REACTIVE MESSAGING PATTERNS

with the

ACTOR MODEL

APPLICATIONS AND INTEGRATION IN SCALA AND AKKA

VAUGHN VERNON

Foreword by Jonas Bonér, Founder of the Akka Project

FREE SAMPLE CHAPTER

SHARE WITH OTHERS
Reactive Messaging Patterns with the Actor Model
This page intentionally left blank
Reactive Messaging Patterns with the Actor Model
Applications and Integration in Scala and Akka

Vaughn Vernon
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data
Vernon, Vaughn.
   Reactive messaging patterns with the Actor model : applications and integration in Scala and Akka / Vaughn Vernon.
   pages cm
   Includes bibliographical references and index.
QA76.73.S28V47 2016
005.2'762—dc23
2015016389

Copyright © 2016 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, 200 Old Tappan Road, Old Tappan, New Jersey 07675, or you may fax your request to (201) 236-3290.

ISBN-10: 0-13-384683-0

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

First printing, July 2015
To my dearest Nicole and Tristan.
Your continued love and support are uplifting.
This page intentionally left blank
Programming with Scala ................................................. 29
  A Condensed Scala Tutorial ..................................... 30
Programming with Akka .................................................. 43
  Actor System ......................................................... 44
  Supervision .......................................................... 55
  Remoting .............................................................. 59
  Clustering ............................................................. 71
  Testing Actors ....................................................... 99
  The CompletableApp ............................................... 102
Summary ...................................................................... 104

Chapter 3 Performance Bent ........................................... 107
  Transistors Matter .................................................... 107
  Clock Speed Matters ............................................... 109
  Cores and Cache Matter .......................................... 111
  Scale Matters ......................................................... 112
  Multithreading Is Hard ............................................ 116
  How the Actor Model Helps ...................................... 122
    Dealing with False Sharing .................................... 124
  The Patterns ............................................................. 126

Chapter 4 Messaging with Actors ...................................... 127
  Message Channel .................................................... 128
  Message ................................................................. 130
  Pipes and Filters ..................................................... 135
  Message Router ....................................................... 140
  Message Translator ................................................ 143
  Message Endpoint ................................................... 145
Summary ...................................................................... 147

Chapter 5 Messaging Channels ........................................ 149
  Point-to-Point Channel ............................................ 151
  Publish-Subscribe Channel ....................................... 154
    Local Event Stream .................................................. 155
    Distributed Publish-Subscribe ................................. 160
  Datatype Channel .................................................... 167
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Invalid Message Channel</strong></td>
<td>170</td>
</tr>
<tr>
<td></td>
<td><strong>Dead Letter Channel</strong></td>
<td>172</td>
</tr>
<tr>
<td></td>
<td><strong>Guaranteed Delivery</strong></td>
<td>175</td>
</tr>
<tr>
<td></td>
<td><strong>Channel Adapter</strong></td>
<td>183</td>
</tr>
<tr>
<td></td>
<td><strong>Message Bridge</strong></td>
<td>185</td>
</tr>
<tr>
<td></td>
<td><strong>Message Bus</strong></td>
<td>192</td>
</tr>
<tr>
<td></td>
<td><strong>Summary</strong></td>
<td>200</td>
</tr>
<tr>
<td>Chapter 6</td>
<td><strong>Message Construction</strong></td>
<td>201</td>
</tr>
<tr>
<td></td>
<td><strong>Command Message</strong></td>
<td>202</td>
</tr>
<tr>
<td></td>
<td><strong>Document Message</strong></td>
<td>204</td>
</tr>
<tr>
<td></td>
<td><strong>Managing Flow and Process</strong></td>
<td>206</td>
</tr>
<tr>
<td></td>
<td><strong>Event Message</strong></td>
<td>207</td>
</tr>
<tr>
<td></td>
<td><strong>Request-Reply</strong></td>
<td>209</td>
</tr>
<tr>
<td></td>
<td><strong>Return Address</strong></td>
<td>211</td>
</tr>
<tr>
<td></td>
<td><strong>Correlation Identifier</strong></td>
<td>215</td>
</tr>
<tr>
<td></td>
<td><strong>Message Sequence</strong></td>
<td>217</td>
</tr>
<tr>
<td></td>
<td><strong>Message Expiration</strong></td>
<td>218</td>
</tr>
<tr>
<td></td>
<td><strong>Format Indicator</strong></td>
<td>222</td>
</tr>
<tr>
<td></td>
<td><strong>Summary</strong></td>
<td>226</td>
</tr>
<tr>
<td>Chapter 7</td>
<td><strong>Message Routing</strong></td>
<td>227</td>
</tr>
<tr>
<td></td>
<td><strong>Content-Based Router</strong></td>
<td>228</td>
</tr>
<tr>
<td></td>
<td><strong>Message Filter</strong></td>
<td>232</td>
</tr>
<tr>
<td></td>
<td><strong>Dynamic Router</strong></td>
<td>237</td>
</tr>
<tr>
<td></td>
<td><strong>Recipient List</strong></td>
<td>245</td>
</tr>
<tr>
<td></td>
<td><strong>Splitter</strong></td>
<td>254</td>
</tr>
<tr>
<td></td>
<td><strong>Aggregator</strong></td>
<td>257</td>
</tr>
<tr>
<td></td>
<td><strong>Resequencer</strong></td>
<td>264</td>
</tr>
<tr>
<td></td>
<td><strong>Composed Message Processor</strong></td>
<td>270</td>
</tr>
<tr>
<td></td>
<td><strong>Scatter-Gather</strong></td>
<td>272</td>
</tr>
<tr>
<td></td>
<td><strong>Routing Slip</strong></td>
<td>285</td>
</tr>
<tr>
<td></td>
<td><strong>Process Manager</strong></td>
<td>292</td>
</tr>
<tr>
<td></td>
<td><strong>Message Broker</strong></td>
<td>308</td>
</tr>
<tr>
<td></td>
<td><strong>Summary</strong></td>
<td>310</td>
</tr>
</tbody>
</table>
Contents

Chapter 8 Message Transformation ................................................. 313
  Envelope Wrapper .............................................................. 314
  Content Enricher .............................................................. 317
    Immutable DoctorVisitCompleted ....................................... 320
    Should the AccountingEnricherDispatcher Be Local? ............. 321
  Content Filter ............................................................... 321
  Claim Check ................................................................. 325
  Normalizer ................................................................. 332
  Canonical Message Model .................................................. 333
    Actor Systems Require a Canon ....................................... 335
  Summary ................................................................. 336

Chapter 9 Message Endpoints ................................................... 337
  Messaging Gateway .......................................................... 338
  Messaging Mapper .......................................................... 344
  Transactional Client/Actor .................................................. 351
    Transactional Client ....................................................... 353
    Transactional Actor ....................................................... 354
  Polling Consumer ........................................................... 362
    Resource Polling .......................................................... 367
  Event-Driven Consumer ..................................................... 371
  Competing Consumers ....................................................... 371
  Message Dispatcher .......................................................... 374
  Selective Consumer .......................................................... 377
  Durable Subscriber .......................................................... 379
  Idempotent Receiver .......................................................... 382
    Message De-duplication .................................................... 383
    Design Messages with Identical Impact ............................... 384
    State Transition Renders Duplicates Harmless ...................... 384
  Service Activator ............................................................ 390
  Summary ................................................................. 391

Chapter 10 System Management and Infrastructure ......................... 393
  Control Bus ................................................................. 394
  Detour ................................................................. 395
  Wire Tap ................................................................. 397
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Metadata/History</td>
<td>398</td>
</tr>
<tr>
<td>Message Journal/Store</td>
<td>402</td>
</tr>
<tr>
<td>Smart Proxy</td>
<td>406</td>
</tr>
<tr>
<td>Test Message</td>
<td>411</td>
</tr>
<tr>
<td>Channel Purger</td>
<td>414</td>
</tr>
<tr>
<td>Summary</td>
<td>416</td>
</tr>
<tr>
<td>Appendix A Dotsero: An Akka-like Toolkit for .NET</td>
<td>417</td>
</tr>
<tr>
<td>Dotsero Actor System</td>
<td>417</td>
</tr>
<tr>
<td>Actors Using C# and .NET</td>
<td>420</td>
</tr>
<tr>
<td>Dotsero Implementation</td>
<td>425</td>
</tr>
<tr>
<td>Summary</td>
<td>427</td>
</tr>
<tr>
<td>Bibliography</td>
<td>429</td>
</tr>
<tr>
<td>Index</td>
<td>433</td>
</tr>
</tbody>
</table>
This page intentionally left blank
Foreword

When Carl Hewitt invented the Actor model in the early 1970s he was way ahead of his time. Through the idea of actors he defined a computational model embracing nondeterminism (assuming all communication being asynchronous), which enabled concurrency and, together with the concept of stable addresses to stateful isolated processes, allowed actors to be decoupled in both time and space, supporting distribution and mobility.

Today the world has caught up with Hewitt’s visionary thinking; multicore processors, cloud computing, mobile devices, and the Internet of Things are the norm. This has fundamentally changed our industry, and the need for a solid foundation to model concurrent and distributed processes is greater than ever. I believe that the Actor model can provide the firm ground we so desperately need in order to build complex distributed systems that are up for the job of addressing today’s challenge of adhering to the reactive principles of being responsive, resilient, and elastic. This is the reason I created Akka: to put the power of the Actor model into the hands of the regular developer.

I’m really excited about Vaughn’s book. It provides a much-needed bridge between actors and traditional enterprise messaging and puts actors into the context of building reactive systems. I like its approach of relying only on the fundamentals in Akka—the Actor model and not its high-level libraries—as the foundation for explaining and implementing high-level messaging and communication patterns. It is fun to see how the Actor model can, even though it is a low-level computation model, be used to implement powerful and rich messaging patterns in a simple and straightforward manner. Once you understand the basic ideas, you can bring in more high-level tools and techniques.

This book also does a great job of formalizing and naming many of the patterns that users in the Akka community have had to discover and reinvent themselves over the years. I remember enjoying reading and learning from the classic *Enterprise Integration Patterns* [EIP] by Hohpe and Woolf a few years ago, and I’m glad that Vaughn builds upon and reuses its pattern catalog,
Putting it in a fresh context. But I believe that the most important contribution of this book is that it does not stop there but takes the time to define and introduce a unique pattern language for actor messaging, giving us a vocabulary for how to think about, discuss, and communicate the patterns and ideas.

This is an important book—regardless if you are a newbie or a seasoned “hakker”—and I hope that you will enjoy it as much as I did.

— Jonas Bonér  
Founder of the Akka Project
Preface

Today, many software projects fail. There are various surveys and reports that show this, some of which report anywhere from 30 to 50 percent failure rates. This number doesn’t count those projects that delivered but with distress or that fell short of at least some of the prerequisite success criteria. These failures, of course, include projects for the enterprise. See the Chaos Report [Chaos Report], Dr. Dobb’s Journal [DDJ], and Scott Ambler’s survey results [Ambysoft].

At the same time, some notable successes can be found among companies that use Scala and Akka to push the limits of performance and scalability [WhitePages]. So, there is not only success but success in the face of extreme nonfunctional requirements. Certainly it was not Scala and Akka alone that made these endeavors successful, but at the same time it would be difficult to deny that Scala and Akka played a significant role in those successes. I am also confident that those who make use of these tools would stand by their platform decisions as ones that were key to their successes.

For a few years now it has been my vision to introduce the vast number of enterprises to Scala and Akka in the hopes that they will find similar successes. My goal with this book is to make you familiar with the Actor model and how it works with Scala and Akka. Further, I believe that many enterprise architects and developers have been educated by the work of Gregor Hohpe and Bobby Woolf. In their book, Enterprise Integration Patterns [EIP], they provide a catalog of some 65 integration patterns that have helped countless teams to successfully integrate disparate systems in the enterprise. I think that leveraging those patterns using the Actor model will give architects and developers the means to tread on familiar turf, besides that the patterns are highly applicable in this space.

When using these patterns with the Actor model, the main difference that I see is with the original motivation for codifying the patterns. When using the Actor model, many of the patterns will be employed in greenfield applications,
not just for integration. That is because the patterns are first and foremost messaging patterns, not just integration patterns, and the Actor model is messaging through and through. You will also find that when implementing through the use of a Domain-Driven Design [DDD, IDDD] approach that some of the more advanced patterns, such as Process Manager (292), will be used to help you model prominent business concepts in an explicit manner.

Who This Book Is For

This book is for software architects and developers working in the enterprise and any software developer interested in the Actor model and looking to improve their skills and results. Although the book is definitely focused on Scala and Akka, Appendix A provides the means for C# developers on the .NET platform to make use of the patterns as well.

What Is Covered in This Book

I start out in Chapter 1, “Discovering the Actor Model and the Enterprise, All Over Again,” with an introduction to the Actor model and the tenets of reactive software. Chapter 2, “The Actor Model with Scala and Akka,” provides a Scala bootstrap tutorial as well as a detailed introduction to Akka and Akka Cluster. Chapter 3, “Performance Bent,” then runs with a slant on performance and scalability with Scala and Akka, and why the Actor model is such an important approach for accomplishing performance and scalability in the enterprise.

This is followed by seven chapters of the pattern catalog. Chapter 4, “Messaging with Actors,” provides the foundational messaging patterns and acts as a fan-out for the following five chapters. In Chapter 5, “Messaging Channels,” I expand on the basic channel mechanism and explore several kinds of channels, each with a specific advantage when dealing with various application and integration challenges. Chapter 6, “Message Construction,” shows you how each message must convey the intent of the sender’s reason to communicate with the receiver. Chapter 7, “Message Routing,” shows you how to decouple the message source from the message destination and how you might place appropriate business logic in a router. In Chapter 8, “Message Transformation,” you’ll dig deeper into various kinds of transformations that messages may undergo in your applications and integrations. In Chapter 9, “Message
Endpoints,” you will see the diverse kinds of endpoints, including those for persistent actors and idempotent receivers. Finally, I wrap things up with Chapter 10, “System Management and Infrastructure,” which provides advanced application, infrastructural, and debugging tools.

**Conventions**

A major part of the book is a *pattern catalog*. It is not necessary to read every pattern in the catalog at once. Still, you probably should familiarize yourself with Chapters 4 through 10 and, in general, learn where to look for details on the various kinds of patterns. Thus, when you need a given pattern, you will at least know in general where to look to find it. Each pattern in the catalog has a representative icon and will also generally have at least one diagram and source code showing how to implement the pattern using Scala and Akka.

The extensive catalog of patterns actually forms a *pattern language*, which is a set of interconnected expressions that together form a collective method of designing message-based applications and systems. Thus, it is often necessary for one pattern to refer to one or more other patterns in the catalog, as the supporting patterns form a complete language. Thus, when a pattern is referenced in this book, it is done like this: Pattern Name (#). That is, the specific pattern is named and then followed by the page number where the referenced pattern begins.

Another convention of this book is how messaging patterns with the Actor model are expressed in diagrams. I worked on formulating these conventions along with Roland Kuhn and Jamie Allen of Typesafe. They are coauthors of an upcoming book on a similar topic: *Reactive Design Patterns*. I wanted our books to use the same, if not similar, ways to express the Actor model in diagrams, so I reached out to Roland and Jamie to discuss. The following shows the conventions that we came up with.

As shown in Figure P.1, actors are represented as circular elements and generally named with text inside the circle. One of the main reasons for this is that

![Figure P.1 A Sender actor sends a message to a Receiver actor.](image-url)
Gul Agha used this notation long ago in his book *Actors: A Model of Concurrent Computation in Distributed Systems* [Agha, Gul].

Further, a message is represented as a component in much the same way that *Enterprise Integration Patterns* [EIP] does, so we reused that as well. The lines with arrows show the source and target of the message. You can actually distinguish a persistent actor (long-lived) from an ephemeral actor (short-lived) using the notations shown in Figures P.2 and P.3. A persistent actor has a solid circular border. Being a persistent actor just means that it is long-lived. It does not necessarily mean that the actor is persisted to disk, but it could also mean that. On the other hand, an ephemeral actor has a dashed circular border. It is one that is short-lived, meaning that it is created to perform some specific tasks and is then stopped.

One actor can create another actor, as shown in Figure P.4, which forms a parent-child relationship. The act of creation is represented as a small circle surrounded by a large circle and takes the form similar to a message being sent.
CONVENTIONS

from the parent to the child. This is because the process of child actor creation is an asynchronous operation.

Actor self-termination is represented by a special message—a circle with an X inside—being sent from the actor to itself, as shown in Figure P.5. Again, this is shown as a message because termination is also an asynchronous operation.

![Figure P.5 Actor self-termination](image)

One actor terminating another actor is shown as the same special message directed from one actor to another. The example in Figure P.6 shows a parent terminating one of its children.

![Figure P.6 One actor terminates another actor.](image)

An actor’s lifeline can be represented similar to that of a Unified Modeling Language (UML) sequence diagram, as shown in Figure P.7. Messages being received on the lifeline are shown as small circles (like pinheads). You must recognize that each message receipt is asynchronous.

A parent’s child hierarchy can be represented as a triangle below the parent with child actors inside the triangle. This is illustrated by Figure P.8.

An actor may learn about other actors using endowment or introduction. Endowment is accomplished by giving the endowed actor a reference to other actors when it is constructed. On the other hand, an actor is introduced to another actor by means of a message.

Introduction, as shown in Figure P.9, is represented as a dotted line where the actor being introduced is placed into a message that is sent to another
Figure P.7 An actor’s lifeline is shown as two asynchronous messages are received.

Figure P.8 An actor’s child hierarchy

Figure P.9 A child actor is introduced by a sender to the receiver by means of a message.
actor. In this example, it is a child actor that is being created by a parent that is introduced to the receiver.

Finally, message sequence is shown by sequence numbers in the diagram in Figure P.10. The fact that there are two 2 sequences and two 4 sequences is not an error. This represents an opportunity for concurrency, where each of the repeated sequences show messages that are happening at the same time. In this example, the router is setting a timer and sending a message to receiver concurrently (steps 2). Also, the timer may elapse before a response can be sent by the receiver (steps 3). If the receiver’s response is received by the router first, then the client will receive a positive confirmation message as sequence 4. Otherwise, if the timer elapses first, then the client will receive a timeout message as sequence 4.

Figure P.10 Message sequence and concurrency are represented by sequence numbers.
This page intentionally left blank
Acknowledgments

I’d like to express many thanks to Addison-Wesley for selecting this book to publish under their distinguished label. I once again got to work with Chris Guzikowski and Chris Zahn as my editors. I especially thank Chris Guzikowski for his patience while major sections of the book underwent drastic changes in response to modifications to the Akka toolkit. In the end, I am sure it was worth the wait.

What would a book on the Actor model be without acknowledging Carl Hewitt and the work he did and continues to do? Dr. Hewitt and his colleagues introduced the world to a simple yet most ingenious model of computation that has only become more applicable over time.

I also thank the Akka team for the fine work they have done with the Akka toolkit. In particular, Jonas Bonér reviewed chapters of my book and provided his unique perspective as the original founder of the Akka project. Akka’s tech lead, Roland Kuhn, also reviewed particularly delicate parts of the book and gave me invaluable feedback. Also, both Roland Kuhn and Jamie Allen were supportive as we together developed the notation for expressing the Actor model in diagrams. Additionally, Patrik Nordwall of the Akka team reviewed the early chapters.

A special thanks goes to Will Sargent, a consultant at Typesafe, for contributing much of the section on Akka Cluster. Although I wrote a big chunk of that section, it was Will who helped with special insights to take it from ordinary to what I think is quite good.

Two of my early reviewers were Thomas Lockney, himself an Akka book author, and Duncan DeVore, who at the time of writing this was working on his own Akka book. In particular, Thomas Lockney endured through some of the earliest attempts at the first three chapters. Frankly, it surprised me how willing Thomas was to review and re-review and how he consistently saw areas for major improvement.
Other reviewers who contributed to the quality of the book include Idar Borlaug, Brian Dunlap, Tom Janssens, Dan Bergh Johnsson, Tobias Neef, Tom Stockton, and Daniel Westheide. Thanks to all of you for providing the kind of feedback that made a difference in the quality of the book. In particular, Daniel Westheide is like a “human Scala compiler,” highlighting even difficult-to-find errors in written code examples.
About the Author

Vaughn Vernon is a veteran software craftsman and thought leader in simplifying software design and implementation. He is the author of the best-selling book Implementing Domain-Driven Design, also published by Addison-Wesley, and has taught his IDDD Workshop around the globe to hundreds of software developers. Vaughn is a frequent speaker at industry conferences. Vaughn is interested in distributed computing, messaging, and in particular the Actor model. He first used Akka in 2012 on a GIS system and has specialized in applying the Actor model with Domain-Driven Design ever since. You can keep up with Vaughn’s latest work by reading his blog at www.VaughnVernon.co and by following him on Twitter: @VaughnVernon.
This page intentionally left blank
Chapter 6
Message Construction

In Chapter 4, “Messaging with Actors,” I discussed messaging with actors, but I didn’t cover much about the kinds of messages that should be created and sent. Each message must convey the intent of the sender’s reason to communicate with the receiver. As *Enterprise Integration Patterns* [EIP] shows, there may be any number of motivations based on the following:

- **Message intent**: Why are you sending a message? Are you requesting another actor to perform an operation? If so, use a *Command Message* (202). Are you informing one or more other actors that you have performed some operation? In that case, use an *Event Message* (207). Have you been asked for some large body of information that you must convey to the requester via a *Message* (130)? The request can be fulfilled using a *Document Message* (204).

- **Returning a response**: When there is a contract between two actors that follows *Request-Reply* (209), the actor receiving the request needs to provide a reply, or *response*. The request is a *Command Message* (202) and the reply is generally a *Document Message* (204). Since you are using the Actor model, the actor receiving the request knows the *Return Address* (211) of the sender and can easily reply. If there are multiple incoming requests that are related to one another or multiple outgoing replies that are logically bundled, use a *Correlation Identifier* (215) to associate separate messages into one logical package.

- **Huge amounts of data**: Sometimes you need more than a *Correlation Identifier* (215) to bundle related messages. What happens if you can correlate a set of messages but you also need to ensure that they are ordered according to some application-specific sequence? That’s the job of a *Message Sequence* (217).

- **Slow messages**: As I have taken the opportunity to repeat in several places through the text, the network is unreliable. When a *Message* (130) of whatever type must travel over the network, there is always a change that network latency will affect its delivery. Even so, there are also latencies in actors when, for example, they have a lot of work to do before they
can handle your requests. Although this can point to the need to redesign some portion of the application to deal with workload in a more acceptable time frame, you also need to do something about it when encountered. You can use a Message Expiration (218) and perhaps the Dead Letter Channel (172) to signal to the system that something needs to be done about the latency situation, if in fact it is deemed unacceptable.

- **Message version**: Oftentimes a Document Message (204), or actually an Event Message (207) or even a Command Message (202), can have a number of versions throughout its lifetime. You can identify the version of the message using a Format Indicator (222).

In much the same way that you must think about the kind of Message Channel (128) you will use for various application and integration circumstances, you must also think about and design your messages specifically to deal with the reaction and concurrency scenarios at hand.

---

**Command Message**

![Diagram of Command Message]

When a message-sending actor needs to cause an action to be performed on the receiving actor, the sender uses a Command Message.

If you are familiar with the Command-Query Separation principle [CQS], you probably think of a Command Message as one that, when handled by the receiver, will cause a side effect on the receiver (see Figure 6.1). After all, that’s what the C in CQS stands for: a request that causes a state transition. Yet, a Command Message as described by Enterprise Integration Patterns [EIP] may also be used to represent the request for a query—the Q in CQS. Because of the overlap in intended uses by Enterprise Integration Patterns [EIP], when designing with the CQS principle in mind and discussing a message that causes a query to be performed, it is best to instead use the explicit term *query message*. Even so, this is not to say that a message-based actor system must be designed with CQS in mind. Depending on the system, it may work best for a given Command Message to both alter state and elicit a response message, as discussed in Request-Reply (209).
Each Command Message, although sent by a requestor, is defined by the receiver actor as part of its public contract. Should the sent Command Message not match one defined as part of the receiver’s contract, it could be redirected to the *Invalid Message Channel* (170).

Command Messages are designed as imperative exhortations of actions to be performed; that is, the exhortation for an actor to perform some behavior. The Command Message will contain any data parameters and collaborating actor parameters necessary to perform the action. For example, besides passing any data that is required to perform the command, a Command Message may also contain a *Return Address* (211) to indicate which actor should be informed about possible side effects or outcomes.

In essence you can think of a Command Message as a representation of an operation invocation. In other words, a Command Message captures the intention to invoke an operation, but in a way that allows the operation to be performed at a time following the message declaration.

The following case classes implement Command Messages for an equities trading domain:

```scala
case class ExecuteBuyOrder(
  portfolioId: String,
  symbol: String,
  quantity: Int,
  price: Money)

case class ExecuteSellOrder(
  portfolioId: String,
  symbol: String,
  quantity: Int,
  price: Money)
```

Here a *StockTrader* receives the two Command Messages but rejects any other message type by sending them to the *Dead Letter Channel* (172), which doubles as an *Invalid Message Channel* (170):
class StockTrader(tradingBus: ActorRef) extends Actor {
    ... 
    def receive = {
        case buy: ExecuteBuyOrder =>
            ...
        case sell: ExecuteSellOrder =>
            ...
        case message: Any =>
            context.system.deadLetters ! message
    }
}

Normally a Command Message is sent over a Point-to-Point Channel (151) because the command is intended to be performed once by a specific receiving actor. To send a broadcast type of message, likely you will want to use an Event Message (207) along with Publish-Subscribe Channel (154).

Document Message

Use a Document Message to convey information to a receiver, but without indicating how the data should be used (see Figure 6.2). This is different from a Command Message (202) in that while the command likely passes data parameters, it also specifies the intended use. A Document Message also differs from an Event Message (207) in that while the event conveys information without specifying its intended use, an event associates the data it carries with a past occurrence in the business domain. Although a Document Message communicates information about the domain, it does so without indicating that the concept is a fact that expresses a specific past occurrence in the business domain. See also Domain Events, as discussed in Implementing Domain-Driven Design [IDDD].

Oftentimes a Document Message serves as the reply in the Request-Reply (209) pattern. In the implementation diagram the Receiver may have previously sent a Command Message (202) to the Sender to request to data, as in Request-Reply (209), or the Sender may send the Document Message to the Receiver without the Receiver previously requesting the information of the Sender.
The following is a Document Message that conveys data about a quotation fulfillment:

```scala
case class QuotationFulfillment(
  rfqId: String,
  quotesRequested: Int,
  priceQuotes: Seq[PriceQuote],
  requester: ActorRef)
```

The information provided by the `QuotationFulfillment` document includes the unique identity of the request for quotation, the number of quotations that were requested, the number of `PriceQuote` instances that were actually obtained, and a reference to the actor that originally requested the quotations. The `PriceQuote` itself could be considered a full Document Message but in this example is just part of the composed `QuotationFulfillment` Document Message:

```scala
case class PriceQuote(
  quoterId: String,
  rfqId: String,
  itemId: String,
  retailPrice: Double,
  discountPrice: Double)
```

As the `PriceQuote` structure indicates, it is not the size or complexity of the message type that determines whether it is a Document Message. Rather, it is the fact that information is conveyed without indicating intended usage (command) or that it is conveyed as part of an application outcome (event). To reinforce this, consider the following `Command Message (202)` and `Event Message (207)`, respectively, that are used in conjunction with obtaining the `QuotationFulfillment` Document Message:

```scala
case class RequestPriceQuote(
  rfqId: String,
```

![Figure 6.2](image-url) The Sender, by means of a Document Message, provides the Receiver with information but without indicating how it should be used.
itemId: String,
retailPrice: Money,
orderTotalRetailPrice: Money)

case class PriceQuoteFulfilled(priceQuote: PriceQuote)

The first case class, which is a Command Message (202), is used to request a set of quotations for a specific item. The second case class, that being an Event Message (207), is published when a price quotation has been fulfilled. These are both quite different from the QuoteFulfillment Document Message, which merely carries information about the results of the previously requested price quotation.

Managing Flow and Process

You can use a Document Message to assist in managing workflow or long-running processes [IDDD]. Send a Document Message on a Point-to-Point Channel (151) to one actor at a time, each implementing a step in the process. As each step completes, it appends to the document that it received, applying the changes for the processing step as it is completed. The actor for the current step then dispatches the appended Document Message on to the actor of the next processing step. This dispatch-receive-append-dispatch recurrence continues until the process has completed.

It is the final step that determines what to do with the now fully composed Document Message. Since long-running processes may be composed from a few to many different smaller processes, it's possible that the final step in a given process merely completes one branch of a larger process. In all of this, no Document Message is actually mutated to an altered state. Instead, each step composes a new Document Message as a combination of the current document and any new information to be appended. The merging of the current Document Message with new data might be performed as a simple concatenation. Assuming a linear process where each step is responsible for gathering a PriceQuote from a given vendor, this example shows how a QuotationFulfillment can be appended:

case class QuotationFulfillment(
  rfqId: String,
  quotesRequested: Int,
  priceQuotes: Seq[PriceQuote],
  requester: ActorRef) {

  def appendWith(
    fulfilledPriceQuote: PriceQuote):
  QuotationFulfillment {
  QuotationFulfillment(}
The Document Message itself may contain some data describing how each processing step actor is to dispatch to the next step. This might be handled by placing the actor address+name of each step inside the original document in the order in which the steps should occur. As each step completes, it simply looks up the next actor and dispatches, sending it the appended document.

As an alternative to this document-based lookup approach, you may instead choose to use the Akka DistributedPubSubMediator, as discussed in Publish-Subscribe Channel (154), to dispatch to a single actor in the cluster without the need to actually look up the actor. This approach uses the DistributedPubSubMediator.Send router message. If using Send, you would simply place the name of each processing step actor in the document, leaving off the address. The contract of the DistributedPubSubMediator ensures that a matching actor somewhere in the cluster will receive the next Document Message per a specified routing policy.

When a long-running process has a complex routing specification, it would be best to use a Process Manager (292) to coordinate dispatching to each step. Generally, you would need such a Process Manager (292) when the dispatching rules include conditional branching based on values appended to the Document Message by one or more steps.

---

Event Message

Use an Event Message, as illustrated in Figure 6.3, when other actors need to be notified about something that has just occurred in the actor that produces the event. Generally, a Publish-Subscribe Channel (154) is used to inform
interested parties about a given event. Yet, sometimes it may be appropriate to
tell a specific actor about an event or tell the specific actor and also publish to
an abstract set of subscribers. See also Domain Events as discussed in Implementing Domain-Driven Design [IDDD].

For example, when an OrderProcessor receives a RequestForQuotation message, it dispatches a request to fulfill the quotations to any number of product discounters. As each discounter that chooses to participate responds with a PriceQuote Document Message (204) describing the discount offer, the OrderProcessor sends a PriceQuoteFulfilled Event Message to an Aggregator (257).

```
case class PriceQuote(
    quoterId: String,
    rfqId: String,
    itemId: String,
    retailPrice: Double,
    discountPrice: Double) // Document Message

case class PriceQuoteFulfilled(
    priceQuote: PriceQuote) // Event Message
```

In this specific case, it is unnecessary to broadcast the event using a Publish-Subscribe Channel (154) because it is specifically the Aggregator (257) that needs to know about the price quote fulfillment. You could have designed the Aggregator (257) to accept a Command Message (202) or a Document Message (204) rather than an Event Message. Yet, the OrderProcessor need not be concerned with how the Aggregator (257) works, only that it will satisfy
its contract once it has received some required number of PriceQuoteFulfilled events. Also note that the PriceQuoteFulfilled is a Document Message (204) in that the Event Message packs the small PriceQuote Document Message (204) as the PriceQuoteFulfilled event information.

---

**Request-Reply**

![Request-Reply Diagram](image_url)

When a message is sent from one actor to another, it is considered a request. When the receiver of the request message needs to send a message back to the request sender, the message is a reply. As shown in Figure 6.4, a common usage pattern of Request-Reply has the requestor sending a Command Message (202) and the receiver replying with a Document Message (204). In such a case, and as described in Command Message (202), the command is probably a Query Message [IDDD].

While the requestor will normally send a Command Message (202), replying with a Message Document Message (204) is not a strict requirement. Still, if you consider the document payload of the reply to be any simple structure, not necessarily a complex one, then it is often appropriate to refer to the reply as a Document Message (204). The point is that the document carries data but does not indicate what the consumer should do with it.

Request-Reply is quite simple and straightforward to implement using the Actor model. In fact, Request-Reply is considered part of the basic actor semantics. Here is how it works:
package co.vaughnvernon.reactiveenterprise.requestreply

import akka.actor._
import co.vaughnvernon.reactiveenterprise._

case class Request(what: String)
case class Reply(what: String)
case class StartWith(server: ActorRef)

object RequestReply extends CompletableApp(1) {
  val client = system.actorOf(Props[Client], "client")
  val server = system.actorOf(Props[Server], "server")
  client ! StartWith(server)

  awaitCompletion
  println("RequestReply: is completed.")
}

class Client extends Actor {
  def receive = {
    case StartWith(server) =>
      println("Client: is starting...")
      server ! Request("REQ-1")
    case Reply(what) =>
      println("Client: received response: " + what)
      RequestReply.completedStep()
    case _ =>
      println("Client: received unexpected message")
  }
}

class Server extends Actor {
  def receive = {
    case Request(what) =>
      println("Server: received request value: " + what)
      sender ! Reply("RESP-1 for " + what)
    case _ =>
      println("Server: received unexpected message")
  }
}

The following output is produced by the Client and Server:

Client: is starting...
Server: received request value: REQ-1
Client: received response: RESP-1 for REQ-1
Client: is completing...
The three classes at the top of the file are the messages that can be sent. Following the message types there is the application (App) object, and then the Client and Server actors. Note that the use of `awaitCompletion()` in the App bootstrap object makes the application stick around until the two actors complete.

The first message, `StartWith`, is sent to the Client to tell it to start the Request-Reply scenario. Although `StartWith` is a `Command Message (202)` request, note that the Client does not produce a reply to the App. The `StartWith` message takes one parameter, which is the instance of the Server actor (actually an `ActorRef`). The Client makes a request to the Server, and the Server makes a reply to the Client. The `Request` and `Reply` are the other two different message types.

Specifically, a Client knows how to `StartWith` and how to react to `Reply` messages, while a Server knows how to react to `Request` messages. If the Client receives anything but `StartWith` and `Reply`, it simply reports that it doesn’t understand. The Server does the same if it receives anything but a `Request`.

These details notwithstanding, the main point of this simple Scala/Akka example is to show how Request-Reply is accomplished using the Actor model. It’s pretty simple. Wouldn’t you agree? Request-Reply is a natural form of programming using the Actor model. As you can see, the Server doesn’t need to know it is replying to the Client actor. It only needs to know it is replying to the sender of the `Request`, and the sender of the `Request` needs to know that it will receive a `Reply` to its `Request`.

All of this happens asynchronously. The Client and the Server share nothing; that is, their states are completely encapsulated and protected. That, and the fact that each actor will handle only one message at a time, allows the asynchronous message handling to be completely lock free.

---

**Return Address**

When reasoning on Request-Reply (209), what if you want your request receiver to reply to an actor at an address other than the direct message sender? Well, that’s the idea behind Return Address, as shown in Figure 6.5, and one that you can implement in a few different ways.
It’s interesting that the Actor model actually uses addresses to identify how to send messages to actors. You see, each actor has an address, and to send a given actor a message, you must know its address. One actor can know the address of another actor by a few different means.

- An actor creates another actor and thus knows the address of the actors it has created.
- An actor receives a message that has the address of one or more other actors that it will send messages to.
- In some cases, an actor may be able to look up the address of another actor by name, but this may create an unseemly binding to the definition and implementation of a given actor.

The *Enterprise Integration Patterns* [EIP] Return Address fits really well with the fundamental ideas behind the Actor model.

One obvious way to provide a Return Address in a given message is to put the address of the actor that you want to receive the reply right in the message that you send. Recall that you did something similar in the Request-Reply example.

case class StartWith(server: ActorRef)

The first message that the client receives is `StartWith`, and that message must contain the `ActorRef` of the server that the client is to use. That way, the client will know how to make requests of some server. Okay, so that’s not really a Return Address, but you could send a Return Address as part of a message in the same way.
If the client chose to, it could also send messages to the server and provide the Return Address of the actor that should receive the reply. Of course, the request message itself would have to support that protocol and allow the `ActorRef` to be included in the message.

```scala
case class Request(what: String, replyTo: ActorRef)
```

That way, when the server is ready to send its reply to the request, it could send the reply to the `replyTo` actor, like so:

```scala
class Server extends Actor {
  def receive = {
    case Request(what, replyTo) =>
      println("Server: received request value: " + what)
      replyTo ! Reply("RESP-1 for " + what)
    case _ =>
      println("Server: received unexpected message")
  }
}
```

That works, but it does require you to design the message protocol in a certain way. What if you have an existing message protocol and you later decide to redesign the existing receiving actor to delegate some message handling to one of its child actors? This might be the case if there is some complex processing to do for certain messages and you don’t want to heap too much responsibility on your original actor, for example the server. It would be nice if the server could create a child worker to handle a specific kind of complex message but design the worker to reply to the original client sender, not to the parent server. That would free the parent server to simply delegate to the child worker and allow the worker to react as if the server had done the work itself.

```scala
package co.vaughnvernon.reactiveenterprise.returnaddress

import akka.actor._
import co.vaughnvernon.reactiveenterprise._

case class Request(what: String)
case class RequestComplex(what: String)
case class Reply(what: String)
case class ReplyToComplex(what: String)
case class StartWith(server: ActorRef)

object ReturnAddress extends CompletableApp(2) {
  val client = system.actorOf(Props[Client], "client")
  val server = system.actorOf(Props[Server], "server")
```
client ! StartWith(server)

awaitCompletion
    println("ReturnAddress: is completed.")
}

class Client extends Actor {
  def receive = {
    case StartWith(server) =>
      println("Client: is starting...")
      server ! Request("REQ-1")
      server ! RequestComplex("REQ-20")
    case Reply(what) =>
      println("Client: received reply: " + what)
      ReturnAddress.completedStep()
    case ReplyToComplex(what) =>
      println("Client: received reply to complex: " + what)
      ReturnAddress.completedStep()
    case _ =>
      println("Client: received unexpected message")
  }
}

class Server extends Actor {
  val worker = context.actorOf(Props[Worker], "worker")

  def receive = {
    case request: Request =>
      println("Server: received request value: " + request.what)
      sender ! Reply("RESP-1 for " + request.what)
    case request: RequestComplex =>
      println("Server: received request value: " + request.what)
      worker forward request
    case _ =>
      println("Server: received unexpected message")
  }
}

class Worker extends Actor {
  def receive = {
    case RequestComplex(what) =>
      println("Worker: received complex request value: " + what)
      sender ! ReplyToComplex("RESP-2000 for " + what)
    case _ =>
      println("Worker: received unexpected message")
  }
}
This is the output produced by the Return Address example:

```
Client: is starting...
Server: received request value: REQ-1
Server: received request value: REQ-20
Client: received reply: RESP-1 for REQ-1
Worker: received complex request value: REQ-20
Client: received reply to complex: RESP-2000 for REQ-20
```

Note that when the Server is created, it uses its context to create a single child Worker actor. This Worker is used by the Server only when it receives a RequestComplex message. Also note that there is no reason to design the RequestComplex message with a replyTo ActorRef. Thus, as far as the Client is concerned, it is the Server that handles the RequestComplex message.

Now notice that the Server doesn’t just tell the Worker what to do by sending it the RequestComplex message. Rather, the Server forwards the RequestComplex message to the Worker. By forwarding, the Worker receives the message as if it had been sent directly by the Client, which means that the special sender ActorRef has the address of the Client, not of the Server. Therefore, the Worker is able to act on behalf of the Server, as if the Server itself had done the work. Yet, the Server is freed from acting as a mediator between the Client and the Worker, not to mention that the Server is ready to process other messages while the Worker does its thing.

---

**Correlation Identifier**

![Diagram](image)

Establish a Correlation Identifier to allow requestor and replier actors to associate a reply message with a specific, originating request message. The unique identifier must be associated with both the message sent by the requestor and the message sent by the replier, as shown in Figure 6.6.

In its discussion of Correlation Identifier, *Enterprise Integration Patterns* [EIP] suggests creating an independent, unique message identifier on the request message and then using that message identifier as the Correlation Identifier in the reply message. The unique message identifier would generally be
generated by the messaging system and would be attached only to the message header. Additionally, *Enterprise Integration Patterns* (EIP) suggests setting the identifier as the *request ID* on the request message but to be named *correlation ID* on the reply message.

In principle this is also what is done with the Actor model. Yet, modeling messages for use with the Actor model works a bit differently as well. For example, there is no separate message header, unless one is created as part of the message’s type. Thus, it makes more sense to design message types to contain unique *business identities*. In this case, you would not need to name the identifier using different names on each message type. In fact, it would most often be best to name the identifier the same on all message types that contain it. That way, it’s just a unique identity that is business specific.

Each of the following message types are correlated using the *rfqId* (request for quotation ID):

```scala
case class RequestPriceQuote(  
  rfqId: String,  
  itemId: String,  
  retailPrice: Double,  
  orderTotalRetailPrice: Double)

case class PriceQuote(  
  quoterId: String,  
  rfqId: String,  
  itemId: String,  
  retailPrice: Double,  
  discountPrice: Double)

case class PriceQuoteTimedOut(rfqId: String)
```

*Figure 6.6 A Requestor attaches a Correlation Identifier to outgoing Messages in order for the Replier to associate its replies with the originating Message.*
case class RequiredPriceQuotesForFulfillment(
    rfqId: String,
    quotesRequested: Int)

case class QuotationFulfillment(
    rfqId: String,
    quotesRequested: Int,
    priceQuotes: Seq[PriceQuote],
    requester: ActorRef)

case class BestPriceQuotation(
    rfqId: String,
    priceQuotes: Seq[PriceQuote])

Although *Enterprise Integration Patterns* [EIP] focuses on the use of Correlation Identifier with *Request-Reply* (209), there is no reason to limit its use to that pattern. For example, you should associate a Correlation Identifier as a unique business identity with all messages involved in a long-running process [IDDD], whether using ad hoc process management or a formal *Process Manager* (292).

**Message Sequence**

Use a Message Sequence when you need to send one logical *Message* (130) that must be delivered as multiple physical *Messages* (130). Together all the messages in the sequence form a batch, but the batch is delivered as separate elements. Each *Message* (130) will have the following:

- A unique Message Sequence identity, such as a *Correlation Identifier* (215).
- A sequence number indicating the sequence of the particular message in the separated batch. The sequence could run from 1 to N or from 0 to N-1, where N is the total number of messages in the batch.
- Some flag or other indicator of the last message in the batch. This could also be achieved by placing a total on the first message to be sent.

On first considering the way the Actor model messages are sent and received, it may seem unnecessary to use a Message Sequence. Also discussed
in Resequencer (264), Akka direct asynchronous messaging has the following characteristics, as applicable in a discussion of Message Sequence (217):

- Actor Batch-Sender sends messages M1, M2, M3 to Batch-Receiver.

Based on this scenario, you arrive at these facts:

1. If M1 is delivered, it must be delivered before M2 and M3.
2. If M2 is delivered, it must be delivered before M3.
3. Since there is no (default) guaranteed delivery, any of the messages M1, M2, and/or M3, may be dropped, in other words, not arrive at Batch-Receiver.

Although sequencing is not a problem in itself, note that the problem arises if any one message sent from Batch-Sender does not reach Batch-Receiver. Thus, when multiple messages comprising a batch must be delivered to Batch-Receiver for the use case to complete properly, you must assume that Batch-Receiver will be required to interact with Batch-Sender if Batch-Receiver detects missing messages from the batch.

When designing the interactions between Batch-Sender and Batch-Receiver, it may work best to design Batch-Receiver as a Polling Consumer (362). In this case, the Batch-Sender tells the Batch-Receiver that a new batch is available, communicating the specifications of the batch. Then the Batch-Receiver asks for each messages in the batch in order. The Batch-Receiver moves to the next sequence in the batch only once the current message in the sequence is confirmed. The Batch-Receiver can perform retries as needed using schedulers, which is also discussed with regard to Polling Consumer (362).

Otherwise, if the Batch-Sender drives the process by sending the message batch in an enumerated blast, the Batch-Receiver must be prepared to request redelivery for any sequence that it doesn’t receive.

---

**Message Expiration**

If it is possible for a given message to become obsolete or in some way invalid because of a time lapse, use a Message Expiration (218) to control the timeout
(see Figure 6.7). While you have already dealt with the process timeouts in the Scatter-Gather (272) implementation, this is different. A Message Expiration is used to determine when a single message has expired, rather than setting a limit on the completion of a larger process.

When using message-based middleware, it is possible to ask the messaging system to expire a message before it is ever delivered. Currently Akka does not support a mailbox that automatically detects expired messages. No worries, you can accomplish that on your own quite easily. You could create a custom mailbox type or just place the expiration behavior on the message itself. There are advantages to both. Here I explain how to do this using a trait for messages. Whether or not the mailbox supports expiring messages, the message itself must supply some parts of the solution.

It is the message sender that should determine the possibility of message expiration. After all, the sender is in the best position to set the message time-to-live based on some user or system specification for the type of operation being executed. Here is how it can be done. First design a trait that allows an extending message to specify the `timeToLive` value.

```scala
trait ExpiringMessage {
  val occurredOn = System.currentTimeMillis()
  val timeToLive: Long

  def isExpired(): Boolean = {
    val elapsed = System.currentTimeMillis() - occurredOn
    elapsed > timeToLive
  }
}
```

![Figure 6.7 A Message Expiration is attached to a Message that may become stale.](image-url)
The trait initializes its `occurredOn` with the timestamp of when it was created. The trait also declares an abstract `timeToLive`, which must be set by the extending concrete class.

The `ExpiringMessage` trait also provides behavior, through method `isExpired()`, that indicates whether the message has expired. This operation first gets the system’s current time in milliseconds, subtracts the number of milliseconds since the message was created (`occurredOn`) to calculate the elapsed time, and then compares the elapsed time to the client-specified `timeToLive`.

Note that this basic algorithm does not consider differences in time zones, which may need to be given consideration depending on the system’s network topology. At a minimum, this approach assumes that different computing nodes that host various actors will have their system clocks synchronized closely enough to make this sort of calculation successful.

This trait is used in the implementation sample, which defines a `PlaceOrder Command Message (202)`:

```scala
package co.vaughnvernon.reactiveenterprise.messageexpiration

import java.util.concurrent.TimeUnit
import java.util.Date
import scala.concurrent._
import scala.concurrent.duration._
import scala.util._
import ExecutionContext.Implicits.global
import akka.actor._
import co.vaughnvernon.reactiveenterprise._

case class PlaceOrder(
  id: String,
  itemId: String,
  price: Money,
  timeToLive: Long)
  extends ExpiringMessage

object MessageExpiration extends CompletableApp(3) {
  val purchaseAgent =
    system.actorOf(
      Props(PurchaseAgent),
      "purchaseAgent")

  val purchaseRouter =
    system.actorOf(
      Props(classOf[PurchaseRouter],
            purchaseAgent),
        "purchaseRouter")
```
In the MessageExpiration sample runner, you create two actors, a PurchaseAgent and a PurchaseRouter. In a real application, the PurchaseRouter could be a Content-Based Router (228) and route to any number of different purchase agents based on the kind of purchase message. Here you aren’t really concerned about that kind of routing but use the PurchaseRouter to simulate delays in message delivery from various causes.

```scala
class PurchaseRouter(purchaseAgent: ActorRef) extends Actor {
  val random = new Random((new Date()).getTime)
  def receive = {
    case message: Any =>
      val millis = random.nextInt(100) + 1
      println(s"PurchaseRouter: delaying delivery of $message for $millis milliseconds")
      val duration = Duration.create(millis, TimeUnit.MILLISECONDS)
      context.system.scheduler.scheduleOnce(duration, purchaseAgent, message)
  }
}
```

To familiarize yourself even more with the Akka Scheduler, you can see another example in Resequencer (264).

Now, more to the point, this is how the actual PurchaseAgent checks for Message Expiration and branches accordingly:

```scala
class PurchaseAgent extends Actor {
  def receive = {
    case placeOrder: PlaceOrder =>
      if (placeOrder.isExpired()) {
        context.system.deadLetters ! placeOrder
        println(s"PurchaseAgent: delivered expired $placeOrder to dead letters")
      } else {
        println(s"PurchaseAgent: placing order for $placeOrder")
      }
  }
}
If the PlaceOrder message is expired, the PurchaseAgent sends the message to the Akka ActorSystem’s special deadLetters actor, which implements the Dead Letter Channel (172). Note that Enterprise Integration Patterns [EIP] discusses the possibility of expired messages being delivered to a different Message Channel (128) for one reason or another, but the motivation is the same. You also have the option to ignore the message altogether.

Here’s the output from running the process:

PurchaseRouter: delaying delivery of PlaceOrder(1,11,50.0,1000) for 87 milliseconds
PurchaseRouter: delaying delivery of PlaceOrder(2,22,250.0,100) for 63 milliseconds
PurchaseRouter: delaying delivery of PlaceOrder(3,33,32.95,10) for 97 milliseconds
PurchaseAgent: placing order for PlaceOrder(2,22,250.0,100)
PurchaseAgent: placing order for PlaceOrder(1,11,50.0,1000)
PurchaseAgent: delivered expired PlaceOrder(3,33,32.95,10) to dead letters
MessageExpiration: is completed.

Format Indicator

Use a Format Indicator to specify the current compositional definition of a given Message (130) type. This technique is discussed in the “Integrating Bounded Contexts” chapter in Implementing Domain-Driven Design [IDDD] by using a Format Indicator as part of a Published Language [IDDD].
When a Command Message (202), a Document Message (204), or an Event Message (207) is first defined, it contains all the information necessary to support all consumers. Otherwise, the systems depending on the given message—in fact, depending on the many messages needed for a complete implementation—would not work. Yet, within even a short period of time any given message type could fail to pack all of the current information for the changing requirements. I’m not limiting this discussion to just one system but possibly many that are integrated.

Over time, there is simply no way that the original definition of all solutionwide messages will remain unchanged. As requirements change, at least some messages must also change. As new integrating systems are added to the overall solution, new messages must be added, and existing messages must be refined. The use of a Format Indicator, as shown in Figure 6.8, can ease the tension between systems that can continue to use the original or earlier format and those that force changes and thus must consume the very latest definition.

![Figure 6.8](image)

**Figure 6.8** Use a Format Indicator to specify the current compositional definition of a given Message (130) type.

As *Enterprise Integration Patterns* [EIP] asserts, some systems can continue to support the original format of any given message. Even so, newer integrators or subsystems with more demanding refinement goals will force existing message types to be enhanced. Quite possibly no two teams involved in the overall solution development will be able to agree on synchronized release dates, let alone merging the schedules of every team involved.

So, how does a Format Indicator work? *Enterprise Integration Patterns* [EIP] defines three possibilities, and I add a fourth, shown here:

- **Version Number**: This approach is discussed in *Implementing Domain-Driven Design* [IDDD]. Each message type embeds a version number as an integer or a text string. The version allows consuming systems to branch on deserialization or parsing logic based on the indicated mes-

---

1. While parsing may sound evil, *Implementing Domain-Driven Design* [IDDD] discusses a very simple and type-safe approach that is easy to maintain.
message format. Generally, at least some, if not most, of the consuming systems may be able to ignore the version number as long as all message changes are additive rather than subtractive. In other words, don't take current correct information properties away from working subsystems; only add on newly required properties.

- **Foreign Key**: This could be the filename of a schema, a document definition, or other kind of format, such as "messagetype.xsd". It could be a URI/URL or some other kind of identity, such as a key that allows for a database lookup. Retrieving the contents of what the foreign key points to would provide the format's definition. This may be less effective since it requires all message consumers to have access to the location that the foreign key points to.

- **Format Document**: Use this to embed the full format definition, such as a schema, into the message itself. This has the obvious size and transport disadvantages when the containing message must be passed between systems.

- **New Extended Message Type**: This approach actually doesn't modify the older message format at all but instead creates a new message that is a superset of the previous message format. Thus, all subsystems that depend only on the original/current version of a message will continue to work, while all subsystems that require the new message can recognize it by its new and distinct type. The new message type name may be closely associated with the one that it extends. For example, if an original Event Message (207) is named OrderPlaced, the newer extending message could be named OrderPlacedExt2. Adding an increasing digit at the end of the message name will allow it to be enhanced multiple times.

Beware When Defining a New Extended Message Type
Defining a New Extended Message Type (the fourth approach in the previous list) may require subsystem actors that happily consume the older message type and that don't understand the new message type to safely ignore the new ones. This may mean logging any newer message types only as a warning rather than interrupting normal system operations with a fatal error. This approach also assumes that both the older and newer types will continue to be sent, at least until all systems can support the extended message type. Otherwise, all systems that were content with the older message type will have to be enhanced to recognize and consume the newest message type, which defeats the purpose of Format Indicator.
The following uses the Version Number approach to enhance the `ExecuteBuyOrder Command Message (202)`:  

```scala
// version 1
case class ExecuteBuyOrder(
    portfolioId: String,
    symbol: String,
    quantity: Int,
    price: Money,
    version: Int)
{
    def this(portfolioId: String, symbol: String,
              quantity: Int, price: Money)
        = this(portfolioId, symbol, quantity, price, 1)
}

// version 2
case class ExecuteBuyOrder(
    portfolioId: String,
    symbol: String,
    quantity: Int,
    price: Money,
    dateTimeOrdered: Date,
    version: Int)
{
    def this(portfolioId: String, symbol: String,
              quantity: Int, price: Money)
        = this(portfolioId, symbol, quantity, price, new Date(), 2)
}
```

Version 1 of the `ExecuteBuyOrder` message specifies a total of four business properties: `portfolioId`, `symbol`, `quantity`, and `price`. On the other hand, version 2 requires a total of five business properties: `portfolioId`, `symbol`, `quantity`, `price`, and `dateTimeOrdered`. The design of both versions of `ExecuteBuyOrder` allows clients to construct both versions passing only four parameters.

```scala
val executeBuyOrder = ExecuteBuyOrder(portfolioId, symbol, quantity, price)
```

In version 2, the `dateTimeOrdered` is automatically provided by the constructor override. The Format Indicator `version` adds an additional property to each of the message types. An overridden constructor on each version allows for the instantiation of `ExecuteBuyOrder` with the version indicator defaulted to the correct value, either 1 or 2.

Since this is a `Command Message (202)`, you can assume that it is the defining and consuming subsystem (one and the same) that requires the new
dateTimeOrdered property to be provided. Yet, it can still support both versions of the message by providing a reasonable default for all version 1 clients.

class StockTrader(tradingBus: ActorRef) extends Actor {
    ...
    def receive = {
        case buy: ExecuteBuyOrder =>
            val orderExecutionStartedOn =
            if (buy.version == 1)
                new Date()
            else
                buy.dateTimeOrdered
            ...
    }
}

Although all version 1 clients will have their buy orders executed based on a slightly inaccurate orderExecutionStartedOn date and time value, they can continue to function with the enhanced StockTrader actor. It is likely, however, that version 1 of ExecuteBuyOrder will be deprecated and all clients will have to update to version 2 by some near-term cutoff.

Summary

In this chapter, you surveyed the kinds of Messages (130) your actors can send and receive and how the intent of each operation determines the kind of Message (130) you will use. You will use Command Message (202) to request an operation to be performed, a Document Message (204) to reply to a query request, and an Event Message (207) to convey that something has happened in your actor system’s domain model. A Command Message (202) and a Document Message (204) will be used together to form a Request-Reply (209). The Actor model always provides the Return Address (211) of the actor to which the reply part of Request-Reply (209) should be sent. You also saw how to leverage a Correlation Identifier (215) to associate a reply with a given request and how you can use Message Sequence (217) when the order of messages to be handled is important. When messages can become stale, use a Message Expiration (218) to indicate the “shelf life.” You also saw how versions of messages can be set by using Format Indicator (222).
Index

Symbols
! (exclamation character), symbols as method names in tell() method, 47

Numbers
32-bit processors, 111
64-bit processors, 111

A
Actor base class
Dotsero, 421
Implementing Actors, 50
Actor model, introduction to, 4
actions actors perform, 13
characteristics of actors and actor systems, 13–15
concurrency and parallelism in, 16–17
contrasted with general messaging systems, 18–19
disadvantages regarding unbounded nondeterminism, 21
explicit nature of, 22–23
extensions available with Akka, 15
as finite state machines, 18
managing nondeterministic systems, 19–21
Object-Capability model (OCM), 21–22
origin of, 11–13
overview of, 10–11
performance benefits of, 122–124
performance benefits of being lock-free, 16
Request-Reply pattern, 17–18
using EIP patterns with, xv–xvi

Actor system (ActorSystem)
canon required by, 335–336
characteristics of actors, 13–15
creating multiple instances, 45
default actors, 45–47
Dotsero actor system, 417–420
finding actors, 48–50
implementing actors, 50
managing with Control Bus, 395
methods, 50–52
overview of, 44–45
remote access between, 59–63
shutting down Dotsero actor system, 418
supervision of, 55–59
TaskManager in implementation of actors, 52–54
top-level supervisors, 47–48

ActorContext, implementing actors, 52–55
actorof() method, 47
ActorRef
ActorSelection compared with, 68–70
finding actors, 48–50
Return Address pattern and, 212–213, 215
as value holder, 47

Actors. See also Transactional Clients/Actors
actions performed by, 13
behavioral testing, 101–102
C# and .NET, 420–425
class characteristics of, 13–15
implementing, 50–55
life-cycle methods, 51
managing with Control Bus, 395
Object-Capability model (OCM) and, 21–22
origin of, 11–13
persistent, 355–357
transactional, 345–355
transient behavior, 339–344
UnboundedMailbox for, 411
unit testing, 99–101

ActorScript, types of actor programming languages, 25

ActorSelection
ActorRef compared with, 68–70
Dotsero support for, 419
finding actors, 48–50
obstacle to use of Object-Capability model, 22
remote lookup and, 68

Adapters [GoF]
Envelope Wrapper and, 314
Service Activators as, 391

advance() method, in Routing Slip example, 287

Aggregates (IDDD)
Aggregate domain model, 13
mapping to messages. See Messaging Mappers
natural aggregates, 354
in transactional design, 354
transient behavior and, 339–344

Aggregators
combined with Recipient List to create Composed Message Processor, 270, 284
Aggregators (continued)
combined with Recipient Lists to create
Scatter-Gather, 254, 272
event, 257–263
in Process Manager example, 298–299
Publish-Subscribe Channel and, 208–209
in Scatter-Gather example, 276
termination criteria used with, 263
Akka Persistence. See Persistence
Akka toolkit
ActorSystem, 44–50
behavioral testing of message processing, 101–102
council clients, 84–87
council roles and events, 76–78
council sharding, 79–84
council-aware routers, 87–99
clustering in general, 71–75
CompleteApp, 102–104
creating council singletons, 79
creating remote actors, 63–68
Dotsero toolkit compared with, 417–420
extensions to Actor model, 15
implementing actors, 50–55
Java API for, 25
lack explicit support for Durable Subscriber, 379
looking up remote actors, 68–71
methods for nodes to join clusters, 75–76
obstacle to use of Object-Capability model, 22
options for obtaining, 26–28
persistence features, 50, 176
programming with, 43–44
remote access between actor systems, 59–63
Retlang compared with, 425
stashing messages using actor-based toolkits, 18
successes using, xv
supervision of actor system, 55–59
testing actors, 99–101
Allen, Jamie, xvii
Amazon.com, scalability approach, 115–116
Amdahl's law, parallelism and, 16–17
Anti-Corruption Layers (IDDD)
designing, 351
transforming messages and, 143, 332
Apache Kafka (Kafka), 382
API, Java API for Akka toolkit, 25
Application Service (IDDD), 353, 390
Applications
to enterprise, 9–10
managing process-based, 309–310
patterns for design and integration, 126
reactive. See Reactive applications
routing complex application processes, 23
why software development is hard, 1–5
Architectural routers, 228
Artificial intelligence, reactive systems used by, 7
Asynchronous messaging, characteristics of actors and actor systems, 13–14
At-least-once delivery
in Guaranteed Delivery, 177, 179–183
Transactional Client/Actor and, 353
At-most-once delivery
purging messages and, 414
Transactional Client/Actor and, 353
Atomic Scala (Eckel and Marsh), 29
Authenticator filter, 136, 138
Availability, trade-off with consistency, 115–116
Await conditions, in Dynamic Router example, 239–240

B
Backpressure, reducing demand on cluster-aware routers, 98–99
BalancingDispatcher, standard Akka dispatchers, 375
Batch-Receiver, Message Sequence and, 218
Batch-Sender, Message Sequence and, 218
BPEL (Business Process Execution Language), 292
Bridges. See Message Bridges
Bus. See Message Bus
Business Process Execution Language (BPEL), 292

C
C#
actors, 420–425
Dotsero toolkit for, 417
C++, Smalltalk compared with, 29
Index

Cache, CPU, 107, 111–112
Canonical Data Models
Message Bus and Message Broker
dependence on, 310
uses of, 333–334
Canonical Message Model
actor systems requiring a canon, 335–336
case classes, 131–132, 134
defined, 313–314
Message Bus interface for services
sharing, 192–193
overview of, 333–335
TradingBus and, 194–198
CAP theorem (Brewer), 115–116
Case classes
Canonical Message Model, 131–132
in defining message types, 42
local event stream of Publish-Subscribe
Channel, 156–158
matching, 133–134
in Recipient List example, 247
Scala tutorial, 39–40
Catchall actor, in Dynamic Router example,
240
Causal consistency, 402, 405
Channel Adapters
data format translation and, 310
Message Bus and, 192
Message Envelope as, 313
overview of, 183–185
packing/unpacking trading messages, 135
translating data, 143
Channel Purger
defined, 393
example, 414–415
overview of, 414
Channels. See Message Channels
Checked items, accessing with Claim Check,
325
Chip performance, 109
Claim Check
defined, 313
example, 326–332
overview of, 325
source file for ClaimCheck patterns, 30
Classes
case classes. See Case classes
Scala tutorial, 31–32
Classification traits, Publish-Subscribe
Channel, 156–160
Clients
cluster clients, 71, 84–87
transactional clients. See Transactional
Clients/Actors
Clock speed, CPU performance and, 109–111
Cluster clients, 71, 84–87
Cluster sharding, 71, 79–84
Cluster singletons
creating, 79
defined, 71
Cluster-aware routers
defined, 71
groups, 89–90
metrics and, 93–97
overview of, 87–89
pools, 90–93
reducing demand, 97–99
Clustering
clients, 84–87
cluster sharding, 79–84
cluster-aware routers, 87–99
creating singletons, 79
nodes joining, 75–76
overview of, 71
roles and events, 76–78
uses of, 72–75
Code blocks, Scala tutorial, 35–36
combined with Aggregator into Scatter-
Gather, 272
Command Messages
Canonical Message Model and, 334–335
constructing/using, 202–204
designing messages with identical impact,
384
Event-Driven Consumers and, 371
Eventual Consistency pattern and, 360
Format Indicators, 223
journaling/storing, 403
making business concepts explicit, 23
mapping subset of actor state to, 344–345
Message pattern example, 131–132
motivations for message construction,
201–202
persistent actor receiving, 356–360
in Process Manager example, 297
reacting to, 8
risk management and, 66
sending, 209, 211
trading options, 134–135
translating data using Message
Translator, 144
Command pattern [GoF], 3
Command Query Responsibility Segregation (CQRS)
Akka support for, 379
Canonical Message Model and, 335
Command Messages and, 202
journaling/storing messages and, 403
mapping Document Message in response to CQRS query, 345
Companion objects, Scala tutorial, 35
Competing Consumers
defined, 337
example, 373
overview of, 371–372
SmallestMailboxRouter example, 376
CompleteApp, 102–104
Complexity stacks, Actor model and, 3–4
Composed Message Processors
example, 271
overview of, 270–271
types of composed routers, 227
using with Pipes and Filters, 263, 284
Composed routers, types of Message Routers, 227
Computation
actors as computational entities, 13
comparing models for, 12
Concurrency
in Actor model, 16–17
characteristics of actors and actor systems, 14–15
conventions for Actor modeling, xxi
multithreading issues, 121
performance benefits of Actor model, 16
Consistency
causal consistency, 402, 405
Eventual Consistency, 354, 360–361
trade-off with availability, 115–116
Constructors
passing arguments to Message Filter, 236
Scala tutorial, 32–34
Consumer endpoints
Competing Consumers, 371–373
Event-Driven Consumer, 371
Polling Consumer, 362–370
Selective Consumer, 377–379
Content Enrichers
Claim Check used with, 325, 332
defined, 313
example, 318–320
immutable actors in, 320
local vs. remote use of scheduler, 321
Object-Capability model (OCM) and, 324
overview of, 317
Content Filters
defined, 313
example, 323–325
overview of, 321–322
Pipes and Filters compared with, 322
Content-Based Routers
Dynamic Routers compared with, 237
example checking inventory availability, 229–232
Message Broker as, 310
Message Bus as, 199, 310
Message Dispatchers compared to, 337, 374
Normalizer using, 333
overview of, 140, 228–229
PurchaseRouter as, 221
resequencing messages and, 265
Splitters combined with, 270
Splitters compared with, 254
type compatibility as factor in routing, 233
types of Message Routers, 227
Control Bus
defined, 393
scope of, 394–396
what can be managed/observed with, 394
Conventions, for Actor modeling, xvi–xxi
Core Domain, Implementing Domain-Driven Design, 10
Cores, CPU performance and, 111–112
Correlation Identifiers
in Aggregator example, 257–258
constructing/using, 215–217
Message Sequence and, 217
motivations for message construction, 201
Smart Proxy establishing, 409
CPU cache, 107, 111–112
CPUs (central processing units)
benefits of Actor model, 122–124
clock speed, 109–111
cores, 111–112
CPU cache, 107, 111–112
Moore’s law, 108–110
CQRS. See Command Query Responsibility Segregation (CQRS)
Customer registration, Enterprise Integration Patterns, 285
Customer-Supplier Development, Implementing Domain-Driven Design, 186
D
Data formats, translators, 309–310
Database, uses of Transactional Client/Actor, 352
Datatype Channels
   Content Enhancer and, 313
   how it works, 168
   overview of, 167
   RabbitMQ example, 168–170
   Selective Consumer routing message types to, 377–379

DDD. See Domain-Driven Design (DDD)

Dead Letter Channel
   deadLetters actor, 222
   dealing with undeliverable messages, 172–175
   motivations for message construction, 202
   rejecting Command Messages, 203
   when to use, 170

Dead letters
   Dotsero support for, 418
   in Dynamic Router example, 240

Deadlocks
   multithreading issues, 117–118
   parallelism and concurrency and, 122

Decrypter filter, 136–138

De-duplication, Enterprise Integration Patterns, 383

Deduplicator filter, 136, 138–139

Design Patterns [GoF]
   Message Brokers and, 308
   State pattern, 53

Detour
   defined, 393
   example, 396–397
   overview of, 395–396
   Wire Tap compared with, 397

Deviations, managing with Control Bus, 395

Dispatchers. See also Message Dispatchers
   managing with Control Bus, 395
   standard Akka dispatchers, 375

Distributed computing, reliability issues with, 115

Distributed Publish-Subscribe Channel
   operations performed on, 165–167
   overview of, 161–162
   properties, 162–164
   DistributedPublishSubscribeMediator, 161–167, 207

Document Messages
   Canonical Message Model and, 334–335
   constructing/using, 204–205
   Event-Driven Consumers and, 371
   Format Indicators, 223
   managing workflows or long-running processes with Document Messages, 206–207
   mapping subset of actor state to, 344–345
   Message Mapping and, 351
   motivations for message construction, 201–202
   quotation fulfillment example, 205–206
   reacting to messages, 8
   replying via Request-Reply, 209
   translating data using Message Translator, 144–145

Documentation, resulting in Published Language, 334

Domain Events
   as Event Messages, 4, 23, 207
   explicit design with, 23
   immutability of, 320
   Implementing Domain-Driven Design (IDDD), 3, 204
   in reactive stack, 4

Domain objects, mapping to messages, 337

Domain-Driven Design (DDD). See also Implementing Domain-Driven Design (IDDD)
   Bounded Context, 334
   Core Domain, 10
   Domain Events, 3, 204, 320
   making business concepts explicit, 22–23
   messaging patterns, xvi
   modeling actors according to specific roles and, 293
   modeling actors using, 3–5, 13-14, 22–23
   supporting SIS, 10

domain-Model actor, top-level supervisors in actor system, 47–48

Domain-specific language (DSL), 292

Dotsero toolkit
   actor system, 417–420
   actors using C# and .NET, 420–425
   implementing, 425–427
   overview of, 417

DSL (domain-specific language), 292

Durable Subscribers
   defined, 338
   example, 380–382
   idempotent actors and, 383
   Message Journal/Store supporting, 393
   overview of, 379
   Transactional Client/Actor and, 353

Dynamic Routers
   compared with Recipient Lists, 245
Dynamic Routers (continued)
example of rule use, 238–245
overview of, 237–238
Selective Consumer and, 377

E
EAI (enterprise application integration), 309–310
EDA (event-driven architectures), nondeterminism in, 21
EIP. See Enterprise Integration Patterns (EIP) (Hohpe and Woolf)
Endowed, security rules in Object-Capability model, 324
Endpoints. See Message Endpoints
Enterprise application integration (EAI), 309–310
Enterprise applications, 9–10
Enterprise Integration Patterns (EIP) (Hohpe and Woolf)
add-ons to basic message, 135
Canonical Data Models and, 333–334
catalog of integration patterns, xv
Command Messages for queries, 202
components supporting Test Message, 412–413
Content Enricher, 317
Content-Based Router, 228–229
Correlation Identifiers, 215–217
customer registration, 285
de-duplication, 383
designing messages with identical impact, 384–390
expired messages and, 222
last-registration-wins rule base, 238
local vs. remote use of scheduler in Content Enricher example, 321
Message Bridge for integrating two messaging systems, 186
on Message Broker as architecture style, 308
message construction and, 201
message parts, 130
Pipes and Filters, 139–140
Return Address pattern and, 212
themes corresponding to Message Channels, 150–151
translating data using Message Translator, 144
on use of Durable Subscriber pattern, 379
on use of Transactional Client/Actor, 351–352
Entities, transient behavior, 339–344
Entry, cluster sharding, 80
Envelope Wrappers
defined, 313
example, 315–316
overview of, 314–315
in Routing Slip example, 287
sending messages through routers, 89
Smart Proxy service messages and, 409
Erlang
Retlang project and, 425
types of actor programming languages, 25
Event Messages
broadcasting Command Messages, 204
Canonical Message Model and, 334–335
characteristics of reactive applications, 7–8
constructing/using, 207–209
decoupling observer from subjects, 154
Event-Driven Consumers and, 371
Eventual Consistency pattern and, 360
Format Indicators, 223
journaling/storing, 403, 405
long-running transactions and, 354
making business concepts explicit, 23
mapping subset of actor state to, 344–345
in Message pattern example, 132
motivations for message construction, 201–202
persistent actor receiving, 356–360
PersistentView and, 362
in polling example, 365
in Process Manager example, 297
reacting to messages, 8
sending StepCompleted message in Claim Check example, 330
Transactional Client/Actor and, 352
translating data using Message Translator, 144
Event Sourcing (IDDD), use in Transactional Client/Actor example, 355, 357–360
Event streams, local event stream of Publish-Subscribe Channel, 155–160
EventBus trait, Publish-Subscribe Channel example, 154–155
Event-driven architectures (EDA), nondeterminism in, 21
Event-Driven Consumers
defined, 337
overview of, 371
themes corresponding to Message Channels, 151
Events, membership events provided by Akka clustering, 78
Eventual Consistency (IDDD)
in Transactional Client/Actor example, 360–361
in transactional design, 354
exceptions, supervisorStrategy, 58–59
Expiration. See Message Expiration
ExpirationMessage trait, 220
Explicitness, characteristic of Actor model, 22–23
Extensible Markup Language (XML)
integrating two messaging systems and, 190
Message Mapping and, 345
querying messages and, 403
translating XML data using Message Translator, 144
Extensible Stylesheet Language (XSL), 144
Extensions to Actor model, Akka, 15
External Event, termination criteria, 263

F
False sharing
deal with, 124–125
multithreading issues, 118
FIFO. See First-in, first-out (FIFO)
Filters
filter actors, 136
Message Filters, 232–237
order of, 137–139
processing steps in pipeline, 139
Finite state machines, Actor model as, 18
First Best, termination criteria, 263
First-in, first-out (FIFO)
Message Channel, 128
purging messages and, 414–415
sequential message delivery, 151–153
Foreign keys, Format Indicators, 224
Format documents, Format Indicators, 224
Format Indicators
constructing/using, 222–226
distinguishing message types, 134
motivations for message construction, 202
Normalizer and, 333
test messages and, 411
Formats, translators, 309–310
FunSuite tool, testing actors, 101
Futures/promises, Akka extensions to Actor model, 15

G
Gateway, Messaging Gateway, 338–344
Generics, Scala syntax for, 41
Google
GSON parser, 345–350
ProtoBuf library for Java, 176
scalability approach, 113–114
Gossip protocol, for cluster communication, 72, 74
Gradle
obtaining Scala and Akka, 28
testing actors, 100
Groups
message groups as use of Transactional Client/Actor, 351
routers in Akka, 87, 89–90
GSON parser, from Google, 345–350
Guaranteed Delivery
Channel Purger and, 414
collection and override options, 182–183
idempotent actors and, 383
implementing using DeadLetter listener, 174–175
Message Journal/Store support, 393
overview of, 175–176
persistence and, 129, 176–177
steps in sending message with, 177–181
Transactional Client/Actor and, 353

H
Header, message parts, 130
Hewitt, Dr. Carl, 12, 15, 19, 25
Hexagonal architecture, Service Activator, 390–391
Horizontal scalability, 7–8
Hyperic Sigar, metrics for cluster-aware routers, 93–94

I
Idempotent Receivers
de-duplication of message, 383
defined, 338
designing messages with identical impact, 384–390
overview of, 382–383
Immutable
immutable actor used in Content Enricher example, 320
Scala tutorial, 37
Implementing Domain-Driven Design (IDDD). See also Domain-Driven Design (DDD)
Aggregate in the domain model, 13
Aggregates in transactional design, 354
IMPLEMENTING DOMAIN-DRIVEN DESIGN (continued)
Anti-Corruption Layers, 143, 332
Application Service, 353, 390
approaches to message types, 134
Bounded Context, 321, 334, 345
Core Domain, 10
Customer-Supplier Development, 186
Domain Events, 3, 204, 320
Event Sourcing, 355, 357–360
Eventual Consistency, 354, 360–361
Format Indicators and, 222–226
immutability, 320
implementing messaging patterns, xvi
making business concepts explicit, 22–23
Ports and Adapters architecture, 2, 390–391
Published Languages, 334–335, 351
reliability issues with distributed computing, 115
Scheduler Bounded Context, 321
import statement, Scala tutorial, 31
Inefficient code, multithreading issues, 118
Infix notation, Scala tutorial, 34
Infrastructure management. See System management
Inktomi, scalability approach used by, 113
Interpreters, 292
Introduced, security rules in Object-Capability model, 324
Invalid Message Channels overview of, 170–172
rejecting Command Messages, 203
J
Java
API for Akka toolkit, 25
ProtoBuf library for, 176
Java Management Extensions (JMX), 76
Java Native Interface (JNI), 176
Java virtual machines (JVMs) managing JVM nodes, 395
Scala as, 25
JavaScript Object Notation (JSON) approaches to message types, 134
Message Mapping and, 345–351
querying messages and, 403
supporting integration of two messaging systems, 190
JMX (Java Management Extensions), 76
JNI (Java Native Interface), 176
Journal, 129, 175. See also Message Journal/Store
JSON. See JavaScript Object Notation (JSON)
JVMs (Java virtual machines) managing JVM nodes, 395
Scala as, 25
K
Kay, Alan, 10–11, 354
Kuhn, Roland, xvii
L
LevelDB, Akka persistence features, 176, 355, 405
Life-cycle messages, in Process Manager example, 296, 298
Livelocks, multithreading issues, 117–118
Load-balancing, cluster-aware routers and, 93
Location transparency Akka extensions to Actor model, 15
network partitioning challenges and, 8
Lock-free concurrency characteristics of actors and actor systems, 14
performance benefits of Actor model, 16
Logging messages, with Wire Tap, 397
Long-running processes Eventual Consistency pattern and, 360
managing, 206–207
Long-running transactions, in transactional design, 354
Looping/iteration, Scala tutorial, 40–41
M
Manager class, Dotsero, 421–422
Many Integrated Core (MIC) architecture, from Intel, 111
Mapping. See Messaging Mappers
Maven methods for obtaining Scala and Akka, 27
testing actors, 100–101
Membership events, provided by Akka clustering, 78
Message Bridges creating Message Bridge actor, 186–192
exchanging messages between actor and nonactor systems, 134
Message Bus and, 192
overview of, 185–186
Message Brokers overview of, 308–310
types of architectural routers, 228
Message Bus
  Canonical Message Model and, 335
  as Content-Based Router, 199
  design considerations for bus-level messages, 134–135
  example of Channel Adapter use, 184–185
  Message Broker compared with, 228, 310
  overview of, 132
  service interface for disparate business systems, 192–193
  TradingBus, 194–198
Message Channels
  Channel Adapter, 183–185
  Datatype Channel, 167–170
  Dead Letter Channel, 172–175
  defined, 127
  distributed Publish-Subscribe Channel, 160–167
  Durable Subscriber use with, 379
  EIP themes corresponding to, 150–151
  Guaranteed Delivery, 175–183
  Invalid Message Channel, 170–172
  local event stream of Publish-Subscribe Channel, 155–160
  Message Bridge, 185–191
  Message Bus, 192–199, 310
  overview of, 128–129
  PersistentActor as, 380
  Point-to-Point Channel, 151–154
  Publish-Subscribe Channel, 154
  reactive applications anticipating failure, 7
  routers dispatching messages over, 140
  types of, 149–150
Message construction
  Command Messages, 202–204
  Correlation Identifier, 215–217
  Document Messages, 204–206
  Event Messages, 207–209
  Format Indicators, 222–226
  managing workflows or long-running processes with Document Messages, 206–207
  Message Expiration, 218–222
  Message Sequence, 217–218
  motivations for, 201–202
  Request-Reply, 209–211
  Return Address, 211–215
Message content, Message Journal/Store, 404
Message Dispatchers
  Competing Consumers and, 371
  defined, 337
  example, 375–377
  overview of, 374–375
  in polling example, 366–367
Message Endpoints
  Channel Adapter as, 183
  Competing Consumers, 371–373
  Datatype Channel and, 167
  defined, 127
  Durable Subscriber, 379–382
  Event-Driven Consumer, 371
  Idempotent Receiver, 382–390
  Message Dispatcher, 374–377
  Messaging Gateway, 338–344
  Messaging Mapper, 344–351
  overview of, 145–147, 337–338
  Polling Consumer, 362–370
  Selective Consumer, 377–379
  Service Activator, 390–391
  Transactional Client/Actor, 351–362
Message Envelope, as Channel Adapter, 313
Message Expiration
  constructing/using, 218–222
  motivations for message construction, 202
Message Filters
  Content Filter compared with, 313
  Content-Based Routers compared with, 228
  example sending orders to inventory systems, 233–237
  overview of, 232–233
  Selective Consumer as, 338, 377
Message History. See Message Metadata/History
Message Id, Message Journal/Store, 404
Message Journal/Store
  Akka persistence features, 129
  column/segment values, 404
  defined, 393
  deleting messages using Channel Purger, 414–415
  example, 405
  journaling messages in, 175
  overview of, 402–404
  snapshots of actor state in, 358–359
  Transactional Client/Actor and, 353
Message Metadata/History
  defined, 393
  example, 398–402
  overview of, 398
Message pattern
  actors accepting Scala types, 131
  Aggregator combining, 258
  attaching metadata to, 398–402
Message pattern (continued)
Canonical Message Model and, 335–336
Command Messages. See Command Messages
Competing Consumers receiving new work, 372
constructing messages. See Message construction
correlating reply with originating message, 216
defined, 127
defining composition with Format Indicator, 222–223
deleting from Message Store using Channel Purger, 414–415
Document Message. See Document Messages
dynamic routing, 237
epipoints, 145–147
Event Messages. See Event Messages
expired messages. See Message Expiration filtering, 139, 232
guaranteeing delivery, 175–177
inspecting actor-to-actor messages with Wire Tap, 397
journaling. See Message Journal/Store
matching case classes, 133
Messaging Gateways and, 339
overview of, 130
PersistentActor reacting to, 183
publishing, 165
receiving/analyzing with Smart Proxy, 406–411
Recipient Lists, 246
routing, 140, 229
saving snapshots, 359
sending actor-to-actor messages on a detour, 395–397
sending to persistent actors, 380–382
sequences of, 217
storing messages. See Message Journal/Store
terminating messages. See Message Endpoints
testing receipt of, 411–413
transforming messages, 143–145, 313
types of messages, 8
Message Routers
Aggregators, 257–263
for complex application processes, 23
Composed Message Processors, 270–271
Content-Based Routers, 228–232
defined, 127
Dynamic Routers, 237–245
Envelope Wrappers and, 313
Message Brokers, 308–310
Message Bus, 310
Message Filters, 232–237
Normalizer using, 333
overview of, 140, 227–228
Process Managers, 292–307
Recipient Lists, 245–254
Resequencers, 264–270
Routing Slips, 285–292
Scatter-Gathers, 272–284
sending messages between two processors, 141–143
Splitters, 254–257
Message Sequence, 217–218. See also Resequencer
Message Store. See Message Journal/Store
Message transformation
Canonical Message Model, 333–336
Claim Check, 325–332
Content Enricher, 317–321
Content Filter, 321–325
Envelope Wrapper, 314–316
Normalizer, 332–333
overview of, 313–314
Message Translators. See also Message transformation
Anti-Corruption Layer, 351
Channel Adapters using, 184
converting datatypes, 169
defined, 127
exchanging messages between actor and nonactor systems, 134
message transformation and, 313
Normalizer using, 333
overview of, 143–145
variants on, 313
Message type, Message Journal/Store, 404
Message-driven characteristic, of reactive applications, 8–9
Messaging Gateways
defined, 337
example of system of transient actors, 339–344
overview of, 338–339
Messaging Mappers
defined, 337
example mapping domain objects, 344–351
overview of, 344
Messaging patterns, Actor model, xvi
  Message Channel pattern, 128–129
  Message Endpoint pattern, 145–147
  Message pattern, 130–135
  Message Router pattern, 140–143
  Message Translator pattern, 143–145
  overview of, 127–128
Pipes and Filters pattern, 135–140
Metadata, attaching to messages, 393, 398–402
Methods
  actor life-cycle methods, 51
  actor system, 50–52
  Scala tutorial, 35–36
Metrics, cluster-aware routers and, 93–97
MIC (Many Integrated Core) architecture, from Intel, 111
Microprocessors
  benefits of Actor model, 122–124
  cores and CPU cache, 111–112
  Moore’s law, 108–110
  sending messages between two processors, 141–143
  transistors and clock speed and, 107
Microsoft Message Queuing (MSMQ), 192–193
Monitoring, supported by Typesafe Activator, 394
Moore’s law, 108–110
MSMQ (Microsoft Message Queuing), 192–193
Multithreading
  difficulties in, 116–119
  lock-free queue implementations and, 120
  resource sharing and, 120–121
  scalability and, 112
Mutable collections, Scala tutorial, 37
N
.NET
  actors, 420–425
  Dotsero toolkit for, 417
New Extended Message Type, Format Indicators, 224
Nodes
  managing with Control Bus, 395
  methods for joining clusters, 75–76
Non-deterministic systems
  managing, 19–21
  overview of, 19
  use case examples, 19–21
Normalizer
  defined, 313
  exchanging messages between actor and nonactor systems, 134
  overview of, 332–333
Notifications, IDDD messages, 134
O
Object models, Actor model compared with, 21
Object-Capability model (OCM), 21–22, 324
Objects
  companion objects in Scala, 35
  mapping domain objects to messages, 337
  Scala tutorial, 31–32
Observer pattern [GoF], 154
OCM (Object-Capability model), 21–22, 324
OnReceive(), Dotsero methods, 422
OrderAcceptanceEndpoint filter, 136–137
OrderManagementSystem filter, 136, 139
P
Package syntax, Scala, 30
Parallelism
  of actions in Actor model, 13
  Amdahl’s law and, 16–17
  benefits of Actor model, 11
  characteristics of actors and actor systems, 15
  multithreading issues, 121
Parent-child relationships, conventions for Actor modeling, xviii
Parenthood, security rules in Object-Capability model, 324
Pattern language, creating, xvii
Pattern matching, Scala tutorial, 42–43
Patterns. See also Message pattern for application design and integration, 126
catalog of, xvii
Performance
  benefits of Actor model, 122–124
  clock speed, 109–111
  cores and CPU cache, 111–112
  false sharing and, 124–125
  importance of transistors, 107–109
  multithreading, 116–122
  overview of, 107
  scale and, 112–116
Persistence
activating in project, 355
of actor state during transactions, 352
Akka toolkit, 50, 129
conventions for Actor modeling, xviii
Guaranteed Delivery and, 176–177
persistent actors, 355–357
persistent views, 361–362
retaining/purging messages from Message Store, 403–405
PersistentActor
creating, 380
Durable Subscribers, 379
Guaranteed Delivery, 177
retaining/purging messages from Message Store, 405
transactional actors and, 355–357
PersistentView
how it works, 361–362
supporting CQRS, 379
PinnedDispatcher, standard Akka dispatchers, 375
Pipes and Filters
Composed Message Processors and, 263, 270, 284
Content Filter compared with, 322
defined, 127
Detour as form of, 396
filter actors, 136
filter order, 137–139
overview of, 135
pipeline, 136–137
processing steps in pipeline and, 139
types of architectural routers, 228
using with inventory system, 237
Play Framework, example of reactive user interface, 4
Point-to-Point Channels
Event-Driven Consumers and, 371
managing workflows or long-running processes with Document Messages, 206–207
Message Channel, 129
overview of, 151
Publish-Subscribe Channel, 154
sending Command Messages over, 204
sequential message delivery, 151–154
Polling, multithreading issues, 121
Polling Consumer
Batch-Receiver as, 218
Competing Consumers and, 371
defined, 337
example, 363–370
overview of, 362–363
volunteering style of work provider, 367, 376
Pools, routers in Akka, 87, 90–91
Ports and Adapters architecture
enterprise application architecture, 2
Service Activator and, 390–391
translating data using Message Translator, 143
PostgreSQL, 403
Process Managers
activities diagram, 299
building block in Actor model, 17–18
for complex application processes, 23
components of loan rate quotation example, 294
Correlation Identifiers and, 217
Document Message as reply, 204
defined, 294–307
fine-grained control with, 360–361
managing workflows or long-running processes, 207
motivations for message construction, 201
overview of, 292–293
for process-based applications, 309–310
remoting and, 65
top-level supervisors in actor system, 47–48
ProcessManagers actor, 47–48
Processors. See Microprocessors
Programming
with Akka toolkit, 43–44
with Scala programming language, 29–30
ProtoBuf library, for Java, 176
Protocol Buffers, serialization options, 60
Published Languages
crafting, 351
documentation resulting in, 334
Format Indicators, 222–226
Publish-Subscribe Channel
Apache Kafka (Kafka), 382
broadcasting Command Messages, 204
Composed Message Processors and, 284
Distributed Publish-Subscribe Channel, 161–162
DistributedPubSubMediator, 207
Durable Subscriber use with, 379
Event Messages, 207–208
implementing distributed Process Managers, 293
implementing Scatter-Gathers, 263, 272
local event stream of, 153–160
operations performed on
DistributedPubSubMediator, 165–167
overview of, 154
properties of DistributedPubSubMediator,
162–164
Publishers, 154

Q
Queries, 209. See also Command Query
Responsibility Segregation (CQRS)

R
Reactive applications
CompleteApp not for use in, 103
difficulties of, 1–5
elasticity of, 7–8
introduction to, 5–6
message driven characteristic, 8–9
performance benefits of Actor model, 123
resilience of, 6–7
responsiveness of, 6
Reactive Manifesto, 5
Reactive routers. See Message Routers
Reactive stack, domain events in, 4–5
Real, Evaluate, Print, Loop (REPL), 27
receive block, implementing actors, 50
Receivers
Document Message, 205
Message Channel, 128
Recipient Lists
combined with Aggregator into
Composed Message Processor, 270, 284
combined with Aggregator into Scatter-
Gather, 272
combining with Aggregator, 257–263
example performing price quoting,
245–254
overview of, 245
Registration, Enterprise Integration Patterns
(EIP), 285
Remote creation, 62, 63–68
Remote lookup, 62–63, 68–71
Remote procedure calls (RPC), 390
RemoteActorRef, 153–154
Remoting, Akka support
local vs. remote use of scheduler in
Content Enricher example, 321
message ordering and, 153–154
overview of, 59–63
remote creation, 63–68
remote lookup, 68–71
REPL (Real, Evaluate, Print, Loop), 27
RequestForQuotation message, 245
Request-Reply
constructing/using, 209–211
Envelope Wrappers and, 313
mimicking polling, 337, 363–370
performing with Dotsero, 420–425
Smart Proxy and, 393, 406
uses of Transactional Client/Actor, 352
Resequencer
Akka Scheduler and, 221
example, 265–270
Message Sequence and, 218
overview of, 264–265
resilience
characteristics of reactive applications,
6–7
designing for, 124
Responsiveness, characteristics of reactive
applications, 6
RESTful resources
custom media types, 134
enriching content and, 321
Service Activator and, 390
Rettig library, use by Dotsero, 425
Rettig, Mark, 425
Return Address
in Command Message, 203
constructing/using, 211–215
Envelope Wrapper used with, 314–316
motivations for message construction,
201
Smart Proxy and, 406
Risk assessment, Idempotent Receiver,
388–390
Root guardian, default actors in actor system,
45
RoundRobinRouter, standard Akka
routers, 376
Routers
cluster-aware, 87–99
Message Router. See Message Routers
standard Akka, 376
Routing Slips
example, 285–292
overview of, 285
as type of Process Manager, 292
types of composed routers, 227
RPC (Remote procedure calls), 390
Index

Rules
  Dynamic Routers using, 237
  Recipient Lists, 245
  security rules in Object-Capability model, 324

S
  SBE (Specification By Example), 101–102
  sbt (Simple Build Tool)
    build file for Akka clustering, 73
    methods for obtaining Scala and Akka, 26–27
  Scala programming language
    case classes, 39
    classes and objects, 31–32
    code blocks (methods), 35–36
    companion objects, 35
    comprehending recipients based on business rules, 250
    constructors, 32–34
    generics, 41
    immutable/mutable collections, 37
    import statement, 31
    infix notation, 34
    looping/iteration, 40–41
    options for obtaining, 26–28
    overview of, 25
    package syntax, 30
    pattern matching, 42–43
    programming with, 29–30
    source files, 30
    successes using, xv
    symbols as method names, 37
    traits, 38–39
  Scalability
    approach used by Amazon.com, 115–116
    approach used by Google, 113–114
    data format translation and, 309
    elasticity compared with, 7–8
    performance and, 112–113
    remoting and, 66
    scale up vs. scale out, 113
  ScalaTest, testing actors, 101
  Scatter-Gather
    Composed Message Processor and, 271
    example, 273–284
    Message Expiration and, 219
    options for implementing, 263
    overview of, 272–273
    types of composed routers, 227
  Scheduler Bounded Context, IDDD, 321
  Scheduling
    Akka Scheduler and, 221
    local vs. remote use of scheduler in
      Content Enricher example, 321
    multithreading issues, 121
    performance benefits of Actor model, 124
  Scratch, types of actor programming languages, 25
  Search engines, scalability approach used by
    Google and Inktomi, 113–114
  Security rules, Object-Capability model (OCM), 324
  Seed nodes, 75–76
  Selective Consumers
    defined, 338
    example, 377–379
    overview of, 377
  Senders
    Document Message, 205
    Message Channel, 128
  Send-Receive message pairs, in Transactional Client/Actor, 351
  Sequencing, xxi. See also Resequencer
  Sequential programming
    compared with Actor model, 11
    not good at supporting parallelism and concurrency, 122
  Serialization
    message mapper as serializer, 345
    remoting and, 60
  Service Activator
    defined, 338
    overview of, 390–391
  Service autonomy, 309
  Service Layer, 333
  Sharding. See Cluster sharding
  Share nothing
    characteristics of actors and actor systems, 14
    performance benefits of Actor model, 123
  Sharing
    dealing with false sharing, 124–125
    multithreading issues, 120–121
  Simple Build Tool (sbt)
    build file for Akka clustering, 73
    methods for obtaining Scala and Akka, 26–27
  Simplicity stacks, in Actor model, 3–5
  Single Responsibility Principle (SRP), 13
  Singletons. See Cluster singletons
  SIS (Strategic information system), 9–10
INDEX 447

SmallestMailboxRouter
  in Competing Consumers example, 372–373
  standard Akka routers, 376
Smalltalk
  compared with C++, 29
  transactional actors and, 354
  types of actor programming languages, 25
Smart Proxy
  defined, 393
  example, 406–411
  overview of, 406–411
Snapshots, of actor state, 358–359
SOA
  Routing Slips and, 285
  service autonomy and, 309
Software applications. See Applications
Source files, Scala, 30
Specification By Example (SBE), 101–102
Splitters
  breaking large data structures into smaller message types, 322
  combined with Content-Based Router into Composed Message Processor, 270
  Content-Based Routers compared with, 228
  Dynamic Routers compared with, 237
  example splitting OrderPlaced message, 254–257
  overview of, 254
  resequencing messages and, 265
SRP (Single Responsibility Principle), 13
Starvation, multithreading issues, 117–118
Stashing messages, using actor-based toolkits, 18
State machines
  Actor model as finite state machine, 18
  characteristics of actors and actor systems, 14
  State pattern, actor ability for dynamic behavior change, 53
stopProcess(), in Process Manager example, 296
Store, Message Journal/Store, 129, 175
Strategic information system (SIS), 9–10
Streams, Message Journal/Store, 404–405
Subjects, Observer pattern [GoF], 154
Subscribers, Publish-Subscribe Channel, 154
Supervision
  of actor system, 55–59
  Akka extensions to Actor model, 15
  Dotsero support for, 419–420
  top-level supervisors in actor system, 47–48
supervisorStrategy
  exceptions, 58–59
  overriding, 57–58
  supervision of actor system, 55–57
Symbols, as method names in Scala, 37, 47
System guardian, default actors in actor system, 45
System management
  Channel Purger, 414–415
  Control Bus, 394–395
  Detour, 395–397
  Message Journal/Store, 402–405
  Message Metadata/History, 398–402
  overview of, 393–394
  Smart Proxy, 406–411
  Text Message, 411–413
  Wire Tap, 397–398
T
  Task scheduling, multithreading issues, 121
  TaskManager, implementing, 52–54
  TCP (Transmission Control Protocol), 60
  tell() method, Dotsero methods, 422
tell() method, 47
Termination
  actors, 50–55
  Aggregators and, 263
  conventions for Actor modeling, xviii
  in random number example, 368–369
  in Routing Slip example, 291
  in Scatter-Gather example, 278–279
  Test Data Generator, EIP components supporting Test Message, 413
  Test Data Verifier, EIP components supporting Test Message, 413
  Test Message Injector, EIP components supporting Test Message, 413
  Test Message Separator, EIP components supporting Test Message, 413
  Test Messages defined, 393
  EIP support, 412–413
  example, 412
  overview of, 411
  Test Output Processor, EIP components supporting Test Message, 413
Testing actors
  behavioral testing of message processing, 101–102
Testing actors (continued)
  overview of, 99
  unit testing actors, 99–101
TestKit
  behavioral testing of message processing, 102
  importing, 101–102
Threads. See Multithreading
Timeout
  in random number example, 368–369
  in Scatter-Gather example, 278–279
  termination criteria, 263
Timeout with Override, 263
timeToLive value, Message Expiration and, 219
Trading Bus, 194–198, 335
Traits
  PersistentActor, 355
  Scala tutorial, 38–39
  Uniform Access Principle and, 407
Transactional Actors. See Transactional Clients/Actors
Transactional Clients/Actors
  Akka persistence features, 129
defined, 337
  Event Sourcing pattern used with, 357–360
  Eventual Consistency pattern used with, 360–361
  overview of, 351–353
  persistent views, 361–362
  transactional actors, 354–355
  transactional clients, 353
Transforming messages. See Message transformation
Transient behavior, of entities or aggregates, 339–344
Transistors
  history of, 108
  importance of, 107–108
  Moore’s law, 108–110
  overview of, 107
Translators, 309–310. See also Message Translators
Transmission Control Protocol (TCP), 60
Types. See Datatype Channels
  Typesafe, xvii
  Typesafe Activator
    methods for obtaining Scala and Akka, 26
    monitoring support, 394
  Typesafe Console, 394
U
  UnboundedMailbox, for actors, 411
  Uniform Access Principle, 407
  Unit tests, 99–101
  User guardian, default actors in actor system, 45–47
V
  Value objects, in registration message, 285–286
  Version numbers, Format Indicators, 223–225
  Vertical scalability, 7–8
  Volunteering style of work provider, Polling Consumer, 367, 376
W
  Wait
    Await condition in Dynamic Router example, 239–240
    Wait for All termination criteria, 263
  WhitePages, performance benefits of Actor model, 122–123
  Wire Tap
    defined, 393
designing with Smart Proxy, 406, 411
  Detour compared with, 395
  example, 397–398
  overview of, 397
  viewing messages in Message Store, 403
  Workflow, managing with Document Messages, 206–207
  Workstations, clustering, 113
X
  XML (Extensible Markup Language). See Extensible Markup Language (XML)
  XSL (Extensible Stylesheet Language), 144