BUILDING RELIABLE INTEROPERABLE DISTRIBUTED OBJECTS WITH THE MAESTRO TOOLS

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Alexey Vaysburd
May 1998

© Alexey Vaysburd 1998
ALL RIGHTS RESERVED

BUILDING RELIABLE INTEROPERABLE DISTRIBUTED OBJECTS WITH THE MAESTRO TOOLS

Alexey Vaysburd, Ph.D.
Cornell University 1998

This work presents the Maestro Tools for development of reliable interoperable object-oriented distributed applications. We discuss the three fundamental parts of Maestro – the object group tools, the client/object interoperability tools, and the group protocols which implement state machine replication of distributed objects – with a special focus on practical usability and system integration issues.

Biographical Sketch

Alexey was born in the city of Odessa in Ukraine in 1970. He graduated from High School #100 in 1987 and went on to study at Odessa State University. In 1990 Alexey's family emigrated to the United States and he transferred to New York University, from which he graduated in May 1993 with a B.A. in Mathematics and Computer Science. In the fall of that year Alexey entered the PhD program in Computer Science at Cornell. He earned an M.S. in Computer Science in 1995 and graduated with a PhD in May 1998.

To my parents

Acknowledgements

The first thing I did on my first day at Cornell, even before finding out where my own room would be, was to look up the number of the office of Ken Birman, run there, and tell Ken right away that I wanted him to be my advisor. This started my exciting five years in Ken's Horus group. I am grateful to Ken for his guidance throughout those years, for his strategic insights, where sometimes a single phrase mentioned by him in passing would suddenly light up a whole new dimension of which I had not even been aware before, and keep me thinking for days and weeks – which would eventually result in my seeing both a research problem and an opportunity to solve it where I had previously seen neither. I am grateful to Ken for his encouragement and the freedom I've usually had to work on projects interesting to me, in the directions I wanted to pursue.

I knew of Robbert van Renesse even before I came to Cornell. As one of the creators of the Amoeba system he seemed almost a legendary person, and I remember how excited I was when Ken introduced me to Robbert on that same first day and told me I was going to work with him on the Horus project. Spending five years in a graduate school is not just about writing code and papers but, perhaps even more importantly, it is about interactions with people. I am happy that I have had the opportunity to work with Robbert and learn not only from his brilliant ideas and programming style, but also from his humanness and kindness.

I would like to thank Anil Nerode for being on my Special Committee.

I wrote my first paper with Roy Friedman, which has been a great experience for me. I am very grateful to Roy for the opportunity to work with him during those several years that he spent at Cornell.

I want to thank everybody in the Horus group for all the discussions and comments over the five years that have contributed to this work. Many thanks to Tim Clark for maintaining and supporting Maestro code within Ensemble releases. The classes with Sam Toueg and Joe Halpern that I took, and the many discussions we've had, have taught me a lot and shaped my perspective on the area of distributed systems. I am grateful to Silvano Maffeis for the pleasure it has always been to work with him. I am happy to have had a chance to work with Dave Bakken. I also would like to thank BBN for their financial support.

There is no doubt that all the people I have worked with have left their imprint on my way of thinking, the style and approach to research, and I am grateful to

all of them. Going back into the past, however, I must start with expressing my ever deep gratitude to Arkadiy Markovich Alt who, back in HS #100 in Odessa, was first to show me the light of intellectual freedom in the country still entangled in dark ages.

I would not have been able to survive the five years of graduate school without my great officemates and all my friends in Ithaca, in New York, and many other places. I want to thank all of them for being there.

Finally, and most importantly, I want to thank my family for their love and support, which in the end is what has made this dissertation possible.

Table of Contents

Biographical Sketch iii

Dedication iv

Acknowledgements v

List of Figures x

1 Introduction 1

2 Group Communication Tools 4
- 2.1 Maestro Tools and Prior Work . . . 4
- 2.2 Group Members . . . 7
  - 2.2.1 Initialization . . . 7
  - 2.2.2 Participating in the Membership Protocol . . . 7
  - 2.2.3 Sending and Receiving Messages . . . 9
  - 2.2.4 External Failure Detectors . . . 11
  - 2.2.5 Example: A Chat Program . . . 12
- 2.3 Group Clients and Servers . . . 14
  - 2.3.1 Initialization and State Transfer Options . . . 15
  - 2.3.2 Sending Messages . . . 16
  - 2.3.3 Membership Monitoring . . . 16
- 2.4 State Transfer . . . 16
  - 2.4.1 Protocol Framework . . . 17
  - 2.4.2 A Pull-Style Protocol . . . 23

3 Object Interoperability Tools 29
- 3.1 Introduction . . . 29
- 3.2 Related Work: Approaches and Tradeoffs . . . 32
- 3.3 Maestro IIOP Bridge and ORB Framework . . . 37
  - 3.3.1 IIOP Bridge: Server Side . . . 37
  - 3.3.2 IIOP Bridge: Client Side . . . 39
  - 3.3.3 Object Keys . . . 39

3.3.4 Building ORBs over the Maestro IIOP Bridge 40
3.4 Reliable Maestro ORB (Replicated Updates) 44
3.4.1 ORB Interfaces 45
3.4.2 A Look Inside 47
3.4.3 Client Perspective: Failover/Transparency Issues 49
3.5 Building Object Adaptors and Applications with Maestro 50
3.5.1 Implementing Objects/Object Adaptors 50
3.5.2 System Configuration/Initialization 53
3.5.3 Setting Up the Client Side 54
3.6 Performance 58
3.7 The Maestro Wizard 61

4 Protocol Support: Implementing State Machine Replication 66
4.1 Introduction 66
4.2 Background and Related Work 68
4.3 System Model and Protocol Support in Horus 70
4.3.1 Partitionable Group Membership Service 72
4.3.2 Atomic Message Delivery Within a View 75
4.3.3 State Machine Replication Within a View 76
4.4 Globally Consistent State Machine Replication 78
4.4.1 State Version Numbers 79
4.4.2 Installation of Primary Views 80
4.4.3 Globally Safe Delivery and State Transfer 82
4.4.4 Restarting Objects After Crashes 87
4.4.5 State Machine Replication Properties 90
4.5 Performance 91
4.6 Discussion 93

Bibliography 96

List of Figures

2.1 The Maestro Class Hierarchy: Core Classes and Application Examples 6
2.2 The object-adapter approach: Building applications by overloading the `Maestro_GroupMember` class 8
2.3 Maestro downcalls/callbacks and Membership Protocol: Joining the group 9
2.4 Maestro downcalls/callbacks and Membership Protocol: Leaving the group 10
2.5 Maestro downcalls/callbacks and Ensemble messages: Sending a multicast 11
2.6 State Transfer Protocol in Maestro 18
2.7 A State Merge protocol: When two group components merge together, they exchange their states using getState()/sendState() downcalls of Maestro 22
2.8 State transfer protocol: Under certain failure scenarios, state transfer may be canceled and restarted later 23
2.9 A Pull-Style implementation of State Transfer Protocol 24

3.1 CORBA ORB Architecture 30
3.2 Maestro Interoperability Architecture: Multiple Request Managers (ORBs) can be supported over a shared IIOP bridge 31
3.3 CORBA Reliability Solutions 33
3.4 Maestro IIOP Bridge: The Server-Side Interface 38
3.5 Maestro Replicated Updates ORB 48
3.6 Compound Interoperable Object References in Maestro: An IOR may contain multiple IIOP profiles pointing to different copies of the replicated object residing at different processes 49
3.7 Performance of the Maestro Replicated Updates ORB (over Horus) and Orbix 2.2 59
3.8 Performance of the Maestro Replicated Updates ORB (over Ensemble) and Orbix 2.2 60

4.1 The Two-Phase Commit Protocol blocks when the coordinator crashes 67
4.2 The Three-Phase Commit Protocol blocks when the network partitions 68

4.3 State Machine Replication: The group behaves as one reliable highly-available object . . . 69
4.4 Layered protocol architecture of Horus . . . 71
4.5 Partitionable group membership service in Horus: Multiple concurrent views can be installed simultaneously . . . 73
4.6 The View Change Protocol in Horus . . . 74
4.7 View Atomic Message Delivery: Group members included in two consecutive views deliver the same set of messages between the view changes . . . 75
4.8 Total ordering protocols available in Horus . . . 77
4.9 Execution of member objects may diverge when the view partitions into several components . . . 78
4.10 When views merge, the state of the more advanced view is transferred to members of the other view. When a primary view is installed, all included members are always in the same state . . . 79
4.11 When any two different primary views are proposed, they are always assigned different sequence numbers . . . 80
4.12 If objects deliver messages which are not safe (i.e. not acknowledged by a majority), their state may need to be corrected (or rolled back) later . . . 83
4.13 With the globally safe delivery layer, messages are only delivered when they become safe. If an unsafe message is rolled back, the application is notified with an `abortCast()` callback . . . 84
4.14 Integration of the primary views layer, globally safe delivery layer, and the state transfer protocol of Maestro . . . 85
4.15 Maestro-level state transfer and internal transfer of unsafe messages may proceed in opposite directions . . . 87
4.16 If objects restarted after crashes still count when computing a quorum, the state of the group may become inconsistent . . . 88
4.17 When an object is reincarnated after a crash, it remains a zombie (denoted by the dashed line) until it receives an up-to-date state and becomes a member of a primary view . . . 89
4.18 Average round latency for different group sizes . . . 91
4.19 Average throughput per group member for different group sizes . . . 92
4.20 Average throughput for the whole group for different group sizes . . . 93

Chapter 1

Introduction

The focus of this work is on integration aspects in development of reliable object-oriented distributed applications. It appears that complexity of building communication protocols and efficient architectures at different levels of a distributed system is overshadowed by the complexity of putting all individual pieces together and getting them to work coherently as a whole. There seems to be some kind of a "law of diminishing impact", where practical usability of a system component that implements a new technology is significantly decreased with each layer separating the new module from the final application level. This trend becomes especially obvious in systems with intrinsically complex interdependencies between individual layers and components, which in case of reliable distributed systems involve not only compatibility and adequacy of modules' interfaces, but, importantly, a number of subtle sophisticated logical properties of provided protocols and assumptions on the system model and the formal abstractified view of the application.

It is unrealistic to expect that multiple system layers and communication protocols will fit nicely together and provide the expected behavior to the application if they were not designed from the beginning with a "vertical" top-bottom view of the system, with an integration of all components in mind. The focus on individual tools, rather than on their integration within complete applications, may result in a kind of a "virtual existence" paradox, where a piece of software which implements some clearly useful technology and is even released to customers may still be practically unusable when it comes to actual applications.

One reason for the usability problem is specifics of experimental research, which is often more concerned with development of new technologies that could be used as "pieces of the puzzle", rather than with the whole puzzle itself. It would indeed be unnatural to expect that research tools were built to immediately fit actual real-world applications, in particular because design priorities, system requirements, and quality-of-code expectations are very different for a piece of research code and enterprise software, especially when high-reliability issues are involved. However, these differences sometimes result in an unfortunate situation when a new technol-

ogy badly needed by real systems is well understood at the research level and even implemented with a collection of tools, and is shown to work with simple example applications, but is practically inaccessible to enterprise developers, who are not familiar with and scared away by the complexity of the involved technology and the low level of provided interfaces and protocol properties.

The *wrong abstraction level* of distributed tools from perspective of an application developer is a major problem, as it usually means both *underspecification* of application-relevant properties and *overspecification* of low-level implementation details. In other words, the application developer often doesn't have adequate hooks for controlling the behavior of underlying system components and protocols and does not get a high-level view of their composite properties, yet at the same time is overwhelmed by the complex yet irrelevant *component/protocol-level* (but not *application-level*) specifications. The mismatch of abstraction levels unavoidably arises when the last step in a development of a collection of tools – the integration and smooth transfer of technology into applications – is missing.

It would be unrealistic indeed to expect that an application programmer concerned with system reliability should be able to understand the complex group communication protocols and their semantics and interdependencies in order to use them effectively when the high-level properties are not clearly specified. In practice, the users often hope that by merely plugging in the provided tools they will magically obtain high reliability and availability of their application, and become disappointed having realized this is not at all the case. Even if several simple tests seem to be running fine and, for example, a to-be-reliable application appears to survive a couple of manually-engineered process crashes, there is no reason to expect that the logical behavior of the system will be as expected in all cases and scenarios and that the incomprehensible (to the application developer) "black magic" of underlying group communication protocols will miraculously make a perfect fit with the system environment and the abstract model of the given application. This situation can be compared to trying to glue all holes in a sieve so as to close all leaks in it. It may be acceptable for a stand-alone experimental research tool to "leak properties" in some situations as long as the main concept is proved to work correctly, assuming the proof of the concept has been the main goal. However, in order to be usable by other components of the system or by the application, all properties provided by an exported module need to be rigorously specified and all "property leaks" need to be fixed; otherwise the integration of different layers and components will unavoidably falter.

The concerns raised above have motivated the integrated approach to design and implementation of the Maestro tools, which are composed as building blocks for reliable interoperable object-oriented applications. The Maestro tools cover three broad problem areas arising in development of such applications, including (1) interfaces and tools to support creation of reliable distributed objects; (2) interoperable protocols and flexible mechanisms to access the reliable objects from

outside clients; and (3) communication-level protocols to implement a meaningful high-level semantics for object reliability through replication. The three-fold focus of the Maestro tools, with a special attention to the "leak-free" integration of those three system components, is reflected in the structure of this dissertation. We describe Maestro Group Communication Tools in Chapter 2, followed by a discussion of Maestro Interoperability Tools in Chapter3. In particular, we show there how the group tools and interfaces provided by Maestro can be used to implement different styles of client/object interaction, with different reliability requirements and communication patterns. Finally, in Chapter 4 we present group protocols which implement a fundamental group-communication-based reliability paradigm called state machine replication [Sch86,Bir96]. Again, our discussion is concerned with integration of all involved system modules and, in particular, between Maestro group interfaces and tools (such as the *state transfer tool*) and underlying protocol layers. In spite of complexity of the low-level protocols involved, we have been able to express the precise semantics of state machine replication for application objects at the *application level* of abstraction with just three concise properties (Section 4.4.5), which are implemented by our protocols in cooperation with the Maestro group tools and the application itself.

Throughout the presentation, we pay special attention to usability issues, which pertains to both protocols and interfaces provided by the Maestro tools. With usability goals in mind, we illustrate most important application usage patterns with running code examples, and explicate the problems arising in design of state machine replication protocols for partitionable environments with sufficient detail (but without irrelevant implementation-specific distractions) so that they can be reimplemented by an interested reader if necessary. All in all, throughput our work we focus on practical aspects of developing reliable interoperable objects, with the end goal of effective application of the Maestro tools in real-world distributed applications.

Chapter 2

Group Communication Tools

Maestro provides a collection of tools and interfaces enabling a distributed systems developer to program directly with group communication abstractions. The core of the tools is a hierarchy of classes that implement fundamental abstract data types for object groups with integrated state transfer protocols and group-member interfaces.

Maestro supports the "object adapter" programming paradigm, where system functionality is exported via public *downcall* methods in predefined base classes, and application-specific behavior is implemented by overloading protected *callback* methods in used-defined subclasses of the base classes. For example, message-sending downcalls have matching callback methods invoked by Maestro to deliver incoming messages to destination objects. The protocols of Maestro and the group communication system underneath provide guarantees on *ordering and reliability of callback invocations* with respect to corresponding downcalls, and well as other system-level properties. The application, in its turn, specifies *how objects will process callbacks* when messages are actually delivered, by overloading callback methods in user-defined subclasses of Maestro base classes.

In the following sections we will describe the most important abstract data types in the Maestro group communication class hierarchy and discuss their usage.

2.1 Maestro Tools and Prior Work

The conceptual design of Maestro tools follows the *group programming model* of the Isis system [BJ87a,Bir96]. Isis defines a programming interface for creating and joining *process groups* and sending multicast and point-to-point messages to group members. Processes can join Isis groups as either *members* or *clients*, where members are provided with a richer collection of system services and interfaces and stronger guarantees on message ordering and membership consistency, while clients have the advantage of scalability. Isis also provides a collection of tools, including a *state transfer* tool, logging and spooling tools, the coordinator-cohort

tool, and other utilities [Isi92].

Whereas Isis is a complete self-contained distributed communication toolkit (even including its own threads package) targeted to distributed application programmers, Maestro group tools are intended to be used at the application level as well as to develop new protocols and other middleware software, such as, for example CORBA ORBs, which has motivated the differences between the structure of Maestro interfaces and Isis API. Most importantly, whereas Isis is a complete group communication system together with a programming interface, Maestro is mostly a programming environment, including an implementation of higher-level protocols, interfaces and tools, which itself requires a group protocol engine (such as Horus or Ensemble) running underneath.

The Isis programming interface is a flat library of functions that can be used by programmers to quickly develop distributed applications. However, the programmers cannot see what is going on "under the hood" and cannot modify the implementation of functions should they not adequately match their requirements. On the other hand, some higher-level group programming idioms using Isis may recur in many applications, yet would not naturally become a part of Isis toolkit. Maestro, in contrast, has been designed as an *open toolkit*, with multi-layered exported interfaces allowing the user to pick up at the most appropriate level and build new layers with necessary properties. In order to provide maximum generality, flexibility, and reusability of code, Maestro tools are structured as a hierarchy of Abstract Data Types corresponding to fundamental group abstractions (members, clients, servers with state transfer), with class semantics becoming progressively richer and more specific when climbing the class hierarchy.

Usually protocol *policies* implemented by a class become *mechanisms* when used in a subclass in the class hierarchy. For example, a class may implement certain protocol policies on synchronization of state transfer with membership changes, but not stipulate how the state should be actually transferred. Using this state transfer framework as a general mechanism, different subclasses may implement their own policies on transferring state, which, in their turn, will become mechanisms when used from yet higher programming levels. Thus, a programmer using Maestro will normally find the class in the Maestro hierarchy which implements the most specific mechanisms appropriate for the intended application, and use it as a starting point for further development (see Figure 2.1). This programming approach is different from Isis where the API is exported as a single layer and is not as easily extensible as in Maestro.

An important difference from the programming model of Isis is Maestro's object orientation. Whereas Isis implements *process groups*, Maestro's groups are comprised of *objects*. In particular, several group member objects can reside within the same process. Although it is not clear whether real applications will often need to create multiple group members within the same process, this feature is obviously useful for testing and debugging purposes. In general, raising the level of group-

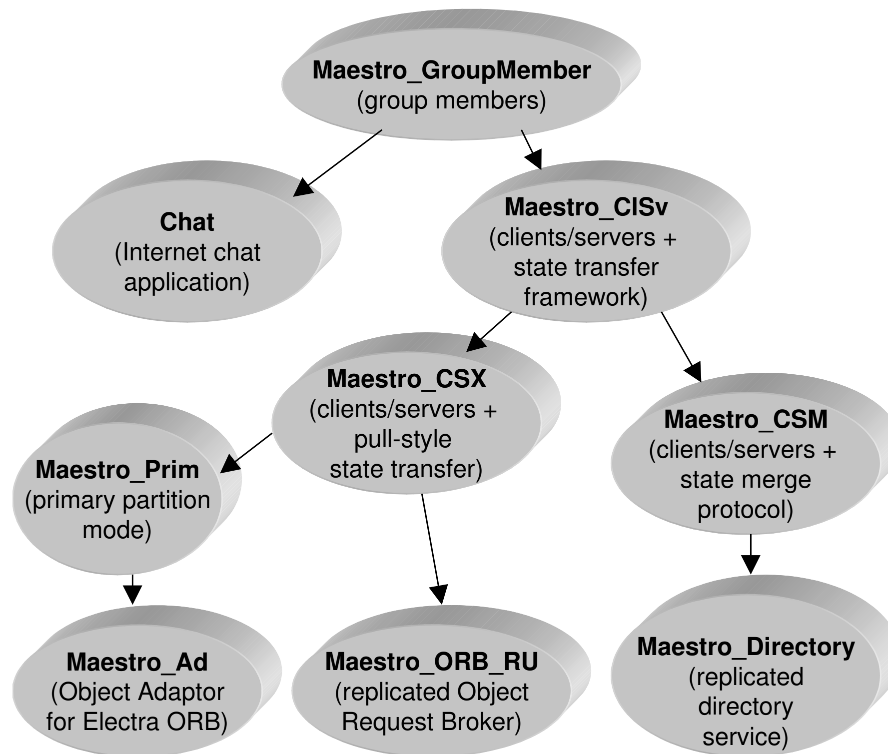

Figure 2.1: The Maestro Class Hierarchy: Core Classes and Application Examples

member abstraction from "processes" to "objects" can help system developers better modularize and encapsulate the implementation of application components and increase flexibility and reusability of code.

Isis supports scalable *client/server groups*, where large sets of clients interact with comparatively small groups of processes acting as servers. A client can pick a member of the server group and issue RPC calls to it or, alternatively, multicast to the entire group. With another group structure implemented in Isis, called *diffusion groups*, clients do not issue requests but passively receive messages multicast by group members [Isi92]. One of the most significant differences between Isis and Maestro is that Maestro primarily focuses on the flat group model, where all members or clients/servers are members of the same object group, and does not support built-in scalable clients the way Isis does. However, Maestro Interoperability Tools (discussed in Chapter 3) provide a general mechanism for connecting external clients to group objects through the standard IIOP protocol. Various protocols for client-to-group communication appropriate for different applications can be built over this mechanism.

Besides Isis, the programming model of Maestro has also been influenced by the design of Basic Object Adaptors (BOA) in the CORBA architecture [OMG97].

With the BOA model, the application defines a subclass of the pre-generated *server skeleton* class which provides a default (usually no-op) implementation of the application object's exported interface. The interface methods are invoked as callbacks when the object receives requests coming from clients. Thus, the application only needs to overload relevant callback methods in order to implement the required functionality. Maestro follows this general paradigm in the design of its group-objects class hierarchy. Namely, each class exports a collection of public *downcall methods* and protected *callback methods*. The callbacks cannot be accessed directly but are automatically invoked by Maestro when corresponding events occur. Using the default implementation, usually no action is taken when an exported callback is invoked. However, the programmer can implement a more specific communication protocol or a higher-level group member semantics by defining a subclass of a Maestro group-object class and overloading the appropriate callback methods. Note that only the callback methods which need to be modified have to be overloaded, thus eliminating unnecessary code and speeding up the development process. This programming model of Maestro is discussed in more detail in the sections below.

2.2 Group Members

Maestro implements the basic object group functionality with the `Maestro_GroupMember` class. In this section we discuss how to use `Maestro_GroupMember`'s features to build object-group applications.

2.2.1 Initialization

Initialization options in the constructor are used to specify the required system-level properties of group member objects, such as the *name of the object group* to join, the *transport protocols* supported by the machine where the object is running, the *group protocol properties*, which include message ordering options (such as safe or totally ordered delivery), security options (message authentication and application data encryption), membership protocol options (failure detection, view synchronization), partitioning behavior (support for group merges and primary views), and a number of others. Those options determine configuration for the group communication system (such as Ensemble) underneath.

2.2.2 Participating in the Membership Protocol

Maestro provides two downcall methods, `join()` and `leave()`, which are invoked by objects to join or leave their group. A `Maestro_GroupMember` object can only be a member of one group, the one whose name is specified when the object is created.

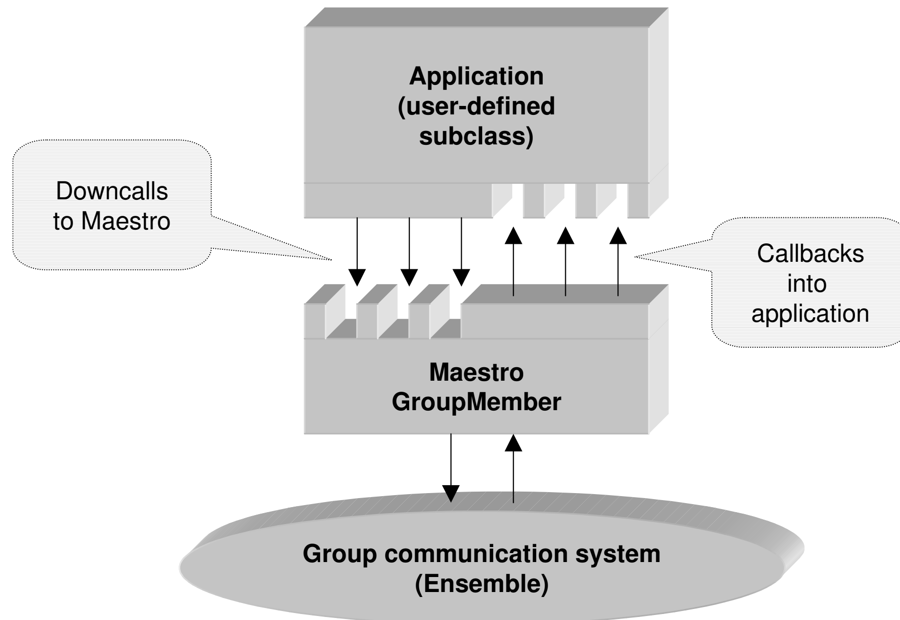

Figure 2.2: The object-adapter approach: Building applications by overloading the `Maestro_GroupMember` class

After an object joins the group, it initially installs a singleton membership view (with itself as the only member). After that, the object may merge with other components of the group and will be receiving ongoing group membership updates. In particular, whenever an object leaves the group or otherwise becomes unavailable and is excluded from the current list of members, the group communication system reports a membership change to Maestro, which passes the information to the application via the `AcceptedView()` callback (see Figure 2.3). The application can overload the callback method (no-op by default) to do whatever is necessary when a membership change is reported.

Parameters in the `AcceptedView()` function specify the current membership in the group, the lists of new and departed members since the previous membership change, and other group information. When a new membership list (a *group view*) is about to be installed, one of the group members is elected to be the *coordinator*. As a part of the view change protocol, the coordinator can multicast a *view message* to all group members included in the new view.

The application may use view messages as a simple mechanism for doing state transfer or distributing system configuration parameters. If necessary, implemen-

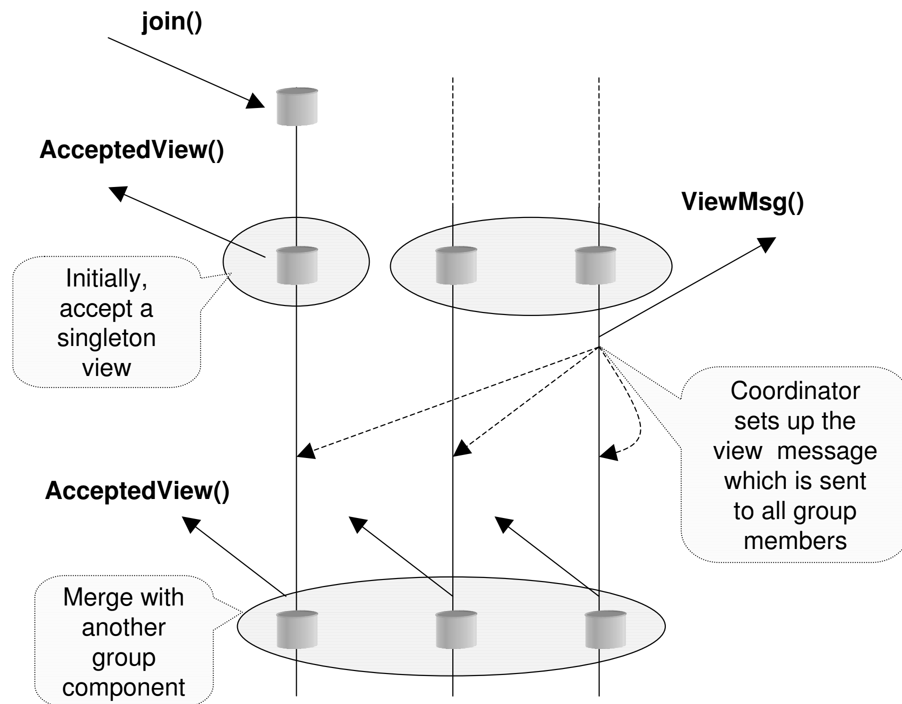

Figure 2.3: Maestro downcalls/callbacks and Membership Protocol: Joining the group

tations of group objects can overload the `ViewMsg()` callback (no-op by default) to push application-specific data into the view message. The message is delivered to member objects along with the `AcceptedView()` callback when the new view is installed.

The `leave()` function is the last downcall a group object may invoke while it is a member of the group. Following a call to `leave()`, the group will reconfigure to exclude the leaving member, which will be notified of completion of the group-leaving protocol by an invocation of the `Exit()` callback by Maestro. The remaining group members will learn of the membership change with the `AcceptedView()` callback, as described earlier (see Figure 2.4). After a group member leaves the group, it can rejoin by calling the `join()` method again.

2.2.3 Sending and Receiving Messages

Maestro supports the closed group model, where object must be members of the group in order to be allowed to send multicast messages. External clients can connect to group objects via external communication channels, for example using the IIOP protocol supported by Maestro Interoperability Tools (see Chapter 3).

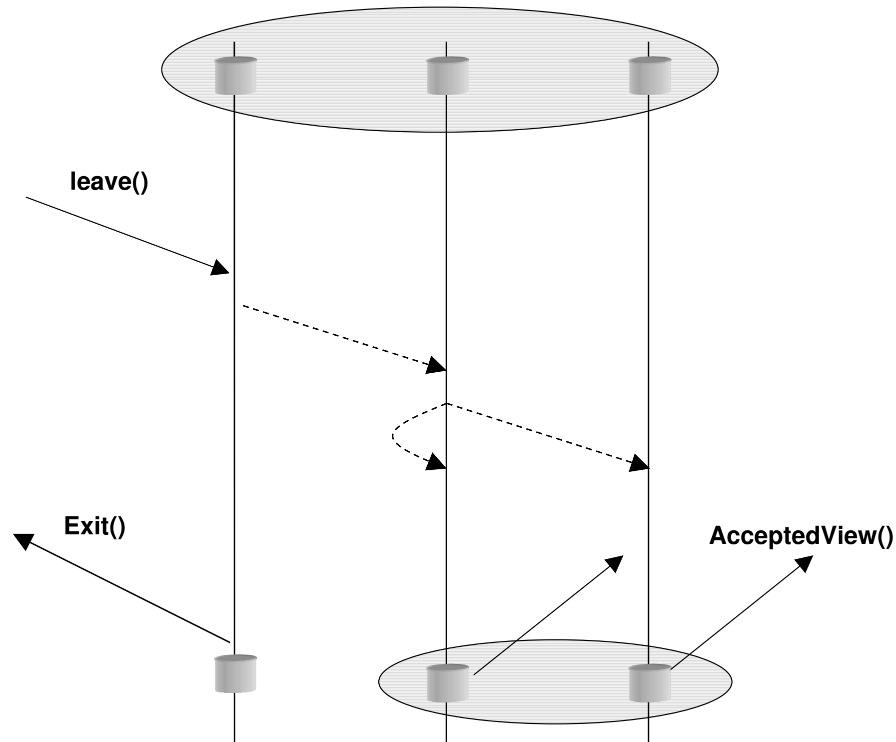

Figure 2.4: Maestro downcalls/callbacks and Membership Protocol: Leaving the group

After a `Maestro_GroupMember` object joins the group, it can send point-to-point and multicast messages to other group members. Maestro provides two downcalls, `cast()` and `send()`, which can be used to send multicast and point-to-point messages respectively. The arguments to the `cast()` and `send()` methods specify the *message* to be sent and, in case of `send()`, the *destination* of the message (the endpoint ID of the receiver object). An invocation of a `cast()` or `send()` downcall by a group member object is eventually followed by a call to the `ReceiveCast` (correspondingly `ReceiveSend`) callback at the destination(s) of the message (see Figure 2.5). However, in certain failure scenarios the message may never be delivered. The specific reliability/fault-tolerance, atomicity, ordering, and security guarantees on message delivery are provided by the group communication system underneath Maestro (such as Ensemble), according to the required group properties specified with `Maestro_GroupMember` configuration options (as described in Section 2.2.1 above).

The arguments in `ReceiveCast` and `ReceiveSend` callbacks specify the *sender* of the message and the *message* itself. By default, no action is taken when a message is received. Message callbacks are intended to be overloaded in subclasses

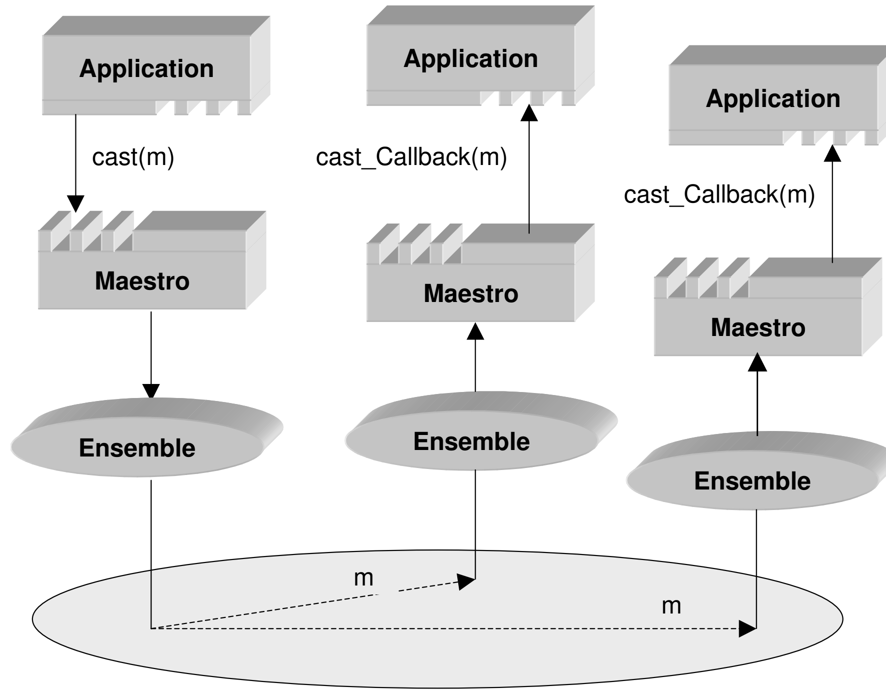

Figure 2.5: Maestro downcalls/callbacks and Ensemble messages: Sending a multicast

of the `Maestro_GroupMember` class to implement application-specific functionality.

2.2.4 External Failure Detectors

Maestro expects that the group communication system underneath (Ensemble) implements a *failure detection* mechanism. Failure detectors are used to track and report unavailability of group member objects. Those reports eventually result in membership reconfigurations, whereby unavailable (either slow, partitioned away, or crashed) members are removed from the group view.

With Maestro running over Ensemble, built-in failure detection can be enabled by requesting a corresponding group property (called *Suspect*) within group member configuration options. The application may also provide its own failure detectors, which can be used as a complement or a replacement of Ensemble's failure detection mechanism. External failure suspicions can be reported to Maestro with the `suspect()` downcall. The parameter to `suspect()` specifies the list of group member objects to be excluded from the membership view (presumably due to their unavailability).

If external (application-supplied) failure detectors are used to completely replace Ensemble's built-in mechanism, they must provide certain logical properties in order for membership protocols of the group communication system underneath to work properly. Two minimal properties required include the *detection of crashes* and *detection of unfair links* [1], which are defined as follows:

Detection of Crashes: A group member that crashes is eventually reported with a `suspect()` downcall by a member that doesn't crash (if such a member exists).

Detection of Unfair Links: A group member that doesn't crash will eventually report with a `suspect()` downcall any member of its view for which the link between the two members is unfair.

These two properties are so called *completeness* properties (in terminology of [CT93]), since they only specify in which cases an object *must* be reported as faulty, but do not set any bounds on *accuracy* of failure detection. For example, it is possible that a group member will be suspected and removed from the group view even though it didn't crash and is well connected to other group members. In general, accuracy requirements are too application-specific to be stipulated at the generic level of Maestro tools. However, the application can expect that built-in failure detection mechanisms of the group communication system underneath (such as Ensemble) are based on fixed timeouts: A group member not heard from for a certain period of time is suspected to be faulty and is removed from the view. The application can set the length of the suspicion timeout with group member configuration options.

2.2.5 Example: A Chat Program

In this section we present an example of a simple application written in Maestro. The program implements the "chat" functionality: Several copies of the program can be started on different machines; the users can type in strings which are multicast to all members in the specified "chat group" and printed out to the standard output. In order to implement the chat application, we only need to define a subclass of the `Maestro_GroupMember` class and overload several relevant callback methods:

```
// A Maestro Example: The Group Chat Application.
#include ''Maestro.h''
// Define a subclass of Maestro_GroupMember.
class Chat: public Maestro_GroupMember {
public:
```

[1] A link is *fair* if any message sent infinitely many times is received infinitely many times by its destination. A link is *unfair* if it is not fair.

```
    Chat(Maestro_GrpMemb_Options &ops):  Maestro_GroupMember(ops) {
        initialized = 0;
        // Join the specified chat group.
        join();
    }
protected:
    // Overload exported protected callbacks of Maestro_GroupMember:
    // Received a multicast message.
    void grpMemb_ReceiveCast_Callback(Maestro_EndpID &origin,
                                      Maestro_Message &msg) {
        Maestro_String str;
        // Extract the multicast string from the msg and print it out.
        msg >> str;
        cout << str << endl;
    }
    // Installed a new view.
    void grpMemb_AcceptedView_Callback(Maestro_GrpMemb_ViewData &viewData,
                                       Maestro_Message &msg) {
        cout << ``Current members in the chat group:''  << endl;
        cout << viewData.members;
        // If this is the first view, start the input processing thread.
        if (!initialized) {
            initialized = 1;
            Maestro_Thread::create(processInput, (void*) this);
        }
    }
private:
    // Input processing thread: read strings from standard input and
    // multicast to the group via the specified Chat object.
    static void processInput(void *arg) {
        Chat *chat = (Chat*) arg;
        char buf[128];
        // In the loop: Read next string from stdin,
        // convert into a Maestro_String, marshal into a message,
        // and multicast to the chat group.
        while (1) {
            cin >> buf;
            Maestro_String str(buf);
            Maestro_Message msg;
            msg << str;
            chat->cast(msg);
        }
    }
    int initialized;
};
```

```
void main(int argc, char *argv[]) {
    if (argc < 2) {
        cerr << ``Usage:  `` << argv[0] << `` <chat-group-name>'' << endl;
        return 1;
    }
    // Setup configuration options.
    Maestro_GrpMemb_Options ops;
    ops.groupName = argv[1];
    ops.transports = ``UDP'';
    ops.argv = argv;
    // Specify required group properties:
    // Membership, failure detection, total ordering etc.
    ops.properties = ``Gmp:Sync:Heal:Switch:Frag:Suspect:Flow:Total'';

    // Create a Chat object. If this operation fails, the panic() method
    // of Chat's error handler will be invoked. In the default
    // implementation, a call to panic() will terminate the process.
    Chat *chat = new Chat(ops);

    // Block the main thread forever.
    Maestro_Semaphore sema;
    sema.dec();
}
```

2.3 Group Clients and Servers

Maestro group communication tools include an implementation of abstract data types for group *clients* and *servers*. The interfaces to both clients and servers are almost identical, which reflects the Maestro's approach of treating clients as merely group members without the (replicated) *group state*. Thus, Maestro clients are fully aware of group semantics and are notified of all membership changes in the group, including addition and departure of both client and server members. Also, both servers and clients can explicitly send messages to other group members, and have the same ordering, reliability, and security guarantees on message delivery (as provided by the group communication system underneath)[2]. Furthermore, Maestro group clients may be upgraded to the server status upon request, in which case they proceed through the *state transfer protocol* so as to bring themselves up to the current state of server members in the group. On the other hand, server members may temporarily lose their server status and become "clients" in certain group-merge scenarios, in which case they, too, will need to participate in the

[2]This is different from semantics of *external clients* discussed in Chapter 3, which do not join Maestro object groups directly but access them through a representative member using the IIOP protocol.

state transfer protocol to receive the current state of the more updated merging component, after which they may again be assigned the server status.

Maestro implements group clients and servers with the `Maestro_ClSv` class, which is defined as a subclass of `Maestro_GroupMember`. Similarly to `Maestrp_GroupMember`, the application will normally define a subclass of `Maestro_ClSv` which will overload the exported protected callbacks so as to implement the required event-processing functionality. The interface and implementation details of the `Maestro_ClSv` class, including the state transfer protocol used by server members, are discussed in the sections below.

2.3.1 Initialization and State Transfer Options

The options to the constructor method for the `Maestro_ClSv` class specify (in addition to configuration options for `Maestro_GroupMember`) whether the group object should be created in the *client* or *server* mode. Only server members participate in the state transfer protocol. Another initialization option specifies *state-transfer safety level*. Maestro supports several safety levels, which constrain the types of messages that can be sent by group members during execution of the state transfer protocol:

Free State Transfer: With *free* state transfer, all types of application messages can always be sent, even during execution of the state transfer protocol.

Protected State Transfer: During *protected* state transfer, the application may only send messages that do not result in modifications to the state of group server objects. Those messages are marked as "safe" by the application and are immediately sent out by Maestro. All other messages generated by the application during state transfer are buffered in Maestro and sent out only after the state transfer protocol completes.

Atomic State Transfer: With *atomic* state transfer, only messages generated as a part of the state-transfer protocol are allowed to be sent immediately. Those are internal state-transfer protocol messages sent by Maestro and application's messages marked as belonging to the "state-transfer" type. All other messages sent by the application are delayed if sent during execution of state transfer protocol.

The goal of Maestro has been to provide maximum flexibility with respect to protocol properties of state transfer, so as to allow applications to implement their own state transferring schemes. This flexibility is enabled, in particular, by letting the application choose the appropriate safety level for the state transfer protocol and letting it have explicit control over safety types of individual messages.

While providing support for potentially sophisticated state transfer schemes, the default mechanism implemented in Maestro offers a simple interface and is adequate for many applications. We will discuss the implementation of state transfer in Maestro and available interfaces in Section 2.4.

2.3.2 Sending Messages

In addition to `send()` and `cast()` downcalls of `Maestro_GroupMember`, the `Maestro_ClSv` class provides methods for server-casting (sending messages to server members only) and subcasting (sending messages to subsets of group members), called `scast()` and `lsend()` correspondingly.

The options to message downcalls (`cast()`, `send()`, `scast()`, and `lsend()`) specify the state-transfer safety level of messages, which can be of three types: *generic*, *safe*, and *state-transfer*. The safety types have the following semantics: *generic* messages are delayed during *protected* and *atomic* state transfers; *safe* messages are delayed during *atomic* state transfers; finally, *state-transfer* messages are not delayed in state transfers of all types.

Another option, used with the `lsend()` downcall, specifies the *destination list* for the message to be sent. Using `lsend()`, a message can be sent to an arbitrary subset of objects in the group, either servers or clients. Similarly, by using destination list option with a servercast, the message can be sent to all servers and the specified clients.

When messages are delivered at the destinations, Maestro notifies the application via corresponding callback methods, namely `ReceiveCast()`, `ReceiveSend()`, `ReceiveScast()`, and `ReceiveLsend()` respectively matching `cast()`, `send()`, `scast()`, and `lsend()` downcalls. The message callbacks are defined in `Maestro_ClSv` as no-op functions which can be overloaded in subclasses of the `Maestro_ClSv` class so as to implement the required application functionality.

2.3.3 Membership Monitoring

As with the `Maestro_GroupMember` class, Maestro reports membership changes to group client/server objects with the `AcceptedView()` callback. However, in addition to the information provided to `Maestro_GroupMember` objects, `AcceptedView()`'s parameters also specify the lists of new and departed client and server members in the group view and the members doing state transfer, the member's state, and other group status information.

2.4 State Transfer

When a new server object joins the group or, more generally, when several group components merge together, Maestro initiates the *state transfer protocol* to bring

the state of server members in the less advanced component up-to-date. The group communication system underneath (Horus or Ensemble) determines the *direction* of state transfer (which members need to *receive* the state and which members will be *transferring* the state) and provides that information to Maestro. We will discuss an implementation (in Horus) of group protocols for state-machine replication, including policies for merging group components and determining direction of state transfer, in Chapter 4. Maestro, in turn, is responsible for the *mechanism* of state transfer (the state transfer protocol itself), which can be implemented in several ways. In Isis [Bir96], for example, one of the old group members (usually the coordinator) sends the state to new members. This is called the *push* approach. In case the coordinator fails during state transfer, the new coordinator has to restart the protocol from scratch, since it doesn't know what portion of the state has already been transferred to the new member. This scheme can be optimized with a protocol where the new server is responsible for requesting the state from more up-to-date members and notifying the coordinator when state transfer has completed (the *pull* approach [Bir96]).

With either push or pull mechanisms used in a state transfer protocol, there is a number of options for structuring the protocol itself, including the behavior of the application during state transfer and the contents of the state. As discussed in [Bir96], the contents and structure of the state is usually application-specific, and the size of the state can vary from rather small to very large. The availability requirements on the application can be high or moderate. The rate of updates to the state can be very low, or they can happen quite frequently. Finally, the state may or may not be reconstructible in real-time. It is obvious that it is not possible to devise a single protocol that would optimally or at least adequately solve the state transfer problem for all combinations of system parameters listed above. This consideration has motivated the state-transfer approach of Maestro, which does not impose any rigid mechanisms based on ungrounded assumptions concerning the properties and requirements of the application, but instead provides a general protocol framework within which various state transfer schemes can be implemented.

2.4.1 Protocol Framework

The framework for state transfer protocols is implemented in Maestro with the `Maestro_ClSv` class. `Maestro_ClSv` defines so called *state transfer views*, which are transient views in which some of the group members are in the process of doing state transfer. Maestro determines when state transfer needs to be started and keeps track of its completion by all participating members. The application is informed of the state transfer status with `AcceptedView()` callbacks. The `Maestro_ClSv` class does not specify *how* the state should be transferred, however it provides a programming model, with several *safety levels* of state transfer and

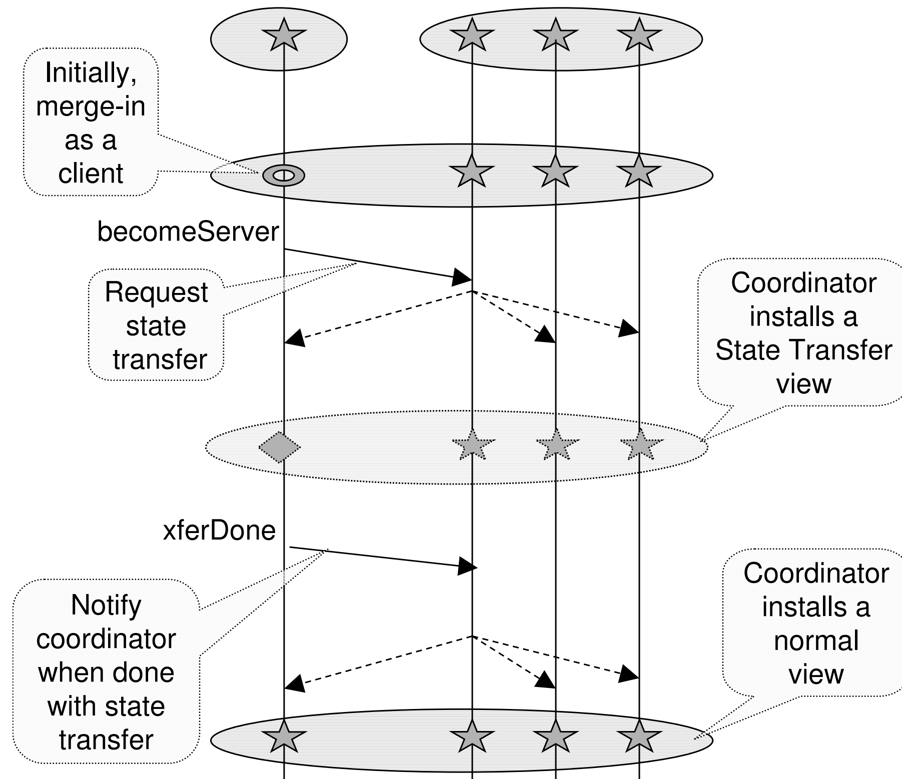

Figure 2.6: State Transfer Protocol in Maestro

ability to control safety on a per-message basis, as discussed above. The only protocol requirement on the application is to notify Maestro when individual group server objects complete updating their state and are ready to be included in the list of normal members. The state-transfer completion notification is done with the `xferDone()` downcall.

The normal sequence of stages for a server object merging into a group is therefore the following. Initially the server joins with the *client* status, which simply means it does not have an up-to-date copy of the group state. Then the new member automatically notifies the group coordinator that it wants to become a server, at which point a *state transfer view* is installed. In that view, the new member is included in the list of *state-transferring servers* (see Figure 2.6). State-transferring servers receive the same messages as normal server members. In particular, they receive all servercasts sent in the group. However, depending on the specific protocol, state-transferring servers may or may not be responsible for processing application requests. Finally, when a new server completes its state transfer, it notifies the coordinator by calling the `xferDone()` function, at which point a new view is installed. In that view, the new member is listed among normal *server members* in the group, thus completing the merge-in protocol.

Observe that Figure 2.6 does not show *how* the state is transferred to the merging-in server. As we mentioned before, `Maestro_ClSv` does not impose a mechanism for transferring state, neither the pull nor the push approach. The actual implementation of state transferring mechanisms is left up to higher levels in the Maestro class hierarchy.

In the example below we show how a simple state transfer policy can be implemented over the `Maestro_ClSv` class. A joining member requests the state from the oldest up-to-date server object in the group. If the selected server crashes, the state request is automatically resubmitted to another up-to-date server:

```
class MyServer: Maestro_ClSv {
public:
    MyServer(Maestro_ClSv_Options &ops): Maestro_ClSv(ops) {
        xferStatus = OFF;
    }

    // Set the replicated state to a new value:
    void setState(Maestro_String &s) {
        Maestro_Message msg;
        msg << s;
        // Multicast the update to all servers.
        scast(msg);
    }

protected:
    // Received a set-state message: update the local state.
    void clSv_ReceiveScast_Callback(
        Maestro_EndpID &origin,
        Maestro_Message &msg)
    {
        msg >> state;
    }

    // A new view has been installed.
    void clSv_AcceptedView_Callback(
        Maestro_ClSv_ViewData &viewData,
        Maestro_Message &viewMsg)
    {
        // (Re)start state xfer if necessary:
        // Request state from the oldest up-to-date server.
        if (viewData.startXfer) {
            xferStatus = ON;
            xferServer = viewData.servers[0];
            Maestro_Message reqMsg;
            askState(xferServer, reqMsg);
```

```
        cout << ``Requesting state from '' << xferServer << endl;
    }

    // Terminate state transfer if necessary.
    if ((xferStatus == ON) &&
        (viewData.state != MAESTRO_CLSV_STATE_SERVER_XFER))
    {
        cout << ``State transfer terminated'' << endl;
        xferStatus = OFF;
    }

    // If this member is waiting for state from a server that has
    // just crashed, resubmit the state request to another server.
    if ((xferStatus == ON) &&
        (viewData.state == MAESTRO_CLSV_STATE_SERVER_XFER) &&
        (!viewData.servers.contains(xferServer)))
    {
        xferServer = viewData.servers[0];
        Maestro_Message reqMsg;
        askState(xferServer, reqMsg);
        cout << ``Resubmitting state request to ''
                << xferServer << endl;
    }
}

// Received a state request. Reply with the current state.
void askState_Callback(
    Maestro_EndpID &origin,
    Maestro_Message &reqMsg)
{
    Maestro_Message stateMsg;
    stateMsg << state;
    sendState(origin, stateMsg);
    cout << ``Received state request from '' << origin << endl;
}

// Received a state message: Update the local state;
// Notify Maestro that state transfer has been completed.
void rcvState_Callback(
    Maestro_EndpID &origin,
    Maestro_Message &stateMsg)
{
    // Use the state message only if it is expected.
    if (xferStatus == ON) {
        cout << ``Received state message from '' << origin << endl;
```

```
            assert(origin == xferServer);
            stateMsg >> state;
            cout << ``My state is:  ''  << state << endl;
            xferStatus = OFF;
            xferDone();
        }
    }

    Maestro_String state;
    Maestro_Endpoint xferServer;
    enum {OFF, ON} xferStatus;
}
```

In the next section we will discuss how a simple pull-style state transfer mechanism with a higher-level interface is implemented in Maestro with the `Maestro_CSX` class (a subclass of `Maestro_ClSv`).

The state transfer protocol of the `Maestro_ClSv` class is fault-tolerant, so that, in particular, if old servers fail or the group partitions before completion of state transfer, the state transfer will be restarted as necessary in the new view. It may also happen that a state transfer will have to be canceled and restarted if a group component undergoing state transfer merges together with another component. In general, when two group components merge, server members in one of them (the one with less updated state) will temporarily lose their server status and be included in the view as client. In this case Maestro will automatically start state transfer from more advanced servers so as to bring all server members up to date. When state transfer completes (as indicated by `xferDone()` downcalls invoked by all state-recipient members), Maestro will reinstall the view with all server members now assigned the normal status.

The `Maestro_ClSv` class defines a generic interface for doing state transfer via normal group transport channels. The exported methods include public downcalls `askState()` and `sendState()` for requesting and sending (portions of) the state respectively, with matching protected callbacks `askState()` and `rcvState()`. The `askState()` downcall specifies the group member from which the state is requested. Another argument is a *request message* identifying the portion of the state to be sent. After a state-receiving server object invokes the `askState()` downcall, the server from which the state is being requested eventually receives the request message via the `askState()` callback invoked by Maestro. The server is expected to eventually reply to the state-requesting member with a `sendState()` downcall. The `sendState()` method specifies the *state message* and the group member to which it is to be sent. After an invocation of the `sendState()` downcall, the destination member will eventually receive the state message through the `rcvState()` callback invoked by Maestro. At that point it can extract the state from the specified message and merge it with its local state.

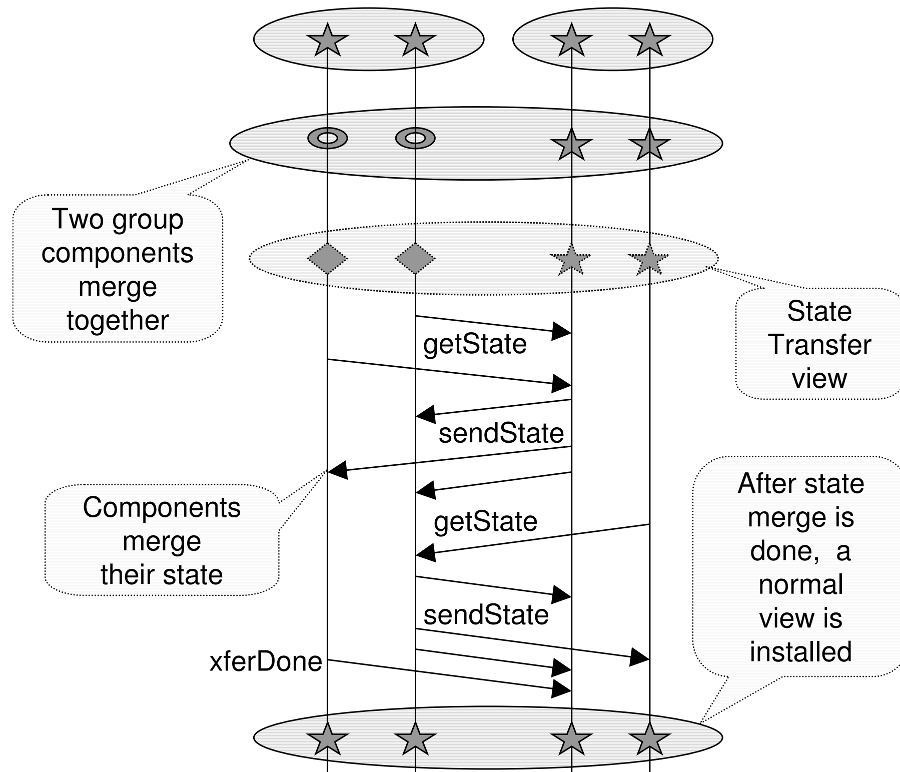

Figure 2.7: A State Merge protocol: When two group components merge together, they exchange their states using getState()/sendState() downcalls of Maestro

While an application can use the provided downcalls (`askState()` and `sendState()`) to perform state transfer via the standard group transport, it is also feasible (and recommended in [Bir96] for a certain type of applications) to use an out-of-band channel so as not to interfere with the normal ongoing communication in the group.

It is important to note that `Maestro_ClSv` does not impose any rules on the contents of request messages and state messages and does not stipulate how state-receiving group members should choose up-to-date servers from which to receive the state. The state transfer framework provided by `Maestro_ClSv` can be used to implement different state transferring policies and schemes and, in particular, can be used to perform not only *state transfer* from a more update group component to a less updated one but, more generally, *state merge*, for applications which allow progress in multiple concurrent group components and need to somehow reconcile their state during merges (Figure 2.7). However, the required semantics of state transfers and merges and optimal protocol tradeoffs depend heavily on the nature and properties of the application and therefore are not implemented at the generic level of the `Maestro_ClSv` class.

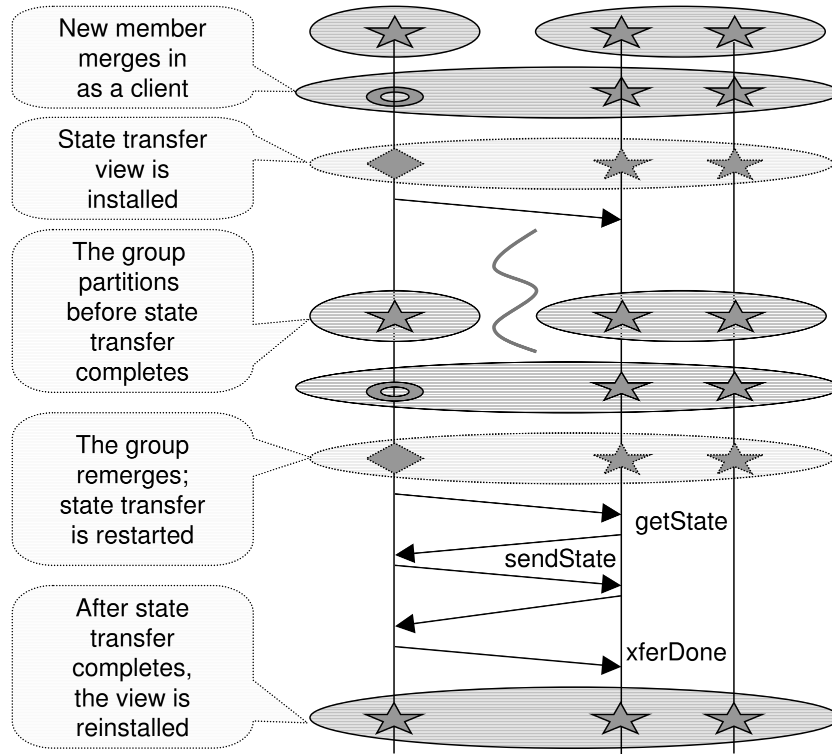

Figure 2.8: State transfer protocol: Under certain failure scenarios, state transfer may be canceled and restarted later

2.4.2 A Pull-Style Protocol

In this section we discuss a simple push-style state transfer protocol implemented in Maestro with the `Maestro_CSX` class. The protocol is appropriate for applications with a sufficiently small state which can be transferred with one or several messages, and where it is acceptable for the application not to perform any state updates until completion of state transfer. For applications with a large or frequently-changing state, or those with extremely high availability requirements, more sophisticated state transferring schemes would need to be advised. However, it appears that an important class of system management, membership service, and replicated control applications do fall under the small-state category and their state does not change too frequently. In particular, a fraction-of-a-second unavailability period due to state transfer may be quite acceptable in those systems.

According to the state transfer protocol implemented in the `Maestro_ClSv` class, initially a new server member joins the group with the client status. Then the `becomeServer` request is automatically sent to the coordinator and a `state transfer view` is installed, in which the new member is listed as a

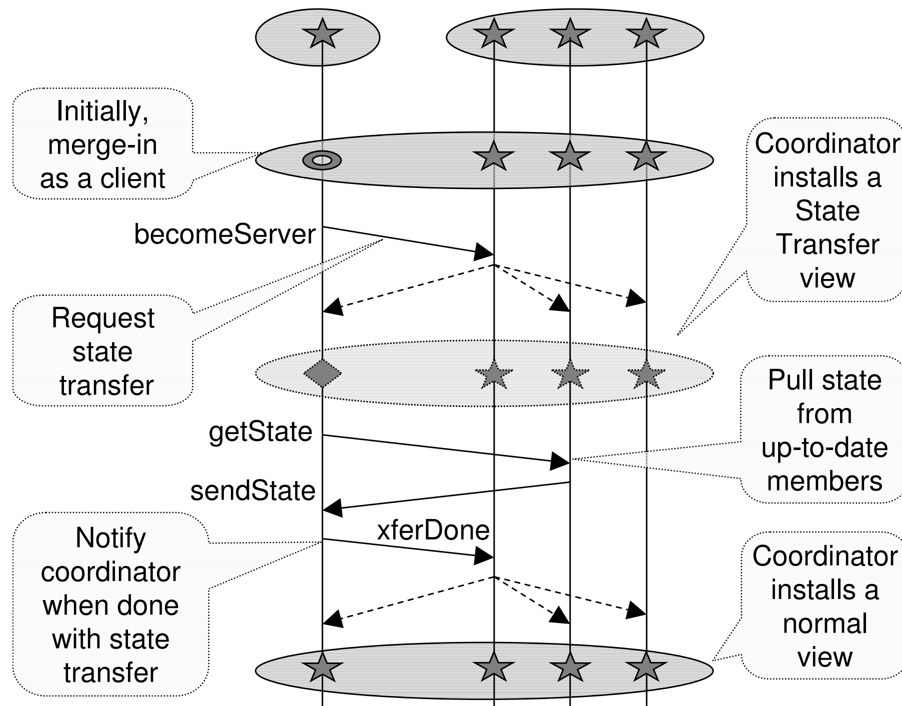

Figure 2.9: A Pull-Style implementation of State Transfer Protocol

`state-transferring server`. At that point the `Maestro_ClSv` class notifies the application (with a corresponding flag set in the `AcceptedView()` callback) that state transfer needs to be restarted. It may happen, however, that state transfer will need to be canceled before it completes, in which case it may be restarted again later. For example, the joining server member may partition away from up-to-date servers before it obtains the state from them and then merge back (see Figure 2.8). It may also happen that old servers will all crash before state transfer completes. The application written directly over the `Maestro_ClSv` class would need to correctly handle all those scenarios and reset its state/restart state transfer when necessary. The protocol involved adds substantial complexity and is certainly not at the "application" level of abstraction. The `Maestro_CSX` class, defined as a subclass of `Maestro_ClSv`, takes care of all special cases, including cancellation/restart of state transfer, and presents a convenient high-level interface to the application.

The pull-style state transfer scheme implemented in the `Maestro_CSX` class is illustrated in Figure 2.9. With the implementation of `Maestro_CSX`, each instance of state transfer is treated as a *transaction* and is assigned a unique *state transfer ID*. All state transfer messages are sent in the context of a particular state transfer

transaction. When a state transfer transaction is canceled or restarted, the application is notified with a corresponding callback, which in particular specifies the state transfer ID. This transaction-style approach turns out to be especially convenient in the normal situation when state transfer is performed asynchronously in a separate thread. It may happen, in particular, that an old state transfer thread will not yet be aware of cancellation of its transaction, while another state transfer thread may have been already created concurrently and performing the new state transfer transaction. However, since each state transfer is identified by a unique ID, the application can always distinguish between different state transfer transactions, even if they overlap in time due to asynchronous threading.

The state transfer interface exported by the `Maestro_CSX` class is as follows. The application is notified of a new state transfer transaction with the `stateTransfer()` callback. The `stateTransfer()` method is invoked in a separate thread, so that it can perform blocking or long-duration operations without blocking other Maestro callbacks. The `xferID` argument of the `stateTransfer()` function identifies the state transfer transaction being started [3]. The joining member can request the state from the oldest up-to-date server with the `getState()` downcall. The server will receive the state request with the `askState()` callback and can send the requested portion of the state to the joining member with the `sendState()` downcall. When the member receives the state message, the `getState()` function returns with the normal status, with the `stateMsg` parameter containing the received message. The joining member can invoke `getState()` multiple times if the group state is too large to fit in a single message and needs to be transferred in several portions. Each time, the state portion being requested can be specified with the `requestMsg` parameter [4]. When state transfer completes, the joining member notifies Maestro with the `xferDone()` downcall. At that point a new view is installed, in which the joining member is listed among normal (up-to-date) group servers. It is essential, however, that `getState()`, `askState()`, `sendState()`, and `xferDone()` methods all explicitly specify the ID of the state transfer transaction being performed, so that all state messages are sent in the specific context of that transaction. In case state transfer is aborted, the application is notified by the `getState()` function returning with the state-transfer-terminated status.

An example of the *Grid* application with replicated state doing state transfer is shown below. The group state is represented by a 10x10 integer matrix (the grid). Two operations, `set(x,y,val)` (set the value of cell `(x,y)` to `val`) and

[3] In the current implementation, `xferID` is an opaque type used to match state requests with state reply messages and identify completed or canceled state transfer transactions. Assuming that at most one state transfer transaction can be started in any group view, `xferID`'s can be implemented as matching view ID's.

[4] In order to prevent corruption of the state in scenarios when state transfer is terminated prematurely, state requests are linearly ordered, and the application is not allowed to request a new portion of the state until it receives a reply to the preceding state request. In other words, the `getState()` downcall is not reentrant.

`get(x,y)` (get the value of cell `(x,y)`) are supported. The grid application is implemented with the `Grid` class defined as a subclass of `Maestro_CSX`. The `get()` operation simply returns the value of the cell `(x,y)` in the local replica of the grid. To perform the update operation (`set()`), a totally ordered multicast message is sent to the group. When a grid replica receives the message, it performs the local update.

In the overloaded `stateTransfer_Callback()` method, a joining server requests the up-to-date grid from an old server with the `getState()` downcall. The server from which the state is asked writes the local copy of the grid into a state message and sends it to the joining member, as implemented in the overloaded `askState_Callback()`. If the state is transferred successfully, the joining member extracts the grid from the state message and applies it to the local state:

```
// Maestro State Transfer Example: The Grid Application.
#include ``Maestro.h''
// Define a subclass of Maestro_CSX.
class Grid: public Maestro_CSX {
public:
    Grid(Maestro_CSX_Options &ops): Maestro_CSX(ops) {
        initialized = 0;
        // Initialize the grid.
        grid = new int[100];
        memset(grid, 0, sizeof(int)*100);
        // Join the specified grid application group.
        join();
    }
    ~Grid() { delete [] grid; }
protected:
    // Overload exported protected callbacks of Maestro_CSX:
    // Received a multicast message.
    void csx_ReceiveCast_Callback(Maestro_EndpID &origin,
                                  Maestro_Message &msg) {
        int x, y, val;
        // Unmarshal the (x,y,val) parameters and update the grid.
        msg >> x >> y >> val;
        cout << ``Setting grid['' << x << ``,'' << y << ``] to '';
        cout << val << endl;
        mutex.lock();
        grid[x*10+y] = val;
        mutex.unlock();
    }
    // Installed a new view.
    void csx_AcceptedView_Callback(Maestro_CSX_ViewData &viewData,
                                   Maestro_Message &msg) {
        cout << ``Current members in the grid group:'' << endl;
```

```
        cout << viewData.members;
        // If this is the first view, start the input processing thread.
        if (!initialized) {
            initialized = 1;
            Maestro_Thread::create(processInput, (void*) this);
        }
    }
    // Starting state transfer.
    void stateTransfer_Callback(Maestro_XferID &xferID) {
        Maestro_Message requestMsg, stateMsg;
        Maestro_XferStatus xferStatus;
        // Request the state from an up-to-date server replica.
        getState(xferID, requestMsg, stateMsg, xferStatus);
        if (xferStatus == MAESTRO_XFER_TERMINATED) {
            cout << ``State Transfer terminated'' << endl;
        }
        else {
            mutex.lock();
            // Extract the grid from stateMsg.
            stateMsg.read(grid, sizeof(int)*100);
            mutex.unlock();
            xferDone(xferID);
        }
    }
    // Received a state request. Send the grid back.
    void askState_Callback(Maestro_EndpID &origin,
                          Maestro_XferID &xferID,
                          Maestro_Message &requestMsg) {
        Maestro_Message stateMsg;
        mutex.lock();
        stateMsg.write(grid, sizeof(int)*100);
        mutex.unlock();
        sendState(origin, xferID, stateMsg);
    }
private:
    // Input processing thread: read commands from standard input and
    // multicast updates to the group via the specified Grid object.
    static void processInput(void *arg) {
        Grid *gr = (Grid*) arg;
        char buf[128];
        int x, y, val;
        // In the loop: Read next command from stdin;
        // perform "get" requests based on local state;
        // multicast "set" requests to the grid group.
        while (1) {
```

```
            cin >> buf;
            if (strcmp(buf, ``get'') == 0) {
                // "get" request: print out the current value at grid[x,y].
                cin >> x >> y;
                gr->mutex.lock();
                cout << ``The value of grid['';
                cout << x << ``,'' << y << ``] is '';
                cout << gr->grid[x*10+y] << endl;
                gr->mutex.unlock();
            }
            else if (strcmp(buf, ``set'') == 0) {
                // "set" request: marshal (x,y,val) parameters into a message
                // and multicast it to the group.
                cin >> x >> y >> val;
                Maestro_Message msg;
                msg << val << y << x;
                gr->cast(msg);
            }
            else {
                cout << buf << ``:  unknown request type'' << endl;
            }
        }
    }
    int initialized;
    Maestro_Lock mutex;
};
void main(int argc, char *argv[]) {
    // Setup configuration options:
    Maestro_CSX_Options ops;
    ops.groupName = ``Grid'';
    ops.transports = ``UDP'';
    // Join the grid group as a server; request atomic state transfer.
    ops.mbrshipType = MAESTRO_SERVER;
    ops.xferType = MAESTRO_ATOMIC_XFER;
    ops.argv = argv;
    // Specify required group properties:
    // Membership, failure detection, total ordering etc.
    ops.properties = ``Gmp:Sync:Heal:Switch:Frag:Suspect:Flow:Total'';

    // Create a Grid object.
    Grid *grid = new Grid(ops);
    // Block the main thread forever.
    Maestro_Semaphore sema;
    sema.dec();
}
```

Chapter 3

Object Interoperability Tools

3.1 Introduction

The distributed systems community has shown considerable interest in integrating interoperable object-oriented technologies, such as CORBA [OMG97] and DCOM [Mic98], with technologies for building secure, reliable distributed systems [Bir96,MMSN97, NMMS97b,NMMS97a,VB97,MFSW95,Hay97,Maf95a,Maf95b]. Maestro interoperability tools provide such integration mechanisms, which can be used to glue together object-oriented distributed applications with reliable group communication systems (such as Horus or Ensemble [vRBM96,Hay97]) while maintaining application component interoperability through the use of a standard communication protocol (IIOP [OMG97]).

The Maestro tools can be used directly to implement reliable interoperable objects in distributed applications, or they can be integrated within higher-level distributed object technologies such as CORBA Object Request Brokers. The ORB implements an infrastructure gluing together a client application accessing an object with the object implementation. The object can reside within the client's process, or on the client's machine, or within the domain of the client's LAN, or anywhere on Internet. Regardless of an object's location, the ORB is responsible for locating the object, delivering client requests to it, performing requested operations, and returning results to clients, all of this seamlessly from the perspective of application. The high-level CORBA ORB architecture is shown in Figure 3.1. The client application doesn't access the object directly but instead invokes methods on the *client stub*, which marshals request parameters into a message and passes it to the *ORB core*. The ORB delivers the request message to the object implementation's process, where it is dispatched through the *object adaptor*. Before the requested operation is performed upon the object, the *object skeleton* unmarshals parameters from the request message and converts the request into a method invocation. Upon completion of the operation, the Skeleton marshals return parameters into a reply message and passes it to the ORB, which sends the message back to

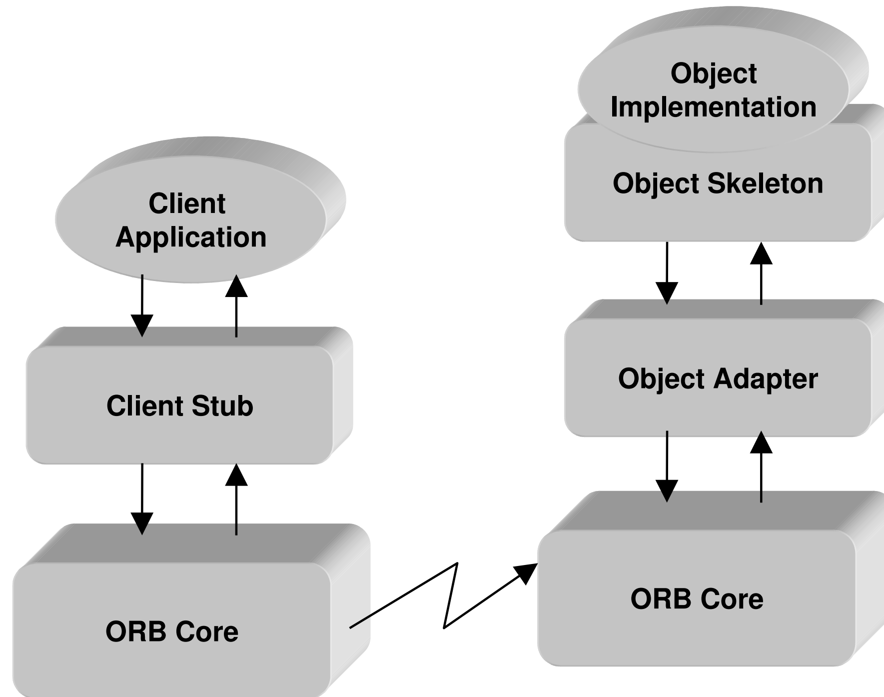

Figure 3.1: CORBA ORB Architecture

the client process. When the ORB at the client side receives the reply message, it hands it to the client stub, which unmarshals return parameters from the message and passes them to the client application.

An Object Request Broker includes both a *transport layer* and a *request manager* which dispatches incoming requests to corresponding object adapters. The functionality of a typical request manager in a CORBA implementation is quite simple (look up the target object and, if found, pass the request to its object adapter). Therefore in the standard CORBA architecture request managers are not treated as a separate abstraction layer, and the notion of a request manager is not a part of the common CORBA terminology.

In our work, the request manager takes on an enlarged role. Maestro can support multiple request managers with more sophisticated message-dispatching mechanisms and policies, which can be used to provide reliability/high availability of IIOP-based services (see Figure 3.2). The mechanisms implemented by those request managers are orthogonal to the functionality provided by the transport layer (which in Maestro is essentially an *IIOP bridge*). We will use terms *request manager* and *ORB* interchangeably throughout this chapter when focusing on request processing policies of ORBs rather than transport-layer issues.

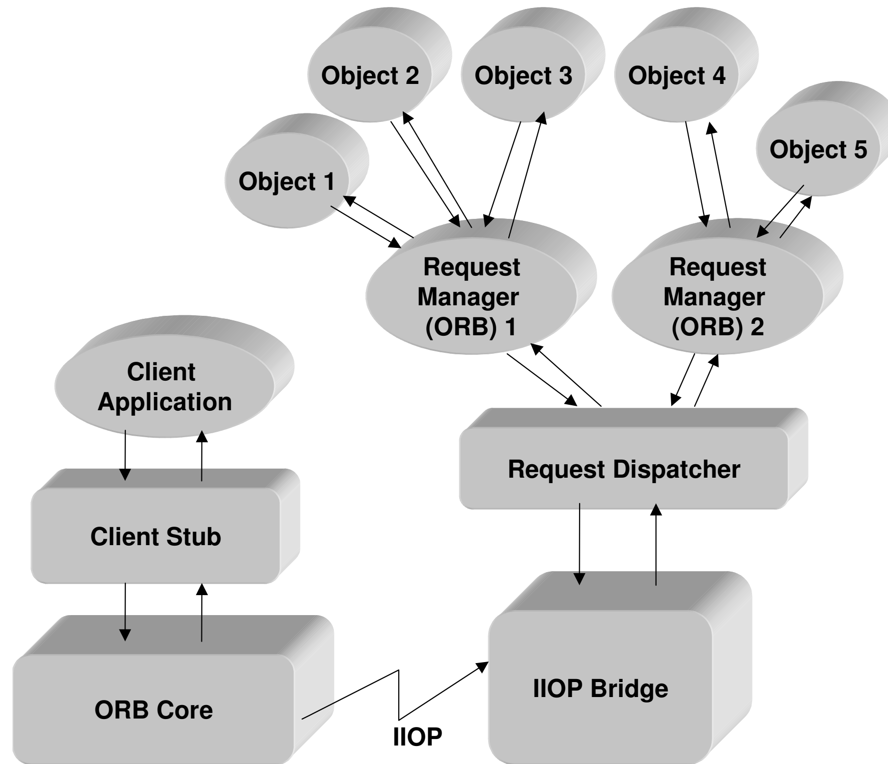

Figure 3.2: Maestro Interoperability Architecture: Multiple Request Managers (ORBs) can be supported over a shared IIOP bridge

Maestro includes a repository of ORBs supporting several execution styles and request-processing policies. The ORBs are discussed in detail below and in following sections. The toolkit respects an open architecture, whereby multiple ORBs can be plugged into the shared IIOP bridge/dispatcher as long as they implement a common interface specified by Maestro. Flexibility is one of the primary goals of this architecture, so that a distributed system developer using Maestro can select the request manager most closely matched to the requirements and communication pattern of the application. For example, the Replicated Updates ORB implemented in Maestro uses Horus or Ensemble to provide high availability of services by actively replicating server objects. Those systems, in turn can be customized to use a distributed protocol optimized to the communication pattern or "system" properties (such as failure detection, message ordering guarantees, security requirements, scalability etc.) desired for the application. Other request managers might employ different techniques or replication paradigms, such as the coordinator-cohort execution style [Bir96].

Maestro interoperability tools include an implementation of IIOP Bridge and a framework for building customized Object Request Brokers [OMG97]. The ORBs

included with Maestro are a reliable Replicated Updates ORB (with object implementations actively replicated over a group communication system underneath), and a Simple ORB with no reliability mechanisms provided. Both ORBs are discussed in detail in the following sections and can be used as a reference/example when building custom ORBs or other tools over Maestro.

Although both client and server sides of the Maestro IIOP Bridge have been implemented, Maestro is generally targeted at the server side of distributed applications, with an intention that clients can be built with any CORBA/IIOP-compliant ORB, such as Iona's Orbix 2.2 for C++ or OrbixWeb 3.0 for Java [1], and seamlessly invoke objects implemented in Maestro. From this perspective, Maestro can be used as the transport/communication layer of a reliable ORB, or it can be used directly to program interoperable objects. In order to extend Maestro to a complete ORB, it would be necessary to write a CORBA-interface layer over Maestro, supply an IDL compiler, and provide standard CORBA services and tools. Maestro itself provides an equivalent of CORBA's Dynamic Skeleton Interface, where parameter marshaling and request-to-object-method dispatching at the server side are done manually (in other words, there is no pre-generated skeleton code).

Maestro is IIOP-compliant but not fully CORBA-compliant in the following sense. CORBA specifications can be broadly divided in two parts, one dealing with application-level interfaces and the other (the IIOP part) defining data representation and message formats. The goal of Maestro has not been to follow the CORBA standard with respect to required interfaces. Rather, Maestro focuses on interoperability aspects of CORBA. Maestro itself offers the server developer a simple, general, light-weight interface which is not CORBA-compliant (and also not comprehensive). If necessary, a standard CORBA interface can be built on top of Maestro. One reason *not* to use standard CORBA interfaces is because they are unjustifiably bulky and complex yet not adequate, and adversely affect performance. With respect to interoperability, on the other hand, Maestro implements an IIOP Bridge compliant with the Version 1.0 of the IIOP protocol.

3.2 Related Work: Approaches and Tradeoffs

Several research projects, including some ongoing ones, have focused on the use of object-oriented paradigms to hide communication-level specifics of distributed applications. This makes integration easier when different interchangeable system layers must interact within an application. Interoperability is another major concern, as distributed applications often run in heterogeneous environments, with system components written in different languages and developed by different vendors. Besides protocol interoperability, standardization of interfaces at the

[1] The data representation/message format standard in the IIOP protocol is language independent.

Provided properties	Electra, Orbix+Isis	Object Group Service	Eternal	Maestro
CORBA-interoperable (via IIOP)	+	+	+	+
Based on CORBA-compliant interfaces	+	+	+	−
Clients don't need to be modified	+	+	+	+
Clients don't need to be rebuilt	−	−	−	+
Solution is OS-independent	+	+	−	+
Supports replicated clients	−	+	+	−

Figure 3.3: CORBA Reliability Solutions

application level turns out to be important in reducing maintenance costs of distributed systems deployed over a long time span. In particular, large corporate system developers are sometimes weary of depending upon proprietary interfaces provided by commercial vendors' tools and prefer to rely on industry standards. This explains, to some extent, why CORBA [OMG97] specifications have been concerned mostly with application-level interfaces, with protocol-level IIOP specifications having been added relatively recently.

Adding reliability to CORBA ORBs has been an active area of research. We believe the first CORBA-compliant reliable ORB was Electra [Maf95a,Maf95b]. It separated CORBA-specific abstractions (the ORB and the Basic Object Adaptor) from the communication substrate, which could be any group-communication system exporting a common interface. Electra was initially based on Isis and subsequently ported to run over the Horus [vRBM96] and Ensemble [Hay97] systems. With the Isis-based implementation, Electra creates a group per replicated object. Client objects join the group as Isis clients, and server objects (object implementations) join as Isis group members. In order to scale over a large number of replicated objects, multiple object groups can be multiplexed over a small number of core Isis groups [GR97,RGS+96].

The Electra approach is attractive in providing high reliability/fault tolerance guarantees to clients, in particular continuous availability of server objects with fully transparent failover. However, all client processes have to link with Electra libraries and depend on Electra daemons and services (e.g. the Electra daemon and Naming Service). Dependency on Electra libraries and processes would not normally be a problem for cluster-based distributed applications, however it would make it difficult to deploy Electra over the Internet to transparently increase avail-

ability of services accessed by remote CORBA clients. In terms of interfaces, Electra has added certain proprietary extensions to object-adaptor interfaces in order to provide functionality specific to group communication, namely methods for state transfer and membership-change notifications [Bir96], which can be overloaded by object implementations if necessary.

Two other reliable CORBA ORB implementations based on Isis are RDO/C++ [Isi94] and Orbix+Isis [LM97,II94]. Similarly to Electra, they map replicated objects to Isis groups, and provide group-communication-specific extensions to CORBA interfaces. Both client and server sides of an application need to link with ORB-specific libraries and depend on a number of daemon processes. Despite being CORBA-compliant, these three ORBs (Electra, RDO/C++ and Orbix+Isis) require a number of proprietary extensions and hooks into the system, in particular those imposing awareness of replication unto object implementations. Such considerations throw a shadow of doubt on the "inter-ORB portability of application code" argument behind CORBA standardization of application-level programming interfaces.

A CORBA reliability solution based on interception of low-level system calls has been implemented in the Eternal system [MMSN97,NMMS97b,NMMS97a]. At run time, Eternal intercepts all *read()/write()* system calls made by the UNIX process, including the calls made by the ORB to send and receive IIOP messages. The IIOP messages are recognized by their standard four-byte "magic" prefixes and are channeled through the Totem group communication system [MMSA+96,MMSA+95] to transparently replicate the ORB. All other messages are let through untouched. The Eternal approach is attractive because it can be used to add reliability to any IIOP-compliant CORBA ORB, without any application-code portability problems and related costs. Note that although replication is transparent to the ORB itself, object implementations still need to be modified to implement state transfer functionality. Also, like Electra, Eternal requires that both server *and* client sides of the application link with the Eternal/Totem library.

The Object Group Service (OGS) approach [FGS98] extends CORBA with a collection of reliability services which implement object multicast, failure detection, and other object-group functionality. The advantage of OGS is that it is CORBA-compliant and can be ported to any compatible ORB without modifications. OGS requires that both clients and servers be either compiled with the OGS library or use the OGS daemon process. However, the code of client applications does not need to be modified unless the clients want to use some advanced features, in which case they may need to make direct calls to OGS.

Eternal and OGS support replication of CORBA clients, so that replicated objects can act both as clients and as servers (at different times). This symmetric approach is more general than that adopted in Electra, Orbix+Isis, and Maestro, which assign fixed roles to objects as either clients or servers, and do not automatically handle situations such as when a replicated object needs to update a repli-

cated name server. Naive implementations with cascading replicated invocations of objects could result in substantial performance degradation and inconsistent state. However, Maestro could handle cascading-replicated-invocations scenarios by using an appropriate ORB "communication style", for example one based on the coordinator-cohort paradigm [Bir96]. In general, the Maestro approach permits a great degree of customization: although designed to support transparency, Maestro aware applications can tap into a rich collection of distributed computing tools and services to implement load-balancing schemes or other options not normally available in systems that provide only a transparent form of replication-based reliability.

There is an obvious tradeoff between degree of standardization of interfaces and their flexibility, applicability, and expressibility. In particular, CORBA application interfaces cannot naturally express nontrivial distributed-system semantics beyond the simple RPC-style client/server paradigm, where a client obtains service by sending a request message to one server object over a reliable connection and blocks until the server sends a reply. There are no provisions in CORBA interfaces as regards reliability and other "system properties". As we have seen, in order to provide fault-tolerance based on an active replication execution style, CORBA interfaces need to be augmented with non-standard extensions encapsulating at least state transfer and membership-change-notification functionality. Other programming styles, such as group communication or publish/subscribe paradigms [Bir96], can only be expressed in CORBA by resorting to alternative interfaces implemented as CORBA services, which remain proprietary until formally standardized by OMG.

Transparency is another concern. For instance, a CORBA client unaware of group communication would have to be modified or at least recompiled in order to take advantage of a standalone group communication service, which may not be feasible for Internet-based applications where the set of clients accessing a publicly advertised service can be changing in time and not known in advance.

Because of the insufficiency of CORBA interfaces, some vendors already offer server-side products which are compliant with the CORBA standard at the protocol/message format/data representation level (IIOP/GIOP), but provide proprietary extensions or alternatives to CORBA interfaces at the server side [Vit98]. The BBN QuO project [ZBS97] is exploring another dimension of the interfaces issue, with a focus on augmenting "functional" CORBA interfaces (based on IDL) with orthogonal interfaces that describe system-level properties (specifically those concerning quality-of-service and reliability) using a System Definition Language.

While standardization of inflexible heavy-weight interfaces at the application level is problematic, CORBA interfaces are in fact *understandardized* at the "bottom" level, in particular as regards request injection/dispatching functionality [Wan97]. Consequently, ORB components provided by different vendors (such as IIOP bridges or object adaptors) are generally not interchangeable.

For example, the IIOP bridge of VisiBroker [Vis97] cannot be used with Orbix [ION98] so as to receive IIOP requests and inject them into the ORB. Because of incompatibility of proprietary ORB architectures, it is hard or impossible to replace individual components of ORBs, in particular those responsible for injection of incoming IIOP messages and dispatching of client requests. By contrast, the interfaces of internal layers in Maestro interoperability tools (such as those of the IIOP bridge) are open and fully documented, so that a developer can either build applications starting from the high-level classes, or "fork off" a custom-defined class hierarchy from any layer in Maestro.

The Maestro interoperability tools differ from systems such as Electra, Eternal, or OGS by not requiring any modifications or proprietary library/daemon dependencies at the client side. This feature makes the Maestro approach suitable for Internet/Web-based distributed CORBA applications, such as online trading systems. Applications of this type are usually based on HTTP (using HTML forms) or, more recently, on Java applets running at the client side and communicating with servers via proprietary protocols. In both cases the clients are restricted to using fixed Web-browser-based visual interfaces implemented by service providers.

Switching from HTTP or proprietary protocols to IIOP would substantially increase flexibility in using internet services from clients' perspective and, in particular, would unbundle the *functionality* of those services from their client-side *interfaces*. However, the internet/web-based applications live in an "open world", where sets of clients accessing services are dynamic, geographically dispersed, and not known in advance. This makes it infeasible to require any code modifications or library/daemon dependencies at the client side in order to take advantage of increased quality of service (such as higher availability).

On the other hand, whereas heterogeneity of system environment and server implementations are common in this setting, making the use of interoperable communication protocols crucial, it is not obvious that compliance to CORBA *interfaces* will be of utmost concern to service providers. The vast amount of non-CORBA legacy systems which may need to be connected to the Internet, the added complexity and performance cost of standard CORBA interfaces, and the lack of portability of application code even between CORBA-compliant ORBs, are just several reasons which may potentially render IIOP-interoperability much more important than complete CORBA compatibility. These considerations have motivated the Maestro's focus on interoperability and complete client transparency rather than conformity with CORBA interfaces and server transparency.

The properties provided by available CORBA-reliability solutions are summarized in Figure 3.3.

3.3 Maestro IIOP Bridge and ORB Framework

Maestro implements an IIOP bridge with a framework for constructing custom ORBs over it. Although both server and client functionality of the IIOP bridge have been implemented, Maestro is mostly targeted towards the server side.

3.3.1 IIOP Bridge: Server Side

The server functionality of an IIOP bridge is implemented in Maestro with the `Maestro_IIOP_Server` class. An instance of an IIOP bridge is identified by the *hostname* of the machine it is running on, and the *port* on which it listens to incoming IIOP messages. The port to be used can be specified in the constructor. For example,

```
Maestro_IIOP_Server server(9876);
```

creates an IIOP server object bound to port 9876. The `port` argument is optional: If omitted, the IIOP server will use the value specified in the `MAESTRO_IIOP_PORT` environment variable.

The `Maestro_IIOP_Server` class exports two public downcall methods corresponding to two types of messages that can be sent by IIOP servers (`requestReply` and `locateReply`), and four protected callback methods corresponding to the four types of IIOP messages that can be received (`request`, `locateRequest`, `cancelRequest`, and `messageError`)

```
class Maestro_IIOP_Server {
 public:
    Maestro_Status requestReply(
        Maestro_CORBA_ULong request_id,
        Maestro_GIOP_ReplyStatusType reply_status,
        Maestro_CORBA_Message &reply_body,
        Maestro_IIOP_ConnId cid);

    Maestro_Status locateReply(
        Maestro_CORBA_ULong request_id,
        Maestro_GIOP_LocateStatusType locate_status,
        Maestro_IIOP_ConnId cid);
protected:
    void request_Callback(
        Maestro_CORBA_ULong request_id,
        Maestro_CORBA_Boolean response_expected,
        Maestro_CORBA_OctetSequence &object_key,
```

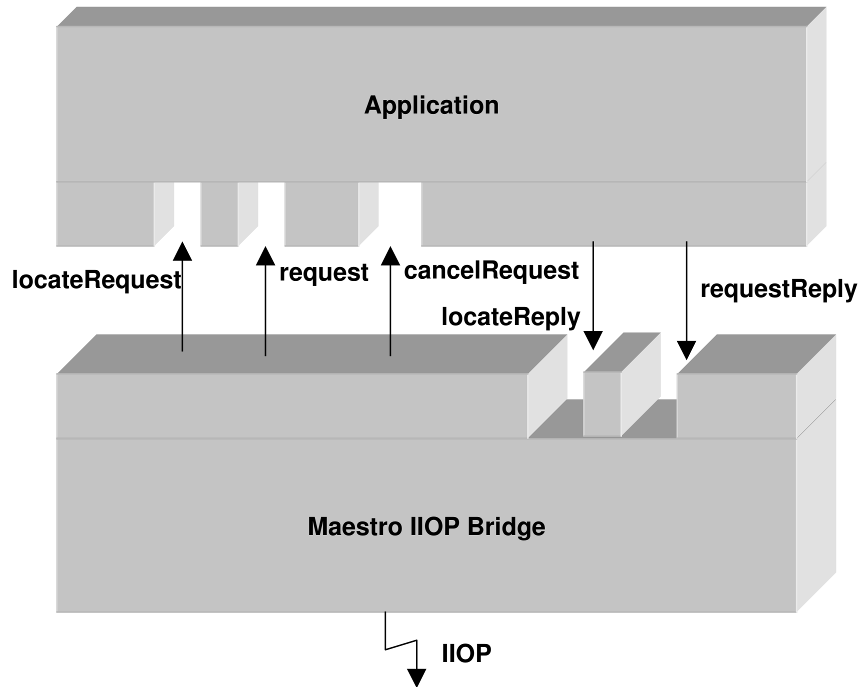

Figure 3.4: Maestro IIOP Bridge: The Server-Side Interface

```
        Maestro_CORBA_String &operation,
        Maestro_GIOP_Principal &requesting_principal,
        Maestro_CORBA_Message &msg,
        Maestro_IIOP_ConnId cid) {}

    void locateRequest_Callback(
        Maestro_CORBA_ULong request_id,
        Maestro_CORBA_OctetSequence &object_key,
        Maestro_IIOP_ConnId cid) {}

    void cancelRequest_Callback(
        Maestro_CORBA_ULong request_id,
        Maestro_IIOP_ConnId cid) {}

    void messageError_Callback(Maestro_IIOP_ConnId cid) {}
}
```

The default implementation of IIOP callbacks is no-op. An application will typically define a subclass of `Maestro_IIOP_Server` which will overload the (vir-

tual) protected callback methods so as to implement required functionality (see Figure 3.4).

An IIOP connection can be closed (for example, in response to a `messageError` callback) with a call to `closeConnection(cid)`, which is another public method of `Maestro_IIOP_Server`.

There are several options for dispatching incoming IIOP messages via `Maestro_IIOP_Server`'s callbacks. A new thread can be allocated for each callback, or the callback functions can be invoked directly by the IIOP bridge's dispatcher thread. Maestro can support both options. However, by default, IIOP callbacks are invoked directly and therefore must not block. In particular, if blocking during processing of requests is possible, the callback functions (overloaded by the application) must pass incoming requests to dedicated processing threads, and return without blocking.

3.3.2 IIOP Bridge: Client Side

The client side of an IIOP bridge is implemented with the `Maestro_IIOP_Client` class. Similarly to `Maestro_IIOP_Server`, this class exports public downcall methods corresponding to three types of messages sent by IIOP clients (`request`, `locateRequest`, and `cancelRequest`) and protected callback methods matching the four types of IIOP messages that can be received (`reply`, `locateReply`, `closeConnection`, and `messageError`).

3.3.3 Object Keys

The IIOP protocol uses *object keys* to identify objects targeted by different types of messages. Maestro implements object keys with the `Maestro_ORB_ObjectKey` class. `Maestro_ORB_ObjectKey` supports key initialization from predefined strings and octet sequences, for example:

```
// Create an object key containing the string "my_key".
Maestro_CORBA_String str(``my_key'');
Maestro_ORB_ObjectKey key(str);
```

`Maestro_ORB_ObjectKey` can also be used to generate new globally unique object keys. In particular, the keys assigned to objects accessed through Maestro-based ORBs should be generated with `Maestro_ORB_ObjectKey`, for example:

```
Maestro_ORB_ObjectKey key;// Create an uninitialized object key.
key.init(``Grid'');// Initialize as a new globally unique object key.
```

The keys generated with `Maestro_ORB_ObjectKey::init()` are assigned random 32-bit hash values. The hash value of an object key can be retrieved with the `hash()` method:

```
Maestro_CORBA_Long hash_value = key.hash();
```

Finally, Maestro provides a utility called `newkey` which can be used to generate new globally unique object keys in the string format, for example (to create a new object key for the `Grid` application):

```
taurus% newkey Grid
MAE:074b8d6ea791947c34da7cbd0007d5aeef:Grid
```

3.3.4 Building ORBs over the Maestro IIOP Bridge

In this section we discuss base interfaces which can be used to build custom ORBs over the Maestro IIOP Bridge. We then describe in more detail the implementation of a simple Maestro ORB. Throughout this section we will be concerned exclusively with the server side; the client side can be implemented with any available CORBA-compliant ORB, such as Orbix.

Base Interfaces

Maestro implements the server-side IIOP functionality with the `Maestro_IIOP_Server` class, as discussed in Section 3.3.1. `Maestro_IIOP_Server` defines methods corresponding to both outgoing and incoming IIOP messages. The former are defined as public downcalls, and the latter are protected callback methods with no-op bodies. We want to stress that the essential IIOP-bridge functionality is implemented with the downcalls, and the callback methods are provided only as hooks for the application. Although an ORB can be implemented by overloading the callbacks directly, it is more convenient (and allows for greater flexibility) to separate IIOP downcalls and callbacks into two different interfaces. This is done in Maestro with the `Maestro_ORB_IIOPDispatcher` and `Maestro_ORB_Base` classes.

`Maestro_ORB_IIOPDispatcher` is a subclass of `Maestro_IIOP_Server`. It defines public downcall methods corresponding to outgoing IIOP messages, namely `requestReply` and `locateReply`:

```
Maestro_Status requestReply(
    Maestro_ORB_RequestId &reqId,
    Maestro_GIOP_ReplyStatusType reply_status,
    Maestro_CORBA_Message &reply_body);

Maestro_Status locateReply(
    Maestro_ORB_RequestId &reqId,
    Maestro_GIOP_LocateStatusType locate_status);
```

The `Maestro_ORB_Base` class defines public callback methods corresponding to incoming messages of types `request`, `locateRequest`, and `cancelRequest` [2]:

[2] See section 3.3.1 for more details

```
void request_Callback(
    Maestro_ORB_RequestId &reqId,
    Maestro_ORB_ObjectKey &objKey,
    Maestro_CORBA_Boolean response_expected,
    Maestro_CORBA_String &operation,
    Maestro_GIOP_Principal &requesting_principal,
    Maestro_CORBA_Message &msg) {}

void locateRequest_Callback(
    Maestro_ORB_RequestId &reqId,
    Maestro_ORB_ObjectKey &objKey) {}

void cancelRequest_Callback(Maestro_ORB_RequestId &reqId) {}
```

Besides the IIOP downcalls, the `Maestro_ORB_IIOPDispatcher` class provides two other public methods, `bind()` and `unbind()`. These can be used to attach/detach *(object key, ORB)* pairs to/from an IIOP Dispatcher:

```
Maestro_Status bind(Maestro_ORB_ObjectKey &objKey, Maestro_ORB_Base *orb);
 Maestro_Status unbind(Maestro_ORB_ObjectKey &objKey);
```

An *(object key, ORB)* mapping specifies the ORB assigned to handle requests targeted for the object with the given key. When an IIOP message is received, the Dispatcher looks up the ORB based on the object key included in the message header and, if a mapping exists, invokes the corresponding callback method on the ORB. It is up to the ORB to process the request and, if necessary, send a reply to the client (via IIOP Dispatcher's public downcall methods). If an *(object key, ORB)* mapping for the key specified in an incoming IIOP message is not found, the Dispatcher will send a reply message with an exception status to the client.

In order to build a custom Object Request Broker over the Maestro IIOP Bridge, all that is needed is to define a subclass of `Maestro_ORB_Base` and overload the three IIOP callback methods (`request_Callback`, `locateRequest_Callback`, and `cancelRequest_Callback`) so as to implement the required request-processing functionality. We show an example in the next section.

Example: A Simple ORB

In this section we look at the implementation of a simple Object Request Broker called Simple ORB.

Objects accessed through the Simple ORB must be implemented as subclasses of the
`Maestro_SimpleORBObjectAdaptor` abstract base class.
`Maestro_SimpleORBObjectAdaptor` declares the following generic `update()` operation, which must be overloaded by all object implementations:

```
Maestro_GIOP_ReplyStatusType update(
    Maestro_CORBA_String &operation,
    Maestro_CORBA_Message &request,
    Maestro_CORBA_Message &reply);
```

The `operation` argument in the `update()` function is the name of the requested operation, as specified in the IIOP request message. The `request` parameter is the body of the IIOP request message, containing the marshaled-in `in` and `inout` parameters of the operation. When a call to the `update()` function returns, the `reply` parameter should contain the body of the IIOP reply message, containing the marshaled-in return value and `inout`/`out` parameters of the operation.

The encoding of parameters in IIOP request/reply messages is specified in the CORBA standard [3]. Namely, the request body includes all `in` and `inout` parameters in the order in which they are specified in the operation's IDL definition, from left to right. If the value of reply status is `MAESTRO_GIOP_REPLY_STATUS_NO_EXCEPTION`, the reply body includes the operation's return value (if any) followed by all `inout` and `out` parameters in the order in which they appear in the operation's IDL definition, from left to right. If the value of reply status is `MAESTRO_GIOP_REPLY_STATUS_USER_EXCEPTION` or `MAESTRO_GIOP_REPLY_STATUS_SYSTEM_EXCEPTION`, then the reply body contains the marshaled exception structure. Refer to the CORBA specification for more details.

The `Maestro_GIOP_ReplyStatusType` type specifies the range of valid return values:

```
typedef enum {
    MAESTRO_GIOP_REPLY_STATUS_NO_EXCEPTION,
    MAESTRO_GIOP_REPLY_STATUS_USER_EXCEPTION,
    MAESTRO_GIOP_REPLY_STATUS_SYSTEM_EXCEPTION,
    MAESTRO_GIOP_REPLY_STATUS_LOCATION_FORWARD
} Maestro_GIOP_ReplyStatusType;
```

An object may implement an IDL interface with multiple functions, in which case it will be necessary to marshal/unmarshal parameters and do the processing individually for all supported operations [4].

The Simple ORB itself is implemented with the `Maestro_ES_SimpleORB` class. `Maestro_ES_SimpleORB` defines public methods `bind()` and `unbind()` which can be used by objects/object adaptors to attach/detach themselves to/from the ORB. After all objects have been initialized and attached to the ORB, the ORB can be activated with the `activate()` method. The `deactivate()` method can be used to disable the ORB at any time [5].

[3] CORBA 2.0 Specification, pp. 12-18/19.

[4] See Section 3.5.1 for an example of an object/object adaptor implementation.

[5] See Section 3.5.2 for an example of object/ORB initialization code.

As a subclass of `Maestro_ORB_Base`, `Maestro_ES_SimpleORB` implements two IIOP callback functions, `request_Callback()` and `locateRequest_Callback`. These functions are invoked by the IIOP bridge/dispatcher to pass the incoming IIOP requests to the ORB. The implementation of `request_Callback()` and `locateRequest_Callback` methods defines the request-processing functionality of an ORB. For example, the Simple ORB implements `request_Callback()` as follows:

```
// request_Callback: Process an incoming IIOP request message.
// This method is invoked by the IIOP Bridge (Dispatcher).
void Maestro_ES_SimpleORB::request_Callback(
    Maestro_ORB_RequestId &reqId,
    Maestro_ORB_ObjectKey &objKey,
    Maestro_CORBA_Boolean response_expected,
    Maestro_CORBA_String &operation,
    Maestro_GIOP_Principal &requesting_principal,
    Maestro_CORBA_Message &reqBody
) {
    mutex.lock();

    // Return an exception if the ORB is not activated.
    if (!active) {
        mutex.unlock();
        if (!response_expected) {
            return;
        }
        Maestro_CORBA_String exc_name(``SystemException'');
        Maestro_CORBA_Exception exc(
            exc_name,
            MAESTRO_CORBA_EXCEPTION_CODE_INV_OBJREF,
            MAESTRO_CORBA_COMPLETION_STATUS_NO);
        Maestro_CORBA_Message reply;
        reply << exc;
        // Use IIOP Dispatcher's requestReply downcall to send the IIOP reply.
        dispatcher->requestReply(
            reqId,
            MAESTRO_GIOP_REPLY_STATUS_SYSTEM_EXCEPTION,
            reply);
        return;
    }

    // Lookup the target object (object adaptor) based on the key.
    Maestro_SimpleORBObjectAdaptor *obj = lookup(objKey);

    // Return an exception if the object is not currently bound to the ORB.
```

```
    if (obj == NULL) {
        mutex.unlock();
        if (!response_expected) {
            return;
        }
        Maestro_CORBA_String exc_name(``SystemException'');
        Maestro_CORBA_Exception exc(
            exc_name,
            MAESTRO_CORBA_EXCEPTION_CODE_INV_OBJREF,
            MAESTRO_CORBA_COMPLETION_STATUS_NO);
        Maestro_CORBA_Message reply;
        reply << exc;
        dispatcher->requestReply(
            reqId,
            MAESTRO_GIOP_REPLY_STATUS_SYSTEM_EXCEPTION,
            reply);
        return;
    }

    // Invoke object's update() method directly.
    // Upon return, replyBody should contain the body of the IIOP reply message.
    Maestro_CORBA_Message replyBody;
    Maestro_GIOP_ReplyStatusType status =
        obj->update(operation, reqBody, replyBody);

    // Use IIOP Dispatcher's requestReply downcall to send the IIOP reply.
    if (response_expected) {
        dispatcher->requestReply(reqId, status, replyBody);
    }
    mutex.unlock();
}
```

3.4 Reliable Maestro ORB (Replicated Updates)

Maestro tools include a reliable Object Request Broker called Replicated Updates ORB. The ORB provides higher availability of server objects by *actively replicating* them over several processes. In the following sections we will discuss the interfaces of the Replicated Updates ORB and important implementation details.

It is important to stress that Replicated Updates is not a fully CORBA-compliant ORB. In particular, it does not implement standard CORBA *interfaces* and *services*, however it does support *object interoperability* via IIOP. A full CORBA ORB, complete with CORBA interfaces, services and an IDL compiler,

can be built on top of Replicated Updates, using Maestro as a middle layer.

3.4.1 ORB Interfaces

The Replicated Updates ORB is implemented with the `Maestro_ES_ReplicatedUpdates` class. Objects accessed through the ORB must be implemented as subclasses of the `Maestro_RUObjectAdaptor` class, which is defined with the following callback methods:

```
class Maestro_RUObjectAdaptor {
public:
    Maestro_GIOP_ReplyStatusType update(
        Maestro_CORBA_String &operation,
        Maestro_CORBA_Message &request,
        Maestro_CORBA_Message &reply) = 0;

    void pushState(Maestro_CORBA_Message &msg) {}
    void getState(Maestro_CORBA_Message &msg) {}
};
```

The `update()` method is a generic IIOP callback, which is invoked by the ORB when the object receives a request message from a CORBA/IIOP client. Objects must overload the `update()` method to implement their request-processing functionality. As with the Simple ORB (Section 3.3.4), the `operation` parameter identifies the name of the IDL function to be invoked on the object, and `request` contains the marshaled `in` and `inout` parameters. When a call to `update()` returns, the `reply` argument should contain the marshaled return value and `inout`/`out` parameters of the invoked operation. See Section 3.3.4 for details.

The `pushState` callback should be overloaded to marshal the object's application state into the `msg` argument. Similarly, in the `getState` callback, the object should retrieve the application state contained in the `msg` argument and update its local state accordingly. The `pushState` and `getState` callbacks are invoked by the Replicated Updates ORB during *state transfer* to bring the state of new instances of an object up to date with the states of previously created object replicas, or to reconcile the states of replicas remerging together after a network partition.

Instances of the `Maestro_RUObjectAdaptor` class are initialized with the following constructor:

```
Maestro_RUObjectAdaptor(Maestro_RUObjectAdaptor_Options &ops);
```

with the `Maestro_RUObjectAdaptor_Options` structure defined as follows:

```
struct Maestro_RUObjectAdaptor_Options {
    Maestro_ORB_ObjectKey key;
    Maestro_ES_ReplicatedUpdates *orb;
};
```

The `key` identifies the object (this is the `object_key` specified in IIOP messages); the `orb` parameter specifies the ORB through which the object should be accessed (the object adaptor will automatically attach to the ORB).

The initialization options for the Replicated Updates ORB, and its public downcalls including the constructor, are defined as follows:

```
struct Maestro_ReplicatedUpdates_Options {
    // The IIOP Bridge/Dispatcher through which this ORB should be accessed.
    Maestro_ORB_IIOPDispatcher *dispatcher;

    // Set if updates are allowed only in the primary (quorum) component.
    int progressInPrimaryOnly;

    // Number of replicas for objects accessed through this ORB.
    // The value of nReplicas determines the quorum size (= majority of replicas).
    int nReplicas;

    // Set if state transfer is required.
    // If stateTransfer flag is not set, the getState()
    // and pushState() callbacks will not be invoked.
    int stateTransfer;

    // Set if total ordering of request invocations
    // between object replicas is not required.
    // This flag should be reset if all replicas must process
    // all operations in the same order.
    int requestsCommute;

    // Set if compound IORs of all bound objects should be reinstalled
    // in the etc directory after membership changes.
    int reinstallIOR;

    // The Maestro_Etc object which should be used to install object IORs.
    Maestro_Etc *etc;

    // The name of the ORB group.
    Maestro_String ORBName;
};

class Maestro_ES_ReplicatedUpdates {
public:
    Maestro_ES_ReplicatedUpdates(Maestro_ReplicatedUpdates_Options &ops);

    // Activate the ORB: Join the ORB group and
    // start accepting IIOP requests to bound objects.
```

```
    // No new objects will be allowed to bind while the ORB is active.
    void activate();

    // Deactivate the ORB: Stop accepting IIOP requests; leave the ORB group.
    void deactivate();

    // Bind an object to the ORB.
    // Objects can only be bound while the ORB is not active.
    // 'key' identifies the object; 'obj' points to the object instance.
    Maestro_Status bind(Maestro_ORB_ObjectKey &key,
                        Maestro_RUObjectAdaptor *obj);

    // Unbind an object from the ORB. Objects can only be unbound
    // while the ORB is not active.
    Maestro_Status unbind(Maestro_ORB_ObjectKey &key);

    // Unbind all objects from the ORB. Objects can only be unbound
    // while the ORB is not active.
    Maestro_Status unbindAll();
};
```

In a typical initialization sequence, objects will be initialized first and bound to an ORB instance with the `bind()` method. After all objects have been bound, the ORB will be enabled with a call to `activate()`. See Section 3.5.2 for an example.

3.4.2 A Look Inside

With the Replicated Updates ORB, multiple replicas of server objects can be created at different processes and bound to local ORB instances, which join together to form a *group* over the Ensemble or Horus system. The processing of incoming IIOP requests is performed as follows (see Figure 3.5). When a Maestro IIOP Bridge/Dispatcher receives a client's request message (1), it invokes the corresponding callback method on the target object's ORB to pass it the message. The ORB then relays/multicasts the message to the entire ORB group (2,3). After an ORB instance receives a relayed message (4), it dispatches the request to the local target object attached to it (5). The replies (6) are suppressed at all ORBs except the one which got the original IIOP request. This ORB sends the reply message back to the client (7).

Observe that when requests are relayed with totally ordered multicasts, they are delivered to all objects in the same order, so that all object replicas perform the same sequence of operations (the *active replication* model [Bir96]).

The architecture of the Replicated Updates ORB scales well in the number of objects residing within a process, since all objects bound to an ORB instance are

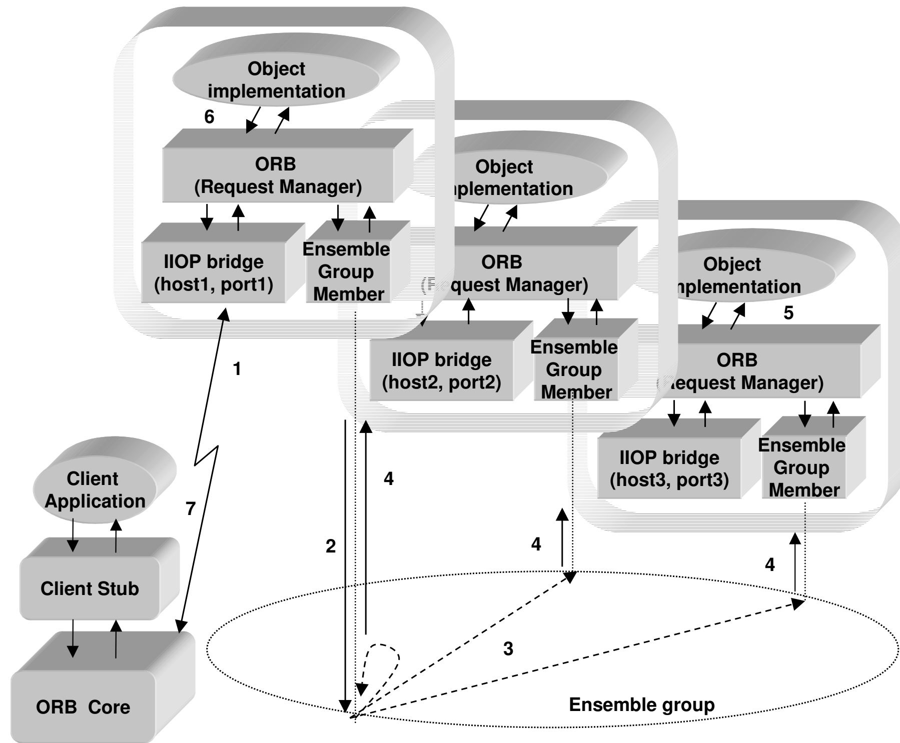

Figure 3.5: Maestro Replicated Updates ORB

multiplexed over a single Ensemble (or Horus) group. This effectively provides light-weight group semantics for free. Maestro also implements inter-object consistency guarantees on group joins/leaves and state transfer: All objects attached to a Replicated Updates ORB join or leave the system and do state transfer in one atomic step. In particular, a single state transfer protocol is run for all objects bound to an instance of the ORB. States of all local object replicas are packed in one message and sent in one step.

Whenever new ORB instances along with attached object replicas join or leave the system (as mapped to membership changes in the corresponding ORB group), the updated IOR's for all bound replicated objects are installed in the `etc` directory or published elsewhere. The IOR's generated by the Maestro Replicated Updates ORB are *compound*: They contain IIOP profiles for all object replicas (see Figure 3.6). The IOR's are published in the standard CORBA format and can be accessed from any CORBA-compliant ORBs. The client side of a CORBA application can bind to any one of the object replicas included in the target object's compound IOR. See Section 3.4.3 for details.

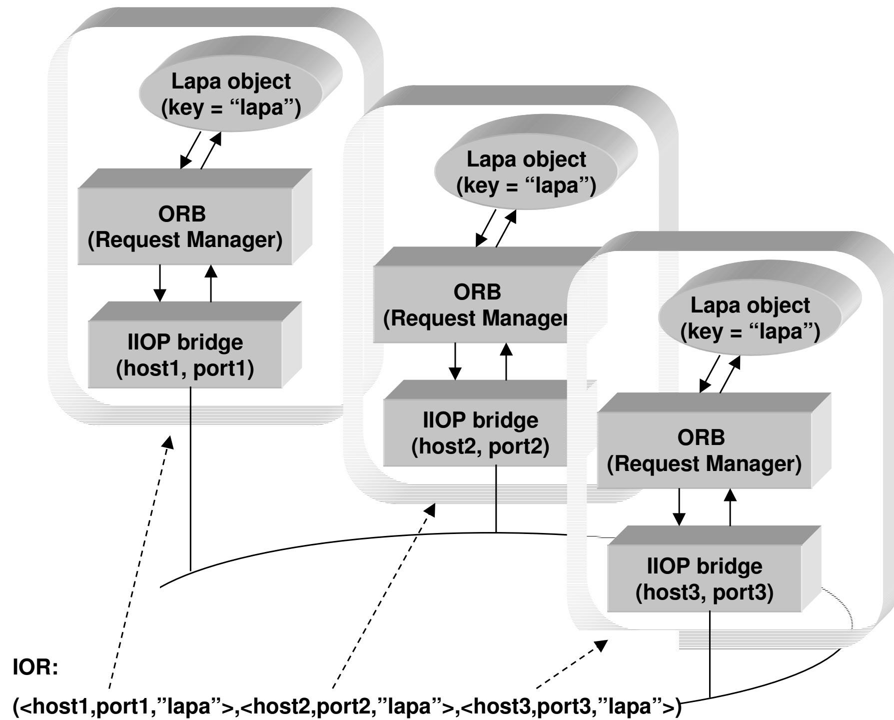

Figure 3.6: Compound Interoperable Object References in Maestro: An IOR may contain multiple IIOP profiles pointing to different copies of the replicated object residing at different processes

3.4.3 Client Perspective: Failover/Transparency Issues

The CORBA/IIOP standard specifies that if a client's connection with the server breaks abnormally, the client should treat the condition as an error and report communication-failure exceptions for all pending requests on the connection [6]. This requirement limits possibilities for transparent failover in cases when an object is implemented with several replicas and the replica to which the client is connected happens to fail. Even without that limitation, the available commercial ORBs (such as Orbix) simply do not expect object implementations to be replicated and therefore do not have built-in mechanisms for server failover in cases of process crashes or link failures. For example, even if Maestro advertises a replicated object by publishing a compound IOR containing an IIOP profile for each object replica, the client running on Orbix will always use only one profile in an IOR (the last profile with Orbix 2.2) and ignore the rest. In case of a failure of an object replica, the standard Orbix client will have to wait until Maestro reconfigures the object

[6]CORBA 2.0 Specifications, p. 12-25.

group and publishes the updated IOR, at which point the client can look up the new IOR and reconnect to another object replica. Although there is no *transparency of failover* here, the client still benefits from *higher availability* of the service, including state consistency guarantees for object replicas provided by active replication.

As concerns transparency of failover, the quality of service provided to CORBA/IIOP clients obviously depends on clients' awareness of object replication. In order to fully exploit the potential of replication, the ORBs at the client side will have to be capable of using multiple profiles in compound IORs so as to hide server failures and reconnect to available object replicas transparently from the application.

3.5 Building Object Adaptors and Applications with Maestro

3.5.1 Implementing Objects/Object Adaptors

In this section we give an example of a simple IIOP-interoperable application built directly over Maestro using the Replicated Updates ORB. The server object implements the following CORBA IDL interface:

```
interface Grid {
    readonly attribute short height;
    readonly attribute short width;

    void set(in short n, in short m, in long value);
    long get(in short n, in short m);
};
```

The client side of the `Grid` application can be implemented in any CORBA-compliant ORB, e.g. Orbix (see Section 3.5.3 for Java and C++ examples).

The server object is implemented in Maestro as a subclass of the `Maestro_RUObjectAdaptor` class, which declares generic `update` and state-transfer callback methods to be overloaded by the application. The `update` callback is invoked by the ORB when an IIOP request from a client is received. Parameters of the request are extracted from the message according to the specified operation name. After performing the operation, the `out/inout` parameters and the return value (if any) are marshaled into the reply message, which is then sent by the ORB back to the client. The structure of the object implementation in the example below is similar to the Dynamic Skeleton Interface approach of CORBA. The marshaling and dispatching code can be written manually or, alternatively, generated with an IDL compiler (not included with the current version of Maestro).

The `Grid` server objects are implemented as follows:

```
class Grid: public Maestro_RUObjectAdaptor {
    Grid(Maestro_RUObjectAdaptor_Options &ops):
        Maestro_RUObjectAdaptor(ops)
    {
        // Initialize the 10x10 grid (implemented as a 100-element array).
        grid = new Maestro_CORBA_Long[100];
        memset(grid, 0, sizeof(Maestro_CORBA_Long) * 100);

        // Operations supported by the Grid interface.
        height = ``_get_height'';
        width = ``_get_width'';
        set = ``set'';
        get = ``get'';
    }

    ~Grid() { delete [] grid; }

    // Generic request handler:
    //     Dispatch the request based on the operation name;
    //     Extract the parameters;
    //     Perform the operation;
    //     Setup the reply message.
    Maestro_GIOP_ReplyStatusType update(
        Maestro_CORBA_String &operation,
        Maestro_CORBA_Message &request,
        Maestro_CORBA_Message &reply)
    {
        // Perform the "_get_height" operation.
        if (operation == height) {
            Maestro_CORBA_Short result = 10;
            reply << result;
            return MAESTRO_GIOP_REPLY_STATUS_NO_EXCEPTION;
        }
        // Perform the "_get_width" operation.
        else if (operation == width) {
            Maestro_CORBA_Short result = 10;
            reply << result;
            return MAESTRO_GIOP_REPLY_STATUS_NO_EXCEPTION;
        }
        // Perform the "set" operation.
        else if (operation == set) {
            Maestro_CORBA_Long value;
            Maestro_CORBA_Short n, m;

            // Unmarshal the in and inout parameters, from left to right.
```

```
            request >> n >> m >> value;

            // Update the grid.
            grid[10*n + m] = value;
            return MAESTRO_GIOP_REPLY_STATUS_NO_EXCEPTION;
        }
        // Perform the "get" operation.
        else if (operation == get) {
            Maestro_CORBA_Long value;
            Maestro_CORBA_Short n, m;

            // Unmarshal the in and inout parameters, from left to right.
            request >> n >> m;
            value = grid[10*n + m];

            // Marshal the return value and inout/out parameters into the reply.
            reply << value;
            return MAESTRO_GIOP_REPLY_STATUS_NO_EXCEPTION;
        // Unsupported operation – return an exception.
        else {
            Maestro_CORBA_String exc_name(``SystemException'');
            Maestro_CORBA_Exception exc(
                exc_name,
                MAESTRO_CORBA_EXCEPTION_CODE_BAD_OPERATION,
                MAESTRO_CORBA_COMPLETION_STATUS_NO);
            reply << exc;
            return MAESTRO_GIOP_REPLY_STATUS_SYSTEM_EXCEPTION;
        }
    }

    // Write object's state (the grid) into the msg.
    void pushState(Maestro_CORBA_Message &msg) {
        msg.write(grid, sizeof(Maestro_CORBA_Long) * 100);
    }

    // Read/update object's state (the grid) from the msg.
    void getState(Maestro_CORBA_Message &msg) {
        msg.read(grid, sizeof(Maestro_CORBA_Long) * 100);
    }

    Maestro_CORBA_Long *grid;
    Maestro_CORBA_String height, width, set, get;
};
```

Note that when using Maestro (at least, when using it in the manner of this ex-

ample), a server object does not need to be aware of reliability issues. For example, specific replication policies of the ORB can be varied while keeping the server implementation unchanged. The Replicated Updates ORB only requires that objects implement state marshaling/unmarshaling routines (`getState` and `pushState`), since the contents of the replicated state is in general application-dependent. All other aspects of object replication, including object-group membership protocol, message ordering, and state-transfer protocol, are transparent to object implementations.

3.5.2 System Configuration/Initialization

In this section we will discuss the initialization sequence for the server side of a Maestro application using the Replicated Updates ORB. The `Grid` application will serve as an example.

The `main()` function of the `Grid` server application initializes a Replicated Updates ORB, creates a `Grid` object and binds it to the ORB, and finally activates the ORB. If needed, multiple server objects could be bound to one ORB replica. The initialization code at the server side is as follows:

```
// Server side of the Grid application.
main(int argc, char *argv[]) {
    // Create an IIOP Dispatcher.
    Maestro_IIOPDispatcher dispatcher;

    // Setup ORB configuration options.
    Maestro_ReplicatedUpdates_Options orb_ops;
    orb_ops.dispatcher = &dispatcher;
    orb_ops.progressInPrimaryOnly = TRUE;
    orb_ops.nReplicas = 5;
    orb_ops.stateTransfer = TRUE;
    orb_ops.requestsCommute = FALSE;
    orb_ops.reinstallIOR = TRUE;
    orb_ops.etc = &Maestro_DefaultEtc;
    orb_ops.ORBName = ``Grid'';

    // Create a Replicated Updates ORB.
    Maestro_ES_ReplicatedUpdates orb(orb_ops);

    // Create a Grid object
    // (it automatically binds to the specified ORB).
    Maestro_ReplicatedUpdates_Options obj_ops;
    Maestro_CORBA_String keyStr(
        ``MAE:3f809d92a791947c346778da00069523b3:Grid'');
    Maestro_ORB_ObjectKey key(keyStr);
```

```
    obj_ops.key = key;
    obj_ops.orb = &orb;
    Grid obj(obj_ops);

    // Activate the ORB.
    orb.activate();

    // Block the main thread.
    Maestro_Semaphore sema;
    sema.dec();
}
```

The `Grid` server developed in the above example is interoperable, via IIOP, with all CORBA-compliant client applications.

3.5.3 Setting Up the Client Side

The client side of an application which accesses objects running on Maestro via IIOP can be implemented with any CORBA/IIOP-compliant ORB, such as Orbix. Since the data representation/message format specifications of IIOP are language-independent, the ORB and the client-side application can be implemented in any language for which CORBA type mapping is defined. In particular, we will show two versions of a client applications written in Java and C++.

The client application is simple. First, it reads-in the IOR file for the server object. The file should be created by the ORB at the server side. The Maestro Simple ORB and Replicated Updates ORB use the following convention for IOR file naming: The IOR for an object which implements interface `Foo` is placed in file `Foo.ior` in the `etc` directory specified by the `MAESTRO_ETC` environment variable.

After the client retrieves a stringified IOR, it converts the IOR into an object reference and invokes remote operations on the object.

In the following example, it is assumed that the server object implements the `Grid` IDL interface as defined in Section 3.5.1, and that the size of the grid is 10.

The client application can be built and executed with commercially available ORBs such as OrbixWeb 3.0 (the Java version) or Orbix 2.2 (the C++ version). The Java and C++ versions of the client code are shown below. The code is a modified version of the `Grid` demo included with Orbix/OrbixWeb distribution.

Client code in Java:

```
// Modified file javaclient1.java from the grid demo of OrbixWeb 3.0.
package gridtest;

import java.io.*;
import org.omg.CORBA.ORB;
import IE.Iona.OrbixWeb._CORBA;
```

```
public class javaclient {
    public static void main(String args[]) {
        if (args.length == 0) {
            System.out.println(``Usage:  javaclient IOR-file-name'');
            System.exit(1);
        }

        // Initialize the ORB. Specify that IIOP protocol should be used for binding.
        ORB orb = ORB.init();
        _CORBA.Orbix.setConfigItem (``IT_BIND_USING_IIOP'',
                        String.valueOf(true));

        // Read-in the IOR.
        FileInputStream f;
        byte buf[] = new byte[1024];
        try {
            f = new FileInputStream(args[0]);
            f.read(buf);
        }
        catch (FileNotFoundException e) {
            System.out.println(``Could not open IOR file '' + args[0]);
            System.exit(1);
        }
        catch (IOException e) {
            System.out.println(``Error reading IOR file '' + args[0]);
            System.exit(1);
        }

        // Get the object reference and narrow to the Grid type.
        String ior = new String(buf);
        org.omg.CORBA.Object objRef = orb.string_to_object(ior);

        if (objRef == null) {
            System.out.println(``objRef is null'');
            System.exit(1);
        }
        grid p = gridHelper.narrow(objRef);
        if (p == null) {
            System.out.println(``p is null'');
            System.exit(1);
        }

        // Get the height and width of the remote grid object.
        short w, h;
```

```
        try {
            w = p.width();
            h = p.height();
        } catch (org.omg.CORBA.SystemException ex) {
            System.out.println(``FAIL: Exception during width,height'');
            System.out.println(ex.toString());
            return;
        }
        if (w != 10 || h != 10) {
            System.out.println(``FAIL: width is '' + w +
                               ``, height is '' + h);
            return;
        }
        System.out.println(``PASS: width is '' + w + ``, height is '' + h);

        // Set a value in the remote grid and check correctness.
        int val = 0;
        try {
            p.set((short)2, (short)4, 123);
            p.set((short)0, (short)0, 0);
            val = p.get((short)2, (short)4);
        } catch (org.omg.CORBA.SystemException ex) {
            System.out.println(``FAIL: Exception during set,get'');
            System.out.println(ex.toString());
            System.out.println(``Minor='' + ex.minor);
            System.exit(1);
        }
        if (val != 123) {
            System.out.println(
                ``FAIL: bad value returned after set,get '' + val);
            return;
        }
        System.out.println(``PASS: grid[2,4] is '' + val);
        }
    }
}
```

Client code in C++:

```
// Modified file iiopcli.cc from the grid_iiop demo of Orbix 2.2.
 #define USE_IIOP

#include ``CORBA.h''
#include ``grid.hh''
#include <sys/types.h>
#include <sys/time.h>
```

```
#include <iostream.h>
#include <stdio.h>
#include <fstream.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[]) {
    CORBA::Object_ptr objPtr;
    grid_ptr gridPtr;
    if (argc < 2) {
        cout << ‘‘Usage:  client IOR-file-name’’);
        exit(1);
    }

    // Read-in the IOR.
    ifstream iorFile(argv[1]);
    if (!iorFile) {
        cout << ‘‘client:  Unable to open IOR file ’’ << iorFile << endl;
        exit(1);
    }
    char refStr[500];
    iorFile >> refStr;

    // Get the object reference and narrow to the Grid type.
    try {
        objPtr = CORBA::Orbix.string_to_object(refStr);
    } catch (CORBA::SystemException& se) {
        cerr << ‘‘Exception:  ’’  << &se << endl;
        exit(1);
    }
    if (CORBA::is_nil(objPtr)) {
        cerr << ‘‘objPtr is null’’ << endl;
        exit(1);
    }
    try {
        gridPtr = grid::_narrow(objPtr);
    } catch (CORBA::SystemException& se) {
        cout << ‘‘exception trying to _narrow:  ’’  << &se << endl;
    }
    if (CORBA::is_nil(gridPtr)) {
        cerr << ‘‘gridPtr is null’’ << endl;
        exit(-1);
    }

    // Get the height and width of the remote grid object.
```

```
    CORBA::Short h, w;
    try {
        h = gridPtr->height();
        w = gridPtr->width();
    } catch (CORBA::SystemException &sysEx) {
        cerr << ``Unexpected system exception:  ''  << &sysEx << endl;
        exit(1);
    } catch(...)  {
        cerr << ``Unexpected exception'' << endl;
        exit(1);
    }
    if (h != 10 || w != 10) {
        cout << ``FAIL: width is '' << w << ``, height is '' << h << endl;
        return;
    }
    cout << ``PASS: width is '' << w << ``, height is '' << h << endl;

    // Set a value in the remote grid and check correctness.
    CORBA::Long val;
    try {
        gridPtr->set(2, 4, 123);
        val = gridPtr->get(2, 4);
    } catch (CORBA::SystemException &sysEx) {
        cerr << ``Unexpected system exception:  ''  << &sysEx << endl;
        exit(1);
    } catch(...)  {
        cerr << ``Unexpected exception'' << endl;
        exit(1);
    }
    if (val != 123) {
        cout << ``FAIL: bad value returned after set,get:  ''
                << val << endl;
        return;
    }
    cout << ``PASS: grid[2,4] is '' << val << endl;

    return 0;
}
```

3.6 Performance

We have measured performance numbers for synchronous remote method invocations with Orbix 2.2 and the Maestro Replicated Updates ORB running over Horus.

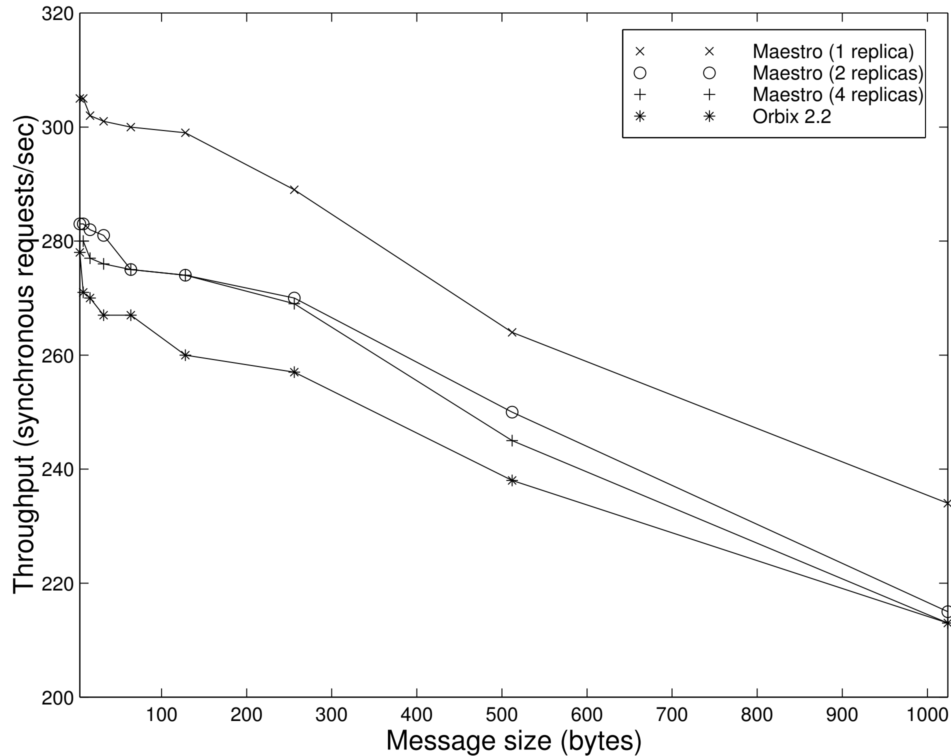

Figure 3.7: Performance of the Maestro Replicated Updates ORB (over Horus) and Orbix 2.2

The same Orbix-based CORBA client was used to issue series of IIOP requests to the server object built with either Orbix or Maestro. The test application was based on the following IDL interface:

```
interface PerformanceTest {
    attribute string value;
};
```

The `PerformanceTest` interface maps to two synchronous operations, `void _set_value(char*)` and `char* _get_value(void)`, which respectively modify or retrieve the value of the string maintained by the object implementation.

The client side of the performance-test application was making a series of 1000 invocations of the `_set_value` operation on a remote object. We computed the total number of requests completed per second, and took the average value after five runs of the application. The tests were performed for strings of different lengths, corresponding to application-level messages of sizes 4, 8, 16, 32, 64, 128, 256, 512, and 1024 bytes. The client side of the application was the same (Orbix-based) for all tests. The server side used either Orbix or Maestro. With Maestro-based tests, we measured performance for different degrees of object replication (1, 2, 3, or 4

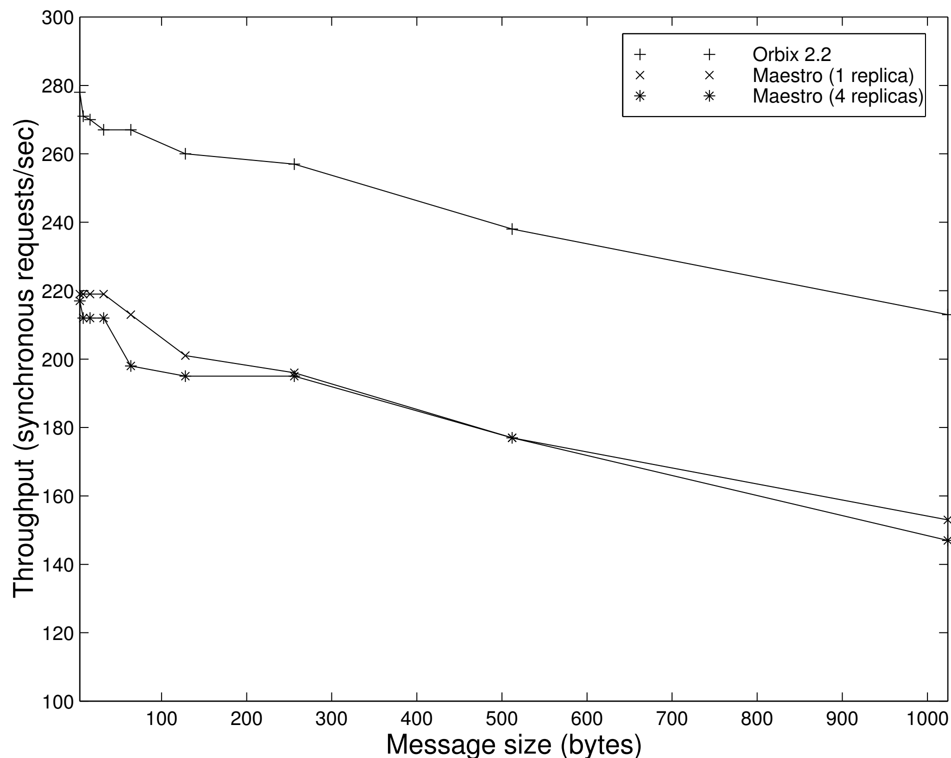

Figure 3.8: Performance of the Maestro Replicated Updates ORB (over Ensemble) and Orbix 2.2

replicas). We executed the tests on a cluster of sparc10 machines running Solaris 2.5 or SunOS 4.1.3 and connected by a lightly loaded 10Mbps Ethernet. In all tests, client and server processes were running on different machines, with at most one server replica per machine.

The throughput results measured for the `_set_value` operation are shown in Figure 3.7. Somewhat surprisingly, the Replicated Updates ORB outperforms Orbix even with four object replicas. We believe this is due to the combined effect of lighter-weight interfaces of Maestro and a specific communication pattern of the application, for which Horus was successfully optimized. As we mentioned earlier, Maestro complies to the IIOP *protocol* but does not implement CORBA *interfaces*, which are quite bulky and involve a lot of redundant function calls and data copying. With regard to the communication level, we observed that the Orbix client always binds to the server object referenced by the *last* IIOP profile included within a combined IOR. This means there is only one replica in the object group which receives IIOP requests and relays them to other replicas via totally ordered multicasts. The Dynamic Sequencer total ordering layer, used in the Horus protocol stack underneath Maestro, works best with precisely this one-sender communication pattern: The active sender becomes the sequencer and can multicast totally

ordered messages immediately, without any delays or additional communication rounds [Bir96]. In particular, the latency of local message delivery is minimal and depends mostly on internal processing costs within Horus layers. Also, the combination of the Dynamic Sequencer protocol and one-sender communication pattern scales extremely well. The measured performance of the test application was almost the same with 2, 3, or 4 replicas[7].

The elapsed time per request was in the range of 3.6-4.7 msec for Orbix and 3.5-4.7 msec for Maestro with three object replicas, for message sizes in the range of 4-1024 bytes. For comparison, the cost of the `bind` operation in the CORBA-compliant Electra Naming Service [Maf96] with replication degree 3 is 6.6 msec when run over Horus and 14.2 msec over Isis.

Performance numbers for the Maestro Replicated Updates ORB running over Ensemble are shown in Figure 3.8. We have measured a lower throughput with Ensemble than with Horus, apparently because the total ordering protocol used in Ensemble has not yet been optimized to minimize the latency of local delivery of messages [Cla98]. However, the protocols scale quite well: The throughput measured with four object replicas is almost the same as with one replica.

3.7 The Maestro Wizard

Maestro Wizard is a tool which aids in development of distributed object-oriented applications by guiding the programmer through a series of design choices which pertain to such dimensions as execution style, communication properties, quality of service requirements, and object state of the application. The programmer is also able to fill in the code for Maestro callback methods from within the wizard. Remember that callbacks are invoked when corresponding events are received by the object, which included such events as multicast or point-to-point message delivery, group membership changes, state requests, and others. The full set of callbacks that can be filled-in by the developer to implement application-specific functionality varies depending on the Maestro class to be used with the application. The choice of the specific class is made by Maestro Wizard based on the execution style selected by the developer. Among possible options are CSCW (cooperative-work), client/server, and publisher/subscriber execution styles.

Maestro Wizard allows the programmer to choose their consistency requirements for message delivery (available options include virtual synchrony, active replication, etc.) and global safety (whether or not support for primary partitions is required, and how the progress of non-primary partitions should be restricted). Based on choices made by the developer, Maestro Wizard generates the template code, which can be either compiled directly into the application, or be further mod-

[7] Figure 3.7 does not show the graph for Maestro performance with 3 replicas, which was very close to that for 2 replicas.

ified by the developer as necessary. It is envisioned that for complex distributed applications off-line editing will be necessary, after the first step in the development process (generation of the template code with Maestro Wizard) has been completed. However, the editing will mostly consist in connecting Maestro objects to other components of the application, without need to do manual configuration of Maestro itself.

The initial version of Maestro Wizard has been implemented with a simple text-based interface. As a direction for future work we can envision a more sophisticated GUI-based visual environment where a developer will be able to perform all aspects of development work, including implementation of individual distributed objects and their integration into the whole system, initialization and configuration of system parameters, and high-level control over quality-of-service properties of the application.

A transcript of a Maestro Wizard session is shown below. The programmer can select the ORB-specific wizard (among those available in the Maestro Wizard Repository) that best matches the execution style/request processing policy required for the application. In this example, the wizard for the Replicated Updates ORB is selected. As we discussed before, the Replicated Updates ORB provides reliability by actively replicating server objects. The wizard guides the developer through a series of questions and generates server application stub and configuration/initialization code according to the options selected:

```
$ wizard
Searching wizard repository in /home/alexey/iiop/WizardRepository/...
Wizard Repository:
1:  Replicated Updates
2:  Replicated Data
3:  Simple

Available options:
s[elect] <#> -- select a wizard to use for the application
d[escribe] <#> -- describe a wizard
Please enter your choice:  select 1
Please enter the name of the application:  Grid

MAESTRO REPLICATED UPDATES ORB WIZARD
* The wizard will guide you through a series of
  configuration choices for the Replicated Updates ORB.
* A skeleton for the application, including ORB initialization/
  configuration code, will be generated and placed in the file Grid.C.

HOW SHOULD THE APPLICATION BEHAVE IN CASE NETWORK PARTITIONS OCCUR?
Available options:
1 - Allow progress in ALL partitions
```

```
2 - Allow progress in one partition only (the majority partition)
Please choose an option: 2

WITH HOW MANY REPLICAS DO YOU INTEND TO RUN THE APPLICATION?
Please enter the value: 3

WHEN A NEW REPLICA JOINS THE GROUP, OR WHEN SEVERAL PARTITIONS MERGE,
IS IT NECESSARY TO TRANSFER THE CURRENT STATE OF THE APPLICATION TO
THE NEW/LESS UPDATED REPLICAS?
Available options:
1 - Transfer state to less updated replicas
2 - No state transfer is needed
Please choose an option: 1

IS IT REQUIRED THAT ALL OBJECT REPLICAS RECEIVE ALL REQUESTS
IN THE SAME ORDER?
Available options:
1 - All replicas must receive all requests in the same order
2 - It's OK if replicas receive requests in a different order
Please choose an option: 1

COMPOUND IORS FOR REPLICATED OBJECTS CAN BE UPDATED AND REINSTALLED
IN THE ETC DIRECTORY (SPECIFIED WITH THE MAESTRO_ETC ENVIRONMENT VARIABLE)
WHENEVER THERE IS A CHANGE IN MEMBERSHIP IN THE ORB/OBJECT GROUP.
Available options:
1 - Automatically reinstall IORs after membership changes
2 - Don't (re)install IORs
Please choose an option: 1

The skeleton file Grid.C has been generated.
```

Based on the options selected in the Maestro Wizard session above, the wizard generates a skeleton file called `Grid.C`. The file includes ORB initialization/configuration code and a definition of the `Grid` class with generic `update()` and state transfer methods (`pushState()` and `getState()`) which have to be overloaded by the developer so as to implement the application's functionality. An example of a server implementation for the `Grid` interface based on the wizard-generated skeleton code is shown in Section 3.5. The code of the generated file `Grid.C` is shown below:

```
// Grid.C: Skeleton file for the Grid application.
// Generated by the Maestro Replicated Updates ORB Wizard
#include ``Maestro_CORBA.h''
#include ``Maestro_GIOP.h''
#include ``Maestro_ORB.h''
```

```
#include ``Maestro_ES_ReplicatedUpdates.h''

// Skeleton for Grid object implementation.
class Grid:  public Maestro_RUObjectAdaptor {
public:
    Grid(Maestro_RUObjectAdaptor_Options &ops):
        Maestro_RUObjectAdaptor(ops) { /* fill in... */ }

    Maestro_GIOP_ReplyStatusType update(
        Maestro_CORBA_String &operation,
        Maestro_CORBA_Message &request,
        Maestro_CORBA_Message &reply) { /* fill in... */ }

    virtual void pushState(Maestro_CORBA_Message &msg) { /* fill in... */ }
    virtual void getState(Maestro_CORBA_Message &msg) {  /* fill in... */ }
};
// Skeleton of the application.
main(int argc, char *argv[]) {
    // Create an ORB dispatcher.
    Maestro_ORB_IIOPDispatcher dispatcher;

    // Create an ORB.
    Maestro_ReplicatedUpdates_Options orb_ops;
    orb_ops.dispatcher = &dispatcher;
    orb_ops.progressInPrimaryOnly = TRUE;
    orb_ops.nReplicas = 5;
    orb_ops.stateTransfer = TRUE;
    orb_ops.requestsCommute = FALSE;
    orb_ops.reinstallIOR = TRUE;
    orb_ops.etc = &Maestro_DefaultEtc;
    orb_ops.ORBName = ``Grid'';

    Maestro_ES_ReplicatedUpdates orb(orb_ops);

    // Create an object (it automatically binds to the specified ORB).
    Maestro_RUObjectAdaptor_Options obj_ops;
    Maestro_CORBA_String keyStr(
                ``MAE:3f809d92a791947c346778da00069523b3:Grid'');
    Maestro_ORB_ObjectKey key(keyStr);
    obj_ops.key = key;
    obj_ops.orb = &orb;
    Grid obj(obj_ops);

    // Activate the ORB.
    orb.activate();
```

```
    // Block the main thread.
    Maestro_Semaphore sema;
    sema.dec();
}
```

Chapter 4

Protocol Support: Implementing State Machine Replication

4.1 Introduction

The Maestro group tools and interoperability tools discussed in previous chapters can be used as building blocks for developing distributed applications and gluing different pieces together. Maestro group tools, however, rely on the group communication system underneath to provide necessary *system membership* and *message delivery* properties within a coherent programming model. In this chapter we will discuss one such model, called *state machine replication* [Sch86], which plays the role of a fundamental building block in design of reliable/highly available distributed applications with strong consistency requirements on membership and communication. In the following sections we will describe an implementation of replicated state machines within the quorum-based *virtual synchrony* [Bir96] protocol framework, which happens to be the only technology that provides both *high availability* and *global consistency* properties [Bir96] (observe that transactions offer the latter but not the former property).

To put virtual-synchrony-based state machine replication in the proper context, it is useful to compare and contrast it with classic distributed-commit protocols, which play a similar role of fundamental building blocks in the field of distributed transaction systems. The most widely used protocol there is the *Two Phase Commit* [Bir96]. The protocol proceeds in two phases. In the first phase, the *coordinator* of the transaction solicits votes ("OK to commit?") from participating members and waits for their replies. In the second phase, the coordinator multicasts a "commit" or an "abort" message, according to the votes received in the first phase. The protocol guarantees that all participating members which come to a decision (abort or commit) will make the *same* decision, thus providing *global consistency*. However, the two-phase commit is not fault-tolerant and may block even if a single failure, such as the coordinator's crash, occurs during the execution of the protocol

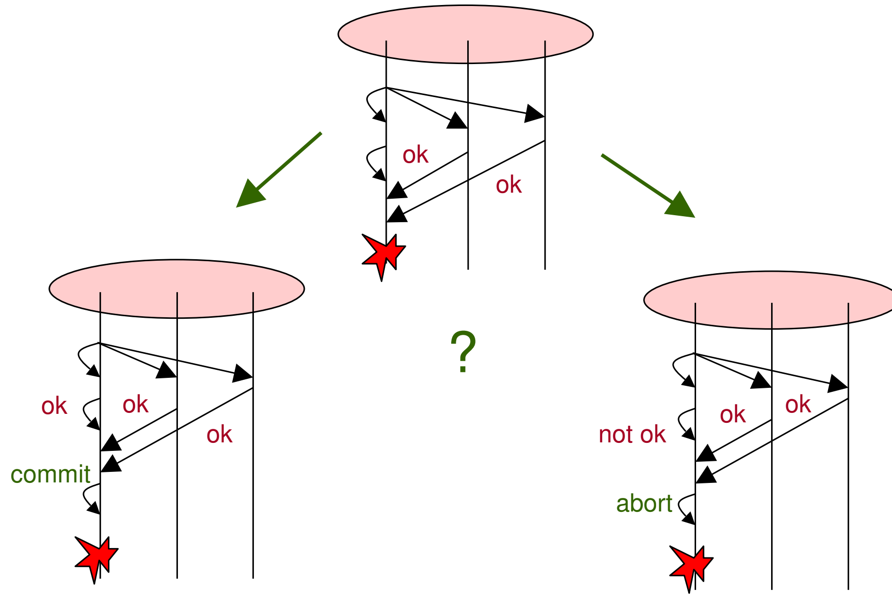

Figure 4.1: The Two-Phase Commit Protocol blocks when the coordinator crashes

(see Figure 4.1).

The *Three Phase Commit Protocol* [Ske85] was designed to provide higher availability and, in particular, to tolerate crash failures during execution of the protocol. However, despite the increased performance cost due to the added "prepare-to-commit" phase, the Three Phase Commit protocol assumes an idealized *fail-stop failure model* [Sch84] (where participants fail by crashing and failure detection is immediate and perfectly accurate) and may block in a real setting due to network partitions or inaccurate failure detection (see Figure 4.2).

Because of its limitations (in particular, lacking fail-stop, Three Phase Commit is no better than Two Phase Commit) and prohibitive performance costs, the Three Phase Commit protocol is rarely used in practice. On the other hand, the Two Phase Commit, being the most widely used protocol for distributed transactions, poses serious availability problems which may be unacceptable for an important class of distributed control/system management applications, which may require some form of globally consistent behavior but do not tolerate blocking of the whole system due to individual failures or network partitions. This kind of application is best targeted by the virtual synchrony/state machine replication paradigm which, as we mentioned before, provides *both* high availability and global consistency and,

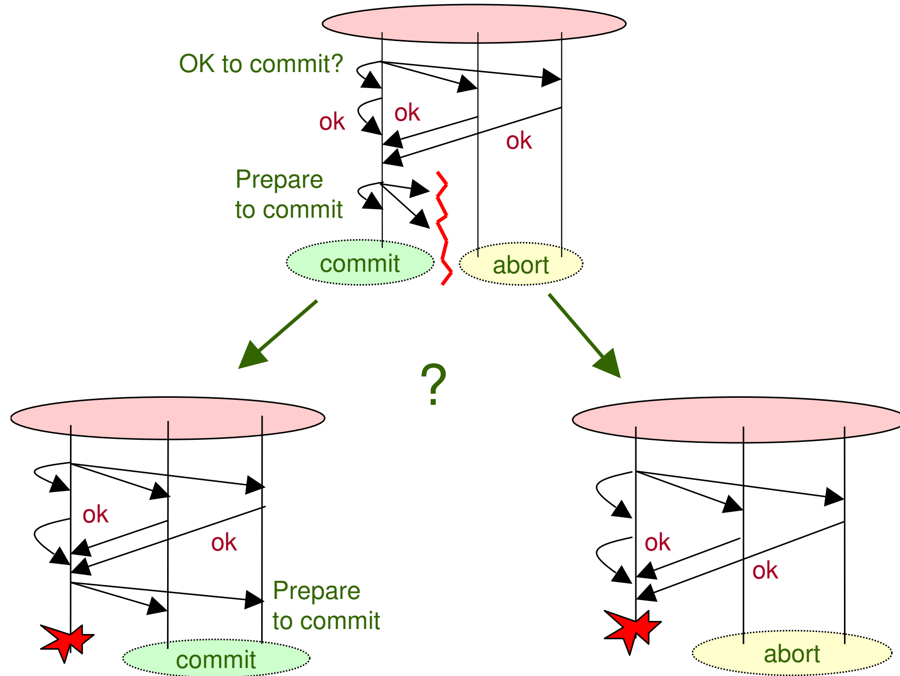

Figure 4.2: The Three-Phase Commit Protocol blocks when the network partitions

incidentally, offers a significantly better performance than the Two Phase Commit Protocol. In the following sections we will describe the state machine replication model in detail and present its virtual-synchrony-based implementation for the most general setting, namely a partitionable network environment with inaccurate failure detection, with excellent performance characteristics.

4.2 Background and Related Work

The *replicated state machine* model proposed in [Sch86] defines *consistent behavior* of a collection of distributed objects. With this model, the objects run identical *state machines* and perform the same sequence of operations, thus producing the same sequence of *outputs* and transitioning through the same sequence of *states*. To the outside observer, the behavior of the system as a whole is indistinguishable from that of a single highly available/reliable object (Figure 4.3). State machine replication is generally considered to be the most elementary paradigm for reliability through object replication and is therefore one of the fundamental low-level building blocks which can be used in development of complex distributed

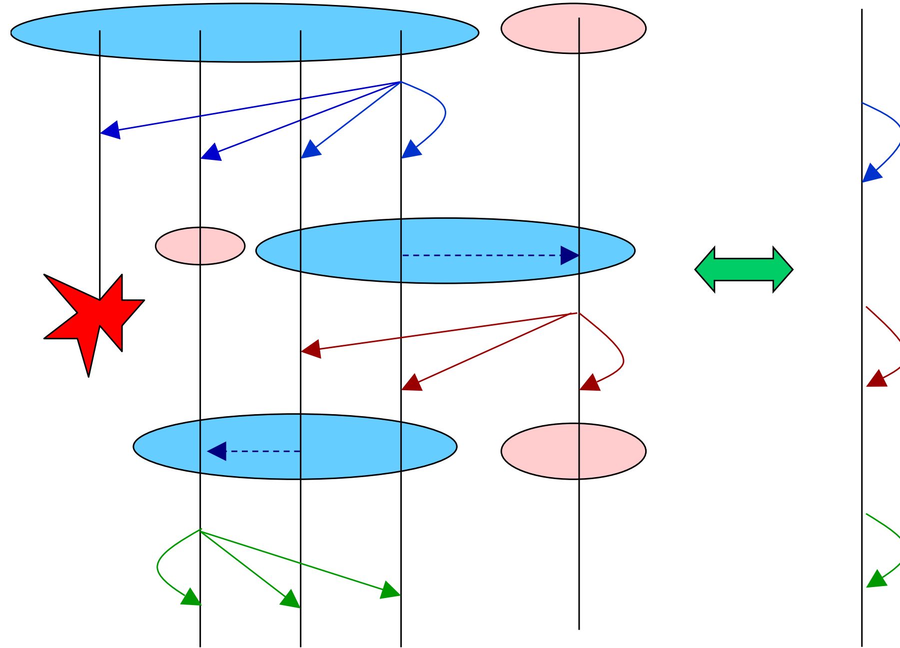

Figure 4.3: State Machine Replication: The group behaves as one reliable highly-available object

systems with strong global consistency *and* high availability requirements.

The first software-based implementation of state machine replication in an asynchronous distributed environment was provided in the Isis system [BJ87b], which has pioneered the *virtual synchrony* model within which both high availability and global consistency of replicated objects could be achieved. The achievement of the virtual synchrony paradigm has been in integration of *system membership* and *communication* within a single formal model and, in particular, in considering all communication within a *membership context*. Isis uses *totally ordered multicasts* to guarantee that all objects in the system (the *object group*) receive messages (operation requests) in the same order, so that all objects perform the same sequence of operations. However, in order to be able to make progress while guaranteeing global consistency, Isis requires a *quorum* of group members, which is computed dynamically as a majority of the last membership configuration (the *group view*) of the system. In case of a network partition or a "virtual" logical partition due to inaccurate failure detection, the system may irreversibly lose the quorum and force a global restart, even if the partitioned objects could eventually have merged back together. Our solution, presented in this chapter, uses a different method for

computing quorum and can tolerate group partitions [FV97]. In particular, even if the quorum is lost, it will be automatically restored when a majority of group members can again communicate and form a view, at which point the system will be able to continue to make progress.

Several implementations of *global total ordering* (a communication-level equivalent of state machine replication) that can tolerate network partitions have been proposed in [Ami95,ADMSM94,Kei94,MMSA93]. These solutions require all messages to be logged on stable storage, which adds a substantial performance overhead. In contrast, our implementation only requires logging of a single bit, which is performed by every process only once at initialization. A tradeoff of our approach is that the state of a member is lost in a crash failure and can only be recovered by means of state transfer from surviving members. If a majority of processes in the group crash simultaneously, the protocol will block.

Differently from our protocol, the implementation in [Kei94] can sustain any number of simultaneous crashes, assuming that failed processes are eventually restarted (this is equivalent to the "no-crashes" assumption). Also, the protocols in [Kei94] can make progress even if a majority view can never be formed due to perpetual partitioning of the network, which makes that approach suitable for WAN environments with very low quality of communication. However, the cost of message logging and extra communication rounds make the solution of [Kei94] impractical for applications with demanding performance requirements. Those applications will usually be deployed in environments with higher quality of communication links, where a majority of processes can reasonably be expected to be almost always connected. Our protocols are best suited for use in such high-performance systems, running over partitionable yet high-quality networks.

The protocols described in this chapter have been implemented within the Horus group communication system [vRBM96]. However, their properties and implementation are quite generic and can be easily ported to other systems, such as Ensemble [Hay97].

In summary, the trademarks of our implementation of state machine replication are *tolerance of network partitions* (the system will be available whenever a majority of group objects can merge together) and *high performance* (no message logging is required; messages can often be delivered within one phase). We will describe the details of our approach in the following sections.

4.3 System Model and Protocol Support in Horus

In this section we will describe a layered implementation of state machine replication based on the Horus group communication system [vRBM96]. The architecture of Horus supports multiple *protocol stacks*, where the high-level group seman-

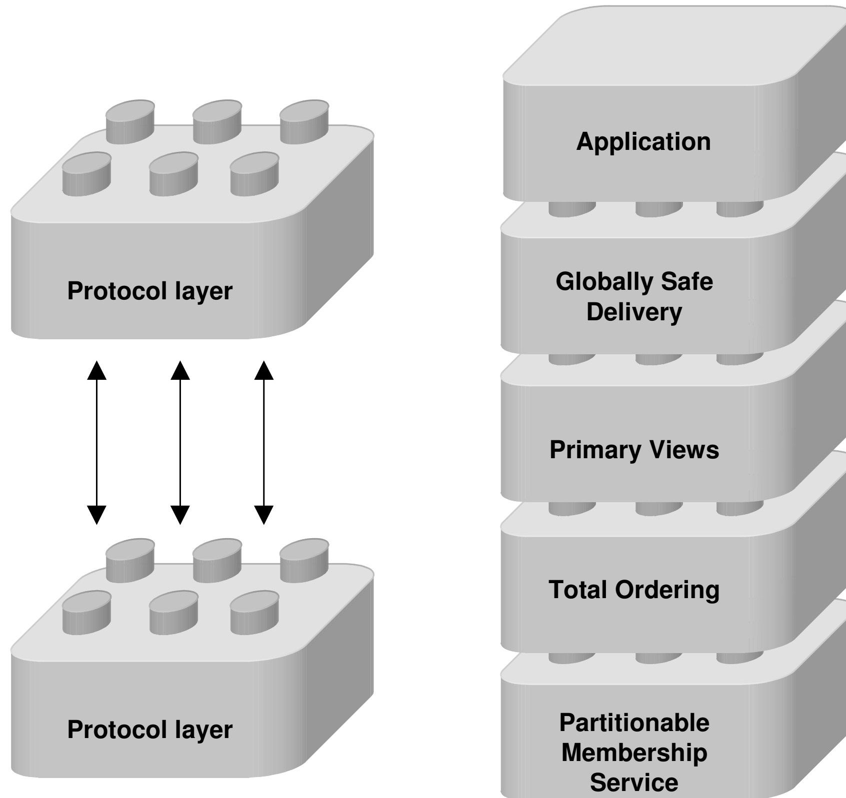

Figure 4.4: Layered protocol architecture of Horus

tics provided to the application is built as a composite function of properties of individual *protocol layers* included in a particular stack. Each layer implements a different mini-protocol and thus differ *semantically*. However, all layers implement the same *common group interface* and thus are compatible *syntactically* and can be composed together in various ways like Lego blocks (Figure 4.4). The *application* is represented by a `Maestro_GroupMember` object (or an object of a subclass of `Maestro_GroupMember`), which is mapped to the top layer in a protocol stack.

Our implementation of replicated state machines in Horus uses available protocol layers responsible for partitionable group membership, total ordering of messages, and other properties, as building blocks with which the high-level state machine replication semantics is obtained. In the reminder of this section we will discuss the properties of those provided layers and see how they are used by our *primary views* and *globally safe delivery* layers to implement replicated state machines.

It is important to remember that state machine replication is an *application-level* rather then *communication-level* property. Our implementation therefore addresses not merely group properties of several additional protocol layers but their *integration*, including and together with application-level abstractions and tools,

such as the state transfer tool implemented in Maestro. Thus, the cooperation of the application with underlying group protocols is important and necessary in order to ensure that the high-level replicated state machine semantics will be maintained throughout an execution.

Throughout the following discussion, we assume an asynchronous distributed system prone to process crashes, link failures, and network partitions. The system consists of a group of application processes, each running a deterministic *state machine* [Sch86], and communicating by sending multicast *messages* to each other. Messages are sent via `cast()` downcalls in Maestro and delivered via invocation of corresponding callback methods at destination objects. An application object's state machine is specified by a set of its internal configurations, or *states*, and a set of *transitions* between states. During a transition, an object may deterministically perform a set of *actions* (produce outputs). Each group object runs the same state machine and starts in the same initial state. The contents of the state and rules for state transitions and actions/outputs are purely application-specific. However, it is essential that the state changes only as a result of delivering multicast messages (via `cast()` callbacks), and those changes/actions be deterministic. It is a responsibility of the application object to guarantee that those restrictions will be enforced throughout the execution. Then the following assertion will be true: *Any two group objects which have seen the same sequence of* `cast()` *callbacks will be in the same state.*

4.3.1 Partitionable Group Membership Service

Horus provides a collection of protocol layers which implement a *partitionable group membership service* [FvR95b,MAMSA94,BDGS95]. Within the partitionable membership model of Horus, each group object has a view of the group (list of "accessible" members) at any time. A view is typically installed in two phases. In the first phase, a group member *proposes* a view by sending a view message to the list of object included in the view. In the second phase, when a member object receives a view message, it may either *accept* it, *i.e.* commit to the new view locally, or to reject the proposed view. A group object may need to reject a proposed view in order to preserve view consistency properties, as discussed below.

The protocol used in Horus to install new views proceeds normally in two phases (see Figure 4.6). In the first phase, a group member called the *coordinator* installs a *flush view*, which is a subset of the last regular view installed at group members. If a member object accepts a flush, it must eventually reply with a **flush_ok** message. When the coordinator receives **flush_ok** replies from all members included in the last flush, it proposes a new regular view (the second phase). If additional group members become unavailable during execution of the view change protocol, more than one flush phase may be necessary before the protocol completes.

The flushes are not visible at the application level. However, the application

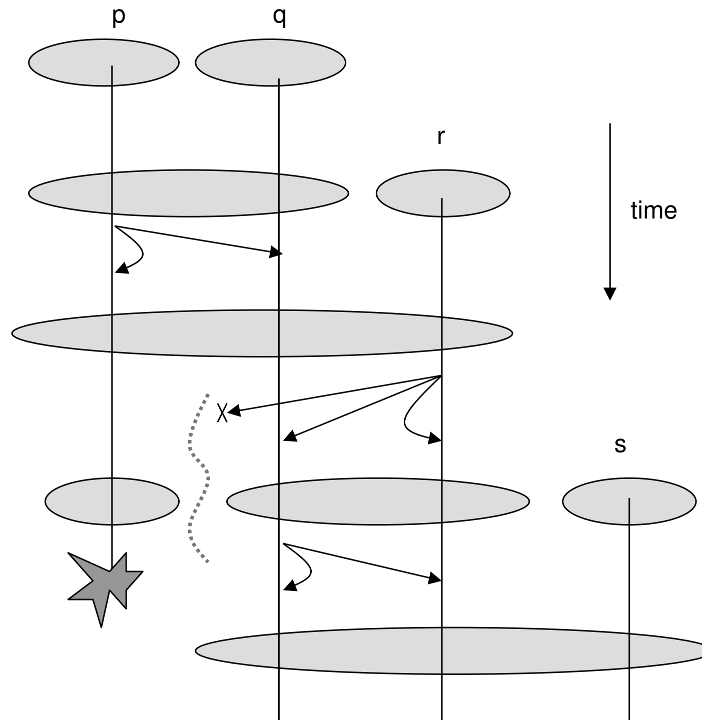

Figure 4.5: Partitionable group membership service in Horus: Multiple concurrent views can be installed simultaneously

(a subclass of the `Maestro_GroupMember` class) is notified of regular view changes with an invocation of the `AcceptedView` callback by Maestro.

When a new group member object is created, it initially installs a *singleton view* of the group and later on merges with other views as they discover each other and can communicate. If network partitions occur during an execution, group views can split into several new components, which will perhaps merge back together when they can communicate again. It is thus inherent in the partitionable membership model that multiple concurrent views of the same group can simultaneously exist in the system (Figure 4.5). Since failure detection is realistically assumed to be unreliable [CT93] and it is often not possible to distinguish crash failures from link failures or network partitions (which all manifest themselves as performance failures), a group component cannot automatically determine whether it is the only active view in the system or whether other group members are currently operational but just happen to be partitioned away. It is important, however, that at most one group component in the system is allowed to make progress, since executions of group objects in disconnected components can diverge, thus violating the replicated state machine semantics.

Figure 4.6: The View Change Protocol in Horus

While the partitionable group membership service of Horus does not provide mechanisms for achieving globally consistent system behavior, it implements protocol properties which can be used in higher layers to build state machine replication *within a view* and to select a *primary view* to represent execution of the group as a whole. The following properties of the partitionable group membership service in Horus are particularly important:

Validity: If a group member proposes or accepts a regular view or a flush, it is included in it.

Causality: Regular views and flushes are installed in the causal order with respect to **view** and **flush** messages.

Consistent Ordering: If two group members both accept any two views (regular or flush), they accept them in the same order.

Agreement on Successors: Any two regular views consecutively accepted by a group member are consecutive at any other group member which accepts both of them.

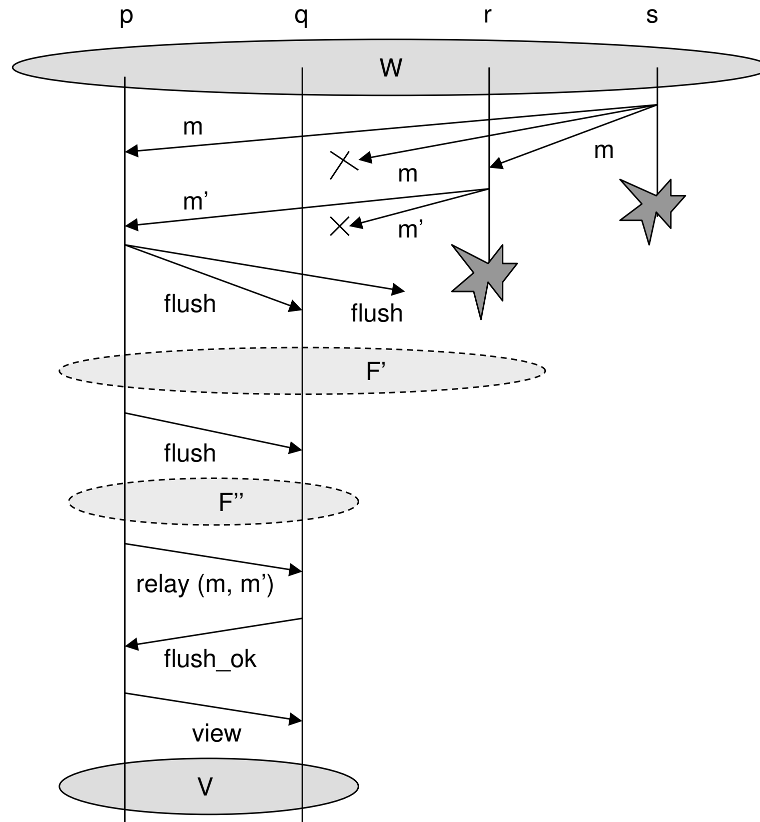

Figure 4.7: View Atomic Message Delivery: Group members included in two consecutive views deliver the same set of messages between the view changes

Agreement on Flushes: All processes included in two consecutive regular views must have accepted the same sequence of flushes between those views [1].

4.3.2 Atomic Message Delivery Within a View

The *Agreement on Successors* property guarantees that the notion of *consecutive views* is well defined and, in particular, does not depend on the choice of a particular group member for which the two views are consecutive. Relying on *Agreement on Successors*, the following property is also well defined and is implemented in Horus with several protocol layers above the partitionable membership service[2]:

View Atomic Message Delivery: Any two group members which are included in two consecutive regular views V_1 and V_2 deliver the same set of messages

[1] For example, in the scenario shown in Figure 4.5, group members p, q and r are included in consecutive regular views W and V and therefore must have accepted the same sequence of flushes between W and V (namely, F' and F'').

[2] In the context of implementing state machine replication we always assume that all messages are multicast to the entire current view of the sender.

after accepting V_1 and before accepting V_2.

The *View Atomic Message Delivery* property is implemented in Horus by delaying the installation of a new view until all multicast messages sent in the context of the old view are known to have been delivered by all group members included in both views. For example, in the scenario shown in Figure 4.7, group members s and r crash before their multicasts m and m' are received by all members in the view. Therefore, before a new view (V) is installed, the member p which received unstable messages has to relay them to the group members (namely, q) which did not receive them. When all surviving group members (i.e. the members included in the last flush) know that all messages they are aware of have become stable among them, the flush can complete and a new view (V) can finally be installed.

4.3.3 State Machine Replication Within a View

The *View Atomic Message Delivery* property guarantees that all members in a view agree on the set of messages they deliver in the context of that view. Now suppose that all group objects accept the view in *the same state* and agree not only on the set of delivered messages but also on their ordering, so that all members deliver *the same sequence* of messages in the view. It follows then that the replicated state machine semantics holds for the execution of the group within that view.

The agreement on sequence of messages delivered within a view is provided with one of the total ordering layers available in Horus. Those protocols usually fall into one of the two categories, the *sequencer-based protocols* and *token-based protocols* [FvR95a,Bir96]. With sequencer-based protocols, multicast messages are first sent point-to-point to a special member (the *sequencer*) which orders the messages and relays them to all members in the view. With token-based protocols, the members are allowed to multicast to the group only when they have a token, which rotates between the group members, thus naturally establishing a causality-induced ordering (Figure 4.8).

Total ordering protocols impose an additional delay for message transmission, namely between an invocation of the `cast()` downcall by the application and reception of the token by the group member (in case of a token-based protocol) or the ordering of the message by the sequencer (in case of a sequencer-based protocol). It is possible that during that waiting interval the group will split, so that the sender will partition away from the member currently holding the token or from the sequencer. When this situation occurs, each partition can deterministically order the remaining "suffix" of messages that were waiting for the token or for the sequencer's ordering at the time of the view split. This ordering will preserve replicated state machine semantics *within* the new views, however, executions *between* the views will obviously diverge, thus violating global consistency (Figure 4.9). It is therefore necessary for state machine replication purposes to strengthen the semantics of total ordering layers to require that when a view splits, at most one

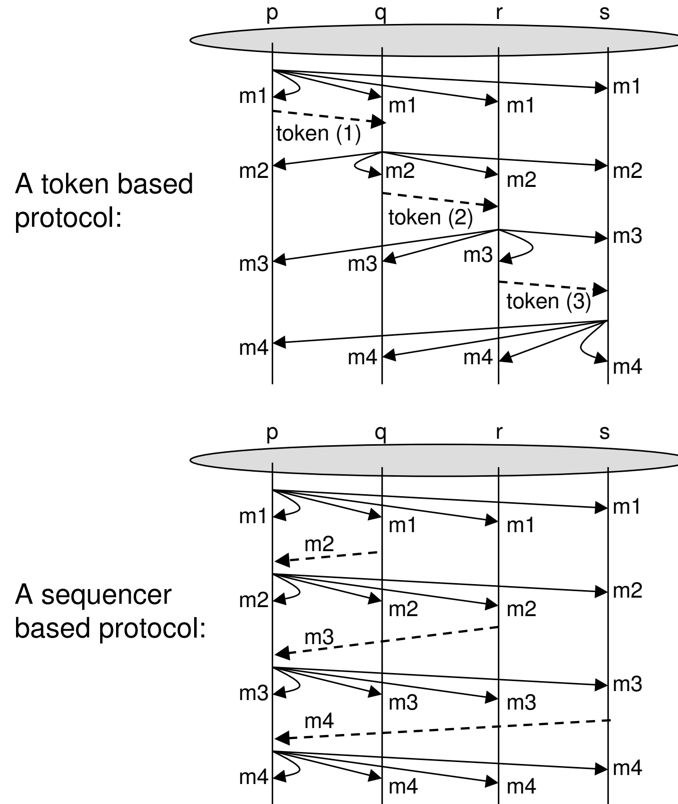

Figure 4.8: Total ordering protocols available in Horus

partition (the *primary view*) will actually deliver the "suffix" messages. The other partitions will report the aborted "suffix" messages to the application (a Maestro group member object) with an `abortCast()` callback. The handling of aborted messages is up to the application. Thus, we require the total ordering layer to provide the following property:

Strong Prefix: For any two group members in a view, the sequence of messages delivered in the context of that view by one of the members is a prefix of the sequence of messages delivered in that view by the other member, or vice versa.

In the next section we will describe our implementation of primary views which, together with the *Strong Prefix* property, guarantees that replicated state machine semantics is maintained not only within an individual view but on the global basis, despite possible link failures and partitions of the group.

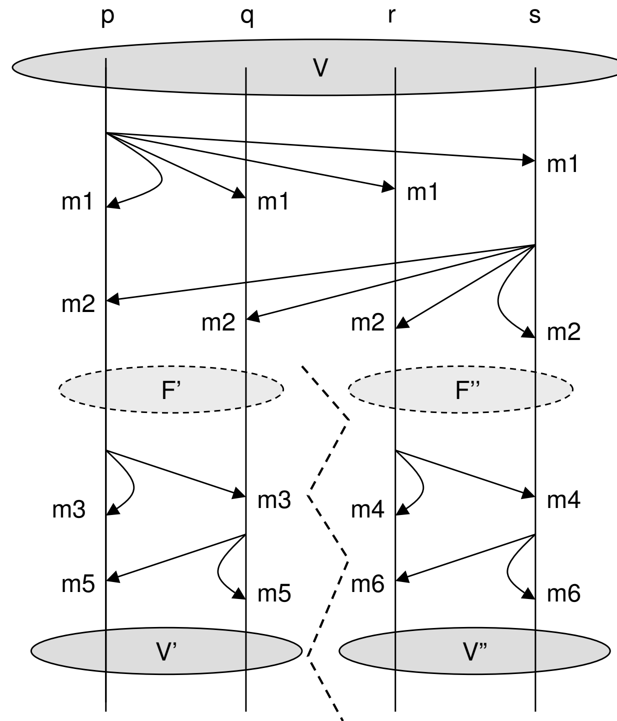

Figure 4.9: Execution of member objects may diverge when the view partitions into several components

4.4 Globally Consistent State Machine Replication

In this section we describe our implementation of globally consistent state machine replication with the *primary views* and *globally safe delivery* layers. The primary views protocol guarantees that at most one view in the group will be marked as *primary* at any time, and that whenever a majority of group members can merge together and form a stable view, this view will eventually become primary. We will also show that if group objects deliver multicast messages (and thus modify their application states) only when they are in a primary view, the execution of the group, as represented by executions of individual members in primary views, is indeed equivalent to an execution of a single reliable highly available object. Finally, we will discuss the role of state transfer in the state machine replication protocol and, in particular, the integration of protocol layers, state transfer mechanisms of Maestro, and the application.

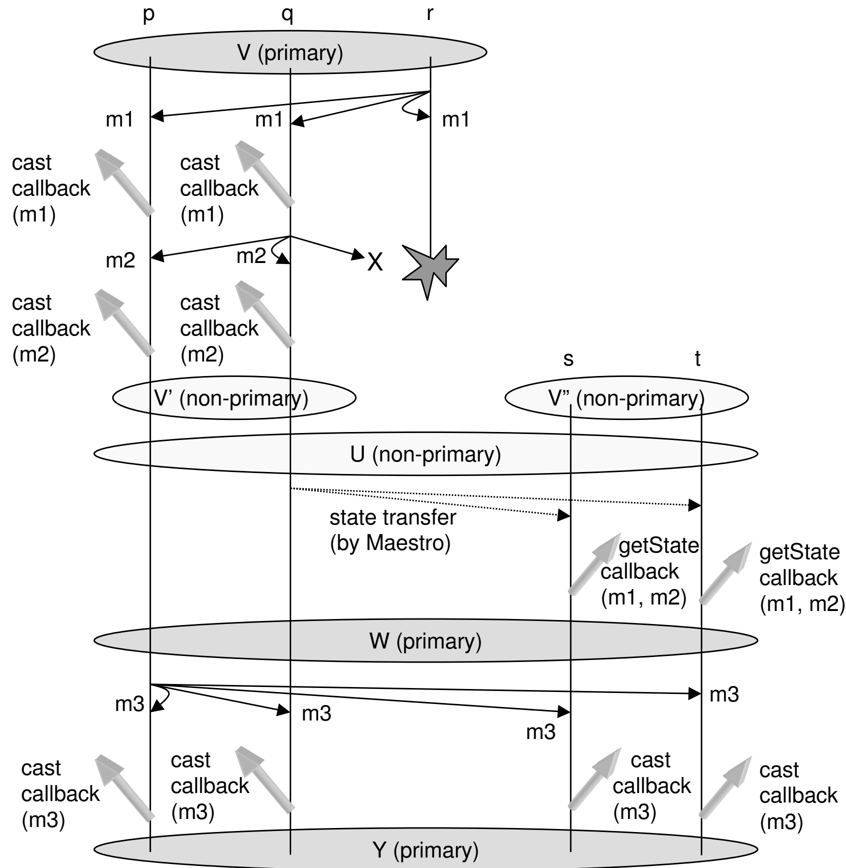

Figure 4.10: When views merge, the state of the more advanced view is transferred to members of the other view. When a primary view is installed, all included members are always in the same state

4.4.1 State Version Numbers

The primary views layer maintains *state version numbers* for all group members during their execution. A state version number is a pair of two integer values, the *primary view sequence number* and the *message sequence number*.

The primary view sequence number identifies the last primary view the object has been a member of. The message sequence number is the number of multicast messages the object has received in the context of the last primary view. The primary views layer guarantees that each primary view is assigned a unique sequence number and the ordering of sequence numbers agrees with the causal ordering of the views. The details of the protocol, in particular the assignment of sequence numbers, are discussed below.

The primary views layer ensures that the lexicographic ordering of state version numbers agrees with the degree of advancement of group members: The objects with higher state version numbers are more advanced ("up-to-date") in the execution of the group. In particular, since group members are allowed to send messages only when they are in primary views, which are uniquely identified by

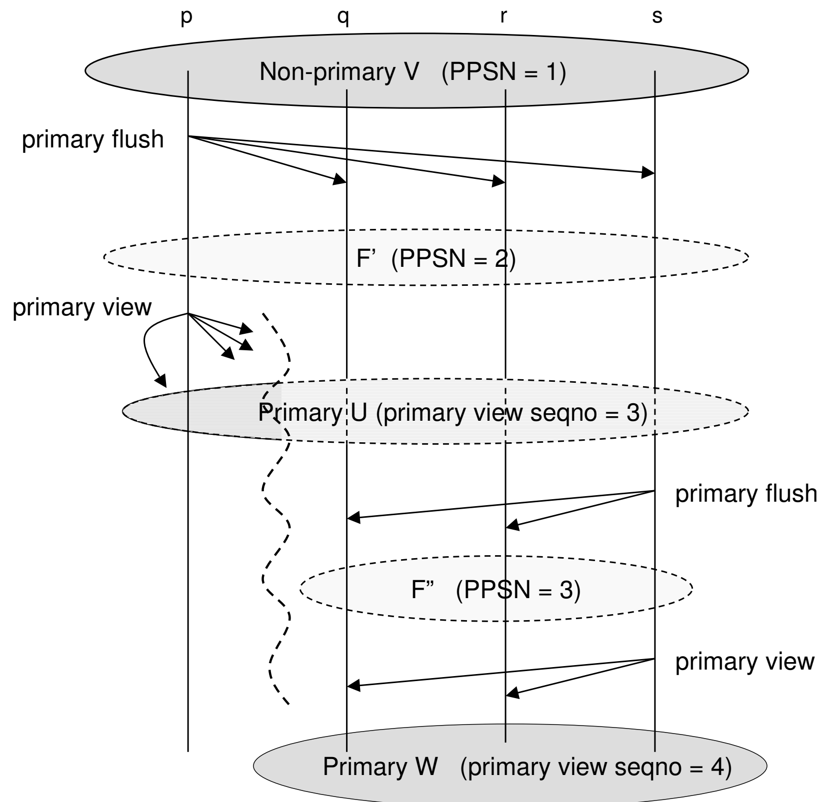

Figure 4.11: When any two different primary views are proposed, they are always assigned different sequence numbers

their sequence numbers, and messages are delivered by all view members in the same order (the *Strong Prefix Property*), it follows that any two group objects with identical state version numbers are in the same state, as long as they were in the same state when they accepted the last primary view. Below we will show how the primary views layer guarantees that group members are indeed in the same state when they install a primary view, so that the following *State Consistency* property is maintained throughout the execution of a group:

State Consistency: Any two group objects with identical state version numbers are in the same state.

4.4.2 Installation of Primary Views

The primary views protocol is illustrated in Figure 4.10. A view is installed as *primary* only if (1) it includes a quorum (a majority of group members) and (2) all members are in the same state. In order to guarantee the second property, the primary views layer initiates *state transfer* whenever two views with different states (as represented by their state version numbers) merge together. For example, in

the scenario shown in Figure 4.10, when views V' and V'' merge together, the new view U is installed as a non-primary view, even though it does include a majority of group members. After U is installed, the application (Maestro) is notified that state transfer is required. The primary views layer specifies the direction of state transfer and the set up-to-date of group members (p and q in this case) from which the state can be requested. The application is responsible for performing state transfer and notifying the primary views layer when the transfer is completed[3]. At this point the states of all members in the view (p, q, s, and t) are assumed to be identical, so the view is safely reinstalled as *primary* W. Once in a primary view, group members can multicast new messages (e.g. $m3$).

In order to implement the *State Consistency* property, the primary views layer must ensure that whenever primary views are proposed, they are assigned unique sequence numbers in such a way that the ordering of sequence numbers agrees with the linear ordering of primary views induced by the causal ordering of view messages. Otherwise, if group members were allowed to propose different primary views with identical sequence numbers, the execution of group objects, as reflected in their state version numbers, might diverge, thus violating consistency of the global group state.

The protocol for installation of primary views and assignment of primary view sequence numbers is illustrated in Figure 4.11. In addition to state version numbers, group members maintain so called *potential primary sequence numbers* (PPSN), which are used to assign sequence numbers to new primary views. Initially the value of the PPSN of a group object is set to 0. Whenever two views merge, the members in the new view adopt the higher value of the PPSN's of the merging views. Finally, when all members in a non-primary majority view complete state transfer and converge to the same state, the view is reinstalled as primary as follows. First, the coordinator performs a *primary flush*, which includes a majority of group members. Observe that primary flushes accepted by all included members (these are called *completed flushes*) can be assigned a natural linear ordering: The order of two completed primary flushes is the order in which they were accepted by a member that installed both of them[4]. When a group member accepts a primary flush, it increases the value of its PPSN by one. If more than one flush needs to be performed before the view change protocol can finalize, each following primary flush will increase the PPSN of an accepting member by one. When the flush protocol completes and the new primary view is proposed, it is assigned the primary view sequence number equal to the current PPSN value of the coordinator, increased by one.

For example, in Figure 4.11, the PPSN of members in the non-primary view

[3]See Chapter 2 for details of the state transfer protocol in Maestro and provided interfaces.

[4]By the *Consistent Ordering* property of the partitionable membership service underneath, this ordering is well defined, i.e. it does not depend on the choice of a member that accepts the two flushes.

V is 1. Therefore, when p flushes the view, the PPSN assigned to the flush F' is 2. When the flush completes (i.e. all group members included in F' respond with **flush_ok** messages), p proposes the new primary view U and assigns sequence number 3 to it. In the scenario shown in Figure 4.11, the group happens to partition right at the point when p proposes U, so that p is the only group member which ever accepts this view. After the other group members included in U (namely, q, r, and s) detect that p is unavailable, the new coordinator, s, starts a new primary flush, F'', so as to install a new primary view without p. By the *Agreement on Flushes* property implemented by the underlying partitionable group membership service of Horus, it is guaranteed that since s was included in U, it must have accepted the flush F' installed by p before proposing U. Thus, the current PPSN at s when it flushes the view is equal to 2, so that the flush F'' sets the PPSN value of accepting members to 3. When the flush completes and s proposes a new primary view W, its sequence number is set to the current PPSN value of s plus one, that is 4, thus guaranteeing that primary views U and W are indeed proposed with different sequence numbers. Observe that the following points of the protocol are essential: (1) all members in the view initially have the same PPSN value; (2) primary flushes include a quorum (a majority of group members); and (3) the *Agreement on Flushes* property holds.

In summary, the primary views layer provides the following properties:

Validity: A group member proposes a primary view only when the view includes a majority of group members and all members are currently in the same state.

Linear Ordering: All proposed primary views are assigned unique sequence numbers. The ordering of primary views, based on their sequence numbers, agrees with the causal linear ordering of their respective completed primary flushes.

Progress: Whenever a majority of group members can merge together into a view which remains stable for sufficiently long time, the view will be eventually reinstalled as *primary*.

4.4.3 Globally Safe Delivery and State Transfer

The primary views layer allows messages to be delivered immediately once they are totally ordered within the primary view. This results in a good performance, as it normally takes only one or two one-way transmissions for a message to be ordered and received at all destinations, depending on the total ordering protocol in use and application's communication pattern. However, this "optimistic" message delivery protocol must cope with a situation when a multicast is sent in a primary view and is totally ordered right before the sender partitions away, so that the message is never received by a majority of group members. In this case the progress of members which do deliver the message will diverge from the execution of the

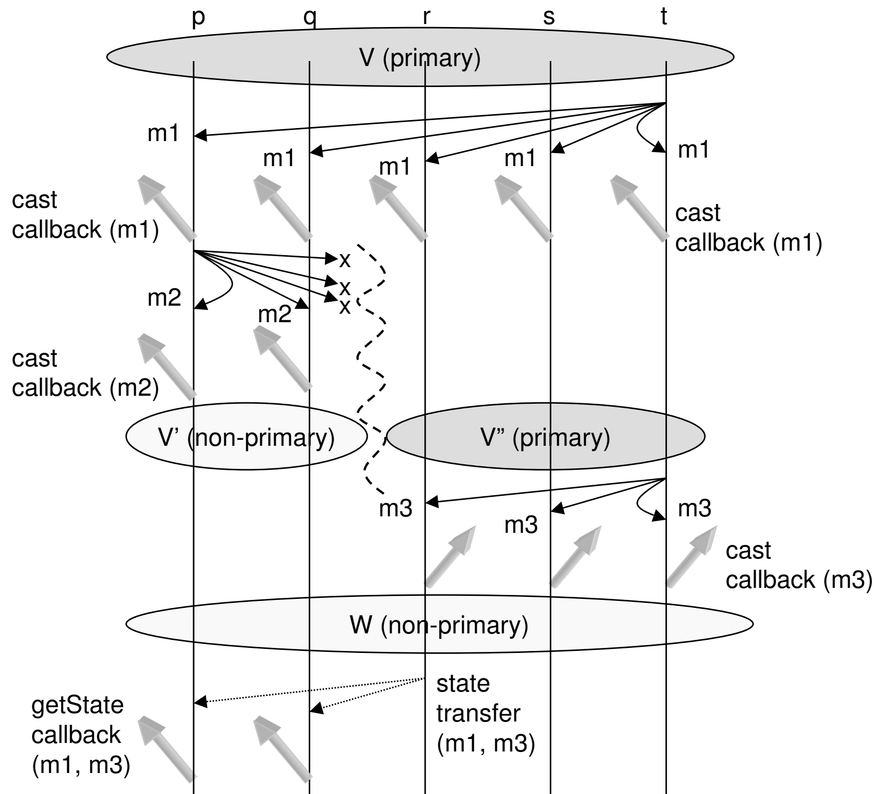

Figure 4.12: If objects deliver messages which are not safe (i.e. not acknowledged by a majority), their state may need to be corrected (or rolled back) later

group as a whole as represented by history of messages delivered by a majority of group members. When the diverged objects merge back into a primary view, they may need to roll back the effect of delivering the eventually-canceled messages.

An example of a scenario where optimistic message delivery is followed by a state rollback is shown in Figure 4.12. The message $m1$ sent in the primary view V is eventually received by a majority of group members and will therefore never be rolled back. However, the multicast $m2$ is sent by member p and is totally ordered (with a token or by the sequencer) just before p and q partition away from the other group members, so that p and q are the only objects which deliver $m2$ to the application with a `cast()` callback. The other group members are able to form a new primary view, V'', in which they deliver another message, $m3$, thereby diverging from the execution of p and q. When views V' and V'' merge back together into the view W, the more advanced state of V'', as represented by message history ($m1$,$m3$), is transferred to members p and q to overwrite the effect of applying $m2$ to their state with the effect of delivering $m3$.

The "optimistic" message delivery semantics with a possibility of rollback may be quite appropriate for some type of applications and has an advantage of faster

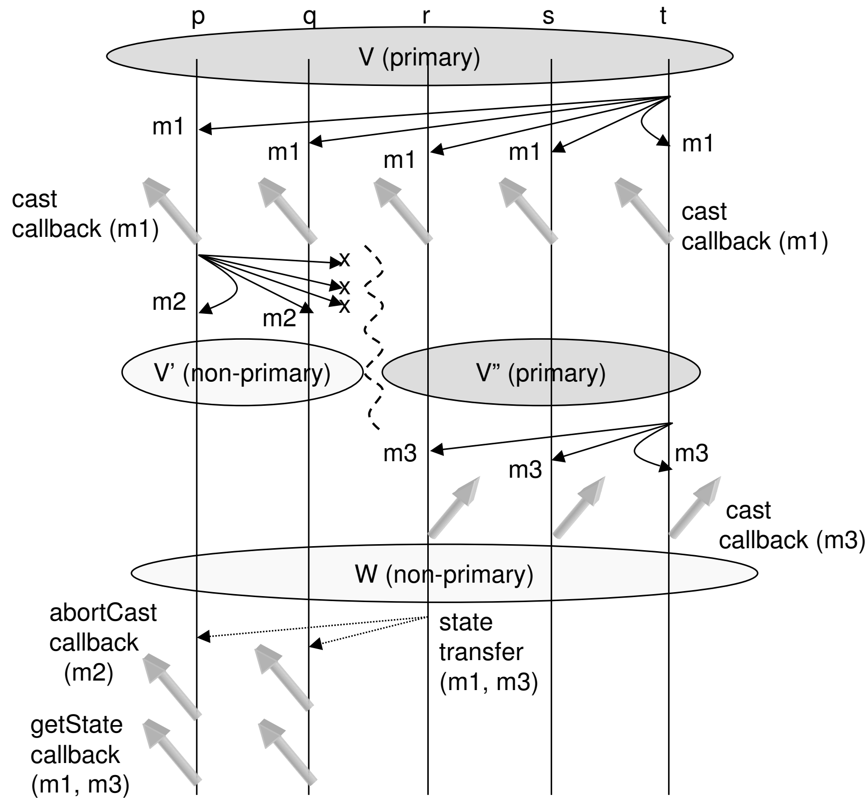

Figure 4.13: With the globally safe delivery layer, messages are only delivered when they become safe. If an unsafe message is rolled back, the application is notified with an `abortCast()` callback

message delivery (no need to wait for stability information). However, sometimes the effects of delivering a message cannot be meaningfully reversed, especially when they include externally observable actions triggered by the message. The applications in which state rollback is undesirable or simply not acceptable can run over the *globally safe delivery layer* [Ami95], which delays delivery of messages until they become acknowledged by a majority of group members and thus are guaranteed to never have to be rolled back in any partitioning/remerging scenario. If a message is never received by a majority and eventually needs to be rolled back, the application is notified with the `abortCast()` callback and simply removes it from its list of pending unsafe messages. An example of such a situation is illustrated in Figure 4.13.

The scenario shown in Figure 4.13 is identical with that of Figure 4.12, except that the globally safe delivery layer is now in use, stacked over the primary views layer in the application's protocol stack. The multicast $m1$ is eventually acknowledged and consequently delivered by all group members. However, the message $m2$ is never received by a majority of members since the sender p partitions away before

Figure 4.14: Integration of the primary views layer, globally safe delivery layer, and the state transfer protocol of Maestro

the multicast stabilizes. Therefore, although the objects p and q have received $m2$, they will never deliver it. At some point $m2$ becomes overwritten by a diverging execution in the primary view V'' (namely, by the multicast $m3$). When views V' and V'' merge together and the up-to-date state is transferred to group objects p and q, they notify the application with `abortCast()` callbacks that the message $m2$ has been aborted and will never be delivered. It is up to the application to determine an appropriate policy for handling aborted messages.

As discussed before, the primary views layer notifies the application when two views with different state version numbers merge together and state transfer is required (Figure 4.10). With the globally safe delivery layer, the state transfer protocol is more complicated as it proceeds at two levels. The *application (Maestro)* performs state transfer for the execution history represented by the sequence of safe message delivered to group objects in the more updated view. In addition to the application-level state transfer, the globally safe delivery layer performs an *internal transfer* of unsafe messages, hidden from the application, to the less advanced view. As a result of the internal state transfer, some of the unsafe messages received by group members in the previous primary view may now become safe and be delivered

to the application. When both Maestro-level and internal state transfer complete, all members in the view are in the same state. At this point, if the view includes a majority of group members, it is safely reinstalled as primary.

An example of a view merging scenario illustrating the integration of the primary views layer, the globally safe delivery layer, and the state transfer protocol of Maestro, is shown in Figure 4.14. After the multicast message $m1$ sent in the primary view V becomes acknowledged by a majority of group members, it is delivered to the application with the `cast()` callback. The other message sent in V, $m2$, does not become majority-safe since one of the view members, r, crashes before it receives $m2$. Observe that the total group size is five, so that a message needs to be acknowledged by at least three members before it becomes safe and can be delivered to the application. At some point view V' and V'' merge together into the view U. the globally safe delivery layer notifies Maestro that the application state of the more advanced view (V') should be transferred to members in the other view. After application-level state transfer completes, the globally safe delivery layer transfers the unstable messages ($m2$ in our example), transparently from the application, to members in the less advanced view. After the message is acknowledged by group objects in U, it becomes majority-safe and is delivered to the application with the `cast()` callback. This completes the state transfer protocol, at which point the view U, as containing a majority of group members, with all members being in the same state, is reinstalled as primary W.

In certain scenarios the transfer of the application-level state corresponding to delivered safe messages and the internal transfer of unsafe messages will proceed in the opposite directions. In the example shown in Figure 4.15, the message $m1$ is sent in the view V and is mutually acknowledged by members r, s, and t, which then safely deliver it to the application. However, the other two members of the view, p and q, partition away before they can collect the minimum number of acknowledgements in order to deliver $m1$ locally. Also, before p installs a new (non-primary) view V', it multicasts and totally orders another message, $m2$, in the context of the primary view V. Thus, both p and q receive $m2$ in V but do not deliver it since $m2$ does not become majority-safe. At some point V' and V'' merge together into the view W. From perspective of the primary views layer, the state of the view V' is more advanced than the state of V'', as represented by the state version numbers of the two views. However, the globally safe delivery layer correctly notifies Maestro that application-level state transfer should proceed from V'' to V', since from the application's point of view, the members in V'' have delivered more messages than p and q have done and therefore are more up-to-date. Thus, Maestro transfers the state corresponding to $m1$ to the members p and q, whereas the globally safe delivery layer transparently transfers $m2$ to the members in V''. After an exchange of acknowledgements, $m2$ becomes safe and is delivered to the application at all group member objects. This completes the state transfer, and the non-primary view W is reinstalled as primary U.

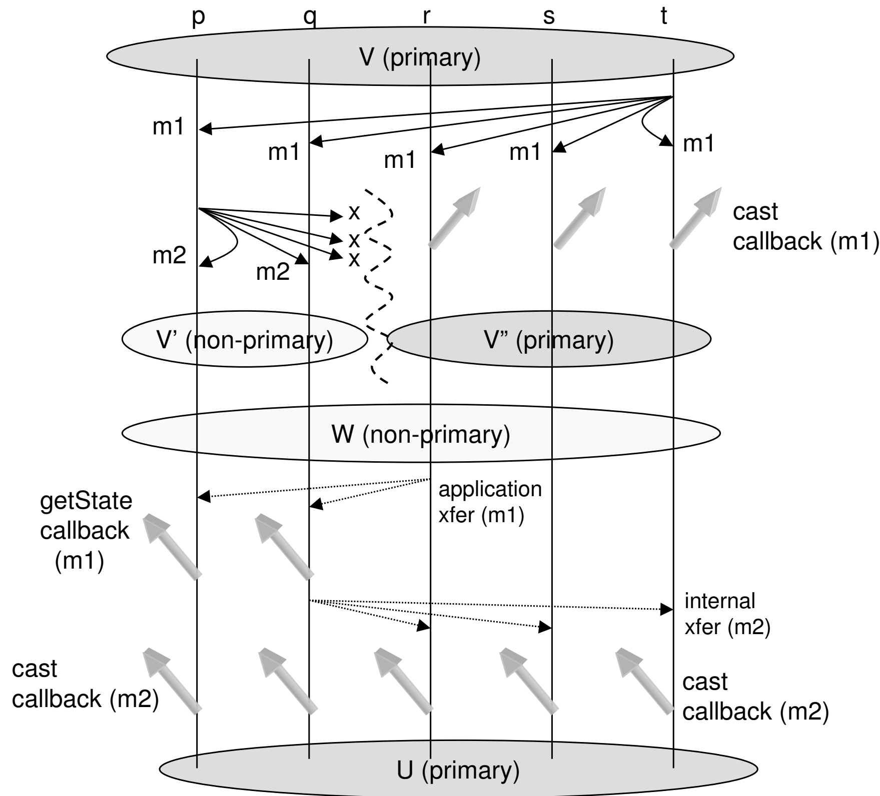

Figure 4.15: Maestro-level state transfer and internal transfer of unsafe messages may proceed in opposite directions

4.4.4 Restarting Objects After Crashes

Since the probability of a crash failure of a process is never zero, a long-running application must rely on some membership monitoring service to restart objects after crashes. It is important to maintain the required degree of object replication and prevent a majority of group objects from having failed simultaneously, which would block the progress of the group forever. While a membership service could be implemented at the tools level based on various object monitoring and reincarnation policies, it is a responsibility of the primary views layer to ensure that when objects are restarted and join the group, their state is reinitialized carefully so as to preserve the replicated state machine semantics of the application.

For performance reasons, our protocols do not require that messages be synchronously logged on a stable storage before they can be sent or received. This is in contrast to other approaches, such as [Kei94]. The tradeoff of this decision is that when a process crashes, the state of the object(s) residing on it may be lost. However, if an object is restarted with an initial state and is allowed to immediately rejoin the group, the global consistency of the application can be violated.

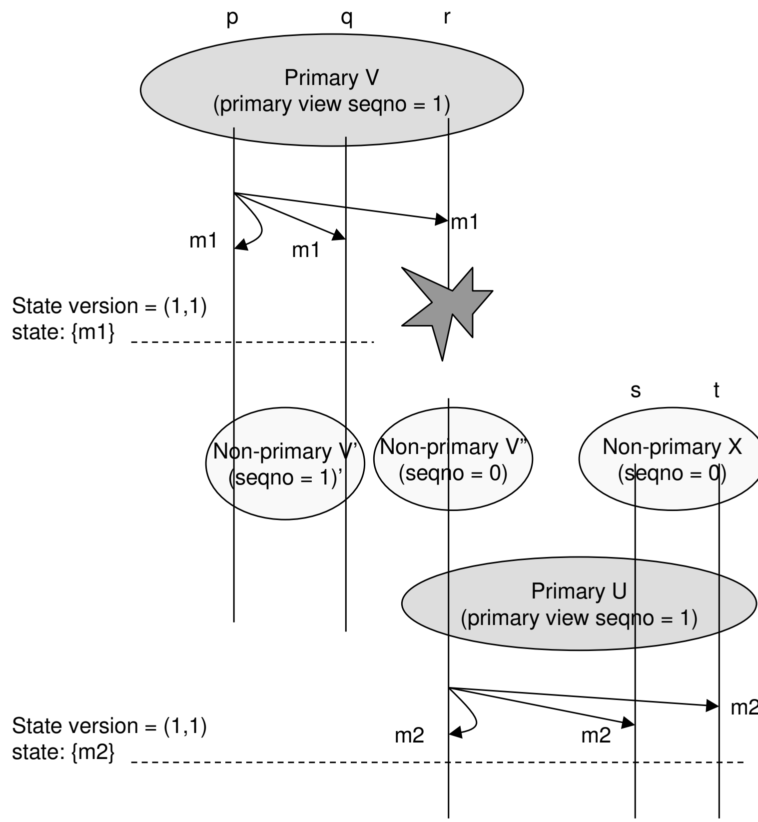

Figure 4.16: If objects restarted after crashes still count when computing a quorum, the state of the group may become inconsistent

An example of such "bad" scenario is shown in Figure 4.16. There, the object r is included in the primary view V with sequence number 1, in which it receives one multicast message, $m1$. At that point the state version number of r becomes $(1,1)$, with the state defined by the message history of $\{m1\}$. Suppose now that r crashed, was eventually restarted in an initial state with state version number $(0,0)$, and merged with two other member objects s and t with the same state version number $(0,0)$, to form a majority view U. Since r, s, and t would be in the same state $(0,0)$, the view U would be installed as *primary*, and it would be assigned the sequence number 1 as if it were the first primary view ever installed in the group. This would immediately violate the *Linear Ordering* property of primary views which the primary views layer must provide. Furthermore, if r received a multicast message $m2$ in the context of view U, its state version number would again become $(1,1)$, but now the state would be defined by the message history of $\{m2\}$, different from $\{m1\}$. This would be a violation of the *State Consistency* property, which is clearly not acceptable.

In order to maintain global consistency of the group in spite of object crashes/ reinitializations, the primary views layer uses *reincarnation bits*, which are logged

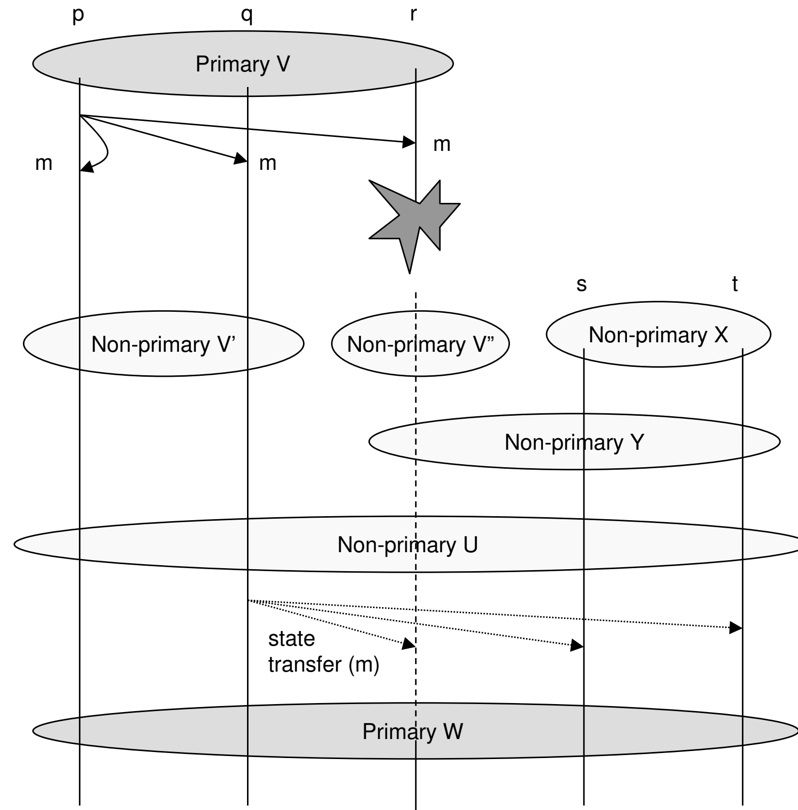

Figure 4.17: When an object is reincarnated after a crash, it remains a zombie (denoted by the dashed line) until it receives an up-to-date state and becomes a member of a primary view

by member objects on stable storage during initialization. Before an object replica is started for the first time, its reincarnation bit is set to 0. When an object is (re)started, it atomically reads the old value of its reincarnation bit and sets the new value to 1. If the old value was not 0, the object is assumed to be a reincarnation of a previously failed replica, in which case it is assigned the *zombie* status[5]. Zombie members do not count when a quorum is computed. Only after a zombie object receives an up-to-date state of the group and is included in a primary view, it will again become a normal member and will count when a group majority is computed. For example, in the scenario shown in Figure 4.17, the object r is restarted after a crash and merges together with two other group members, s and t, as in the scenario of Figure 4.16. However, this time r is assigned the zombie status and does not count when determining whether the new view Y includes a majority of group members. Since Y only includes two normal members (out of five object

[5]It is a responsibility of the membership monitoring service to guarantee that at most one incarnation of an object replica will be active at any time.

replicas in the group), it is installed as a non-primary view. However, after the group members merge to form a majority view U and state transfer is completed, the view is safely reinstalled as primary W, in which all members, including r, have the normal status.

4.4.5 State Machine Replication Properties

We say that (M_p, S_p) is an *execution history* of object replica p if M_p (the *message history* of p) is the sequence of messages delivered by p and S_p (the *state history* of p) is the sequence of states through which p has transitioned during its execution. We assume that all messages in a message history and all states in a state history are unique, i.e. they can appear at most once. As discussed before, all object replicas run the same state machine, and state transitions happen only as a result of a message delivery or a state transfer from another replica[6].

In integration with the membership and total ordering layers, the Maestro tools (including the state transfer protocol), and the application, the primary views layer and the globally safe delivery layer implement the *state machine replication semantics* as follows.

For any fixed execution of the group, there is an object execution history (M_G, S_G) such that the following properties hold:

Linearizability: If M_p were the message history of an object replica p and p never participated in a state transfer as a recipient, then the state history of p would be S_G.

Completeness: For any object replica q, if (M_q, S_q) is the execution history of q, then M_q is a subsequence of M_G and S_q is a subsequence of S_G.

Validity: For any state s in S_G there is an object replica q such that s is included in S_q, and for any message m in M_G there is an object replica r such m is included in M_r.

When the globally safe delivery layer is not included in the protocol stack, the *Completeness* property is replaced with the following:

Majority Completeness: If state s is included in state histories of a majority of group members, then s is included in S_G. If message m is included in message histories of a majority of group members, then m is included in M_G.

The pair (M_G, S_G) is called the *group execution history*, with the following meaning: *The group emulates an execution of a single reliable highly available object with execution history of* (M_G, S_G). Observe, however, that it is possible there will be no single object replica p in the group such that $M_p = M_G$ and $S_p = S_G$.

[6] In particular, due to state transfer, two object replicas with identical state histories may have different message histories.

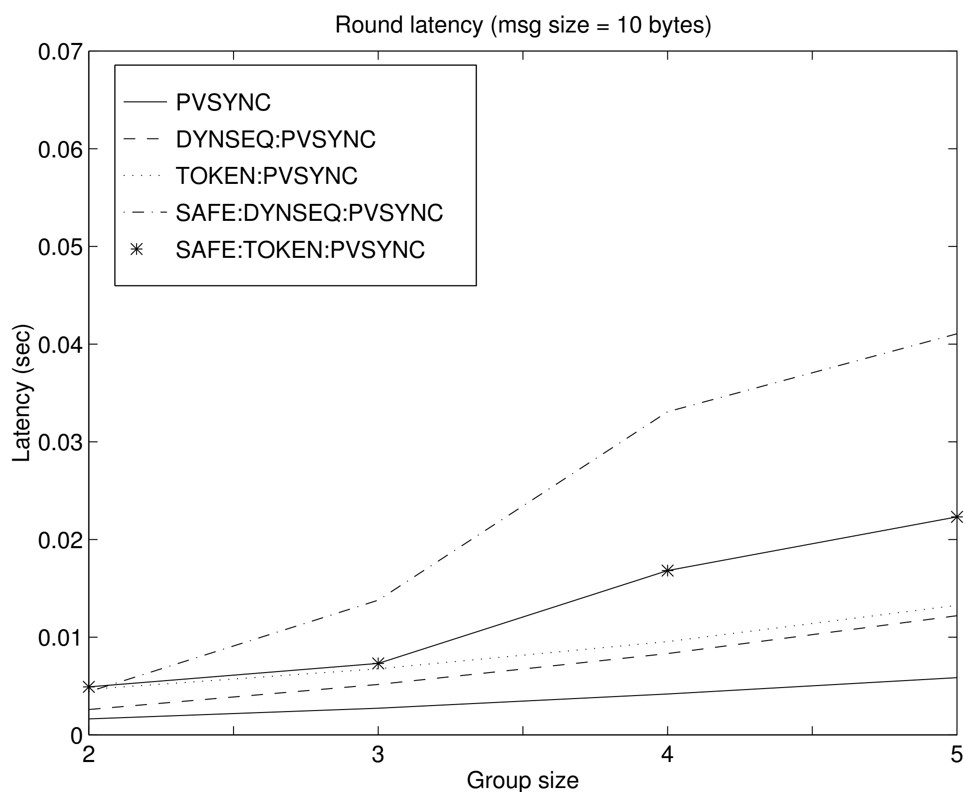

Figure 4.18: Average round latency for different group sizes

4.5 Performance

We have measured performance of our Horus-based implementation of replicated state machine protocols with a cluster of Sparc10 and Sparc20 machines connected by a 10Mbps Ethernet. The tests were performed in rounds (using the `ring` application of Horus), where in each round group members were synchronized and every member sent a series of 10-byte multicast messages. A round would complete when all transmitted messages became fully stable. Thus, the average duration of a round measured performance of the system under maximum load, with everybody sending bursts of messages to everybody else.

The effect of the *primary views layer* on performance has proved to be insignificant, which was not a surprise since most of the work performed by this layer is done during installation of new views and state transfer. During normal execution within a view, the primary views layer does not add any headers on messages and has little overhead for event processing. We have found, however, that the choice of a particular total ordering protocol [AMMS+93,FvR95a,KTHB89] and the use of message packing techniques [FvR95a] play a decisive role in setting the system's performance. Also, when the *globally safe delivery layer* is included in the stack, the protocol it uses for fast propagation of stability information proves to significantly influence the resulting performance.

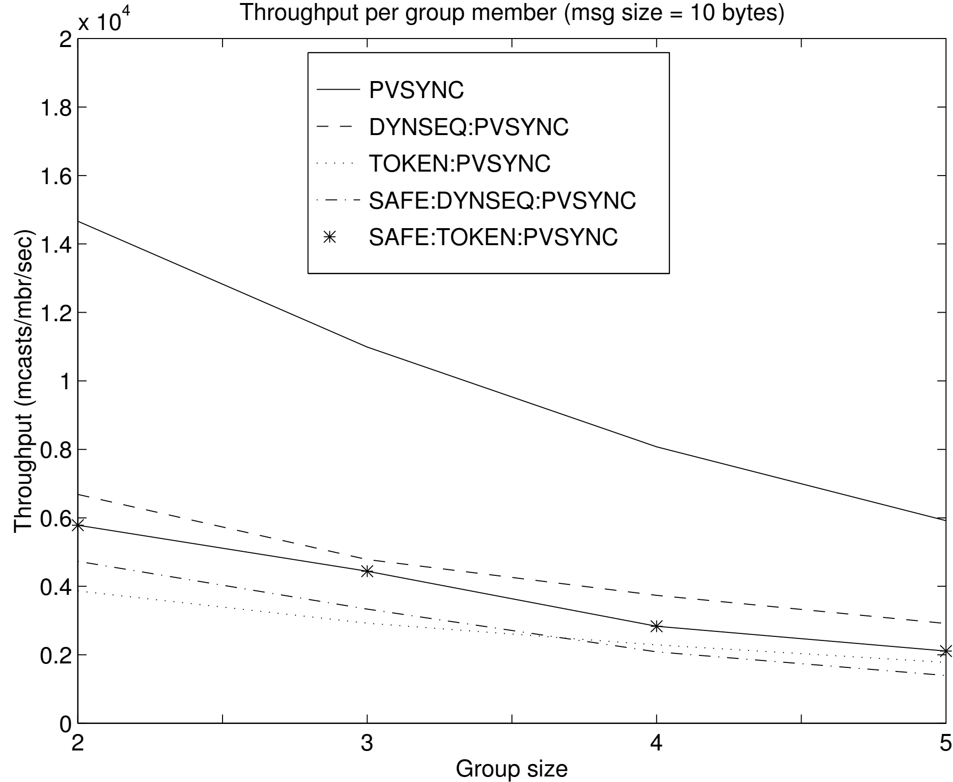

Figure 4.19: Average throughput per group member for different group sizes

Generally, the *dynamic-sequencer* total ordering protocol outperformed the rotating-token protocol in protocol stacks providing *optimistic message delivery* (without the globally safe delivery layer), in terms of both latency and throughput. On the other hand, the *rotating-token* total ordering protocol provided better latency and throughput than the dynamic-sequencer protocol when *safe delivery* was enabled (i.e. the globally safe delivery layer was included in the protocol stack).

We have experimented with a number of protocols for fast propagation of message stability information required by the globally safe delivery layer. Based on performance measurements, we have found that the protocol based on piggybacking the stability vector on a rotating token performed best, with respect to latency as well as throughput, regardless of the total ordering protocol being used. However, when the rotating-token total ordering layer is included in the protocol stack, the same token can be utilized for both total ordering of messages *and* for stability propagation, thus increasing the efficiency of the protocols.

The numbers obtained in performance measurements are shown in Figures 4.18, 4.19, and 4.20. The protocol stack which provides partitionable group membership with view-atomic message delivery ("partitionable virtual synchrony") is denoted as `PVSYNC`. The stacks which implement optimistic message delivery with primary views, based on dynamic-sequencer or rotating-token total ordering protocols, are

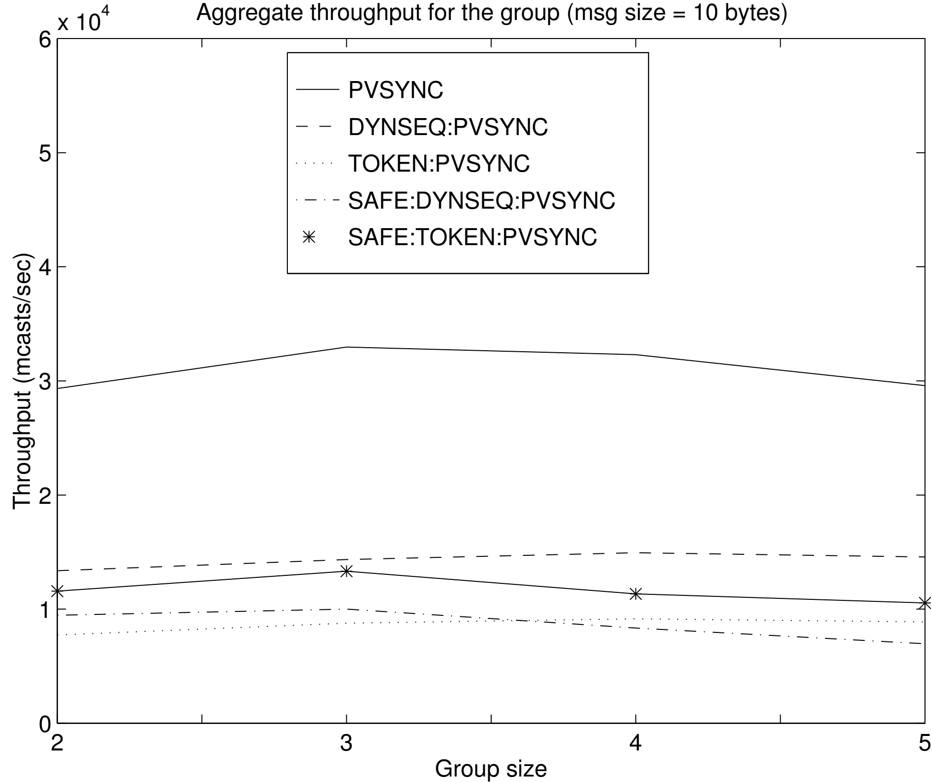

Figure 4.20: Average throughput for the whole group for different group sizes

denoted as `DYNSEQ:PVSYNC` and `TOKEN:PVSYNC` respectively. Similarly, the two variations of protocol stacks which implement globally safe message delivery (with both the primary views layer and the globally safe delivery layer included) are denoted as `SAFE:DYNSEQ:PVSYNC` and `SAFE:TOKEN:PVSYNC`, corresponding to the total ordering protocols being used.

4.6 Discussion

We have described our implementation of state machine replication for distributed objects, which can tolerate network partitions and will be able to make progress whenever a majority of group members can form a stable view. Our solution is based on an integration of the group protocols provided in Horus, the Maestro tools, and the application. The protocol does not require message logging and exhibits excellent performance characteristics. One of the reasons of good performance is *optimistic ordering of messages* within a primary view, even when globally safe delivery is enabled. With our protocol, messages are delivered with fewer communication rounds than, for example, in the more conservative global total ordering protocol of [Kei94]. As a tradeoff, optimistically ordered messages may

need to be aborted in certain partitioning scenarios, which would never happen in the solution of [Kei94]. We therefore target our protocols for high-performance applications running in environments where real and virtual (logical) partitions are possible but unlikely and where there is a meaningful way to handle aborted multicast messages.

The primary views protocol used in our implementation requires a majority of the entire group in order to get a quorum and install a view as primary. An alternative approach, called *dynamic voting*, computes the quorum as a majority of members in the *immediately preceding primary view*, rather than a majority of the whole group [Bir96]. With a dynamic-voting-based quorum, it can be easier to form a primary view, since a majority of the previous primary view may be smaller than a majority of the entire group. On the other hand, with dynamic voting, the primary view can become small enough (namely, to include only two members) so that a single crash failure will block the group forever. It is possible of course to require a certain minimum view size in order to install the view as primary, however this solution eliminates the advantage of dynamic voting against the group-majority-based protocol. Indeed, in order to tolerate n simultaneous crash failures, a modified dynamic-voting-based protocol would install a view as primary only if it contained a majority of the previous primary view *and* included at least $2n + 1$ members. With the group size of $2n + 1$, the availability provided by this protocol is in fact *lower* than the availability of the simple majority-based protocol which, too, will tolerate n simultaneous crashes when the group size is $2n + 1$. Increasing the size of the group will improve availability of the dynamic-voting-based protocol but degrade its performance. Since the minimum group size required by both protocols in order to tolerate n crashes is $2n + 1$, it follows that the majority-based protocol performs better.

The limitations of distributed commit protocols (the Two Phase Commit and Three Phase Commit) discussed in the introduction to this chapter served as a motivation for suggesting a partition-tolerant state machine replication paradigm as an alternative. It is clear, however, that state machine replication is not simply an improvement of previously developed distributed commit protocols, since it assumes a somewhat different programming model. The protocols which implement distributed transactions usually include the solicitation of votes from participating members ("OK to commit?") as the first phase. The solicitation of votes is necessary since applications running at object replicas are often not deterministic, so the coordinator of a transaction cannot assume *a priori* that all participants will be able to commit just because it itself can do it. The state machine replication model is different, however, since it assumes that all group members are running identical deterministic programs. Consequently, for any message sent to the group, it will always be the case that either all or none of the participating objects will be able to perform the action triggered by the delivery of that message. This observation lets us eliminate the solicitation-of-votes phase and consider each multicast as a

complete one-message transaction. The delivery of a message by a group member (which is essentially performed with a *one-phase protocol*) is then equivalent to a local commit action. A transaction of this type can still be aborted, however, if the multicast is canceled in a certain group partitioning scenario, as discussed earlier.

In terms of availability, the state machine replication approach clearly outperforms distributed commit protocols, since a multicast may be blocked (without either delivering or aborting) only when and while the sender is partitioned away from a majority of group members. As long as a majority of group members can communicate, they will be able to make progress. For comparison, recall that with distributed commit protocols, the crash or partitioning-away of a single object replica can block the *entire group* until the failed member is restarted or the link failure is healed. These considerations make it clear that state machine replication is a preferred paradigm for applications that require both global consistency and high availability, expect high performance, and must tolerate object crashes and network partitions. In practice, however, this solution is usually best applied not directly to existing applications (which are often not deterministic) but to application management/control infrastructures, which in many cases can indeed be naturally modeled as replicated state machines. An example of a successful use of state machine replication in such a setting is described in [FB96].

In conclusion, we want to put our implementation of partition-tolerant replicated state machines in the perspective of the famous distributed-consensus-impossibility result of [FLP85]. As shown in [FLP85], a protocol that implements distributed consensus in an asynchronous system may not guarantee termination if crash failures are possible. This limitation obviously applies to our protocols, which guarantee progress in a globally consistent replicated state machine execution only as long as a majority of group members can communicate with each other and form a stable view. Observe that stability of views directly depends on accuracy of failure detection. In an asynchronous system, typical timeout-based failure detectors are inherently inaccurate. However, the impossibility result of [FLP85] can be rendered irrelevant by strengthening the system model [CT93]. This is indeed usually done in practice, since even though group protocols themselves are asynchronous, the failure detection (a mechanism orthogonal to protocols themselves) is often based on specific timing assumptions and its accuracy is quite predictable. In particular, when network performance is within expected bounds (in terms of latency, bandwidth, message loss rate etc.), group members are usually able to form stable views and make progress. The accuracy of failure detection and progress guarantees can further be quantified probabilistically [SM87,MSS96]. From practical perspective, probabilistic real-time performability guarantees on a group execution are much more relevant than formal yet somewhat remote from reality consensus impossibility results.

Bibliography

[ADMSM94] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication Using Group Communication. Technical Report CS94-20, Institute of Computer Science, the Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[Ami95] Y. Amir. *Replication Using Group Communication Over a Partitioned Network*. Ph.D. dissertation, Institute of Computer Science, the Hebrew University of Jerusalem, 1995.

[AMMS+93] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. In *Proc. of the 13th International Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[BDGS95] Ö. Babaoğlu, R. Davoli, L. Giachini, and P. Sabattini. The Inherent Cost of Strong-Partial View-Synchronous Communication. Technical Report UBLCS–95–11, Department of Computer Science, University of Bologna, April 1995.

[Bir96] K. P. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Company and Prentice Hall, December 1996.

[BJ87a] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, December 1987.

[BJ87b] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[Cla98] Tim Clark. Private communication, April 1998.

[CT93] T. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. *Journal of the ACM*, 1993.

[FB96] R. Friedman and K. Birman. Using Group Communication Technology to Develop a Reliable and Scalable Distributed IN Coprocessor. In *Proc. of the TINA 96 Conference*, pages 25–41, September 1996.

[FGS98] P. Felber, R. Guerraoui, and A. Schiper. The Implementation of a CORBA Object Group Service. *Theory and Practice of Object Systems*, Vol. 4(2) 1998.

[FLP85] M. Fischer, N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FV97] R. Friedman and A. Vaysburd. Fast Replicated State Machines Over Partitionable Networks. In *Proc. of the IEEE 16th International Symposium on Reliable Distributed Systems*, October 1997.

[FvR95a] R. Friedman and R. van Renesse. Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. Technical Report TR95-1527, Department of Computer Science, Cornell University, July 1995. Submitted for publication.

[FvR95b] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95–1491, Department of Computer Science, Cornell University, March 1995.

[GR97] K. Guo and L. Rodrigues. Dynamic Light-Weight Groups. In *17th IEEE International Conference on Distributed Computing Systems, p.33-42*, Baltimore, Maryland, May 1997.

[Hay97] Mark Hayden. *The Ensemble System*. Ph.D. dissertation, Department of Computer Science, Cornell University, Forthcoming, Dec. 1997.

[II94] IONA and Isis. An Introduction to Orbix+Isis. IONA Technologies and Isis Distributed Systems, 1994.

[ION98] IONA. Orbix. IONA Technologies, http://www.iona.com/Products/Orbix/, 1998.

[Isi92] Isis. The Isis Distributed Tolkit Version 3.0, User Reference Manual. Isis Distributed Systems, Inc., 1992.

[Isi94] Isis. Reliable Distributed Objects for C++. User's Guide. Isis Distributed Systems, Inc., April 1994.

[Kei94] I. Keidar. A Highly Available Paradigm for Consistent Object Replication. Master's thesis, Institute of Computer Science, the Hebrew University of Jerusalem, 1994.

[KTHB89] F. Kaashoek, A. Tanenbaum, S. Hummel, and H. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[LM97] S. Landis and S. Maffeis. Building Reliable Distributed Systems with CORBA. In *Theory and Practice of Object Systems*, John Wiley and Sons, 1997.

[Maf95a] Silvano Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *Proc. of the 1995 USENIX Conference on Object-Oriented Technologies*, Monterey, CA, June 1995.

[Maf95b] Silvano Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. Ph.D. dissertation, University of Zurich, Switzerland, 1995.

[Maf96] S. Maffeis. A Fault-Tolerant CORBA Name Server. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, Niagara-on-the-Lake, Canada, October 1996.

[MAMSA94] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended Virtual Synchrony. In *Proc. of the 14 International Conference on distributed Computing Systems*, June 1994.

[MFSW95] C. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A Tollkit for Building Fault-Tolerant Distributed Application in Large Scale. Technical report, Department d'Informatique, Ecole Polytechnique Federale de Lausanne, July 1995.

[Mic98] Microsoft. Component Object Model. Microsoft Corporation, http://www.microsoft.com/activex/, 1998.

[MMSA93] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Asynchronous Fault-Tolerant Total Ordering Algorithm. *SIAM Journal of Computing*, 22(4):727–750, August 1993.

[MMSA+95] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos, and T. P. Archambault. The Totem System. In *Proc. of the 25th Annual International Symposium on Fault-Tolerant Computing*, pages 61–66, Pasadena, CA, June 1995.

[MMSA+96] L. Moser, P. M. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.

[MMSN97] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. The Eternal System. In *Workshop on Dependable Distributed Object Systems, OOPSLA'97*, Atlanta, Georgia, October 1997.

[MSS96] L. Malhis, W. Sanders, and R. Schlichting. Numerical Performability Evaluation of a Group Multicast Protocol. *Distributed Systems Engineering, Special Issue on Performance Modelling (ed. Peter G. Harrison), vol. 3, no. 1, pp. 39-52*, March 1996.

[NMMS97a] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance. In *Third USENIX Conference on Object-Oriented Technologies and Systems*, Portland, Oregon, June 1997.

[NMMS97b] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. The Interception Approach to Reliable Distributed CORBA Objects. In *Panel on Reliable Distributed Objects, Third USENIX Conference on Object-Oriented Technologies and Systems*, Portland, Oregon, June 1997.

[OMG97] OMG. CORBA/IIOP 2.1 Specification. Object Management Group, http://www.omg.org/corba/corbiiop.htm, 1997.

[RGS+96] L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Verisimo, and K. Birman. A Transparent Light-Weight Group Service. In *15th IEEE Symposium on Reliable Distributed Systems, p.130-139*, Niagara-on-the-Lake, Canada, October 1996.

[Sch84] Fred B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems 2:2, pp.145-154*, May 1984.

[Sch86] Fred B. Schneider. The state machine approach: a tutorial. Technical Report TR 86-800, Department of Computer Science, Cornell University, December 1986. Revised June 1987.

[Ske85] D. Skeen. Determining the Last Process to Fail. *ACM Transactions on Computer Systems, Vol. 3:1, pp.15-30*, February 1985.

[SM87] W. Sanders and J. Meyer. Performability Evaluation of Distributed Systems Using Stochastic Activity Networks. In *Proceedings of the*

International Workshop on Petri Nets and Performance Models, pp. 111-120, Madison, WI, August 1987.

[VB97] A. Vaysburd and K. Birman. Building Reliable Adaptive Distributed Objects with the Maestro Tools. In *Workshop on Dependable Distributed Object Systems, OOPSLA'97*, Atlanta, Georgia, October 1997.

[Vis97] Visigenic. VisiBroker. Visigenic Software, http://www.visigenic.com/prod/, 1997.

[Vit98] Vitria. Vitria Velocity. Vitria Technology, Inc., http://www.vitria.com, 1998.

[vRBM96] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[Wan97] Yi-Min Wang. Private communication, November 1997.

[ZBS97] J. Zinky, D. Bakken, and R. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, April 1997.