seen as a frequent phenomenon, and replication as an effective means to cope with partial failures. We therefore introduced the concept of object-group communication to support the realization of replicated and parallel system services.

By showing that OODP can be done in an efficient, reliable, and generic way, we demonstrated the usefulness and desirability of an EOM-compliant toolkit. We think that ELECTRA is a very promising approach for scalable, fault-tolerant, open, and reusable distributed applications. We now summarize the goals and results of our research.

7.1 Research Goals

7.1.1 Execution Model

The Electra Object Model (EOM) we proposed in Chapter 3 defines the low-level system support we regard as indispensable for any OODP environment. Active and passive objects, typed interfaces, remote method calling, object-groups, a failure detection service, consistent group management, ordering of events, and Virtual Synchrony are the foundation of EOM. In EOM, program executions are carried out in a virtually synchronous way — this means that all significant events, for instance the delivery of unicasts, multicasts, view-changes, and failures, appear as if each event had occurred at the same logical instant in all processes.

The key benefit of EOM is that it enables the building of efficient and reliable object-oriented applications. Efficient in the sense that the model is based on non-blocking communication, so that the parallelism inherent to an application can be exploited. Reliable in that Virtual Synchrony guarantees a consistent ordering of the events, despite failures and application reconfiguration, i.e., objects leaving or joining the system dynamically.

Most components of EOM are results of research done in other laboratories. Our contribution consists in elaborating design alternatives, selecting the important EOM components, and aggregating them into a well-rounded model which supports OODP. EOM is formulated in terms of "need-to-have" and "nice-to-have" requirements, these are summarized in Appendix A.

7.1.2 Architectural Model

In Chapter 4 we described a flexible run-time system design to support OODP. The run-time system is structured in an object-oriented fashion and comprises an invariable interface as well as *adaptor objects* to map this generic interface onto the one provided by the underlying operating system. This model requires only minimal effort to adapt a run-time system to a new hardware platform or operating system, and is valuable not only for run-time systems but for other applications as well.

Our contribution consists in devising the mentioned *adaptor model*, which in our opinion and experience is more general and flexible than many run-time system designs which were proposed recently, and which allows to reuse adaptor objects. Furthermore, by adhering to the adaptor model, we have designed, implemented, and documented a run-time system for an EOM-compliant toolkit.

7.1.3 Electra Toolkit

An implementation of EOM, called the ELECTRA toolkit, was described in Chapter 5. ELECTRA adheres to the CORBA standard and is structured along the adaptor model. Thus, ELECTRA differs from other OODP environments in that group communication is supported, in that a configurable system architecture is employed, and in that it is CORBA compliant. ELECTRA can be configured for different operating systems and various state-of-the-art platforms such as HORUS and ISIS.

Our contribution consists in describing the design of ELECTRA and the interfaces of its components, and in demonstrating that EOM can be realized with about one man-year of effort, provided that the implementation is based on a technology such as HORUS or ISIS. For the benchmark application considered, the performance of ELECTRA was assessed and found to be higher than the performance of several distributed computing platforms in use today. This was true even though the API of most of the other platforms we benchmarked offered a lower abstraction level. The mapping of group operations onto the C++ programming language was also elaborated and described. We believe that ELECTRA may serve as starting point for a production-quality object request broker for reliable distributed systems.

7.2 Future Work

We intend to continue our research in several areas. First, we will finish the ELECTRA toolkit and make it accessible to endusers. Building components and frameworks for distributed systems is a further goal. The Information Space framework described in Section 6.4 is enticing and has great practical relevance. Generally speaking, we are interested in generic, reusable components for robust distributed systems.

Another topic of future research is related to the adaptor model described in Chapter 4. Here, we want to configure ELECTRA for other platforms and operating systems and further refine its design. Along with ELECTRA, we are planning to build other configurable distributed systems, for example a generic multicast transport service based on the preliminary work outlined in Section 4.1.1. Finally, we will work on further comparison of our approach with those taken in other projects. We also intend to implement distributed applications on various toolkits in order to compare the results with our methods.

TO BE CONTINUED

Appendix A

EOM Requirements

Active and Passive Objects

Need-to-have:

The ELECTRA Object Model consists of active and passive objects. Active objects communicate by interchanging passive objects. Passive objects provide methods for marshaling their state into a sequence of bytes, and for unmarshaling their state out of a sequence of bytes.

Active objects are not tied to a client or server role: any active object can invoke operations on other active objects and handle incoming requests. EOM objects are not fragmented, but completely contained in one address space. The granularity of EOM objects varies between very small and very large. Object-references can be transmitted across node boundaries, allowing applications to operate on distributed objects in a uniform way.

Nice-to-have:

Operations to convert active objects into passive ones and *vice versa* may be provided.

Interfaces

Need-to-have:

In EOM, the operations of an active object are specified with an Interface Definition Language (IDL). The IDL supports interface declarations, common basic data-types, constructed types, user-defined objects as parameters to remote operations, and interface-inheritance. EOM is neither tied to a specific target language nor to a specific IDL.

Nice-to-have:

Legal behaviors of an object can be specified formally, and a verification tool checks whether the object implementation is correct in respect to its specification.

Remote Method Calling

Need-to-have:

Basically, communication is carried out by Remote Method Calling (RMC). Therefore, a client object invokes the methods of a local representative of the remote object, called the client proxy. The proxy supplies the same set of operations as the remote object. An RMC obtains an arbitrary number of input parameters and returns an arbitrary number of results. RMCs can be carried out in a synchronous, in an asynchronous, and in a deferred synchronous way. The exact mapping of the RMC types and the syntax of remote method invocations depends on the target language and is thus not specified by EOM. Communication with an object is carried out through a lossless and non-generating FIFO channel.

Nice-to-have:

The programmer can specify weaker semantics such as best-effort delivery and unordered communication. Stronger semantics such as atomicity (*zero-or-once* delivery) can be specified for applications requiring a transactional communication style.

Object-Groups

Need-to-have:

EOM supports collections of objects, so-called object-groups. Proxy objects hide replication to a large extent and allow clients to interact with object-

groups analogously to the way interaction with singleton objects occurs. All operational objects deliver the same set of messages, and this set includes all messages multicast by operational objects, and no spurious messages (reliable multicast).

Object-groups are open in the sense that also non-members may issue multicasts on the group. Object-groups are dynamic in that objects may join or leave a group at any time, and an object may be member of several groups simultaneously. Group RMCs return responses as long as at least one group member is operational, and group members fail independently from each other.

Where an RMC produces a set of results, programmers can specify that the results be collated to a single termination, or that a specific number of results, a majority for instance, be returned.

Nice-to-have:

Unreliable multicast can be selected for applications which do not require the strong guarantees provided by reliable multicast. The programmer can specify that a voting mechanism be employed to compare incoming results of a group operation and to select the most frequent one.

Failure Suspector

Need-to-have:

In EOM, a Failure Suspector Service (FSS) is used to suspect failures and to propagate failure beliefs. Failure beliefs are gathered mainly with timeout mechanisms and hints from the operating system; they cannot be completely accurate since the EOM is based on an asynchronous system model. This does not cause any problems as long as incorrect beliefs are handled and propagated in a consistent way. Failure beliefs are piggybacked onto the messages exchanged by the objects in the application and travel along causal chains of events. A close cooperation takes place between the FSS, the Group Membership Service, and the Reliable Multicast Component. In analogy to ISIS, we assume a *primary-partition* model.

Nice-to-have:

Similar to HORUS and to TRANSIS, a multiple-partition model could be supported.

Group Management

Need-to-have:

Object-group members register with a Group Membership Service to be notified of membership changes. The service is implemented by a Strong Group Membership Protocol, which ensures that membership changes are seen in the same order by all members. Furthermore, membership changes are synchronized with the application's RMCs (point-to-point and multicast), meaning that at the moment an RMC is sent or delivered by the run-time system, views are consistent and the latest view is installed in the run-time system.

Nice-to-have:

Weaker group membership protocols may be provided in addition to the strong one, and programmers can select a protocol which best fits their applications' requirements.

Ordering of Events

Need-to-have:

In EOM, communication is inherently asynchronous, which is why we require that the run-time system provides causal delivery (CMCAST) for all kinds of object invocations. Additionally, programmers can specify that object-group multicasts originating in different senders are seen in exactly the same order by all receiver objects (ACMCAST), or they can choose a FMCAST protocol for applications which are not sensitive to ordering. Overlapping groups are supported, but objects being simultaneously member of two different groups might receive the same set of RMCs in different orders. EOM does not assume the existence of a synchronized-clocks service.

Nice-to-have:

Programmers can influence the underlying causality mechanism to give hints about "hidden" causal relationships. Furthermore, it is possible to specify that ordering semantics hold true even in presence of overlapping groups. Large applications may be structured using causality domains.

Virtual Synchrony

Need-to-have:

Significant events like the delivery of unicasts, multicasts, view-changes, and failures appear as if each event had occurred at the same logical instant in all objects, i.e., executions are virtually synchronous. Virtual Synchrony is the low-level synchronization model of EOM.

Nice-to-have:

Other synchronization forms such as Linearizability, R-Linearizability, or Serializability can be provided as well and should be based on the Virtual Synchrony model.

Appendix B

Virtual Machine Interface

```
//
// Class VirtualMachine defines the generic "instruction set"
// Electra is based on. The idea is to make Electra
// independent of the underlying platform (HORUS, ISIS, etc.).
// To let Electra run on a specific platform, one needs
// to create a subclass of VirtualMachine, which maps the
// virtual operations on the operations provided by the platform.
// With the VirtualMachine class below, adaptors for MUTS, for
// the HORUS group socket interface, and for ISIS V3.1 were
// implemented.
//
// (C) Copyright Silvano Maffeis 1995
//

class VirtualMachine: public ElectraObject{
public:
  VirtualMachine();
  virtual ~VirtualMachine();

  // Initialization:
  //
  virtual ORBStatus machInit();
  virtual ORBStatus machQuit();
```

```
virtual ORBStatus electraInit(const AppPolicy&) = 0;
virtual ORBStatus electraQuit();

// Entities:
//
virtual ORBStatus entityCreate(Entity&, const ProtocolPolicy&) = 0;
virtual ORBStatus entityDestroy(Entity&) = 0;
virtual boolean entityEqual(const Entity&, const Entity&) = 0;

// Asynchronous, reliable message passing:
//
virtual ORBStatus msgSend(Entity& dest, const Message& m,
    tUpcall sendDone, void *sEnv) = 0;
virtual ORBStatus msgReceive(Entity& listenOn, tUpcall rcv,
    void *rEnv, tUpcall usr = 0, void *uEnv = 0) = 0;
virtual ORBStatus msgReceiveDone(void *) = 0;

// Group management:
//
virtual ORBStatus grpCreate(Entity& grp,
    const ProtocolPolicy&) = 0;
virtual ORBStatus grpJoin(Entity& grp, Entity& e,
    tUpcall mon, void *,
    tUpcall getState, void *,
    tUpcall setState, void *) = 0;
virtual ORBStatus grpLeave(Entity& grp, Entity& e) = 0;
virtual ORBStatus grpDestroy(Entity& grp) = 0;

// Thread management:
//
virtual ORBStatus threadDeclare(Thread&,
    tUpcall f,
    void *env = 0,
    tPrio p = eMid,
    int stack = MIN_STACK) = 0;
virtual ORBStatus threadCreate(Thread& t, void *param) = 0;
virtual ORBStatus threadDestroy(Thread& t) = 0;

// Failure Suspectors:
```

```
  //
  virtual ORBStatus monMonitor(Monitor&, const Entity&,
     tUpcall, void *mEnv)= 0;
  virtual ORBStatus monCancel(Monitor&) = 0;

  // Synchronization:
  //
  //     - semaphores:
  //
  virtual ORBStatus semaCreate(Sema& s, int value) = 0;
  virtual ORBStatus semaDestroy(Sema& s) = 0;
  virtual ORBStatus semaInc(Sema& s) = 0;
  virtual ORBStatus semaDec(Sema& s) = 0;

  //     - locks:
  //
  virtual ORBStatus lockCreate(Lock& l);
  virtual ORBStatus lockDestroy(Lock& l);
  virtual ORBStatus lockAcquire(Lock& l);
  virtual ORBStatus lockRelease(Lock& l);

  //     - event counters:
  //
  virtual ORBStatus ecCreate(ECount& e);
  virtual ORBStatus ecDestroy(ECount& e);
  virtual ORBStatus ecInc(ECount& e);
  virtual ORBStatus ecWait(ECount &e, int value);
};
```

Appendix C

Isis Adaptor for Electra

Declaration

```
//
// This module contains an Electra adaptor for ISIS V 3.1.
// It mainly supplies the operations defined in the
// VirtualMachine baseclass. The adaptor also does the
// multiplexing necessary to have several Electra objects per
// ISIS process.
//
// Layout of tha data_ part in an ISIS entity:
// address, int object_id
//
// (C) Copyright Silvano Maffeis 1995
//

class IsisAd;

// class to represent a communication endpoint within
// the ISIS adaptor:
//
class EntityAddr {
  friend class IsisAd;
```

```
public:
  EntityAddr();

private:
  address addr_;                                    // ISIS address.
  int oid_;                                 // Object ID. Unique for a proc.
  static int nextOid_;                          // Next object id to assign.
};

// IsisSema helps in mapping Electra semaphores on Isis
// conditions:
//
class IsisSema {
public:
  IsisSema(int val);
  int val_;
  condition cond_;
};

// class to represent an object-group member:
// The ISIS adaptor allows to have many objects per ISIS process.
// Therefore, we need some kind of multiplexing mechanism to
// dispatch incoming messages to the destination object(s) within
// an ISIS process. Class Member, and class MemberList help in
// implementing the multiplexing.
//
class Member {
public:
  Member(const Entity&,
         tUpcall viewChange, void *,
         tUpcall tellState, void *,
         tUpcall setState, void *);

  Entity eid_;                                          // member object.
  tUpcall viewChange_;                          // to notify view changes.
  void *vEnv_;
  tUpcall tellState_;                           // state management ops.
  void *tEnv_;
  tUpcall setState_;
```

```
  void *sEnv_;
};

// An instance of class MemberList contains the members of an
// object group. When a request arrives, the adaptor searches
// the target MemberList and invokes the request handling
// method of all list elements.
//
class MemberList {
public:
  MemberList(address& group, Member& e);

  address group_;                          // associated ISIS group,
  List <Member> members_;                  // objects in this process
                                           // being members of
                                           // the group.
};

// The ISIS Adaptor itself:
//
class IsisAD: public VirtualMachine{
public:
  IsisAD(char *name);
  ~IsisAD();

  // Initialization:
  //
  ORBStatus machInit();
  ORBStatus machQuit();
  ORBStatus electraInit(const AppPolicy&);
  ORBStatus electraQuit();

  // Entities:
  //
  ORBStatus entityCreate(Entity&, tPrio,
     const ProtocolPolicy&);
  ORBStatus entityDestroy(Entity&);
  boolean entityEqual(const Entity&, const Entity&);
```

```
// Asynchronous, reliable message passing:
//
ORBStatus msgSend(Entity& dest, Message& m,
    tUpcall sendDone, void *sEnv);
ORBStatus msgReceive(Entity& src, tUpcall rcv, void *rEnv,
                     tUpcall usr = 0, void *uEnv = 0);
ORBStatus msgReceiveDone(void *);

// Group management:
//
ORBStatus grpCreate(Entity& grp, const ProtocolPolicy&);
ORBStatus grpJoin(Entity& grp, Entity& e,
                  tUpcall mon, void *,
                  tUpcall getState, void *,
                  tUpcall setState, void *);
ORBStatus grpLeave(Entity& grp, Entity& e);
ORBStatus grpDestroy(Entity& grp);

// Thread management:
//
ORBStatus threadDeclare(Thread&, tUpcall f,
                        void *env = 0,
                        tPrio p = eMid,
                        int stack = MIN_STACK);
ORBStatus threadCreate(Thread& t, void *param);
ORBStatus threadDestroy(Thread& t);

// Failure Suspectors:
//
ORBStatus monMonitor(Monitor&, const Entity&, tUpcall, void *);
ORBStatus monCancel(Monitor&);

//    - semaphores:
//
ORBStatus semaCreate(Sema& s, int value);
ORBStatus semaDec(Sema& s);
ORBStatus semaInc(Sema& s);
ORBStatus semaDestroy(Sema& s);
```

```
protected:
  // convert an ISIS error code to a string:
  static char *ierr();
  // convert an Entity to an EntityAddr:
  static EntityAddr *e2e(const Entity& e);
  // get the MsgReceiveData structure for a given oid:
  static MsgReceiveData *getMsgReceiveData(int oid);
  // set the MsgReceiveData structure for a given oid:
  static void setMsgReceiveData(MsgReceiveData *, int oid);
  // register a member of an object group. Invoke pg_join
  // if required:
  static ORBStatus registerMember(char *name, const address&,
                                  const Entity&,
                                  tUpcall viewChange, void *,
                                  tUpcall getState, void *,
                                  tUpcall setState, void *);
  // unregister a member of an object group. Invoke pg_leave
  // if required:
  static ORBStatus unregisterMember(address&, Entity&);
  // destroy an object group:
  static ORBStatus unregisterGroup(address&);
  // invoke the request handling method of an object:
  static boolean invokeUpcall(int oid, char *msgBuf, boolean);

  // invoked when an ISIS message arrives for this process:
  THREAD static void msgReceiveThread(message *msg);
  THREAD static void electra_pre_main(void *, void *);
  // ISIS main task:
  THREAD static void maintask(void *);
  // ISIS group monitor:
  THREAD static void groupmon(groupview *gview_p, int arg);
  // helps in creating Electra threads:
  THREAD static void threadWrap(void *v);

  // helps in performing state transfer:
  THREAD static void tellState(int, address*);
  THREAD static void setState(int, message *);

private:
```

```
  // MsgReceiveData structures of the Electra objects in this process:
  //
  static MsgReceiveData *mrds_[MAX_MRDS];
  static List <MemberList> isisGroups_;
};
```

Implementation

```
#include <ElectraInc.hh>
#include <orb/CORBA.h>
#include <VMachine.hh>
#include <List.hh>

#include <isis.h>
#include "IsisAD.hh"


////////////////////////////////////////////////////////
//
// Class IsisAD
//
//

IsisAD::IsisAD(char *name)
  : VirtualMachine(name)
{
  ::memset(mrds_, 0, MAX_MRDS);
}

IsisAD::~IsisAD()
{
  semaDestroy(waitS_);
  for(int i = 0; i < MAX_MRDS; i++){
    delete mrds_[i];
  }
}

ORBStatus
IsisAD::machInit()
{
  ::isis_remote_init(0, 0, 0, ISIS_PANIC);
  ::isis_task((void (*)(...)) maintask, "IsisAd::maintask");
  ::isis_task((void (*)(...)) groupmon, "IsisAd::groupmon");
  ::isis_entry(1, msgReceiveThread, "IsisAd::msgReceiveThread");
```

```
  semaCreate(waitS_, 0);
  ::isis_mainloop(maintask);
  return okStat;
}

ORBStatus
IsisAD::machQuit()
{
  return okStat;
}

ORBStatus
IsisAD::electraInit(const AppPolicy& ap)
{
  if(!compatible(ap.threadModel(), CORBA::eNonPreemptive)
      || !compatible(ap.eventOrder(), CORBA::eVsync)
      || !compatible(ap.groupMgmt(), CORBA::eStrongGmp)){
    return apNotSupported;
  }

  init_ = TRUE;
  return okStat;
}

ORBStatus
IsisAD::electraQuit()
{
  return okStat;
}

ORBStatus
IsisAD::entityCreate(Entity& e, tPrio prio,
                     const ProtocolPolicy& proto)
{
  log→junk("IsisAD::entityCreate");

  EntityAddr eaddr;

  e.setPriv((char*)&eaddr, sizeof(EntityAddr));
```

```
  e.setVsync();

  return okStat;
}

ORBStatus
IsisAD::entityDestroy(Entity& e)
{
  log.junk("IsisAD::entityDestroy");
  // do nothing.
  return okStat;
}

boolean
IsisAD::entityEqual(const Entity& a, const Entity& b)
{
  return
    addr_isequal(&e2e(a)→addr_, &e2e(b)→addr_)
      && e2e(a)→oid_ == e2e(b)→oid_;
}

// send a message to destination "dest". Invoke "sendD" when
// finished. ("finished" means that resources associated with
// the message can be freed, and not that an ack has been received!).
//
ORBStatus
IsisAD::msgSend(Entity& dest, Message& m, tUpcall sendD, void *sEnv)
{
  EntityAddr *eaddr = (EntityAddr *)dest.getPriv();

  log→junk("IsisAD::msgSend called");

  // "fork-off" broadcast. Rendez-vous is handled by the RPC module
  // using semaphores. Therefore we don't need bc_wait().
  //
  // eaddr->oid_ is needed by msgReceiveThread only when the
  // destination is a singleton object:
  //
  if(::bcast_l("f", &eaddr→addr_, 1, "%A[1]%d%C", &eaddr→addr_,
```

```
                eaddr→oid_, m.getTotalBuf(), m.getTotalSize(), 0)
                == -1){
    log→panic("IsisAD::msgSend:  bcast:  %s", ierr());
  }

  // invoke the send-done upcall:
  //
  (*sendD)(sEnv, 0);
  return okStat;
}

// Registers a thread which will be invoked when a message
// arrives on Entity listenOn:
//
ORBStatus
IsisAD::msgReceive(Entity& listenOn,
                   tUpcall rcv, void *rEnv,
                   tUpcall usr, void *uEnv)
{
  log→junk("IsisAD::msgReceive");

  struct MsgReceiveData *mrd = new MsgReceiveData(rcv, rEnv,
      usr, uEnv, listenOn, FALSE);

  setMsgReceiveData(mrd, e2e(listenOn)→oid_);
  return okStat;
}

// invoked by the RPC layer when it is done with an RPC
// request or reply:
//
ORBStatus
IsisAD::msgReceiveDone(void *adaptorData)
{
  delete (char*) adaptorData;
  return okStat;
}

// create an object-group:
```

```
//
ORBStatus
IsisAD::grpCreate(Entity& grp, const ProtocolPolicy& p)
{
  address *addr;

  // first create an entity for the group:
  //
  entityCreate(grp, eMid, CORBA::group_proto);

  // Note that the first pg_join automatically creates an
  // ISIS process group:
  //
  if((addr = ::pg_join(grp.getName(), PG_XFER, 0, tellState,
                       setState, PG_MONITOR, groupmon, 0, 0))
                       == &NULLADDRESS){
    log->warn("IsisAD::grpCreate:  pg_join:  %s", ierr());
    return ORBStatus(SYSTEM_EXCEPTION, INTERNAL);
  }
  e2e(grp)->addr_ = *addr;

  return okStat;
}

// join an object-group:
//
ORBStatus
IsisAD::grpJoin(Entity& g, Entity& e,
                tUpcall viewChange, void *vEnv,
                tUpcall getState, void *tEnv,
                tUpcall setState, void *sEnv)
{
  log->junk("IsisAD::grpJoin:  joining group %s", g.getName());
  return registerMember(g.getName(), e2e(g)->addr_, e,
                        viewChange, vEnv,
                        getState, tEnv,
                        setState, sEnv);
}
```

```
// leave an object-group:
//
ORBStatus
IsisAD::grpLeave(Entity& g, Entity& e)
{
  log→junk("IsisAD::grpJoin:  leaving group %s", g.getName());
  return unregisterMember(e2e(g)→addr_, e);
}

// destroy an object-group:
//
ORBStatus
IsisAD::grpDestroy(Entity& g)
{
  address *addr;
  log→junk("IsisAD::grpDestroy:  destroying group %s",
     g.getName());
  return unregisterGroup(e2e(g)→addr_);
}

ORBStatus
IsisAD::threadDeclare(Thread& t, tUpcall f, void *env, tPrio p, int
stack)
{
  threadGet(t, f, env, p, stack);
  return okStat;
}

ORBStatus
IsisAD::threadCreate(Thread& t, void *param)
{
  t.setParam(param);
  ::t_fork(threadWrap, ((void*)&t));
  return okStat;
}

ORBStatus
IsisAD::threadDestroy(Thread& t)
{
```

```
  return okStat;
}

ORBStatus
IsisAD::monMonitor(Monitor& m, const Entity& e, tUpcall f, void *env)
{
  EntityAddr *eaddr = (EntityAddr *)e.getPriv();
  int wid;

  if((wid = ::proc_watch(&eaddr->addr_, f, env)) == -1){
    log.warn("IsisAD::monMonitor:  proc_watch:  %s", ierr());
    return ORBStatus(SYSTEM_EXCEPTION, INTERNAL,
        MAYBE, 0);
  }

  m.setPriv((char *)&wid, sizeof(wid));

  return okStat;
}

ORBStatus
IsisAD::monCancel(Monitor& m)
{
  int wid = *(int *)m.getPriv();

  if(::proc_watch_cancel(wid) == -1){
    log.warn("IsisAD::monMonitor:  proc_watch_cancel:  %s",
        ierr());
    return ORBStatus(SYSTEM_EXCEPTION, INTERNAL,
        MAYBE, 0);
  }

  return okStat;
}

ORBStatus
IsisAD::semaCreate(Sema& s, int value)
{
  s.setInitialized();
```

```
  IsisSema *sema = new IsisSema(value);
  s.setPriv((char *)sema);

  return okStat;
}

// decrement a semphore. Trying to decrement a semaphore whose
// value is 0 blocks the issuing thread.
//
ORBStatus
IsisAD::semaDec(Sema& s)
{
  if(!s.getInitialized()){
    log->panic("IsisAD::semaDec:  semaphore uninitialized");
  }

  IsisSema *sema = (IsisSema*)s.getPriv();

  if(--sema->val_ < 0){
    // wait until val_ >= 0
    //
    ::t_wait(&sema->cond_);
  }

  return okStat;
}

// increment a semaphore. If the value was 0 and one or more
// threads are blocked on this semaphore, unblock one thread:
//
ORBStatus
IsisAD::semaInc(Sema& s)
{
  if(!s.getInitialized()){
    log->panic("IsisAD::semaInc:  semaphore uninitialized");
  }

  IsisSema *sema = (IsisSema*)s.getPriv();
```

```
  if(++sema→val_ ≤ 0){
    // resume one blocked task:
    //
    ::t_sig(&sema→cond_, 0);
  }

  return okStat;
}

ORBStatus
IsisAD::semaDestroy(Sema& s)
{
  if(!s.getInitialized()){
    log→panic("IsisAD::semaDestroy:  semaphore uninitialized");
  }

  delete s.getPriv();

  return okStat;
}

// ——————————- UTILITIES ————————————
//

// wrapper method used by threadCreate to create an Electra thread:
//
void
IsisAD::threadWrap(void *v)
{
  Thread *t = (Thread*)v;
  (*t→getFunc())(t→getEnv(), t→getParam());
}

// msgReceiveThread is invoked by ISIS whenever a message
// arrives for this ISIS process. In case of a point-to-point
// message, %d contains the object ID of the target object,
// which is unique per process. In case of a group invocation,
// we iterate over isisGroups_, search the target object group,
```

```
// and invoke the upcalls of all members.
//
THREAD void
IsisAD::msgReceiveThread(message *msg)
{
  int oid;
  int msgLen;
  MsgReceiveData *pass, *pass2;
  char *msgBuf;
  boolean found = FALSE;
  address dest;

  log->junk("IsisAD::msgReceiveThread called");

  // read the received message. oid is the object id of the
  // receiver-object in this process:
  //
  if(::msg_get(msg, "%a%d%+C", &dest, &oid, &msgBuf, &msgLen) < 1){
    log->panic("IsisAD::msgReceiveThread:  malformed message");
  }

  // find all object-group members in this process which are
  // interested in the message:
  //
  ListIter <MemberList> iter(isisGroups_);
  MemberList *elem;

  while(elem = iter.next()){
    // find the destination process-group:
    //
    if(addr_isequal(&dest, &elem->group_)){
      found = TRUE;

      ListIter <Member> aIter(elem->members_);
      Member *aElem;

      // invoke the upcall of all objects registered for the group:
      //
      while(aElem = aIter.next()){
```

```
        if(!invokeUpcall(e2e(aElem→eid_)→oid_, msgBuf, FALSE)){
          log→panic("IsisAD::msgReceiveThread:  bad addr:  %d",
                    e2e(aElem→eid_)→oid_);
        }
      }
    }
  }

  if(!found){
    // If we get here, the destination was a singleton object. Simply
    // invoke its upcall.
    //
    if(!invokeUpcall(oid, msgBuf, TRUE)){
      log→panic("IsisAD::msgReceiveThread:  bad addr:  %d",
          oid);
    }
  } else {
    delete msgBuf;
  }
}

// invoke the request handling upcall of the Electra object whose
// ID is "oid":
//
boolean
IsisAD::invokeUpcall(int oid, char *msgBuf, boolean deleteBuf)
{
  MsgReceiveData *pass, *pass2 = new MsgReceiveData;

  if(!(pass = getMsgReceiveData(oid))){
    return FALSE;
  }

  pass→buf_ = msgBuf;
  pass→deleteBuf_ = deleteBuf;
  pass→priv_ = 0;

  // save the MsgReceiveData object:
  //
```

```
  *pass2 = *pass;

  // invoke the registered user thread:
  (pass2→rcv_)(pass2→rEnv_, pass2);

  return TRUE;
}

// state transfer. Isis invokes tellState to obtain a
// group-member's state when a new process joins the group.
// All we have to do is searching an arbitrary object which
// is member of the group, and invoke its tellState method:
//
void
IsisAD::tellState(int, address *grp)
{
  address *addr = 0;
  ListIter <MemberList> iter(isisGroups_);
  MemberList *elem;
  Sequence <CORBA::Any,0ul> state;

  // Pick the first object which is member of the group:
  //
  while(elem = iter.next()){
    // does the group exists?
    //
    if(addr_isequal(grp, &elem→group_)){
      (*elem→members_.getFirstElem()→tellState_)
        (elem→members_.getFirstElem()→tEnv_, &state);

      u_int size;
      addr = &elem→group_;

      char *buf = new char [size = ::dumpSize(state)];
      ::dump(state, buf);
      ::xfer_out(0, "%A[1]%C", addr, buf, size);
      delete buf;
      return;
    }
```

```
  }

  log.warn("IsisAD::tellState:  couldn't get state");
}

// invoked by Isis when a new process joins a process group.
// All we have to do is to search the object which joined the
// group and invoke its setState method:
//
void
IsisAD::setState(int, message *msg)
{
  Sequence <CORBA::Any,0ul> state;
  char *stateBuf;
  int stateBufLen;
  address addr;
  ::msg_get(msg, "%a%+C", &addr, &stateBuf, &stateBufLen);
  ::undump(state, stateBuf);
  delete stateBuf;

  ListIter <MemberList> iter(isisGroups_);
  MemberList *elem;

  // search the object which joined the group:
  //
  while(elem = iter.next()){
    // does the group exists?
    //
    if(addr_isequal(&addr, &elem->group_)){
      ListIter <Member> aIter(elem->members_);
      Member *aElem;

      // is the object a member of the group?
      //
      while(aElem = aIter.next()){
        (*aElem->setState_)(aElem->sEnv_, &state);
      }
    }
    return;
```

```
  }

  log.warn("IsisAD::setState:  no object found");
  return;
}

void
IsisAD::maintask(void *v)
{
  ::isis_start_done();
  electra_main(gArgc, gArgv);
  quit_adaptors();
  ::isis_disconnect();
}

char *
IsisAD::ierr()
{
  return IE_errors[-::isis_errno];
}

EntityAddr *
IsisAD::e2e(const Entity& e)
{
  return (EntityAddr*)e.getPriv();
}

void
IsisAD::setMsgReceiveData(MsgReceiveData *mrd, int oid)
{
  mrds_[oid] = mrd;
}

MsgReceiveData *
IsisAD::getMsgReceiveData(int oid)
{
  return mrds_[oid];
}
```

```
// join an object to an ISIS process group. Since a process
// might contain several objects, and two object may
// want to join the same group, registerMember() takes care
// that pg_join is issued only once per process.
//
ORBStatus
IsisAD::registerMember(char *name, const address& g, const Entity& e,
                       tUpcall viewChange, void *vEnv,
                       tUpcall tState, void *tEnv,
                       tUpcall sState, void *sEnv)
{
  ListIter <MemberList> iter(isisGroups_);
  MemberList *elem;

  while(elem = iter.next()){
    // does the group exists already?
    //
    if(addr_isequal(&g, &elem->group_)){
      ListIter <Member> aIter(elem->members_);
      Member *aElem;

      // is the object member of the group already?
      //
      while(aElem = aIter.next()){
        if(e2e(aElem->eid_)->oid_ == e2e(e)->oid_){
          log.warn("IsisAD::registerMember:  already member");
          return ORBStatus(SYSTEM_EXCEPTION, IMP_LIMIT);
        }
      }

      // register the object with the group:
      //
      elem->members_.append(Member(e,
                                   viewChange, vEnv,
                                   tState, tEnv,
                                   sState, sEnv));

      // state transfer:
      //
```

```
      if(elem→members_.getNumElems() > 1){
        Sequence <CORBA::Any,0ul> state;
        (*elem→members_.getFirstElem()→tellState_)
          (elem→members_.getFirstElem()→tEnv_, &state);
        (*elem→members_.getLastElem()→setState_)
          (elem→members_.getLastElem()→sEnv_, &state);
      }

      return okStat;
    }
  }

  // If we get here, the process is not member of the group:
  //
  isisGroups_.append(MemberList(g, Member(e, viewChange, vEnv,
                                          tState, tEnv,
                                          sState, sEnv)));

  if(::pg_join(name, PG_XFER, 0, tellState, setState, PG_MONITOR,
               groupmon, 0, 0) == &NULLADDRESS){
    log→warn("IsisAD::registerMember:  pg_join:  %s", ierr());
    return ORBStatus(SYSTEM_EXCEPTION, INTERNAL);
  }

  return okStat;
}

// leave an object group. Issue pg_leave after the last
// member-object in this process left:
//
ORBStatus
IsisAD::unregisterMember(address& g, Entity& e)
{
  ListIter <MemberList> iter(isisGroups_);
  MemberList *elem;

  while(elem = iter.next()){
    // does the group exists?
    //
```

```
    if(addr_isequal(&g, &elem→group_)){

      ListIter <Member> aIter(elem→members_);
      Member *aElem;

      // is the object a member of the group?
      //
      while(aElem = aIter.next()){

        if(e2e(aElem→eid_)→oid_ == e2e(e)→oid_){
          aIter.remove();
          if(elem→members_.getNumElems() == 0){
            iter.remove();
            if(::pg_leave(&g) == -1){
              log→warn("IsisAD::unregisterMember:  %s",
                  ierr());
              return ORBStatus(SYSTEM_EXCEPTION,
                  INTERNAL);
            }
            return okStat;
          }
        }
      }
    }

  // illegal group or illegal object specified
  //
  return ORBStatus(USER_EXCEPTION, INV_OBJREF);
}

// delete a whole object group:
//
ORBStatus
IsisAD::unregisterGroup(address& g)
{
  ListIter <MemberList> iter(isisGroups_);
  MemberList *elem;
```

```
  while(elem = iter.next()){
    // does the group exists?
    //
    if(addr_isequal(&g, &elem→group_)){
      iter.remove();

      if(::pg_delete(&g) == -1){
        log→warn("IsisAD::unregisterGroup:  pg_delete:  %s",
            ierr());
        return ORBStatus(USER_EXCEPTION, INV_OBJREF,
            NO, 0);
      }

      return okStat;
    }
  }

  return ORBStatus(USER_EXCEPTION, INV_OBJREF, NO, 0);
}

void
IsisAD::groupmon(groupview *gview_p, int arg)
{
  address *group = &gview_p→gv_gaddr;
  ListIter <MemberList> iter(isisGroups_);
  MemberList *elem;
  GroupInfo *gInfo;

  // iterate through all object groups:
  //
  while(elem = iter.next()){
    // search the object-group whose view changed:
    //
    if(addr_isequal(group, &elem→group_)){
      ListIter <Member> aIter(elem→members_);
      Member *aElem;
      // invoke the viewChange method of all member objects:
      //
      while(aElem = aIter.next()){
```

```
        gInfo = new GroupInfo(addr_isnull(&gview_p→gv_joined)
            ? -1 : 1);
        (*aElem→viewChange_)(aElem→vEnv_, gInfo);
      }
    }
  }
}

MsgReceiveData *
IsisAD::mrds_[MAX_MRDS];

List <MemberList>
IsisAD::isisGroups_;


//////////////////////////////////////////////////////
//
// Class EntityAddr
//
//

EntityAddr::EntityAddr()
{
  addr_ = ::my_address;
  oid_ = nextOid_++;

  if(oid_ == MAX_MRDS){
    log.panic("EntityAddr::EntityAddr:  out of object ids");
  }
}

int
EntityAddr::nextOid_ = 0;

//////////////////////////////////////////////////////
//
// Class IsisSema
//
//
```

```
IsisSema::IsisSema(int val)
  : val_(val), cond_(0)
{
}

////////////////////////////////////////////////////////
//
// Class Member
//
//

Member::Member(const Entity& eid,
               tUpcall viewChange, void *vEnv,
               tUpcall tellState, void *tEnv,
               tUpcall setState, void *sEnv)
  : eid_(eid), viewChange_(viewChange), vEnv_(vEnv),
    tellState_(tellState), tEnv_(tEnv), setState_(setState),
    sEnv_(sEnv)
{
}

////////////////////////////////////////////////////////
//
// Class MemberList
//
//

MemberList::MemberList(address& group, Member& e)
  : group_(group)
{
  members_.append(e);
}
```

Bibliography

[ABLN85] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE11(1), January 1985.

[ADKM92a] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms for Multicast Communication Groups. In *Proceedings of the 6th Intl. Workshop on Distributed Algorithms (WDAG-6)*. Springer-Verlag, November 1992. LNCS 647.

[ADKM92b] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd International Symposium on Fault-Tolerant Computing*. IEEE, July 1992.

[Agh86] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.

[And94] G. Andert. Object Frameworks in the Taligent OS. In *Spring COMPCON 94*. IEEE Press, February 1994.

[APR93] R. Aiello, E. Pagani, and G. P. Rossi. Causal Ordering in Reliable Group Communications. *ACM SIGCOMM Computer Communication Review*, 23(4), September 1993.

[Ash93] C. Ashford. *Comparison of the OMG and the ISO/CCITT Object Models*. Telecom Canada, Ottawa, Ontario, April 1993.

[Aut92] M. AUTRATA. DME Overview. In *IEEE NOMS92, Proceedings of the Network Operations and Management Symposium*, 1992.

[Avi85] A. AVIZIENIS. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12), December 1985.

[Bal90] H. E. BAL. *Programming Distributed Systems.* Silicon Press, first edition, 1990.

[BC94] K. P. BIRMAN AND T. CLARK. Performance of the Isis Distributed Computing Toolkit. Technical Report 94-1432, Department of Computer Science, Cornell University, June 1994.

[BCG91] K. P. BIRMAN, R. COOPER, AND B. GLEESON. Design Alternatives for Process Group Membership and Multicast. Technical Report 91-1257, Department of Computer Science, Cornell University, December 1991. Submitted to IEEE Transactions on Parallel and Distributed Systems.

[BG93] K. P. BIRMAN AND B. B. GLADE. Consistent Failure Reporting in Reliable Communication Systems. Technical Report 93-1349, Department of Computer Science, Cornell University, May 1993.

[BHG87] P. A. BERNSTEIN, V. HADZILACOS, AND N. GOODMAN. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[BHJ+87] A. BLACK, N. HUTCHINSON, E. JUL, H. LEVY, AND L. CARTE. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, SE13(1), January 1987.

[Bir93a] K. P. BIRMAN. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12), December 1993.

[Bir93b] K. P. BIRMAN. Integrating Runtime Consistency Models for Distributed Computing. Technical Report 91-1240, Department of Computer Science, Cornell University, July 1993. To appear in Journal of Parallel and Distributed Computing.

[BJ87a] K. P. BIRMAN AND T. A. JOSEPH. Exploiting Virtual Synchrony in Distributed Systems. *ACM SIGOPS Operating Systems Review*, 21(5), November 1987.

[BJ87b] K. P. BIRMAN AND T. A. JOSEPH. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1), February 1987.

[BJ89] K. P. BIRMAN AND T. A. JOSEPH. Exploiting Replication in Distributed Systems. In S. MULLENDER, editor, *Distributed Systems*. ACM Press, 1989.

[BKMS95] W. R. BISCHOFBERGER, T. KOFLER, K.-U. MÄTZEL, AND B. SCHÄFFER. Computer Supported Cooperative Software Engineering with Beyond-Sniff. In *Proceedings of the 7th Conference on Software Engineering Environments*, Noorwijkerhout, The Netherlands, 1995.

[BM93] Ö. BABAOĞLU AND K. MARZULLO. Consistent Global States. In S. MULLENDER, editor, *Distributed Systems*, chapter 4. Addison Wesley, second edition, 1993.

[BMST93] N. BUDHIRAJA, K. MARZULLO, F. B. SCHNEIDER, AND S. TOUEG. The Primary-Backup Approach. In S. MULLENDER, editor, *Distributed Systems*, chapter 8. Addison Wesley, second edition, 1993.

[BNOW93] A. BIRRELL, G. NELSON, S. OWICKI, AND E. WOBBER. Network Objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. ACM Press, December 1993.

[BNOW94] A. BIRRELL, G. NELSON, S. OWICKI, AND E. WOBBER. Network Objects. Technical Report 115, Digital Systems Research Center, Palo Alto, CA, February 1994.

[Boo91] G. BOOCH. *Object-Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, 1991.

[BRH+93] R. Bhoedjang, T. Ruhl, R. Hofman, K. Langendoen, H. Bal, and F. Kaashoek. Panda: A Portable Platform to Support Parallel Programming Languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, San Diego, CA, September 1993. USENIX.

[Bro92] M. L. Brodie. The Promise of Distributed Computing and the Challenges of Legacy Information Systems. In P. M. D. Gray and R. J. Lucas, editors, *Advanced Database Systems: Proceedings of the 10th British National Conference on Databases*. Springer-Verlag, 1992.

[BSS91] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[BvR94] K. P. Birman and R. van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[CB94] K. Cho and K. P. Birman. A Group Communication Approach to Mobile Computing — MobileChannel: an ISIS Tool for Mobile Services. Technical Report 94-1424, Department of Computer Science, Cornell University, May 1994.

[CBHRdP93] V. Cahill, R. Balter, N. R. Harris, and X. Rousset de Pina. *The Comandos Distributed Application Platform*. Research Reports ESPRIT, Project 2071, COMANDOS, Volume 1. Springer-Verlag, 1993.

[CFH+93] P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte, and X. Rousset de Pina. Experience with Shared Object Support in the Guide System. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems IV*, San Diego, CA, 1993. USENIX.

[CGA86] N. Carriero, D. Gelernter, and S. Ahuija. Linda and Friends. *IEEE Computer*, 19(8), August 1986.

[CI94] F. A. Cummins and M. H. Ibrahim. Wrapping Legacy Applications. *First Class — The Object Management Group Newsletter*, 4(2), April 1994.

[Cla85] D. CLARK. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, Orcas Island, December 1985.

[CM84] J. CHANG AND N. F. MAXEMCHUK. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3), August 1984.

[CMRS93] C. CAP, S. MAFFEIS, L. RICHTER, AND V. STRUMPEN. Ein Glossar wichtiger Begriffe zu verteilten und parallelen Systemen. Technical Report 93.44, CS Dept. University of Zurich, November 1993. Begleitmaterial zum Fortbildungsseminar Verteilte Systeme, Sept. 1993.

[CMRW93] J. CASE, K. MCCLOGHRIE, M. ROSE, AND S. WALDBUSSER. Introduction to Version 2 of the Internet-Standard Network Management Framework. RFC 1441, Request for Comments, April 1993.

[CNL89] S. T. CHANSON, G. W. NEUFELD, AND L. LIANG. A Bibliography on Multicast and Group Communication. *ACM SIGOPS Operating Systems Review*, 23(4), October 1989.

[Cri89] F. CRISTIAN. Probabilistic Clock Synchronization. *Distributed Computing, Springer-Verlag*, 3(3), July 1989.

[CS93a] C. CAP AND V. STRUMPEN. Efficient Parallel Computing in Distributed Workstation Environments. *Parallel Computing*, 19(11), November 1993. Springer-Verlag.

[CS93b] D. CHERITON AND D. SKEEN. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, North Carolina, December 1993.

[CSB92] G. COULSON, J. SMALLEY, AND G. S. BLAIR. The Design and Implementation of a Group Invocation Facility in ANSA. Technical Report MPG-92-34, Distributed Multimedia Research Group, Department of Computing, Lancaster University, Lancaster, UK, 1992.

[CT93] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. Technical Report 93-1374, Department of Computer Science, Cornell University, August 1993.

[Cus93] H. Custer. *Inside Windows NT.* Microsoft Press, 1993.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, second edition, 1991.

[Dee89] S. Deering. Host Extensions for IP Multicasting. RFC 1112, Request for Comments, August 1989.

[Den92] T. G. Dennehy. Class Libraries as an Alternative to Language Extensions for Distributed Programming. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III).* USENIX Association, March 1992.

[Dig93] Digital Equipment Corp., Hewlett-Packard Co., HyperDesk Corp., NCR Corp., Object Design Inc., SunSoft Inc. *The Common Object Request Broker: Architecture and Specification*, December 1993. Revision 1.2.

[Dij65] E. W. Dijkstra. Co-operating Sequential Processes. In F. Genuys, editor, *Programming Languages.* Academic Press, London, 1965.

[DKM93] D. Dolev, S. Kramer, and D. Malki. Early Delivery Totally Ordered Multicast in Asynchronous Environments. In *The Twenty-Third International Symposium on Fault-Tolerant Computing.* IEEE, June 1993.

[DLA88] P. Dasgupta, R. J. LeBlanc, Jr., and W. F. Appelbe. The Clouds Distributed Operating System. In *The 8th International Conference on Distributed Computer Systems.* IEEE Computer Society Press, June 1988.

[DM92] W. H. Davidow and M. S. Malone. *The Virtual Corporation.* HarperCollins Publishers, 1992.

[DPJ90] F. De Paoli and M. Jazayeri. FLAME: A Language for Distributed Programming. In *Proceedings of IEEE International Conference on Computer Languages.* IEEE Computer Society, March 1990.

[Dra90] R. P. DRAVES. The Revised IPC Interface. In *Proceedings of the First USENIX Conference on MACH.* USENIX, 1990.

[DSC92] A. DAVE, M. SEFIKA, AND R. H. CAMPBELL. Proxies, Application Interfaces, and Distributed Systems. In L.-F. CABRERA, editor, *Proceedings of the Second International Workshop on Object Orientation in Operating Systems.* IEEE Computer Society Press, September 1992.

[EMS91] J. L. EPPINGER, L. B. MUMMERT, AND A. Z. SPECTOR. *Camelot and Avalon.* Morgan Kaufmann Publishers, Inc., 1991.

[ES91] M. A. ELLIS AND B. STROUSTRUP. *The Annotated C++ Reference Manual.* Addison Wesley, 1991.

[FLP85] M. J. FISCHER, N. A. LYNCH, AND M. S. PATERSON. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2), April 1985.

[GGM94] B. GARBINATO, R. GUERRAOUI, AND K. R. MAZOUNI. Distributed Programming in GARF. In *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming*, Lecture Notes in Computer Science 791. Springer-Verlag, 1994.

[GL93] R. A. GOLDING AND D. D. E. LONG. Using an Object-Oriented Framework to Construct Wide-Area Group Communication Mechanisms. Technical Report UCSC-CLR-93-11, University of California, Santa Cruz, March 1993.

[GM93] A. GATTI AND S. MAFFEIS. Effiziente Riskio- und Ertrags-Optimierung. *Schweizer Bank, SHZ Fachverlag*, November 1993.

[GM94a] A. GATTI AND S. MAFFEIS. Erhöhte Flexiblität im Global Risk Management. *Schweizer Bank, SHZ Fachverlag*, February 1994.

[GM94b] A. GATTI AND S. MAFFEIS. Nichts geht ohne Standards und neue Technologien. *Schweizer Bank, SHZ Fachverlag*, March 1994.

[Gol92] R. A. GOLDING. Weak Consistency Group Communication for Wide-Area Systems. In *Proceedings of the 2nd IEEE Workshop on the Management of Replicated Data*. IEEE, November 1992.

[GS91] J. GRAY AND D. P. SIEWIOREK. High-Availability Computer Systems. *IEEE Computer*, September 1991.

[GS94] R. GUERRAOUI AND A. SCHIPER. Transaction Model vs. Virtual Synchrony Model: Bridging the Gap. In K. P. BIRMAN, F. CHRISTIAN, F. MATTERN, AND A. SCHIPER, editors, *Distributed Systems: From Theory to Practice*, Lecture Notes in Computer Science. Springer-Verlag, 1994.

[GT92] R. A. GOLDING AND K. TAYLOR. Group Membership in the Epidemic Style. Technical report, University of California, Santa Cruz, 1992.

[Her94] A. HERBERT. Distributing Objects. In F. BRAZIER AND D. JOHANSEN, editors, *Distributed Open Systems*. IEEE Computer Society Press, 1994.

[HHBC92] O. HAGSAND, H. HERZOG, K. BIRMAN, AND R. COOPER. Object-Oriented Reliable Distributed Programming. In *Proceedings of the IEEE Workshop on Object Orientation in Operating Systems*, Paris, France, September 1992.

[HK93] G. HAMILTON AND P. KOUGIOURIS. The Spring Nucleus: A Microkernel for Objects. Technical Report SMLI TR-93-14, Sun Microsystems Laboratories, Inc., April 1993.

[Höl94] U. HÖLZLE. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. PhD thesis, Computer Science Department, Stanford University, August 1994.

[HR94] G. HAMILTON AND S. RADIA. Using Interface Inheritance to Address Problems in System Software Evolution. *ACM SIGPLAN Notices*, 29(8), August 1994.

[HT93] V. HADZILACOS AND S. TOUEG. Fault-Tolerant Broadcasts and Related Problems. In S. MULLENDER, editor, *Distributed Systems*, chapter 5. Addison Wesley, second edition, 1993.

[HW90] M. P. HERLIHY AND J. M. WING. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[Hyp93] Hyperdesk Corporation. *Revised Submission in Response to the OMG RFP for a C++ Language Mapping*, September 1993. OMG Document 93-9-2.

[IEE85] IEEE, New York. 802.3*: Carrier Sense Multiple Access with Collision Detection*, 1985.

[Int93] International Business Machines Corporation. *SOMobjects Developer Toolkit – An Overview*, Version 2.0 edition, June 1993.

[Isi92] Isis Distributed Systems Inc. *The Isis Distributed Toolkit Version* 3.0*, User Reference Manual*, 1992.

[Isi93a] Isis Distributed Systems, Inc., 111 Cayuga Street, Suite 200, Ithaca, NY 14850. *Preliminary User Documentation for RDO/C++*, Revision 1.0.2 edition, August 1993.

[Isi93b] Isis Distributed Systems Inc., Ithaca, NY. *Object Groups: A Response to the ORB 2.0 RFI*, April 1993.

[J+94] C. JACQUEMOT ET AL. COOL: The CHORUS CORBA Compliant Framework. In *Spring COMPCON 94*. IEEE Press, February 1994.

[JF88] R. E. JOHNSON AND B. FOOTE. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2), June 1988.

[JFR93] F. JAHANIAN, S. FAKHOURI, AND R. RAJKUMAR. Processor Group Membership Protocols: Specification, Design and Implementation. In *Proceedings of the* 12*th Symposium on Reliable Distributed Systems*, Princeton, New Jersey, October 1993. IEEE.

[Kaa92] M. F. KAASHOEK. *Group Communication in Distributed Computer Systems.* PhD thesis, Vrije Universiteit, Amsterdam, 1992.

[Keh92] B. P. KEHOE. Zen and the Art of the Internet, 1992.

[KGR94] F. J. KOVAC, J. H. GILCHRIST, AND W. H. ROBINSON. Goodyear's Virtual Enterprise 2000. *HPCwire Magazine for High-Performance Computing*, Issue 4010, April 1994.

[KL91] D. G. KAFURA AND G. LAVENDER. Recent Progress in Combining Actor-Based Concurrency with Object-Oriented Programming. *ACM OOPS Messenger, Proceedings OOPSLA/ECOOP 90 Workshop on Object-Based Concurrent Systems*, 2(2):55–58, April 1991.

[Kra93] S. KRAKOWIAK. Issues in Object-Oriented Distributed Systems. In *Proceedings of the IFIP WG10.3 International Conference on Decentralized and Distributed Systems*, Palma de Mallorca, Spain, September 1993.

[KTHB89] M. F. KAASHOEK, A. S. TANENBAUM, S. F. HUMMEL, AND H. E. BAL. An Efficient Reliable Broadcast Protocol. *ACM SIGOPS Operating Systems Review*, 23(4), October 1989.

[KvRvST93] M. F. KAASHOEK, R. VAN RENESSE, H. VAN STAVEREN, AND A. S. TANENBAUM. FLIP: An Internetwork Protocol for Supporting Distributed Systems. *ACM Transactions on Computer Systems*, 11(1), February 1993.

[Lam78] L. LAMPORT. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.

[Lam86] B. W. LAMPSON. Designing a Global Name Service. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, Calgary, Canada, 1986.

[Lis88] B. LISKOV. Distributed Programming in Argus. *Communications of the ACM*, 31(3), March 1988.

[LKBT92] W. G. Levelt, M. F. Kaashoek, H. E. Bal, and A. S. Tanenbaum. A Comparison of Two Paradigms for Distributed Shared Memory. *Software – Practice and Experience*, 22(11), 1992.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. *ACM SIGPLAN Notices*, 23(7), July 1988.

[LSP82] L. Lamport, R. Shostak, and M. Paese. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[MADK94] D. Malki, Y. Amir, D. Dolev, and S. Kramer. The Transis Approach to High Availability Cluster Communication. Technical Report CS94-14, The Hebrew University of Jerusalem, 1994.

[Maf92] S. Maffeis. Normen und Standards gewinnen an Bedeutung. *Up to Data – Ruf AG*, 7(1):1–3, January 1992.

[Maf93a] S. Maffeis. Cache Management Algorithms for Flexible Filesystems. *ACM SIGMETRICS Performance Evaluation Review*, 21(2), December 1993.

[Maf93b] S. Maffeis. Electra — Making Distributed Programs Object-Oriented. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems IV*, San Diego, CA, 1993. USENIX.

[Maf93c] S. Maffeis. File Access Patterns in Public FTP Archives and an Index for Locality of Reference. *ACM SIGMETRICS Performance Evaluation Review*, 20(3), March 1993.

[Maf93d] S. Maffeis. Remote Method Calling and Object Group Communication. ECOOP Workshop on Object-based Distributed Programming, Kaiserslautern, Germany, July 1993.

[Maf93e] S. Maffeis. Technologische Grundlagen des Computer Supported Cooperative Work. Technical Report 93.29, CS Dept. University of Zurich, July 1993. Doktoranden Seminar Sommersemester 1993, Prof. Bauknecht und Prof. Tjoa.

[Maf94a] S. MAFFEIS. A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming. In R. GUERRAOUI, O. NIERSTRASZ, M. RIVEILL, editor, *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming*, Lecture Notes in Computer Science 791. Springer-Verlag, 1994.

[Maf94b] S. MAFFEIS. Design and Implementation of a Configurable Mixed–Media Filesystem. *ACM SIGOPS Operating Systems Review*, 28(4), October 1994.

[Maf94c] S. MAFFEIS. System Support for Distributed Computing. In W. GENTZSCH AND U. HARMS, editors, *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe 1994*, Lecture Notes in Computer Science 797. Springer-Verlag, 1994.

[MBM95] S. MAFFEIS, W. BISCHOFBERGER, AND K.-U. MÄTZEL. A Generic Multicast Transport Service to Support Disconnected Operation. In *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, Ann Arbor, MI, April 1995. USENIX.

[MC92] S. MAFFEIS AND C. H. CAP. Replication Heuristics and Polling Algorithms for Object Replication and a Replicating File Transfer Protocol. Technical Report 92.06, CS Dept. University of Zurich, June 1992.

[MCWB94] K. MARZULLO, R. COOPER, M. D. WOOD, AND K. P. BIRMAN. Tools for Distributed Application Management. In T. L. CASAVANT AND M. SINGHAL, editors, *Readings in Distributed Systems*. IEEE Computer Society Press, 1994.

[Mey88] B. MEYER. *Object Oriented Software Constrution*. Prentice Hall, 1988.

[MMSA93] L. E. MOSER, P. M. MELLIAR-SMITH, AND V. AGRAWAL. Necessary and Sufficient Conditions for Broadcast Consensus Protocols. *Distributed Computing, Springer-Verlag*, 7(2), December 1993.

[Moc87] P. V. MOCKAPETRIS. Domain Names – Concepts and Facilities. RFC 1034, Request for Comments, November 1987.

[MPI94] MPI: A MESSAGE-PASSINGE INTERFACE STANDARD. Message Passinge Interface Forum, May 1994.

[MPS93a] S. MISHRA, L. L. PETERSON, AND R. D. SCHLICHTING. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. Technical report, Department of Computer Science, The University of Arizona, 1993.

[MPS93b] S. MISHRA, L. L. PETERSON, AND R. D. SCHLICHTING. Experience with Modularity in Consul. Technical report, Department of Computer Science, The University of Arizona, 1993.

[Mul89] S. MULLENDER. *Distributed Systems.* ACM Press, 1989.

[MvRT+90] S. J. MULLENDER, G. VAN ROSSUM, A. S. TANENBAUM, R. VAN RENESSE, AND H. VAN STAVEREN. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, 23(5), May 1990.

[Nel81] B. NELSON. *Remote Procedure Call.* PhD thesis, Carnegie Mellon University, 1981. CMU-CS-81-119.

[NWM93] J. R. NICOL, C. T. WILKES, AND F. A. MANOLA. Object Orientation in Heterogeneous Distributed Computing Systems. *IEEE Computer*, 26(6), June 1993.

[OPSS93] B. OKI, M. PFLUEGL, A. SIEGEL, AND D. SKEEN. The Information Bus — An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, North Carolina, December 1993.

[Pan] J. PANEPINTO. UNIX Linda Enables Parallel Processing. Digital Equipment Corporation.

[Par92] P. PARDYAK. Group Communication in an Object-Based Environment. In L.-F. CABRERA, editor, *Proceedings of the Second International Workshop on Object Orientation in Operating Systems.* IEEE Computer Society Press, September 1992.

[PBB+94] D. Powell, P. Barrett, G. Bonn, M. Chérèque, D. Seaton, and P. Veríssimo. The Delta-4 Distributed Fault-Tolerant Architecture. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Systems*. IEEE Computer Society Press, 1994.

[PBS89] L. Peterson, N. Buchholz, and R. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, 7(3), August 1989.

[PHOH90] L. Peterson, N. Hutchinson, S. O'Malley, and H. Rao. The x-kernel: A Platform for Accessing Internet Resources. *IEEE Computer*, 23(5), May 1990.

[Pow94] D. Powell. Distributed Fault Tolerance: Lessons Learned with Delta-4. *IEEE Micro*, February 1994.

[PS93] F. Pacull and A. Sandoz. R-Linearizability: An Extension of Linearizability to Replicated Objects. In *Proceedings of the Fourth Workshop on Future Trends of Distributed Systems*. IEEE, 1993.

[PW88] L. J. Pinson and R. S. Wiener. *An Introduction to Object-Oriented Programming and Smalltalk*. Addison Wesley, 1988.

[R+88] M. Rozier et al. Chorus Distributed Operating Systems. *Computing Systems Journal, The Usenix Association*, 1(4), 1988.

[Ric92] A. M. Ricciardi. *The Group Membership Problem in Asynchronous Systems*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, November 1992. No. 92-1313.

[RSB93] A. Ricciardi, A. Schiper, and K. P. Birman. Understanding Partitions and the "No Partition" Assumption. Technical Report 93-1355, Department of Computer Science, Cornell University, June 1993.

[S+89] M. Shapiro et al. SOS: An Object-oriented Operating System – Assessment and Perspectives. *Computing Systems*, 2(4), December 1989.

[Sch86] K. J. SCHMUCKER. *Object-Oriented Programming for the Macintosh.* Hayden Hasbrouck Heights, Hasbrouck Heights, New Jersey, 1986.

[Sch90] G. SCHOINAS. Issues on the Implementation of POSYBL. Technical report, University of Crete, Greece, 1990.

[Sch93a] F. B. SCHNEIDER. Replication Management using the State-Machine Approach. In S. MULLENDER, editor, *Distributed Systems*, chapter 7. Addison Wesley, second edition, 1993.

[Sch93b] F. B. SCHNEIDER. What Good are Models and What Models are Good? In S. MULLENDER, editor, *Distributed Systems*, chapter 2. Addison Wesley, second edition, 1993.

[SDP] S. K. SHRIVASTAVA, G. N. DIXON, AND G. D. PARRINGTON. An Overview of the Arjuna Distributed Programming System. Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK.

[Sem93] Semaphore, 800 Turnpike Street, North Andover, MA 01845. *Glossary of Object-Oriented Terminology*, 1993.

[SH94] S. SANKAR AND R. HAYES. ADL — An Interface Definition Language for Specifying and Testing Software. *ACM SIGPLAN Notices*, 29(8), August 1994.

[Sha86] M. SHAPIRO. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *6th International Conference on Distributed Computer Systems*, May 1986.

[Shr94] S. K. SHRIVASTAVA. To CATOCS or not to CATOCS, that is the .... *ACM SIGOPS Operating Systems Review*, 28(4), October 1994.

[SK92] M. SINGHAL AND A. KSHEMKALYANI. An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 43, 1992. Amsterdam, North-Holland.

[SM91] M. SEQUEIRA AND J. A. MARQUES. Can C++ be Used for Programming Distributed and Persistent Objects? In LOUIS-FELIPE CABRERA, VINCENT RUSSO, AND MARC SHAPIRO, editor, *Proceedings of the International Workshop on Object*

Orientation in Operating Systems. IEEE Computer Society Press, October 1991.

[SM94a] L. SABEL AND K. MARZULLO. Simulating Fail-Stop in Asynchronous Distributed Systems. Technical Report 94-1413, Department of Computer Science, Cornell University, March 1994.

[SM94b] R. SCHWARZ AND F. MATTERN. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing, Springer-Verlag*, 7(3), March 1994.

[Spi88] J. M. SPIVEY. Understanding Z, A Specification Language and its Formal Semantics. *Tracts in Theoretical Computer Science*, 3, 1988. Cambridge University Press.

[SR93] A. SCHIPER AND A. RICCIARDI. Virtually-Synchronous Communication Based on Weak Failure Suspectors. In *Proceedings of the 23rd International Symposium on Fault-Tolerant computing*, Toulouse, June 1993. IEEE.

[SRC84] J. H. SALTZER, D. P. REED, AND D. D. CLARK. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4), November 1984.

[SS83] R. D. SCHLICHTING AND F. B. SCHNEIDER. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems*, 1(3), August 1983.

[SS92] D. P. SIEWIOREK AND R. W. SWARZ. *Reliable Computer Systems: Design and Evaluation.* Digital Press, Bedford, Mass., 1992.

[SS93] A. SCHIPER AND A. SANDOZ. Understanding the Power of the Virtually-Synchronous Model. In *Proceedings of the 5th European Workshop on Dependable Computing*, Lisbon, February 1993.

[Str88] B. STROUSTRUP. Parametrized Types for C++. In *USENIX C++ Conference*, Denver, October 1988. USENIX Association.

[Str93] H. STRICKLAND. Taking the Lid Off C++ — Effectively Using Runtime Type Information. *C++ Report*, 5(2), February 1993.

[Tan92] A. S. TANENBAUM. *Modern Operating Systems.* Prentice Hall, 1992.

[The94a] The Isis Group. *Horus Socket Interface*, May 1994.

[The94b] The Isis Group. *MUTS Documentation*, May 1994.

[TvR85] A. S. TANENBAUM AND R. VAN RENESSE. Distributed Operating Systems. *ACM Computing Surveys*, 17(4), December 1985.

[TvR88] A. S. TANENBAUM AND R. VAN RENESSE. A Critique of the Remote Procedure Call Paradigm. In R. SPETH, editor, *Research into Networks and Distributed Applications.* Elsevier Science Publisher B.V., 1988.

[TvRvS+90] A. S. TANENBAUM, R. VAN RENESSE, H. VAN STAVEREN, G. J. SHARP, S. J. MULLENDER, J. JANSEN, AND G. VAN ROSSUM. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12), December 1990.

[vECGS92] T. VON EICKEN, D. E. CULLER, S. C. GOLDSTEIN, AND K. E. SCHAUSER. Active Messages: a Mechanism for Integrated Communication and Computation. In *19th International Symposium on Computer Architecture.* ACM Press, May 1992.

[Vin93] S. VINOSKI. Distributed Object Computing with CORBA. *C++ Report*, 5(6), August 1993.

[VR92] P. VERÍSSIMO AND L. RODRIGUES. Group Orientation: A Paradigm for Distributed Systems of the Nineties. In *Proceedings of the Third Workshop on Future Trends of Distributed Computing Systems.* IEEE Computer Society, April 1992.

[vR93a] R. VAN RENESSE. A MUTS Tutorial. MUTS documentation, Cornell University, 1993.

[vR93b] R. VAN RENESSE. Causal Controversy at Le Mont St.-Michel. *ACM SIGOPS Operating Systems Review*, 27(2), April 1993.

[vR94] R. VAN RENESSE. Why Bother With CATOCS? *ACM SIGOPS Operating Systems Review*, 28(1), January 1994.

[vRB94] R. VAN RENESSE AND K. P. BIRMAN. Fault-Tolerant Programming using Process Groups. In F. BRAZIER AND D. JOHANSEN, editors, *Distributed Open Systems*. IEEE Computer Society Press, 1994.

[vRBC+92] R. VAN RENESSE, K. P. BIRMAN, R. COOPER, B. GLADE, AND P. STEPHENSON. Reliable Multicast between Microkernels. In *Proceedings of the USENIX Workshop of Micro-Kernels and Other Kernel Architectures*, Seattle, Washington, April 1992.

[vRHB94] R. VAN RENESSE, T. M. HICKEY, AND K. P. BIRMAN. Design and Performance of Horus: A Lightweight Group Communication System. Dept. of Computer Science, Cornell University, July 1994.

[Weg93] P. WEGNER. Towards Component-Based Software Technology. Technical Report CS-93-11, Brown University, Dept. of Computer Science, July 1993.

[Wei93] W. E. WEIHL. Specifications of Concurrent and Distributed Systems. In S. MULLENDER, editor, *Distributed Systems*, chapter 3. Addison Wesley, second edition, 1993.

[WGM88] A. WEINAND, E. GAMMA, AND R. MARTY. ET++ — An Object Oriented Application Framework in C++. In *OOPSLA'88 Conference Proceedings*, September 1988.

[Win89] J. M. WING. Verifying Atomic Data Types. *International Journal of Parallel Programming*, 18(5), October 1989.

[WWC92] G. WIEDERHOLD, P. WEGNER, AND S. CERI. Toward Megaprogramming. *Communications of the ACM*, 35(11), November 1992.

[Zuc94] L. ZUCCONI. Issues Concerning Re-Engineering of Legacy Software in the Federal R&D Environment. *ACM Software Engineering Notes*, 19(3), July 1994.

Index

Curriculum Vitae

name:	Silvano Maffeis
date of birth:	12/16/1965
place of birth:	Männedorf, Switzerland
nationality:	Switzerland
e-mail:	maffeis@acm.org

Education

4/1973	- 3/1979	Primarschule, Männedorf, Switzerland
4/1979	- 3/1981	Sekundarschule, Männedorf, Switzerland
4/1981	- 9/1985	Mathematisch-Naturwissenschaftliches Gymnasium (MNG), Zurich
2/1986	- 5/1986	Military Service
10/1986	- 6/1991	University of Zurich. Studies in Computer Science and Economics (Wirtschaftsinformatik)
7/1991		*lic. oec. publ.* degree in Computer Science and Economics, with *summa cum laude*
10/1991	- 2/1995	Department of Computer Science, University of Zurich

Work Experience

1/1983	- 2/1986	Assembling and testing computer terminals at Ingenieurbüro Pomey,

		Uetikon am See, Switzerland
5/1986	- 10/1991	Part-time employments as programmer, instructor, or consultant in several companies in Switzerland
8/1989	- 10/1989	Student project at UNISYS, Zurich
2/1990	- 5/1990	Student project at ETH, Zurich
1/1992	- 12/1993	Project with Siemens-Nixdorf AG. Topics: Replication in distributed systems; the Vanilla file system; the Electra toolkit. Teaching assistant at University of Zurich
1/1994	- 2/1995	Project with the Union Bank of Switzerland and with Siemens-Nixdorf AG. Topics: Beyond–Sniff — A distributed software engineering environment; the Generic Multicast Transport Service; the Electra toolkit

Publications and Technical Reports

1. Maffeis, Silvano. "Normen und Standards gewinnen an Bedeutung." *Up to data* Januar 1992, Zürich: RUF AG.

2. Maffeis, Silvano, and Clemens Cap. *Replication Heuristics and Polling Algorithms for Object Replication and a Replicating File Transfer Protocol.* University of Zurich Technical Report IFI TR 92.06. Zurich: 1992.

3. Maffeis, Silvano. "File Access Patterns in Public FTP Archives and an Index for Locality of Reference." *ACM SIGMETRICS Performance Evaluation Review* 20(3) (March 1993).

4. Maffeis, Silvano. "Remote Method Calling and Object Group Communication." ECOOP Workshop on Object-Based Distributed Programming. Kaiserslautern, Germany: July 1993.

5. Maffeis, Silvano. "Electra — Making Distributed Programs Object-Oriented." Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems IV. San Diego: The USENIX Association, September 1993.

6. Maffeis, Silvano. *Technologische Grundlagen des Computer Supported Cooperative Work.* Bericht des Institutes für Informatik der Univer-

sität Zürich 93.29. Zürich: 1993.

7. Cap, Clemens, Silvano Maffeis, Lutz Richter, und Volker Strumpen. *Ein Glossar wichtiger Begriffe zu verteilten und parallelen Systemen.* Bericht des Institutes für Informatik der Universität Zürich 93.44. Zürich: 1993.

8. Gatti, Antonio, und Silvano Maffeis. "Effiziente Risiko- und Ertrags-Optimierung." *Schweizer Bank* November 1993, SHZ Fachverlag.

9. Maffeis, Silvano. "Cache Management Algorithms for Flexible Filesystems." *ACM SIGMETRICS Performance Evaluation Review* 21 (2) (December 1993).

10. Gatti, Antonio, und Silvano Maffeis. "Erhöhte Flexiblität im Global Risk Management." *Schweizer Bank* Februar 1994, SHZ Fachverlag.

11. Gatti, Antonio, und Silvano Maffeis. "Nichts geht ohne Standards und neue Technologien." *Schweizer Bank* März 1994, SHZ Fachverlag.

12. Maffeis, Silvano. "A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming." Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming. Springer-Verlag, LNCS 791, April 1994.

13. Maffeis, Silvano. "System Support for Distributed Computing." Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe 1994. Springer-Verlag, LNCS 797, April 1994.

14. Maffeis, Silvano. "Design and Implementation of a Configurable Mixed-Media Filesystem." *ACM Operating Systems Review* 28(4) (October 1994).

15. Maffeis, Silvano. "Electra: An Object Request Broker for Robust Distributed Systems." Proceedings of the '94 SIPAR-Workshop on Parallel and Distributed Computing. University of Fribourg Report 94-19, 1994.

16. Maffeis, Silvano, Walter Bischofberger, and Kai-Uwe Mätzel. "A Generic Multicast Transport Service to Support Disconnected Operation". Proceedings of the 2nd USENIX Symposium on Mobile and

Location-Independent Computing. Ann Arbor, MI: The USENIX Association, April 1995.

Presentations and Invited Talks

various dates	Siemens AG, ZFE, Munich
various dates	Doktoranden Seminar, University of Zurich
various dates	Union Bank of Switzerland, UBILAB, Zurich
7/23/1993	ECOOP Workshop on Object-Based Distributed Programming, Kaiserslautern, Germany
9/3/1993	OBDP Workshop, CHOOSE, ETH Zurich
9/27/1993	USENIX Symposium on Experiences with Distributed and Multiprocessor Systems IV, San Diego
10/19/1993	Fortbildungsseminar Verteilte Systeme, University of Zurich
11/19/1993	Vrije Universiteit, Centrum fur Wiskunde en Informatica, Amsterdam, The Netherlands
1/6/1994	Unité Mixte Bull/IMAG, Grenoble, France
1/26/1994	Neu-Technikum Buchs, Buchs (SG), Switzerland
4/18/1994	International Conference and Exhibition on High Performance Computing and Networking (HPCN) 94, Munich
6/28/1994	EPFL Lausanne, Switzerland
9/29/1994	Object Technology at Work Conference, DBTA & CHOOSE, University of Zurich
10/11/1994	Fortbildungsseminar Verteilte Systeme, University of Zurich
10/14/1994	SIPAR Workshop on Distributed and Parallel Computing, University of Fribourg
12/6/1994	IBM Zurich Research Laboratory