The ROMANCE approach to Replicated Object Management[1] *(version 2.1)*

Luís Rodrigues, Paulo Veríssimo
Technical University of Lisboa - INESC[2]
e-mail:...ler@inesc.pt,paulov@inesc.pt

January, 1994

[1]A shorter version of this report was published in the Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems, Lisboa, Portugal, September, 1993, © 1993 IEEE

[2]Instituto de Engenharia de Sistemas e Computadores. This work was partially supported by the CEC, through Esprit BR 6360 (Broadcast).

Abstract

The ROMANCE project aims at building a Replicated Object MANagement Configurable Environment based on Group Technology. Ultimately, we aim to develop a programming environment to support sharing passive replicated data objects regardless of data persistence or durability requirements. ROMANCE objects are distributed and can be accessed from any node in the system (in a location transparent way). Object access can be implemented either by issuing remote invocations or by maintaining a local copy of the object. Thus, shared objects may be replicated, both in volatile storage, in the context of user processes, and/or in stable memory. Additionally, ROMANCE intends to provide support for fault-tolerance and controlled inconsistency of object replicas in the presence of network partitions.

This paper details the goals of the ROMANCE project, presents its software architecture and current status. A fundamental goal of the project is to experiment with Group Technology. In ROMANCE, distribution and replication are implemented using underlying group communication and membership services. The paper also describes how these tools are used to achieve ROMANCE goals.

Chapter 1

Introduction

The Navigators team at INESC has been working for some years now in Group Technology [5, 33]. As a result, a Group Communications infrastructure optimized for local area networks has been developed [28]. We are now extending this work in two directions: developing a group infrastructure for large-scale networks; and developing higher level tools to help the programmer to incorporate the Group Technology in his/her applications.

In this framework, we have just started the ROMANCE project, aiming at building a Replicated Object MANagement Configurable Environment based on Group Technology. A fundamental goal of the project is to experiment with Group Technology: we intend to make extensive use of group communication and group membership services in all levels of the architecture. Replicated object management will be a critical issue in most future distributed systems and it is an area that is not yet satisfactorily covered by existing systems. We hope that this experiment will help us to clarify the inherent advantages and disadvantages of the use of groups to develop distributed applications, since it is felt that group orientation will be a prevailing paradigm in the future distributed systems [5].

Replication of objects in a distributed system has been used to improve performance and also to provide fault-tolerance in different application areas such as: persistent data management [3], management of replicated computations [27], and distributed shared memory [25]. These three application areas can be considered complementary and none of them fully covers the

spectrum of applications that can use replicated data management. For instance, while persistent data management systems often preclude the efficient sharing of volatile objects, distributed shared-memory systems usually do not offer fault-tolerant features. In ROMANCE, we will search for models that conciliate these different approaches. Ultimately, we aim at developing a programming environment that provides support for: (i) sharing passive replicated objects, independently of their persistence or durability requirements; (ii) objects living in the context of user processes or residing in storage servers; (iii) fault-tolerance; (iv) controlled inconsistency in the presence of network partitions.

This paper presents the ROMANCE system architecture and describes the current prototype implementation. The paper is organized as follows. Section 2 describes the basic ROMANCE model and details how distributed replicated objects can be accessed and how they can be made persistent and/or fault-tolerant. The programmers view of ROMANCE is described in section 3, where the relation between classes, inheritance and distribution is also discussed. A practical example is given in section 4. Current status is reported in section 5, comparison with related work on section 6, and final concluding remarks appear in section 7.

Chapter 2

Basic ROMANCE model

In ROMANCE we consider that the system is composed of a set of distributed processors or nodes, that do not physically share memory and communicate exclusively through message passing (see figure 2.1). The system is populated by ROMANCE objects. ROMANCE objects consist of passive data encapsulated by a set of operations or methods, dubbed the object interface. Our model closely follows recent work on distributed object oriented systems as, among others, Emerald [6], Argus [21], Clouds [12], Comandos [32], SOS [30], and Arjuna [31]. All ROMANCE objects are characterized by a set of attributes:

- All ROMANCE objects are *distributed*. They can be accessed from any node of the system, with location transparency. All ROMANCE objects are accessed as if in the context of its client through the use of an appropriate proxy [24].
- All ROMANCE objects are potentially *replicated*. Several copies, or *images* of each object may exist in the distributed system, either for fault-tolerance or for increased performance.
- All ROMANCE objects are potentially *persistent*. An object may have images exclusively on volatile memory, in this case it is a pure volatile object. However, the object may also maintain one or more images in stable memory, assuring that the object state survives beyond the life of the program that created it.

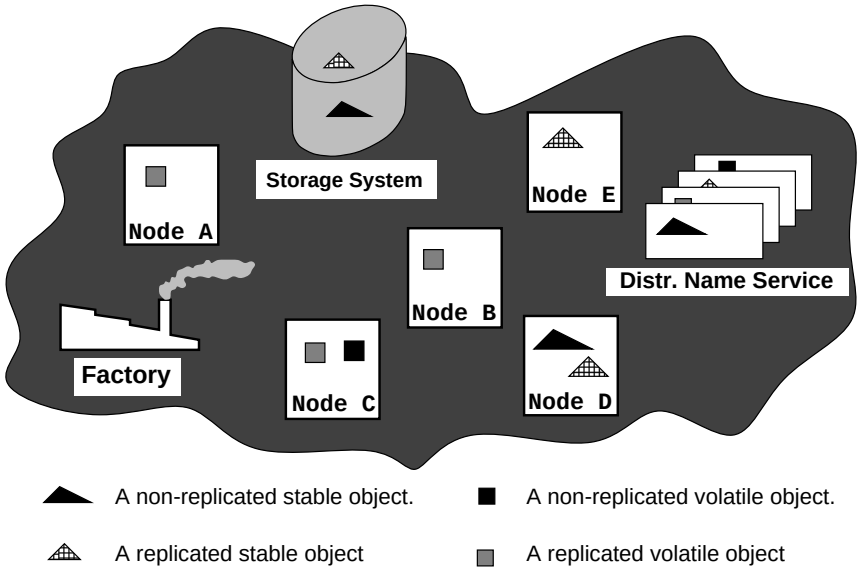

Figure 2.1: System view.

- All ROMANCE objects are potentially *concurrent*. The ROMANCE system does not enforce any concurrency control policy. The object itself must implement its own internal concurrency control policy, using the appropriate mechanisms. Global concurrency control policies, such as *serializability*, for synchronizing activities that access multiple objects, may be enforced but are not considered in the current system stage.

In order to support the access to distributed replicated objects, the ROMANCE system relies on the existence of some pre-defined objects, namely (figure 2.1):

- The ROMANCE Distributed Name Service (RDNS) used to locate ROMANCE objects in the system.
- The ROMANCE Factory, in charge of activating, when needed, a *server* for an object. A server contains a local image of the object and is able to process remote invocations on behalf of remote clients.

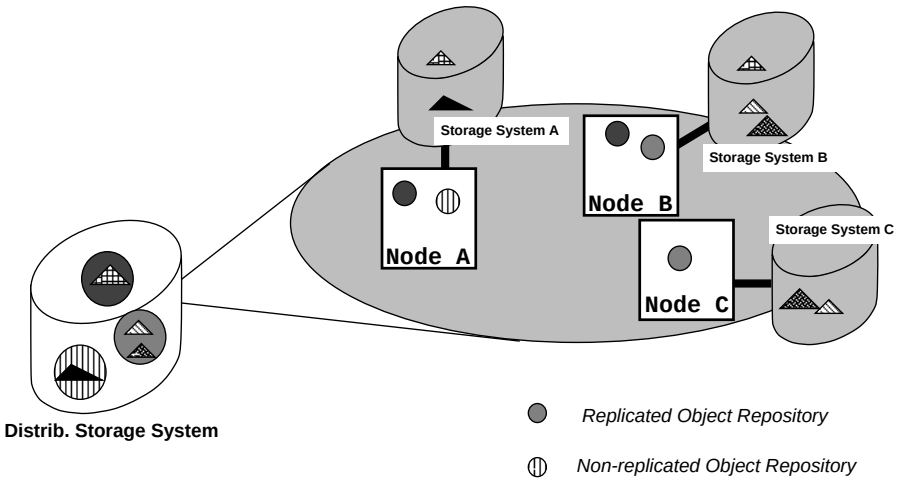

Figure 2.2: Storage system.

- The ROMANCE Storage System, responsible for storing stable images of persistent objects. The Storage System can be decomposed in several *Object Repositories*. Each Object Repository is itself a ROMANCE object, built upon one or more volatile and stable images (figure 2.2). Thus, the storage system may be easily enriched with new Object Repositories, programmed using the core ROMANCE components.

These components cooperate to provide distributed access to ROMANCE objects. In our system, shared data can only be accessed through a set of operations, or *interface*. The interface implementation can support the object access using at least two alternatives: (i) access a remote image through remote invocations; (ii) to create a local image of the object; this image is automatically kept consistent with other images through some selected replicated data management protocol. Both types of access may be available simultaneously for each object and several interface implementations may be linked with any object client since ROMANCE will support run-time selection of the most appropriate implementation. We will first analyze these methods separately and, afterwards, we will describe how they coexist in the ROMANCE environment.

2.1 Remote access

Access to remote objects is performed through remote invocations. This method is supported by two auxiliary blocks: a *Group Remote Invocation Proxy* (GRIP) and a matching *Ambassador* [10]. The GRIP marshals the local client invocation in a message multicasted to the remote ambassador(s), and waits for results, if any (GRIP functionality is similar to the client stub of point-to-point RPC). The ambassador is the GRIP counterpart on the server side, and is responsible for transforming messages in the appropriate method invocation to the local image and to send the results, if any, back to the calling GRIP. An ambassador is usually associated with a volatile object image, although it might be configured to forward the invocations to another GRIP, acting as a protocol gateway (an approach similar to the use of SSP chains [29]). In ROMANCE, a pair consisting of one ambassador with a local object image is called a *server*.

The pair GRIP/Ambassador defines a protocol to access the remote image. This protocol may vary with the object class, and it is possible to support more than one access method for the same object. The ROMANCE library will contain a set of off-the-shelf pairs available for the designer of new distributed replicated objects.

For fault-tolerance, the ambassador may be replicated. In this case, two or more ambassadors will serve remote invocations, and each ambassador will keep a local image of the object. Clients will interact with the group of ambassadors, through replicated remote invocations [9, 20] (see figure 2.3) [1].

2.2 Maintain a local image

Another method to access ROMANCE objects is to maintain a local image of the object. This may improve performance if locality of data can be exploited. However, to use this approach, one has to solve two problems: (i) when the new image is created, its state must be fetched from some valid copy in the system; (ii) the new copy must be kept consistent with the other

[1] When objects are accessed exclusively through remote invocations, the protocol between the GRIPs and the Ambassadors can also be used to maintain replica consistency. However, this is not often the case in ROMANCE.

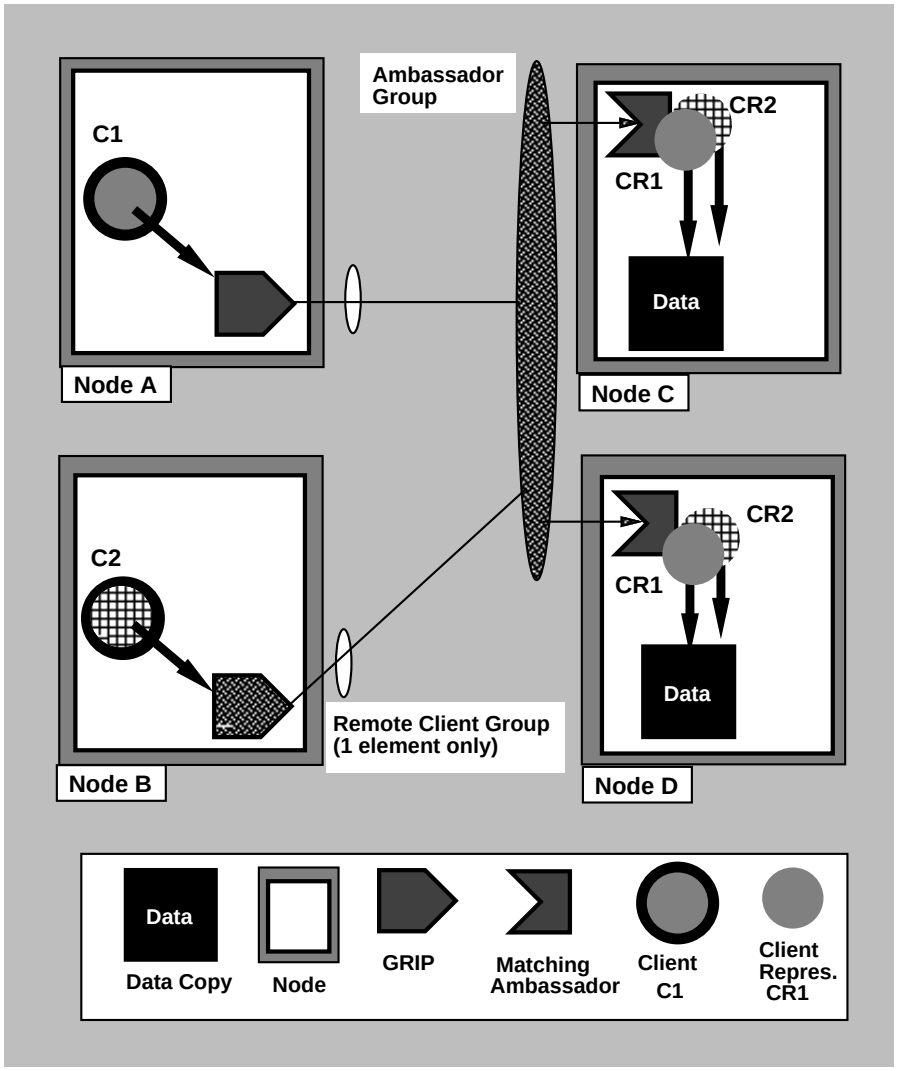

Figure 2.3: Remote copy approach.

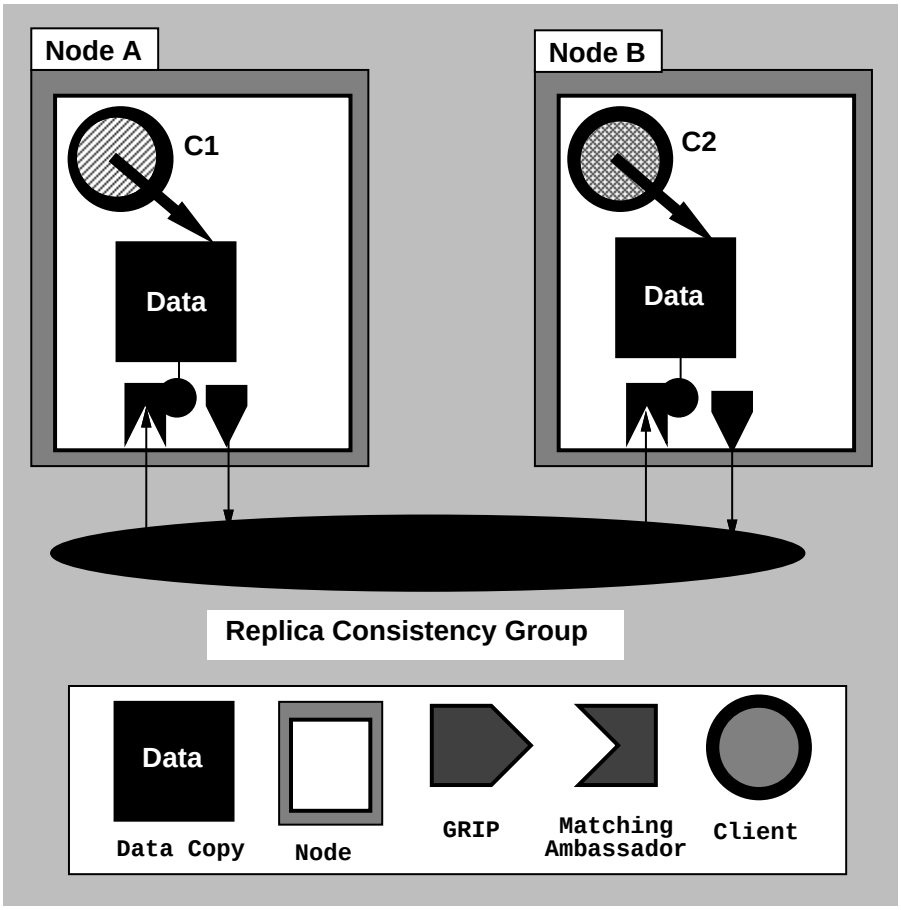

Figure 2.4: Local copy approach.

copies in the system accordingly to some predefined criteria. In ROMANCE we want to give the designer the freedom to select, or implement, the most appropriate consistency criteria for its classes, on a class by class basis. Thus, the same solution cannot be applied to all objects: each class of objects will have its consistency criteria and, consequently, will use different mechanisms to solve the two problems above.

Since active images of the same object need to cooperate to be consistent they need to exchange messages. This communication can also be modeled by replicated remote invocations. Thus, a replicated object, in addition to its public (service) interface, must also define a private interface, with the methods used for inter-replica coordination. As any other interface for a ROMANCE object, this private interface is accessed through GRIP/Ambassador pairs. Since all images can potentially invoke and receive invocations from other images, they will be necessarily associated with an ambassador and a GRIP for that interface (see figure 2.4).

In this model, the object client invokes directly an object's image. The image itself, accordingly with the class consistency criteria and protocol used for replicated data management, decides when it must invoke the private interface of other replicas. Thus, the details of inter-replica coordination are hidden from the object client. It should be emphasized that not all accesses to an image will require interaction between the replicas. Typically, some operations will be purely local (usually, read operations).

2.3 Mixed access

The two types of access previously mentioned are harmonized in ROMANCE in a single model, such that both types of access may coexist simultaneously, as illustrated in figure 2.5. In this approach, several object images may be active in the system (in the figure, on nodes D, E, F, and G). These active images are kept consistent using some replicated data management protocol. To implement this protocol, all images export a private interface, that can only be invoked by other object images. These interactions are performed by GRIPs and Ambassadors supported at run-time by a multicast group that links all replicas (Replica Consistency Group). The purpose of this multicast group is twofold: it maintains membership information about active replicas

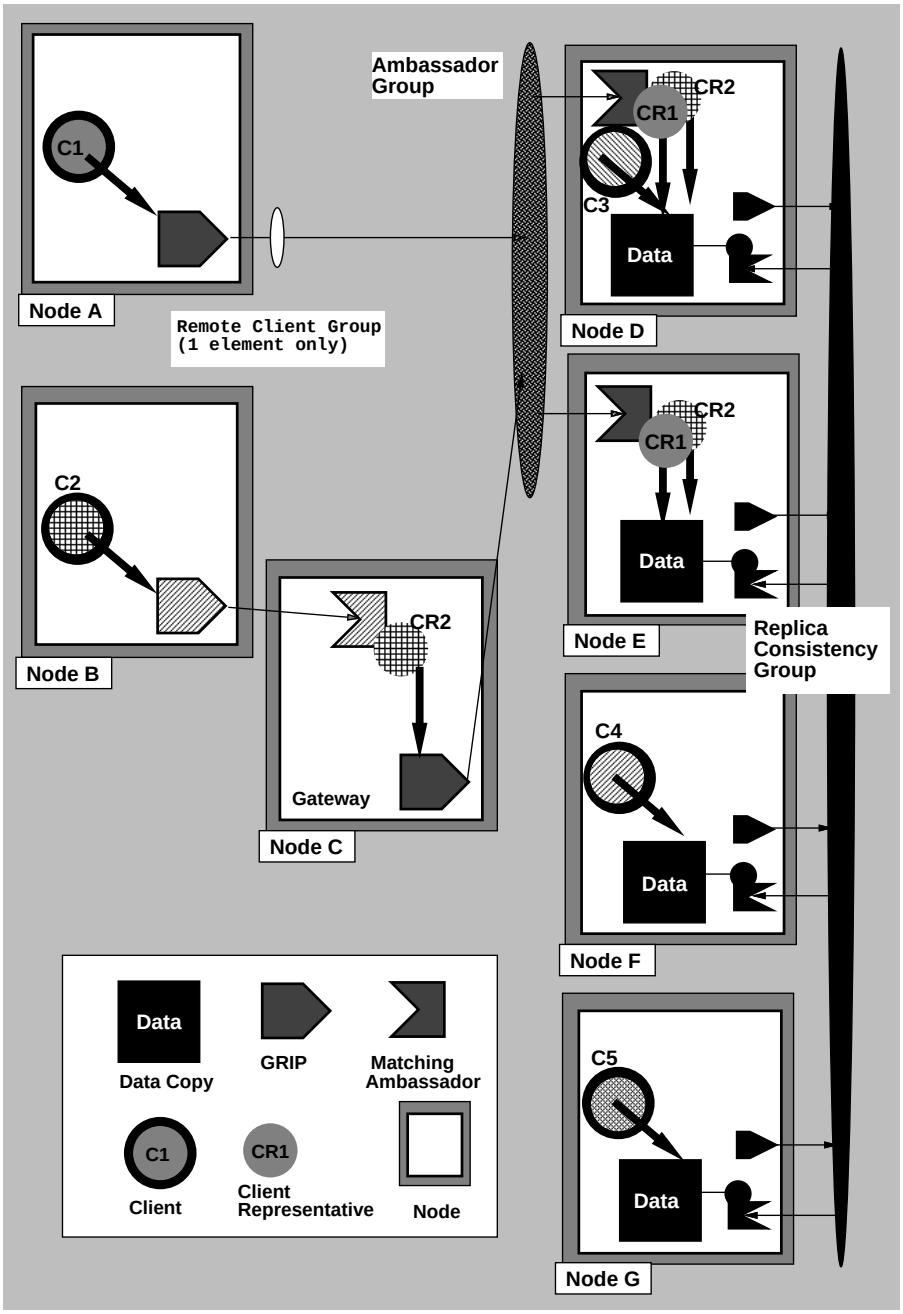

Figure 2.5: Mixed model (local copy and remote access).

and is a private channel for inter-replica coordination.

Some object images are accessed directly from local clients, as illustrated by clients C3, C4, and C5 in figure 5. However, remote clients are also supported, since the object may export its public interface through one or more GRIP/Ambassador pairs. For fault-tolerance, object ambassadors can be replicated, although the number of ambassador replicas do not need necessarily to match the total number of object images in the system (in the example, ambassadors are instantiated on nodes D and E). The figure also illustrates how Ambassadors and GRIPs can be composed to provide a protocol gateway (node C).

In the mixed model, replica consistency must be encapsulated within the image public interface, since local clients access this interface directly. Thus, in principle, there is no need to support consistency at the level of the protocol between the GRIPs and the Ambassadors supporting the public interface. In practice, the Ambassador acts as a local representative of the remote client. However, since the Ambassador is itself replicated, the protocol between Ambassador replicas and client GRIPs must ensure that remote invocations are performed exactly once in a unique object image (as the image is itself responsible for propagate invocation results to other images when needed).

2.4 Persistence

All ROMANCE objects are potentially persistent. In order to make one object persistent one has simply to assign an Object Repository to that object. The Object Repository will keep one or more object images in stable storage. Persistent objects are flushed to disk automatically when their last volatile image is de-activated. However, the object state may be saved in the repository at any other instant. Only one Object Repository may be associated with a given object at any moment. This is not a limitation as Object Repository can be aggregates formed from other Object Repositories.

ROMANCE objects can be replicated both in stable memory and in volatile memory. In the general case, when a new image is activated, its state is fetched from the state of another active image, if any. Otherwise, the state of the new volatile image is retrieved from the object's repository.

2.5 Fault-tolerance

In ROMANCE fault-tolerance is achieved through replication. At this stage, only crash-failures [27] are considered. Objects can be replicated both in volatile memory and in stable memory. Replication of images in volatile memory provides continuity of service in case of processor crashes. Maintaining at least one image in stable store provides tolerance to total processor crashes (as generalized power-failures). Replication of images in stable storage provides tolerance to permanent storage failures. All communication failures, including lost, duplication, re-order, and corruption of messages are handled by the underlying group-communication primitives. Network partitions are the exception to this rule. These are dealt with the mechanisms described in the following section.

2.6 Object Granularity

In ROMANCE we do not enforce any particular granularity for replicated distributed objects. Object size may differ and it is up to the object designer to choose the appropriate granularity of replication, accordingly to the pattern of object accesses. In principle, an object may be as small as a single octet, but such a granularity might prove to be inefficient, as the overhead to maintain replication and distributed access would be too large when compared with object size. Thus, ROMANCE is mainly targeted for coarse grain objects. Objects with very large sizes can also be supported but may require dedicated protocols for state transfer.

2.7 Network partitions

A partition of the distributed system occurs when the system is split into groups of isolated nodes. The nodes in each partition can communicate with each other, but no node in one partition can communicate with nodes in other partitions. If replica divergence must be avoided, at most one partition may be allowed to continue processing invocations. This partition, if existent, is called the *distinguished group*. Naturally, the distinguished group may

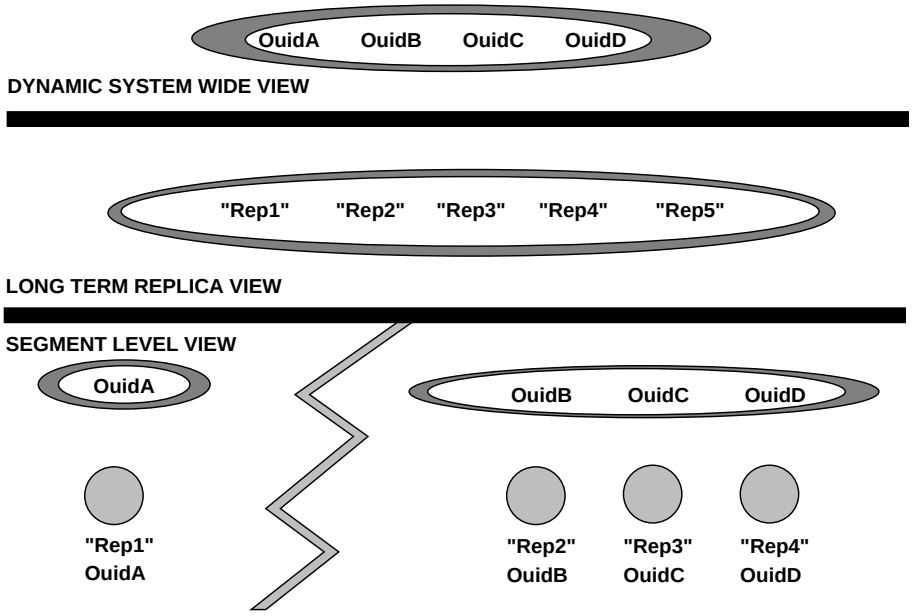

Figure 2.6: Network partitions in ROMANCE.

vary from object to object, depending on the number of images and on their location.

Note that it is impossible for members of one partition to obtain information about other partitions. Thus, the selection of the distinguished group must be made exclusively by the members of the group . Additionally, a group can only be allowed to proceed if it is sure that no other group will be selected. One possible criteria is to allow the work to continue only in a partition with the majority of nodes. Other possible variant, is to assign votes to each node and select the partition with the majority of votes [16]. In any case, to verify it belongs to the distinguished group, an image must have a view of the replicas that it can access, and a view of the replicas it should be able to access if the network were fully connected. In ROMANCE these two views are dubbed respectively, *segment level view* and *long term replica view* (see figure 2.6).

The segment level view keeps the replica membership in each network partition. In absence of partitions, the segment view maintains the list of all object images in the system. When a partition occurs, each image is provided

with a new segment view, containing all the images in its partition. The task of maintaining this view is performed directly by the membership functions of the underlying group communication infrastructure [28].

A long term replica view is associated with every ROMANCE object. It is usually pre-defined or defined by system management components. It contains the list of replicas that are expected to be accessible in absence of network partitions. The availability of an object can then be defined as a function of the long term replica view and the segment view. This can be used to ensure that only one partition is selected as the distinguished group, or to ensure that an object is inaccessible until a minimum level of fault-tolerance is provided.

Additionally, ROMANCE uses a third level of membership dubbed the *dynamic system-wide view*. This view is itself a ROMANCE object, on its own, and may be implemented using the previous views, if replicated. It provides a partition-tolerant system wide registration service, where all images of an object may register. It is intended for objects whose number of images cannot be known "a priori" or may change at run time (for instance a shared editor). The system-wide view provides means for decision making when choosing the distinguished group for such objects, through the comparison of the segment level view with the total number of images in the system, at the moment the partition occurred.

2.8 ROMANCE and Groups

A fundamental goal of the project is to experiment with Group Technology: we intend to make massive use of group communication and group membership services. In particular, every time a component is replicated, a group is created and used to maintain replica location and membership, to exchange information between replicas, and as a method to offer access to a public object interface [33]. We intend to build a set of replica management protocols using groups and compare our results of other algorithms or implementations using different approaches [22, 18, 23]. By using groups in all aspects of the architecture we expect to evaluate the inherent merits and disadvantages of our current Group Technology.

It should be noted that in ROMANCE we look to groups as an archi-

tectural solution to solve distribution and replication problems. Thus, from the point of view of the client of ROMANCE objects, the use of groups is transparent. Group Technology can also be used to support applications where the notion of groups appears explicitly in the problem specification. This kind of support, is provided at a higher layer and is currently being studied in our research group in the framework of defining and implementing a generic groupware platform [11].

Chapter 3

Programming in ROMANCE

The ROMANCE project intends to create an environment that encourages the programmer to *use* and to *develop* distributed replicated objects. These two aspects are supported through different mechanisms. Use of ROMANCE objects is encouraged by making distribution and replication transparent. ROMANCE objects can be created and accessed in the same way as other objects of the designer programming language. From the point of view of object implementor, replication is not transparent. Nevertheless, ROMANCE provides a set of models and tools that ease the task of programming new distributed replicated objects.

Ideally, object interfaces should be specified using the language used to program the application. This is the natural approach when the language is specially designed to support distribution. Unfortunately, this is not a feature of many popular programming languages, like *C* [19] or *C++* [13]. Thus, these languages have to be used with restrictions, for instance, on parameter passing to remote invocations. An alternative approach, is to use a dedicated language to specify the object interface, dubbed an *Interface Description Language* (IDL), and then use a compiler to translate this specification into one or more target programming languages. The later approach is currently being used in many distributed systems and platforms, such as ANSA [1] or CORBA [26]. This approach is also followed in ROMANCE. Currently, we support CORBA as the Interface Description Language and *C++* as the target programming language.

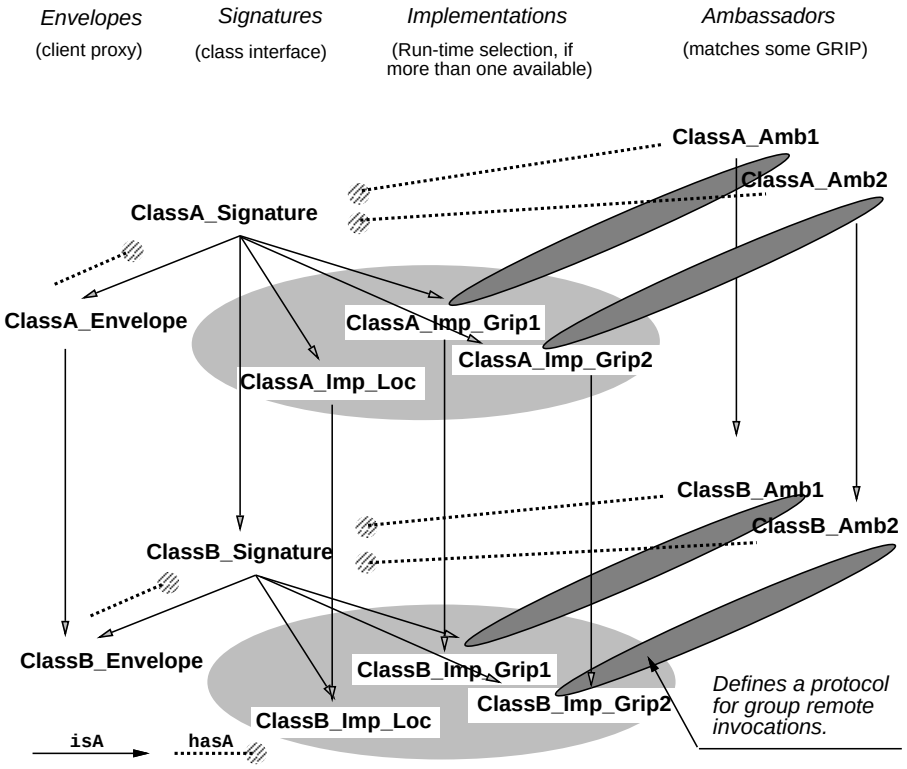

Figure 3.1: Class Hierarchy.

Translation from CORBA IDL to *C++* is provided by a specialized ROMANCE compiler. This compiler was developed using "The IDL Compiler Front End" provided by Sun [1] for the OMG group, from which only the back-end needed to be re-programmed (in fact, our compiler was based on already modified back-end, developed at INESC). We have chosen this compiler since it reduced substantially the programming effort, and by using a standard, OMG provided front-end, we are assured that the ROMANCE specifications conform to CORBA syntax.

From a CORBA IDL specification, the ROMANCE compiler produces a set of *C++* classes, that support the use and programming of distributed replicated objects. These classes are (see also figure 3.1):

[1] Sun Microsystems, Inc.

- A *Signature* class whose public interface, in $C++$, matches the CORBA IDL interface description. From the point of view of the object interface, this class is as a virtual base class for all object implementations. Additionally, this class registers all available implementations for that class, and selects an appropriate implementation when one object of the class needs to be accessed. It keeps also a list of all available subtypes, that is of all Signature classes derived from it. Thus, from some Signature class it is possible to create an object of any specialization of the desired signature. All interfaces are identified, and can be accessed in ROMANCE, by the associated interface name, as defined in its IDL description.

- An *Envelope* class hides from the object client the actual implementation being used. The envelope [10] class is the visible $C++$ interface of the object and is in charge of forwarding the requests to the selected implementation. Its interface must match the object signature, thus it is derived from the Signature class. When an envelope is created, it requests to the interface class to build an appropriate implementation for the object. If desired, it is possible to specify the sub-type of the object being created (by default, an object of the type defined by the IDL source is created).

- A standard *Group Remote Invocation Proxy* (GRIP). This GRIP is automatically generated from the IDL description. The GRIP is responsible for marshaling the invocation input parameters, invoke the remote ambassador(s), collect and un-marshal the results if any. The standard GRIP communicates with the name service in order to obtain the ambassador-group for the target object and establishes a connection with such group. If no image of the object is active, the GRIP can request the remote activation of object servers (which may be replicated), through the services of the ROMANCE factory.

- A *Local* image template, for the replicated image implementation. The template must be filled by the programmer with the code that actually implements the interface functionality. If the object is to be replicated, the programmer has to invoke the object private methods that support replicated data management. Usually, the replication algorithm is encapsulated in one of the object's base class. If the mechanism

to maintain replica consistency requires complex initialization, like instantiation of GRIPs and Ambassadors for inter-replica coordination, this will also be encapsulated in the object's base class initializers.

- A standard *Ambassador* to collect and process remote invocations. Its protocols match those of the standard GRIP. When created, an ambassador can be supplied with any Interface implementation. In the most common case, the ambassador accesses an implementation that contains a local copy of the data. Optionally, it can also access a GRIP of different type, assuming the role of gateway, between two different access methods.

3.1 Inheritance

All ROMANCE objects share some common functionality. For instance all romance objects are potentially persistent, may be linked to an object repository and saved in and retrieved from stable storage. Naturally, this basic functionality can be accessed remotely or locally. It is thus described in a ROMANCE basic interface, from which all ROMANCE interfaces must be derived. Interface inheritance is supported at the IDL level. All new ROMANCE interfaces must be derived directly from this basic interface or from one of the interfaces provided in the ROMANCE library. Although multiple inheritance is supported by CORBA IDL syntax, only simple interface inheritance is supported in ROMANCE. This may prove to be a limitation, and we are currently evaluating such an extension. The ROMANCE compiler generates $C++$ sources in such a way that the code from the base classes is reused in the implementations of the derived classes. Thus, each interface can be translated to $C++$ separately, and there is no need to re-write any code to implement the functionality derived from base classes. This rule applies for local image implementations as well as for class envelopes, GRIPs and Ambassadors.

Each interface defines a type in the system, and is assigned with an unique system-wide identifier that remains unchanged after its creation. All $C++$ methods generated from the interface signature are declared virtual such that derived implementations may specialize the base interface by redefining one or more methods. Additionally, it is possible from the $C++$

classes generated from a given interface, to access an object of any of the interface specializations. Thus, any interface acts as a generalization of all its derived interfaces.

3.2 Replication, inheritance and distribution

The use of inheritence to express properties such as consistency, atomicity, etc, is today commonly used in many projects [31, 14, 23]. In ROMANCE, each replica management policy is defined through a bi-directional interface with the managed data object. Replica consistency is maintained by activating this interface. For instance, a release consistency [15] policy may require the invocation of *acquire* and *release* primitives to bound object access, and needs to access the object state to obtain and update the replica state.

Since to be managed by a given policy, an object has to export some pre-defined operations, all objects managed by a given policy are derived from a virtual base class that defines the agreed interface. Basically, only the implementations that maintain a local copy need to be derived from such class. This has the advantage of making all the other class hierarchies (envelopes, GRIPS and ambassadors) independent of the replication method used. In ROMANCE, we achieve this goal by avoiding any reference to the consistency mechanism at the level of the interface specification. In this way, a type is not bound to any replica management policy at specification time. The compiler generates a template for the "local image" implementation that be customized by deriving it from the appropriate class. In principle, we should be able to support different replication strategies for the same type of objects. However, this raises a new problem: when accessing an object of a given type, it is necessary to know which strategy is being used for that particular object, in order to select the matching implementation. At this moment, we assume that only one replication strategy is associated with each type.

Chapter 4

An example

This section exemplifies how replicated objects are programmed and used in the ROMANCE system. The programmer wants to build a replicated volatile object that exports only two operations *readLong* and *writeLong*. This object should be remotely accessible, through function shipping, but also through a local copy which is kept consistent with all other local copies. The ROMANCE library provides a replica management algorithm to ensure strong replica consistency. This algorithm is available as a ROMANCE class, and described in the file ``Strong.h'' (see figure 4.1). The include file includes a description of the interface used for inter-replica coordination (which is uniquely manipulated by the base class) and a list of methods that can be used by the object implementor to keep the copies consistent.

The implementor may specify a new class, by deriving its interface from the basic Romance interface (``RBasic.hil'') as illustrated by the interface " Explo" on ``Explo.hidl''. He/she can also implement specializations of his/hers own interface by deriving from it new interfaces (``Specialization.hidl''). The ROMANCE compiler, will generate, the appropriate Signature, Envelope, GRIP, Ambassador and Local implementation template. To enforce strong consistency between the volatile copies of the object, the programmer has simply to derive the "local copy" implementation of its type (``ExploLoc_Strong.h'') from the class defined in the ROMANCE library. The figure also shows how the programmer should use the private replicated data management interface to fill the template form of the local implementation (``ExploLoc_Strong.C''). A sketch of how the client envelope looks is

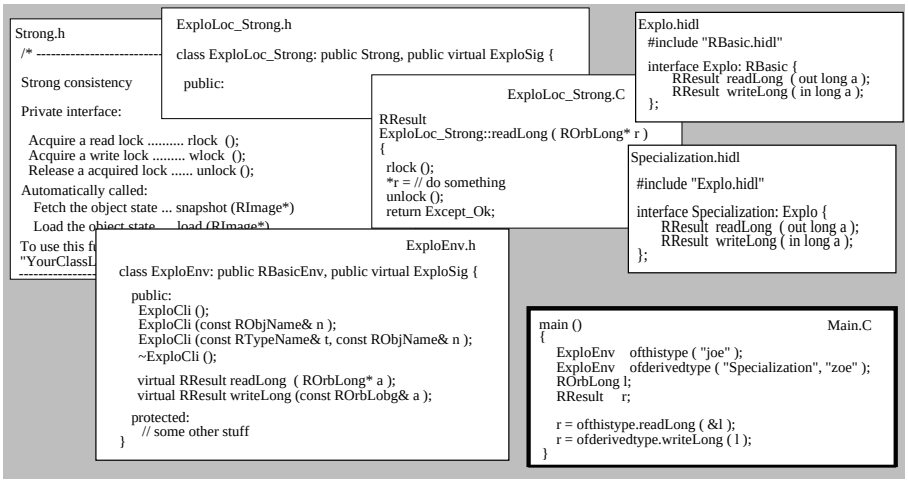

Figure 4.1: An example.

presented in the figure (‘‘`ExploEnv.h`’’). Finally, the figure also illustrates how a program using objects of new classes might be coded (‘‘`Main.C`’’).

Chapter 5

Current status and future work

The current prototype runs on a local-area network using the xAMp group communications service [28] as the underlying group communication infrastructure. The ROMANCE CORBA IDL to $C++$ compiler is implemented. Although it does not yet support the complete set of IDL constructs, it is usable and all the existing objects were built using this compiler. All the classes described in section 3 are generated by the compiler. All objects are programmed in $C++$, and object volatile images live in the address space of UNIX multi-threaded processes. Prototypes of the ROMANCE Name Service and of the ROMANCE Factory are implemented. Thus, ROMANCE objects can be activated and accessed, either by keeping a local image or through remote innovations. Persistence is currently provided through a prototype Object Repository based on the UNIX file system. The ROMANCE library contains only a very limited set of replica management protocols. No partition-tolerant object was implemented yet. At this stage, we are still trying to exercise our basic model, using "toy" implementations of all components to create some practical feedback to refine our model. In the near future, we intend to continue this exercise to test the concepts not experimented until now. In particular, we will implement a set of weak-consistent replication strategies and a set of partition-tolerant objects. Only after doing these experiments, and having stronger confidence in our design choices we will implement sophisticated versions of ROMANCE services. The port of ROMANCE to other group communication infrastructures, like for instance the ISIS system [4, 5], is also part of our long-term goal.

Chapter 6

Related work

We are searching for models that integrate two usually incompatible methods of access to replicated objects. Namely, ROMANCE supports simultaneously remote invocations and local copy access. To our knowledge, the work in this area is very limited. However, each of these methods have been intensively studied in separate. Remote invocations (and remote function calls) to replicated objects has been supported by several systems as *troupes* [9], ARGUS [20], DELTA-4 [27] or Arjuna [31]. More recent research projects include the ELECTRA [23] object oriented toolkit that also provides abstractions for remote method calling or the Object Groups [17] prototype that uses the ISIS system. However, most of these systems preclude local access to replicas and provide support for only a limited set of pre-defined replication strategies. In ROMANCE we follow the path of allowing different replication methods to be applied to different objects, using a problem-oriented shared memory approach [8]. A system that provides support for different shared-memory implementations is Munin [7], that uses *type-specific memory coherence* protocols tailored for different types of data. Additionally, our approach avoids the use of dedicated language support, and uniquely uses CORBA IDL to specify interfaces. Other approaches exist that define new languages. The ORCA programming language supports distributed replicated objects [2] but provides only a single model of consistency. A more versatile approach, that also resources to a specialized compiler, is given by the Fragmented Object model [24].

Chapter 7

Conclusions

This paper presents a Replicated Object MANagement Configurable Environment based on Group Technology. The environment, dubbed ROMANCE, provides support for using and programming distributed replicated objects. It achieves this goal through a set of tools, libraries and run-time support objects that were presented and discussed in this paper. A primary goal of the ROMANCE project is to experiment with the Group Technology. All membership and remote invocation services are based on fault-tolerant Group Technology. Currently, the xAMp protocol suite is being used, but ports to other group toolkits (for instance, ISIS) are foreseen. We hope that in the lifetime of the project, by building a wide set of replicated objects, we will be able experiment the use of groups in a large number of scenarios, and to assess the intrinsic merits and limitations of Group Technology.

Acknowledgments.

We wish to thank all colleagues that have discussed these ideas with us, namely to F.Cosquer, M.Pereira, M.Sequeira, and W.Wogels. The authors are grateful to Mark Little for commenting on earlier versions of this report. The help of P.Trancoso was invaluable in the task of modifying his CORBA IDL compiler.

Bibliography

[1] Architecture Projects Management, Ltd, Cambridge, UK. *The ANSA Reference Manual*, release 1.1 edition, July 1989.

[2] H. Bal, A. Tanenbaum, and M. Kaashoek. Orca: A Language for Distributed Programming. Technical Report Report IR-140, Dept. of Mathematics and Computer Science, Vrije Universiteit, December 1987.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[4] K. Birman, T. Joseph, and F. Schmuck. Isis - a distributed programming environment, user's guide and reference manual. Technical report, The ISIS Project, Dept. of Computer Science, Cornell University, Ithaca, March 1988.

[5] K. Birman, R. Cooper, and B. Gleeson. Programming with process groups: Group and multicast semantics. Technical Report TR-91-1185, Cornell University, Ithaca, USA, January 1991.

[6] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Eng., Special Issue on Distributed Computing*, December 1986.

[7] J. Carter, J. Bennettt, and W. Zwanepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 152–164, October 1991.

[8] D. Cheriton. Problem oriented shared memory revisited. In *Proceedings of the 5th ACM SIGOPS European workshop*, Mont Saint-Michel, France, September 1992. ACM.

[9] E. Cooper. Replicated procedure call. In *Proceedings of the 3rd ACM symposyum on Principles of Distributed Computing*, Berkeley, CA 94720, USA, August 1984. ACM.

[10] J. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley Publishing Co., 1992.

[11] F. Cosquer and P. Veríssimo. Groupware platform definition *(in preparation).* Technical Report RT-63/93, INESC, September 1993.

[12] P. Dasgupta, R. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc, W. Appelbe, J. Bernabeu-Auban, P. Hutto, M. Yousef A. Khalidi, and C.J. Wilkenloh. The design and implementing of the clouds distributed operating system. Technical report, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1990.

[13] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.

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

[15] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibsons, A. Gupta, and J. Henessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Int'l Symp. on Computer Architecture*, pages 15–26, Seattle, Washington, May 1990.

[16] D. Gifford. Weighted Voting For Replicated Data. In *Proceedings of the Seventh Symposium on Operating System Principles*, pages 150–162. ACM, December 1979.

[17] O. Hagsand, H. Herzog, K. Birman, and R. Cooper. Object-oriented reliable distributed programming. In *Proceedings of 2nd International Workshop on Object-Orientation in Operating Systems*, 1992.

[18] O. Hagsand, H. Herzog, K. Birman, and R. Copper. Object groups: An approach to reliable distributed programming. Technical report, Cornell University, October 1991.

[19] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall Software Series, second edition edition, 1988.

[20] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Lazy Replication: Exploiting the Semantics of Distributed Services. Technical Report MIT/LCS/TR-84, MIT Laboratory for Computer Science, 1990.

[21] B. Liskov. The ARGUS language and system. In *Distributed Systems, Methods and Tools for Specification*, volume 190 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[22] M. Little. *Object Replication in a Distributed System*. PhD thesis, Newcastle University Computer Science Department, September 1991.

[23] S. Maffeis. The electra approach to object oriented distributed programming: An overview. Technical Report IFI TR 92.93, Istitut fur Informatik de Universitat Zurich, November 1992.

[24] M. Makpangou, Y. Gourhant, J. Narzul, and M. Shapiro. Fragmented objects for distributed abstractions. In *Advances in Distributed Systems*. IEEE, 1992.

[25] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[26] Object Management Group: OMG. *The Common Object Request Broker Architecture and Specification*. OMG, August 1991.

[27] D. Powell, editor. *Delta-4 - A Generic Architecture for Dependable Distributed Computing*. ESPRIT Research Reports. Springer Verlag, November 1991.

[28] L. Rodrigues and P. Veríssimo. xAMp: a Multi-primitive Group Communications Service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, Houston, Texas, October 1992. IEEE.

[29] M. Shapiro, P. Dickman, and D. Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Technical Report 1799, INRIA, 1992.

[30] M. Shapiro et al. SOS: An object-oriented operating system - assessment and perspectives. *Computing Systems*, 2(4):287–338, 1989.

[31] S. Shrivastava, G. Dixon, and G. Parrington. An Overview of the Arjuna Distributed Programming System. *IEEE Software*, January 1991.

[32] P. Sousa, M. Sequeira, A. Zúquete, P. Ferreira, C. Lopes, P. Guedes, and J. Marques. The IK Distributed and Persistent Platform: Overview and Evaluation. *Usenix Computing Systems*, 6(4), Fall 1993.

[33] P. Veríssimo and L. Rodrigues. Group orientation: a paradigm for distributed systems of the nineties. In *Proceedings of the 3rd IEEE Workshop on Future Trends of Distributed Computing Systems*, Taipe, Taiwan, April 1992. also as INESC AR/67-92.