In the memory of my father Si’Hmanou Tari
To my mother Takh’lit Madaoui

In the memory of my role model, my father,
Abdussalam Ali Bukhres
# Contents

**FOREWORD** xv

**PREFACE** xix

**ACKNOWLEDGMENTS** xxv

**ACRONYMS** xxvii

**PART I  BASICS OF CORBA** 1

1. **Introduction to Distributed Systems** 3
   1.1 Basics of Distributed Systems 3
       1.1.1 Architectures 4
       1.1.2 Characteristics 8
       1.1.3 Advantages and Disadvantages 11
   1.2 Distributed System Technologies 12
       1.2.1 Socket 13
       1.2.2 Remote Procedure Call 17
       1.2.3 Remote Method Invocation 21
       1.2.4 Distributed Computing Environment 23
       1.2.5 Distributed Component Object Model 25
   1.3 Summary 30
   1.4 Review Questions 31
   1.5 Exercises 31

2. **Introduction to CORBA** 32
   2.1 Overall Picture 32
   2.2 CORBA 1, CORBA 2, and CORBA 3 35
2.3 Object Management Group 36
  2.3.1 Reference Object Model 36
  2.3.2 Object Management Architecture 43

2.4 Common Object Request Broker Architecture 46
  2.4.1 ORB Core 48
  2.4.2 Interface Definition Language 49
  2.4.3 Interface and Implementation Repositories 52
  2.4.4 Object Adaptors 54
  2.4.5 CORBA Interoperability 56

2.5 CORBA Binding 57
  2.5.1 Binding of Transient IORs 57
  2.5.2 Binding Persistent IORs 58

2.6 CORBA and Existing Technologies 60
  2.6.1 DCE vs. CORBA 60
  2.6.2 DCOM vs. CORBA 61
  2.6.3 RMI vs. CORBA 62

2.7 Summary 62

2.8 Review Questions 63

2.9 Exercises 63

3. CORBA Programming 64

3.1 Overall Picture 64

3.2 Basic CORBA Programming 67
  3.2.1 Interface Definition Language 67
  3.2.2 Static Invocation Interface 85
  3.2.3 Static Skeleton Interface 90

3.3 Dynamic Types 102
  3.3.1 TypeCode 102
  3.3.2 Type Any 103

3.4 Advanced CORBA Programming 106
  3.4.1 Dynamic Invocation Interface 106
  3.4.2 Dynamic Skeleton Interface 112
  3.4.3 Interface and Implementation Repositories 118

3.5 Summary 124

3.6 Review Questions 125
3.7 Exercises 125

PART II  ADVANCED CORBA 127

4. Object Adaptors 129

4.1 Overall Picture 129

4.2 Architectures 131
  4.2.1 Basic Object Adaptor 132
  4.2.2 Portable Object Adaptor 133

4.3 Technical Issues 138
  4.3.1 Overview 138
  4.3.2 Basic Object Adaptor 141
  4.3.3 Portable Object Adaptor 145

4.4 Database Adaptors 151

4.5 Summary 160

4.6 Review Questions 161

4.7 Exercises 161

5. CORBA Interoperability 162

5.1 Overall Picture 162

5.2 Domain 163

5.3 Bridge 164

5.4 Interoperability Protocols 166
  5.4.1 Internet Inter-ORB Protocol 177
  5.4.2 Environment Specific Inter-ORB Protocol 178

5.5 Interoperable Object Reference 179

5.6 Summary 181

5.7 Review Questions 181

5.8 Exercises 182

6. CORBA Caching 183

6.1 Overall Picture 183

6.2 Caching Issues and Techniques 185

6.3 Cache Replacement 187
8.3 Architecture 246
  8.3.1 Trader Components 249
  8.3.2 Service Type Repository 259
  8.3.3 Dynamic Property Evaluation 262
8.4 Constraints, Policies, and Preferences 263
8.5 Query Propagation 266
  8.5.1 CORBA Query Routing 266
  8.5.2 Semantic-based Query Routing 268
8.6 TOS Implementations 270
  8.6.1 JTrader 270
  8.6.2 TAO Trader 271
  8.6.3 DOK Trader 272
8.7 Summary 274
8.8 Review Questions 275
8.9 Exercises 275

9. Event Service 277
  9.1 Overall Picture 277
  9.2 Push and Pull Models 280
    9.2.1 DCE Event Management Service (XEMS) [81] 282
  9.3 Architecture 284
    9.3.1 Canonical Pull Model 286
  9.4 Untyped Event Channel 287
    9.4.1 Design Aspects 287
    9.4.2 Implementation Aspects 290
  9.5 Typed Event Channel 298
    9.5.1 Design Aspects 298
    9.5.2 Implementation Aspects 302
  9.6 CORBA Event Service Implementations 307
    9.6.1 OrbixTalk 307
    9.6.2 TAOs Event Service 308
    9.6.3 Electra 309
    9.6.4 CyberBus 310
    9.6.5 Orbix+ISIS 311
  9.7 Discussion on QoS Issues 312
9.8 Summary 314
9.9 Review Questions 314
9.10 Exercises 315

10. Object Transaction Service 316
10.1 Overview 316
10.2 Basics of Transactions 317
   10.2.1 Concepts 317
   10.2.2 Concurrency Control Protocols 320
   10.2.3 Commit Protocols 323
10.3 OTS Concepts 326
   10.3.1 Interface Hierarchy 326
   10.3.2 Context Propagation 328
   10.3.3 Transactional Objects 329
   10.3.4 Recoverable Objects 330
   10.3.5 Transactional and Recoverable Servers 332
10.4 Transaction Scenario 333
10.5 OTS Implementations 336
   10.5.1 Iona Transaction Service 336
   10.5.2 Microsoft Transaction Service 338
10.6 Summary 340
10.7 Review Questions 340
10.8 Exercises 341

11. Object Query Service 342
11.1 Overall Picture 342
11.2 Background on Query Processing 345
   11.2.1 Overview 345
   11.2.2 Execution Strategies 347
   11.2.3 Query Architecture 351
   11.2.4 Join Operators 352
11.3 OQS Languages 354
   11.3.1 Object Query Language 354
   11.3.2 OQL Algebra 356
11.4 OQS Components
    11.4.1 QueryEvaluator 360
    11.4.2 Collections and Iterator 361
    11.4.3 QueryableCollection 363
    11.4.4 QueryManager and Query Object 363

11.5 DOK Query Service
    11.5.1 Query Execution Engine 365
    11.5.2 Query Optimization 366

11.6 Summary 367
11.7 Review Questions 368
11.8 Exercises 369

REFERENCES 371

INDEX 377
Foreword

Innovations have been occurring at a predictable rate in certain technology domains for many years. For example, Moore’s Law—where the capacity of general-purpose computer chips has doubled every 18 months—is still going strong after three decades. More recently, the speed of IP networks has been improving at an even faster rate—known as Metcalf’s Law—where bandwidth increases by a factor of ten every two years. At this point there is even a “bandwidth index,” similar to indices that track the price/performance of other commodities, such as petroleum or electricity. The steady advance in these technologies is remarkable and is due in no small part to decades of synergistic research, development, and education by academic, industrial, and government partners around the world.

There are, however, important domains—particularly software-intensive distributed systems in telecommunications, health care, aerospace, and online financial services—that are not improving at the same rate as Moore’s Law or Metcalf’s Law, due to a variety of inherent and accidental complexities, such as partial failures, distributed deadlock, and non-portable programming APIs. Consequently, although computer and network hardware keeps getting smaller, faster, cheaper, and better at a predictable pace, complex distributed software systems seem to get bigger, slower, more expensive, and buggier, and the innovation cycles are hard to predict.

An appropriate metaphor for the challenges of software-intensive distributed systems appears in the movie *Apollo 13*, starring Tom Hanks. After an explosion in the command module forces the crew into the lunar module, the carbon dioxide levels grow dangerously high due to a broken air scrubber. At this crucial moment, a manager at Johnson space center walks into a room full of engineers and scientists sitting around a table and dumps out a bag containing common components—such as toothpaste, Tang, and duct tape—found on the lunar module. He tells the group they’ve got eight hours to take these components and assemble an air scrubber that will fit into the appropriate opening, and if it is not right the first time, everyone is going to die!

Increasingly, developers of complex software-intensive distributed systems—especially large-scale mission-critical “systems of systems”—are facing challenges analogous to those of the Apollo 13 engineers and scientists. In particular, time-to-market pressures and competition for consumers and personnel have created a situation where distributed systems must be developed using a large number of commodity-off-the-shelf (COTS) components, which are not developed in-house.
and whose quality can thus rarely be controlled directly. Yet, just like the Apollo 13 engineers and scientists, we must quickly and robustly master the principles, patterns, and protocols necessary to thrive in a COTS-centric environment because our livelihood—and sometimes even our lives—depend upon our success.

Over the past decade, various techniques and tools have been developed to alleviate many accidental and inherent complexities associated with distributed software systems. Some of the most successful of these techniques and tools center on distributed object computing (DOC) middleware, which resides between applications and the underlying operating systems, protocol stacks, and hardware devices to simplify and coordinate how these components are connected and how they interoperate. Just as communication protocol stacks can be decomposed into multiple layers, so too can DOC middleware be decomposed into the following layers:

- **Infrastructure middleware**, which encapsulates and enhances native OS communication and concurrency mechanisms to create object-oriented (OO) network programming components, such as reactors, acceptor-connectors, monitor objects, active objects, and component configurators. These components help eliminate many tedious, error-prone, and non-portable aspects of developing and maintaining networked applications via low-level OS programming API, such as Sockets or POSIX pthreads. Widely-used examples of infrastructure middleware include Java virtual machines (JVMs) and the ADAPTIVE Communication Environment (ACE).

- **Distribution middleware**, which use and extend the infrastructure middleware to define a higher-level distributed programming model. This programming model defines reusable APIs and components that automate common end-system network programming tasks, such as connection management, (de)marshaling, demultiplexing, end-point and request demultiplexing, and multithreading. Distribution middleware enables distributed applications to be programmed using techniques familiar to developers of standalone applications, i.e., by having clients invoke operations on target objects without concern for their location, programming language, OS platform, communication protocols and interconnects, and hardware. At the heart of distribution middleware are Object Request Brokers (ORBs), such as Microsoft’s Component Object Model (COM)+, Sun’s Java remote Method Invocation (RMI), and the OMG’s Common Object Request Broker Architecture (CORBA), which is a key focus of this book.

- **Common middleware services**, which augment distribution middleware by defining higher-level domain-independent services, such as event notifications, logging, multimedia streaming, persistence, security, global time, real-time scheduling and end-to-end quality of service (QoS), fault tolerance, concurrency control, and transactions. Whereas distribution middleware focuses largely on managing end-system resources in support of an OO distributed programming model, common middleware services focus on managing resources throughout a distributed system. Developers can reuse these services to allocate, schedule, and coordinate global resources and perform common
distribution tasks that would otherwise be implemented in an ad hoc manner within each application.

- **Domain-specific services**, which are tailored to the requirements of particular domains, such as telecommunications, e-commerce, health care, process automation, or aerospace. Unlike the other three OO middleware layers—which provide broadly reusable “horizontal” mechanisms and services—domain-specific services are targeted at vertical markets. Domain-specific services are the least mature of the middleware layers today, due partly to the historical lack of distribution middleware and common middleware service standards, which provide a stable base upon which to create domain-specific services. Since these services embody knowledge of application domains, however, they can significantly increase system quality and decrease the cycle-time and effort required to develop particular types of distributed applications.

As these DOC middleware layers mature they are becoming COTS products that are readily available for purchase or open-source acquisition. COTS DOC middleware has become essential in software development organizations that face stringent time and resource constraints since it helps amortize software life-cycle costs by leveraging previous development expertise and concentrating research efforts that improve quality and performance. Ultimately, this R&D process will result in software-intensive distributed systems that get smaller, faster, cheaper, and better at a predictable pace, just as computer and network hardware do today.

The following factors have helped improve the quality and performance of COTS DOC middleware products during the past decade:

- **Maturation of DOC middleware standards**—DOC middleware standards have matured considerably in recent years. For instance, the OMG has adopted specifications for CORBA that reduce ORB footprint, improve fault tolerant behavior, reserve real-time connection and threading resources, and expose various types of QoS policies to applications.

- **Maturation of DOC middleware patterns and frameworks**—A substantial amount of R&D effort has focused on patterns and frameworks for DOC middleware and applications. As these patterns mature and become instantiated in COTS framework components, they have helped improve the efficiency, scalability, predictability, and flexibility of DOC middleware.

Until recently, however, it has been hard to instructors and students to learn how to use DOC middleware effectively without dedicating substantial time and effort. One problem has been that DOC middleware APIs, capabilities, and best practices have existed largely in the programming folklore, the heads of expert developers, or scattered throughout articles in trade magazines and web sites. Another problem is that existing books on DOC middleware and CORBA are intended as guides for industry practitioners rather than as textbooks for students. Thus, many important theoretical and fundamental distribution issues are not covered in these books.
In a highly competitive information technology economy, educating students to become effective distributed software developers is increasingly important. Premium value and competitive advantage is accruing to individuals, universities, companies, and even countries that can quickly master the principles, patterns, and protocols necessary to integrate COTS middleware to create complex DOC applications that cannot be bought off-the-shelf yet. Success in this endeavor requires close collaboration between researchers and practitioners, which is why I’m delighted that Zahir Tari and Omran Bukhres have written Fundamentals of Distributed Object Systems: The CORBA Perspective to help educate researchers and developers of next-generation information technologies.

This book uses CORBA to illustrate the theory and practice of distribution middleware and many common middleware services, as follows:

- The coverage of CORBA’s distribution middleware is split into two parts: (1) fundamental aspects of the CORBA reference model, such as the CORBA interface definition language (IDL), object references, and standard interoperability protocols and (2) advanced CORBA features, such as portable object adapters, client caching, and enhanced communication protocols. This material provides much more than a rehash of the standard CORBA APIs—it also describes the key technical concepts, underlying theoretical foundations, and common solutions related to challenges encountered when developing and integrating interoperable software.

- The coverage of common middleware services focus on a wide range of CORBA’s objects services, such as the CORBA Naming, Trading, Events, Transaction, and Query services. For most of these services, this book describes the corresponding architectures and basic elements. It also shows how such services can be implemented and presents lessons that can be learned and generalized when developing domain-specific services and distributed applications.

By study, mastering, and applying the material in this book, you’ll be able to design and implement distributed applications more rapidly and effectively.

We are fortunate that Zahir and Omran have found time in their busy professional lives to write an outstanding textbook on DOC middleware and CORBA. If you want thorough coverage of the DOC middleware technologies that are shaping next-generation distributed systems read this book. I’ve learned much from it, and I’m confident that you will too.

Douglas C. Schmidt
University of California, Irvine
Preface

CORBA, the acronym for the Common Object Request Broker Architecture, is the result of a standardization consortium, called OMG (Object Management Group), involving more than six hundred international software companies. OMG aims to produce and set up an architecture and a set of standards for open distributed systems enabling interoperability across different hardware and software vendor platforms.

This book presents the theoretical and technical views of the CORBA technology, including the architecture (the Object Management Architecture), the main technical issues (e.g., adaptor, interoperability, caching) and the different services for the management of heterogeneous and distributed objects (naming, trading, query and transaction management). We also present the technical foundations of the main issues related to the design and implementation of large-scale distributed applications, and give details about how specific parts of a CORBA system can be designed.

This book will be valuable to the reader who is interested in understanding the foundations of the CORBA technology and whose aim is to perform advanced research on one of the technical issues related to such a technology (caching, trading, etc.). Or the reader may just want to learn how to program advanced distributed applications, and therefore the aim is to understand the basic “programming techniques” for building heterogeneous distributed applications. The reader may want to find out as much as possible about the CORBA technology and to get a “technical” inside view of its different aspects (architectural, design etc.).

The eleven chapters of this book provide answers to questions that most people are asking, including our students at RMIT and Purdue University:

• What is CORBA?
• Why do we need CORBA?
• What do we need to know in order to understand the “inside” of CORBA?
• What do we need to know in order to implement distributed applications?

Our aim is to provide detailed and technical answers to these questions. The book is organized into three parts. The first part describes the CORBA basics, such as the foundations of distributed systems, CORBA Architecture details, and CORBA programming. This part provides the first step in understanding all the aspects related to the CORBA technology as well as how to implement distributed applications by
using different CORBA components such as the Static Skeleton Interface, Dynamic Invocation Interface, Dynamic Skeleton Interface, and Interface Repository. The second part covers specific issues related to CORBA, including Object Adaptor, Interoperability, Caching and Load balancing. Finally, the last part describes some important CORBA services, such as naming, trading, transaction and query services.

HOW TO USE THE BOOK

There are many ways to use this book on distributed objects. Basically there are three blocks of chapters, with each referring to a certain level of technicality for CORBA. Block I involves introductory chapters that, as ordered in the figure below, provide a good understanding of the basics of distributed objects. Readers familiar with concepts of distributed systems can skip Chapter 1 and start from Chapter 2 and cover Chapters 3, 5, and 7. These four chapters of the first block cover all the necessary details on the main concepts related to CORBA distributed object systems: the CORBA chapter covers architectural concepts and issues of distributed object systems; Chapter 3 explains how to implement CORBA-based applications; Chapter 5 explains how different distributed object systems communicate with each other, and finally, Chapter 7 explains how objects identified by their “object names” can be used by applications.

After the concepts illustrated in the chapters of Block I are well understood, we suggest readers move to another level of technicality. The chapters of Block II provide a more advanced view of distributed objects.

<table>
<thead>
<tr>
<th>Block I</th>
<th>Chapter 1 (Introduction to Distributed Systems)</th>
<th>(2 Hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Chapter 2 (Introduction to CORBA)</td>
<td>(3 Hours)</td>
</tr>
<tr>
<td></td>
<td>Chapter 3 (CORBA Programming)</td>
<td>(6 Hours)</td>
</tr>
<tr>
<td></td>
<td>Chapter 5 (CORBA Interoperability)</td>
<td>(2 Hours)</td>
</tr>
<tr>
<td></td>
<td>Chapter 7 (Naming Service)</td>
<td>(2 Hours)</td>
</tr>
</tbody>
</table>

| Block II | Chapter 4 (Object Adaptors)                    | (3 Hours) |
|          | Chapter 8 (Trading Object Service)             | (3 Hours) |
|          | Chapter 9 (Event Service)                      | (3 Hours) |

| Block III | Chapter 6 (CORBA Caching)                     | (3 Hours) |
|           | Chapter 10 (Object Transaction Service)        | (4 Hours) |
|           | Chapter 11 (Object Query Service)              | (3 Hours) |
Chapter 4 goes into details on one of the major components of CORBA systems—the Object Adaptors. The remaining chapters of Block II address the issues of service retrieval in large-scale distributed systems as well the communication across heterogeneous distributed systems.

Block III contains chapters that need to be read after the basic and advanced concepts of distributed objects are well understood. They require a deep understanding of different aspects covered in Blocks I and II, including programming aspects and architectural aspects. Chapter 6 shows how distributed object applications can be made efficient by extending client proxies. Chapter 10 covers issues of robustness and reliability of transactions in distributed object applications. Chapter 11 explains how CORBA and Database technologies can be (partially) integrated.

For undergraduate students, all the chapters of Block I will need to be covered in detail. The instructor can add additional chapters from Block II according to the order of how they are listed in the figure. Some chapters of the second and third blocks can be assigned as reading material, for example, Chapters 8, 9, and 6.

For postgraduate students, we suggest covering a few chapters of Block I to be covered in the first two or three lectures. Chapters 2 and 5 can be such chapters. If students are familiar with Java, then some of the implementation concepts of programming can be covered as an assignment (e.g., an assignment covering both DII, DSI and the look up of the Interface Repository, where students will build a complex application using such concepts). The remaining lectures will need to cover the chapters of Block II and Block III.

ORGANIZATION OF THE BOOK

Part I is dedicated to the basics of CORBA and contains technical concepts necessary for a good understanding of distributed systems and distributed object systems. Chapter 1 introduces the main elements of distributed systems and aims at providing readers with appropriate background to better understand the remaining chapters of this book. Detailed descriptions of existing distributed system technologies, such as Socket, Remote Procedure Call, Remote Method Invocation, are also provided. Chapter 2 is the first step into the CORBA world. It draws a general picture of CORBA and describes in more detail the main elements of the CORBA architecture, such as Interface Definition Language (IDL), Object Request Broker (ORB) and Interface and Implementation Repositories. The concept of object binding is presented in the CORBA context, and different approaches supported by existing CORBA-compliant systems are explained, such as the binding of transient object references and the binding of persistent object references. Chapter 3 demonstrates how to design and implement distributed object applications using a Java-based CORBA system, OrbixWeb. It is the only chapter of this book that provides program codes. A step-by-step implementation of a complete application is proposed: first, basic programming techniques are illustrated and later more advanced ones are proposed, such as programming with DII (Dynamic Invocation Interface), programming with DSI (Dynamic Skeleton Interface) and programming with the Interface Repository.
Part II is about advanced topics in CORBA—adaptors, interoperability and caching. CORBA adaptors provide interface flexibility and management. Chapter 4 describes the main issues related to the design of adaptors and later discusses the two architectures: BOA and POA. Both these architectures are explained and compared to each other with regard to a set of criteria, such as object/servant management, object grouping, request redirection and multi-threading. At the end of the chapter, a POA extension to deal with database issues, such as persistence of object references, is given. Chapter 5 explains the inter-ORB communication protocol which is based on the standard IIOP (Internet Inter-ORB Protocol). An ORB can implement additional communication protocols. The structure of IOR (Interoperable Object References), which enables invocations to pass from one ORB to another, is discussed. Chapter 6 is about CORBA performance. It discusses how to make CORBA applications efficient by caching remote objects and therefore make them locally accessible through appropriate proxies. Caching relates to the way proxies (e.g., client proxy and server proxy) and the ORB perform invocations. Chapter 6 describes a specific design for CORBA caching. A FIFO-based removal algorithm is discussed, and this uses a double-linked structure and hash table for eviction. A variation of optimistic two-phase locking for consistency control is proposed. This protocol does not require a lock at the client side by using a per-process caching design.

Part III is about the Common Object Services Specification (COSS). CORBA provided a standard for several distributed services. This part describes some of the important CORBA services, such as the Naming Service, Event Service, Trading Service, Object Transaction Service, and Object Query Service. Chapters 7 and 8 describe the CORBA Naming and Trading services, respectively. These services provide appropriate functionalities to share information available in different servers. Users can browse, retrieve, and update object references by using the different operations of the interface provided by the Naming or Trading Services. Chapter 7 enables a transparent access to objects and offers facilities to retrieve objects based on their names. Chapter 8 provides “matchmaking” services for objects. Each object is recorded with appropriate information and the role of the trader is to find the best match for the client, based on the context of the request service and the offers of the providers. Chapter 8 starts with an illustrative example and shows how a trader can be used to locate services. The different steps of this example are illustrated with JTrader, a Java-based CORBA Trading Service. Later sections describe the elements of the CORBA trader architecture as well as the issues related to efficiency and scalability of CORBA Traders (e.g., clustering, query propagation). Chapter 9 covers the CORBA Event Service, which offers sophisticated communication between ORB objects. This chapter provides details of the following aspects: the event architecture, the different communication semantics (e.g. synchronous, asynchronous, and deferred invocations), and the different communication models (Push/Pull and hybrid models). A few implementations of the CORBA Event Service are described. Chapter 10 is about the CORBA Object Transaction Service (OTS). This service is one of the essential parts of most of distributed systems, including those supporting business applications such as OLTP (On-Line Transaction Processing). This chapter
starts with a background on transactions, including the different transaction models (such flat and nested models), different concurrency control and commit protocols. Later, a detailed discussion on the CORBA OTS is provided, and this covers the description of the different elements of this service. The last section overviews two implementations, one related to the Iona’s product and the other one is the Microsoft counterpart of the OTS, called MTS (Microsoft Transaction Service). Chapter 11, the last chapter of Part III, describes a service which can be qualified as a “data-oriented” service, that is, the Object Query Service. This service enables integration of databases by using a standard query language (such as the ODMG’s Object Query Language, denoted as OQL). The first two sections of this chapter describe the main query processing techniques used in distributed databases. This may help the reader to understand the important issues as well as solutions related the processing of heterogeneous databases. Later sections provide a detailed description of the elements of CORBA’s Object Service.
Acknowledgments

We gratefully acknowledge all the people directly or indirectly involved in the preparation of this book. Many people from RMIT and Purdue helped tremendously. In particular, we would like to thank our former students who have reviewed some sections of this book and made valuable suggestions for improvement. Special thanks go to RMIT students:

- Suryadinata Setiawan, who implemented the CODAR database adaptor, and made contributions to Chapters 1, 2, 3, and 4, as well as reviewed questions, exercises, and answers to both of them for all chapters in the book.
- David Lin, for his work on CORBA caching, including the design of the caching model and the implementation and testing of the model.
- Greg Craske and Kiran, who did extensive work on the CORBA Trading Object Service, and in particular, for the issues of clustering and query routing.
- Sunny Bains for his work on the CORBA Event Service, including the design and implementation of the typed and untyped event channels.

We would like to thank our graduate students Christina Davenport, Srinivasan Sikkupparbathiyam, Marcelo Areal, Eric Lynch, and Lara Atallah for their readings and tremendous feedback. We also would like to thank Frank Oreovicz and Michelle Boshears for their qualified English editing.

The most important support that made this whole book possible came from our families. We would also like to thank our respective employers for providing us with an appropriate environment and their support for this book.

And as always, a special expression of deep and sincere gratitude goes to Allah almighty for all his help and guidance.

Z. Tari
O. Bukhres
<table>
<thead>
<tr>
<th>Acronyms</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOA</td>
<td>Basic Object Adaptor</td>
</tr>
<tr>
<td>COM</td>
<td>Microsoft’s Component Object Model</td>
</tr>
<tr>
<td>CORBA</td>
<td>Object Request Broker Architecture</td>
</tr>
<tr>
<td>COSS</td>
<td>Common Object Services Specification</td>
</tr>
<tr>
<td>DCE</td>
<td>Distributed Computing Environment</td>
</tr>
<tr>
<td>DCOM</td>
<td>Microsoft’s Distributed COM</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>GIOP</td>
<td>General Inter-ORB Protocol</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>IIOP</td>
<td>Internet Inter-ORB Protocol</td>
</tr>
<tr>
<td>LRU</td>
<td>Last Recently Used</td>
</tr>
<tr>
<td>ODMG</td>
<td>Object Database Management Group</td>
</tr>
<tr>
<td>OQS</td>
<td>Object Query Service</td>
</tr>
<tr>
<td>OODBMS</td>
<td>Object Oriented Database Management System</td>
</tr>
<tr>
<td>OMA</td>
<td>Object Management Architecture</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
<tr>
<td>OTS</td>
<td>Object Transaction Service</td>
</tr>
<tr>
<td>02PL</td>
<td>Optimistic 2PL</td>
</tr>
<tr>
<td>2PL</td>
<td>Two Phase Locking</td>
</tr>
<tr>
<td>POA</td>
<td>Portable Object Adaptor</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>STR</td>
<td>Service Type Repository</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transport Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>TOS</td>
<td>Trading Object Service</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>
The advent of computers was motivated by the need to perform complex data calculations and processing quickly. Once their usage to perform these tasks is ubiquitous, the next computing breakthroughs are spurred by the necessity to collaborate with other computers via a network. The earliest solutions are based on a model called centralized systems, in which a single computer with one or multiple CPUs processes all incoming requests. However, reasons such as cost, reliability, and the separate nature of the divisions that makes up organizations using the systems causes this model to be less attractive. Another model, called distributed systems, addresses these issues with its distribution. Instead of having one single powerful computer, distributed systems employ multiple computers communicating to each other via a common network. The independent, distributed, and sometimes heterogeneous nature of these computers also underlies the importance of having a distributed system software to provide a common view of the systems.

This chapter serves as a brief introductory overview of distributed systems. The first section explains the basic ideas of distributed systems. The second section compares different solutions of a distributed system software, called middleware: Sockets, RPC, RMI, DCOM, and CORBA.

1.1 BASICS OF DISTRIBUTED SYSTEMS

A distributed system is a collection of autonomous computers linked by a network and equipped with distributed system software [22]. A distributed system is the opposite of a centralized system, which consists of a single computer with one or multiple powerful CPUs processing all incoming requests. The distributed system software enables the comprising computers to coordinate their activities and to share system resources. A well-developed distributed system software provides the illusion of a single and integrated environment although it is actually implemented by multiple computers in different locations. In other words, the software gives a distribution transparency to the systems.
1.1.1 Architectures

The definition and implementations of distributed systems evolve from the system where remote terminals or minicomputer, independently carrying out some operations and periodically communicating with mainframes in batch mode. Between the late 1970s and the early 1980s, the notion of distributed systems was synonymous with distributed processing, that is, a request processing technique where a request is broken into subtasks which will be processed by multiple machines. During this period, a distributed system was interconnected in either the star, hierarchical, or ring structure. As shown by Figure 1.1, each remote terminal in star structure is connected to the central computer via a modem. Figure 1.2(a) depicts a distributed system with hierarchical structure.

One or more locations have their own minicomputers. Each minicomputer at these locations is connected to a central computer via leased phone lines. The minicomputer periodically sends the required summary data, for example, daily sales, to the central computer. Figure 1.2(b) illustrates a distributed system interconnected in a ring structure consisting of autonomous computers linked in a peer-to-peer fashion. Examples of peers are mainframes, mid-range machines and PCs.

Distributed system types of this period range from functional distribution, centrally controlled, integrated systems, to non-integrated systems. A distributed system with functional distribution has some of its functions distributed, but not the capability to process a complete transaction. The system employs intelligent terminals or controllers to perform message editing, screen formatting, data collection, dialogue with terminal operators, some security functions, and message compaction. A centrally controlled distributed system is viewed as a collection of peripheral small computers, which might be capable to completely process. Each of them is sub-

![Figure 1.1 Star structure.](image-url)
ordinate to a higher level computer in the overall system structure. An integrated
distributed system consists of separate systems which have an integrated design of
the data located in different systems, and possibly of the data in different programs
as well. A non-integrated distributed system comprises independent systems which
are connected by a computer network.

The meaning and implementations of distributed systems started to change in the
period between mid to late 1980s. The idea of distributed systems represents a system
ranging from a separate, geographically dispersed applications cooperating with each
other, to a single application formed by the cooperation of geographically dispersed
but relatively independent, stand-alone, and component programs. A distributed sys-
tem of this period has four types of structural and interconnection configurations.
The first one, which is shown in Figure 1.3(a), is a mainframe connected to personal
computers with certain functions or applications are off-loaded from the mainframe
into the PCs.

Another type is PCs connected to a mid-range machine, which, in turn, is con-
nected to a mainframe. As depicted in Figure 1.3(b), all of them are organized in a
hierarchical structure with one root and as one moves closer toward the root, have an
increasing computing power, function, and possibly control is implied.

Users of these PCs manipulate the mid-range machine to access applications and
data backup of one department. The mid-range delivers data to the mainframe which
handles enterprise level processing. Figure 1.4(a) illustrates the next type of struc-
tural and interconnection configurations.

A distributed system of this type is arranged as a connection of peers which have
different details of interconnection and dependency. However, no clear central point
of control exists in the system. As shown in Figure 1.4(b), the last type is a collec-

---

**Figure 1.2** (a) Hierarchical structure; (b) ring structure.
von peer hierarchies. Each of the mainframe or mid-range machines exhibits a hierarchical configuration and is connected to a collection of PCs.

Distributed systems underwent a final evolution around the late 1980s. As shown in Figure 1.5(a), each site in the early distributed systems of this period contained one or multiple individual software providing access to its resources.

Later, the granularity of distribution control became more fine-grained, enabling functions of a single software to be distributed across the network. Thus, as Figure 1.5(b) illustrates, the current distributed system now consists of autonomous computers which are linked by a network. The software used by this system is divided into multiple components, each residing at a different site.
Figure 1.4  (a) Connection of peers structure and inter-connection configuration; (b) connection of hierarchical peers structure and inter-connection configuration.

Figure 1.5  (a) A distributed system in pre-late 1980s; (b) a distributed system today.
1.1.2 Characteristics

A distributed system has six important characteristics [22]: (1) resource sharing, (2) openness, (3) concurrency, (4) scalability, (5) fault tolerance, and (6) transparency. These characteristics are not automatic consequences of distribution. Instead, they are acquired as a result of a careful design and implementation.

**Resource Sharing**  Resources provided by a computer which is a member of a distributed system can be shared by clients and other members of the system via a network. In order to achieve effective sharing, each resource must be managed by a software that provides interfaces which enables the resource to be manipulated by clients. Resources of a particular type are managed by a software module called resource manager, which performs its job based on a set of management policies and methods.

**Openness**  Openness in distributed systems is the characteristic that determines whether the system is extendible in various ways. This characteristic is measured mainly by the degree to which new resource sharing services can be incorporated without disruption or duplication of existing services. The opposite of an open distributed system is a closed distributed system. The set of features and facilities provided by a closed distributed system stay static overtime. New features and facilities cannot be added into the system. This prevents the system from providing any new resources other than those which are already made available. The openness of a distributed system can be viewed from two perspectives: hardware extensibility and software extensibility. The former is the ability to add hardware from different vendors to a distributed system, while the latter is the ability to add new software or modules from different vendors to a distributed system.

A system is considered being open if its key software interfaces are published, that is, the interfaces are specified, documented, and made available publicly to software developers. This process is similar to the process of standardizing these interfaces since both make the interfaces publicly available. However, the former does not require the interfaces to pass official standardization process before they are made available. An example of more open systems are UNIX systems. Resources of these system are used via system calls, that is, a set of procedures which are made available to programs and other languages that support conventional procedure facilities. UNIX systems are able to handle a newly added type of hardware by adding new systems calls or new parameters to existing system calls. These systems calls are implemented by one module of the UNIX called kernel. However, unless access to the source code is available (like in the case of Linux Operating System (OS)), the kernel is fixed and inextensible. Therefore, the design of the kernel determines the range and the level of supports for different resource types available to UNIX applications. The availability of interprocess communication facilities in UNIX widens the scope for achieving openness. It allows resources that are not accessible through system calls to be manipulated via the interprocess communication facilities instead. It also enables resources on machines with different hardware and software to be used as long as the necessary interprocess communication facilities are available.
Concurrency  Concurrency is the ability to process multiple tasks at the same time. A distributed system comprises multiple computers, each having one or multiple processors. The existence of multiple processors in the computer can be exploited to perform multiple tasks at the same time. This ability is crucial to improve the overall performance of the distributed system. For example, a mainframe must handle requests from multiple users, with each user sending multiple requests at the same time. Without concurrency, performance would suffer, since each request must be processed sequentially. The software used must make sure that an access to the same resource does not conflict with others. All concurrent access must be synchronized to avoid problems such as lost update (two concurrent access update the same data, but one of the updates is lost), dirty read (one access updates the data read by another access, but the former fails and affects the latter), incorrect summary (a set of data is updated by an access while the set is being processed by another access), and unrepeatable read (an access reads data twice, but the data are changed by another access between the two reads) [29].

Scalability  Scalability in distributed systems is the characteristic where a system and application software need not to change when the scale of the system increases. Scalability is important since the amount of requests processed by a distributed system tends to grow, rather than decrease. In order to handle the increase, additional hardware and/or software usually needed. However, this does not mean throwing more hardware and more software into the system would resolve the issue of scalability. In fact, a system which is not scalable does not utilize the additional hardware and software efficiently to process requests. This is because the system is not designed to expand. Such a system will eventually hit its processing capability limits and its performance starts to degrade. On the other hand, a system is said to be scalable if it provides flexibilities to grow in size, but still utilize the extra hardware and software efficiently. The amount of flexibilities the system has determines the level of scalability it provides.

Fault Tolerance  Fault Tolerance in distributed systems is a characteristic where a distributed system provides an appropriately handling of errors that occurred in the system. A system with good fault tolerance mechanisms has a high degree of availability. A distributed system’s availability is a measure of the proportion of time that the system is available for use. A better fault tolerance increases availability. Fault tolerance is achieved by deploying two approaches: hardware redundancy and software recovery. The former is an approach to prevent hardware failures by means of duplication. Although expensive, this technique improves availability since one or multiple dedicated hardware stand ready to take over the request processing task when a failure occurs. For example, an application might use two interconnected computers where one of them is acting as a backup machine in case the other is unable to process requests. Software recovery is an approach where software is designed to recover from faults when they are detected. Even so, a full recovery may not be achieved in some cases. Some processing could be incomplete and their persistent data may not be in a consistent state.
Transparency Transparency is the concealment of the separation of components in a distributed system from the user and the application programmer such that the system is perceived as a whole rather than as collection of independent components. As a distributed system is separated by nature, transparency is needed to hide all unnecessary details regarding this separation from users. The term *information object* is used to denote the entity to which the transparency can be applied to. There are eight forms of transparency in distributed systems. The first one is *access transparency*, which enables local and remote information objects to be accessed using identical operations. The second one is *location transparency*, which enables information objects to be accessed without the knowledge of their location. These first two forms of transparency are also known as *network transparency*. They provide a similar degree of anonymity for resources found in centralized computer systems. Their existence or inexistence have a strong effect on the utilization of a distributed system’s resources. The third transparency is *concurrency transparency*, which enables several processes to operate concurrently using shared information objects without interference between them. *Replication transparency* enables multiple information objects to be used to increase reliability and performance without knowledge of the replicas by users or application programs. *Failure transparency* hides faults and allows users and application programs to complete their tasks even though a hardware or software failure occurs. *Migration transparency* permits information objects to be moved within a system without affecting the operation of users or application programs. *Performance transparency* permits a distributed system to be reconfigured to improve performance as the load fluctuates. Finally, *scaling transparency* permits the system and application to expand in scale without changing the system structure and application algorithms. For example, consider an e-mail address of amazigh@cs.rmit.edu.au. Users need not to know about the physical address of machine that must be contacted to deliver an e-mail to this address, nor do they need to know how the e-mail is actually sent. Thus, an e-mail address has both location and access transparency, that is, network transparency.

The presence or absence of the resource sharing, open, and transparent characteristics explained above influences how heterogeneities of a distributed system are addressed. The existence of resource sharing increases the need for the system to be more open. This is because sharable resources of this system are almost certainly made up of hardware and software from different vendors. Without the openness, these resources cannot be used by clients if they are based on technologies from vendors different from those resources. Another problem is that hardware and software of various vendors will not be able to be incorporated into the system, especially in cases where legacy systems exist. Having an openness in the system facilitates the mix-and-match of hardware and software. This allows the system to take advantage of the best features from different products, regardless who their vendors are. The last characteristic that determines a distributed system’s approach to its heterogeneities is transparency. If the transparency is not available, clients will be exposed to the complexities of multiple technologies underlying the system. As a result, the system becomes harder to use and requires a lot of training. This could reduce, even eliminate the lure of the distributed systems model completely.
1.1.3 Advantages and Disadvantages

The trend of distributed systems is motivated by the potential benefits that they could yield. These benefits are [83][101]:

- **Shareability**: Shareability in distributed systems is the ability that allows the comprising systems to use each other’s resources. This sharing takes place on a computer network connected to each system, using a common protocol that governs communications among the systems. Both the network and the protocol are respectively the common communication medium and the common protocol that facilitate sharing. The Internet is a good example of a distributed system. Each computer that wishes to use and/or share resources must be connected to the network and understand TCP/IP.

- **Expandability**: Expandability of a distributed system is the ability that permits new systems to be added as members of the overall system. Foreseeing all resources that will ever be provided is often not feasible. It also influences the ability to determine what level of processing power the host machines must have. A distributed system might end up providing unused resources on machines with under-capacity utilization. Such waste of time and money is resolved by giving the freedom to add shared resources only when they are really needed.

- **Local Autonomy**: A distributed system is responsible to manage its resources. In other words, it gives its systems a local autonomy of their resources. Each system can apply local policies, settings, or access controls to these resources and services. This makes distributed systems ideal for organizations whose structure consists of independent entities located in different locations. For example, multinational companies might have their systems scattered in different locations, each managing the affairs of a particular branch.

- **Improved Performance**: As the number of clients accessing a resource increases, the response time starts to degrade. The conventional ways of maintaining the response time, for example, upgrading the host machine, can be used to offset this effect. This is further improved with techniques such as replication, which allows the same resources to be copied, and load balancing, which distributes access requests among these copies. The separate nature of distributed systems is also helpful since resources exist in different machines. Requests for these resources are sent to different machines, making the request processing to be naturally distributed. Finally, the number of computers in a distributed system benefits the system, in terms of its processing power. This is because the combined processing power of multiple computers provides much more processing power than a centralized system. The limit of which a single computer can be installed with multiple CPUs prohibit an endless increase of its processing power.

- **Improved Reliability and Availability**: Disruptions to a distributed system do not stop the system as a whole from providing its resources. Some resources may not be available, but others are still accessible. This is because these re-
sources are spread across multiple computers where each resource is managed by one computer. If these resources are replicated, the disruption might cause only minimum impact on the system. This is because requests can be diverted to other copies of the target resources.

• **Potential Cost Reductions:** The first advantage of distributed systems over centralized systems is cost effectiveness. As Grosch’s law states, the computing power of a single CPU is proportional to the square of its price. Thus, one CPU X, which has four times the performance of one CPU Y, can be acquired at twice the cost of CPU Y. However, this law became invalid when microprocessors were introduced. Paying double price yields only a slightly higher speed, not four times of the CPU performance. Therefore, rather than paying more for a single CPU, the most cost-effective way of achieving a better price to performance ratio is to harness a large number of CPUs to process requests. Another potential cost reduction occurs when a distributed system is used to handle request processing shared of multiple organizations. All of these organizations could contribute to the setup and maintenance costs. This reduces the per organization costs down compared to setting and maintaining the system independently.

Beside these advantages a distributed system has the following disadvantages:

• **Network Reliance:** Because all computers in a distributed system rely on a network to communicate to each other, problems on the network would disrupt activities in the system as a whole. This is true especially for physical problems such as broken network cables, routers, bridges, etc. The cost of setting up and maintaining the network could eventually outweigh any potential cost savings offered by distributed systems.

• **Complexities:** A distributed system software is not easy to develop. It must be able to deal with errors that could occur from all computers that make up the distributed system. It must also capable to manipulate resources of computers with a wide range of heterogeneities.

• **Security:** A distributed system allows its computers to collaborate and share resources more easily. However, this convenience of access could be a problem if no proper security mechanism are put in place. Private resources would be exposed to a wider range of potential hackers, with unauthorized accesses launched from any computers connected to the system. In fact, a centralized system is usually more secure than a distributed system.

### 1.2 DISTRIBUTED SYSTEM TECHNOLOGIES

*Middleware* in distributed systems is a type of distributed system software which connects different kinds of applications and provides distribution transparency to its connected applications. It is used to bridge heterogeneities that occurred in the