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

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

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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.
ISBN 978-0-13-384683-6 (hardcover : alk. paper)
1. Scala (Computer program language) 2. Application software—Development. 3. Computer multitasking—Mathematics. 4. Java virtual machine. 5. Business enterprises—Data processing. I. Title.
QA76.73.S28V47 2016
005.2'762—dc23
2015016389

ISBN-13: 978-0-13-384683-6
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

Contents

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

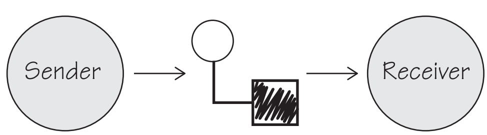


Figure P.1 A Sender actor sends a message to a Receiver actor.

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.

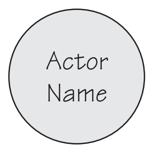


Figure P.2 A persistent or long-lived actor

Figure P.3 An ephemeral or short-lived actor

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

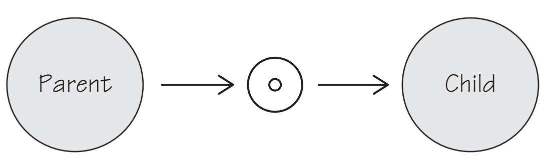


Figure P.4 A parent actor creates a child actor.

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.

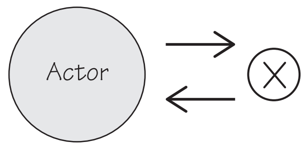


Figure P.5 Actor self-termination

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.

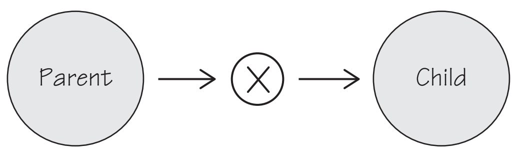


Figure P.6 One actor terminates another actor.

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

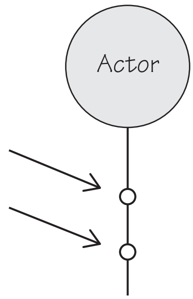


Figure P.7 An actor's lifeline is shown as two asynchronous messages are received.

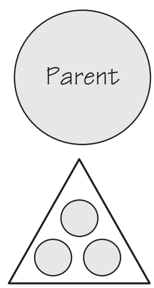


Figure P.8 An actor's child hierarchy

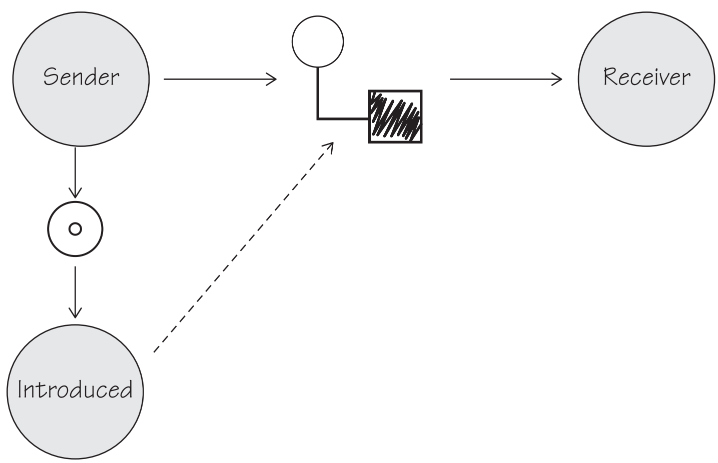


Figure P.9 A child actor is introduced by a sender to the receiver by means of a message.

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.

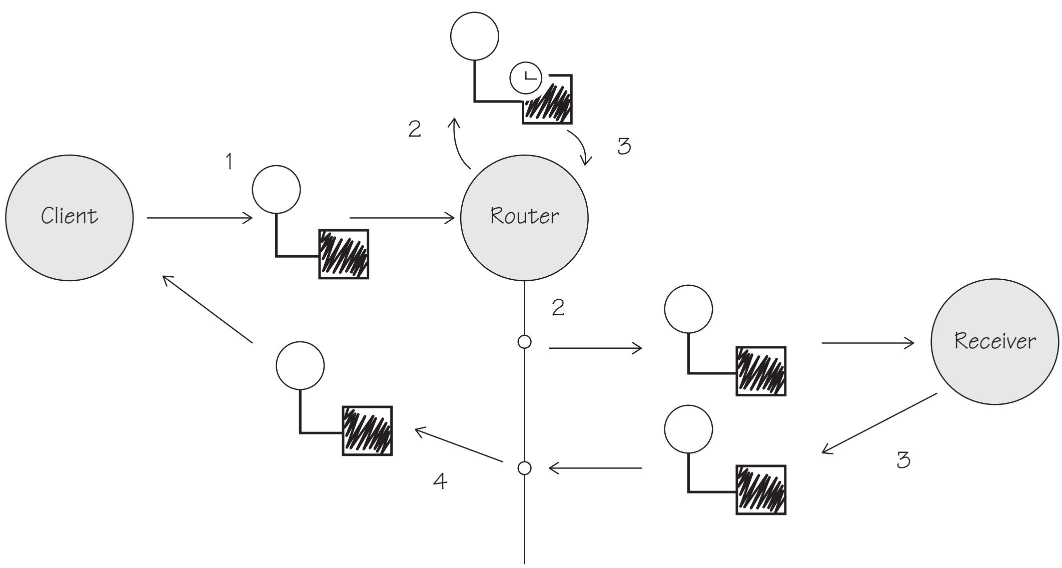


Figure P.10 Message sequence and concurrency are represented by sequence numbers.

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.

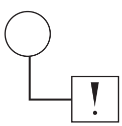
- *Huge amounts of data*: Sometimes you need more than a *Correlation Identifier (215)* to bundle related messages. What happens if you can correlate a set of messages but you also need to ensure that they are ordered according to some application-specific sequence? That's the job of a *Message Sequence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



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)*.

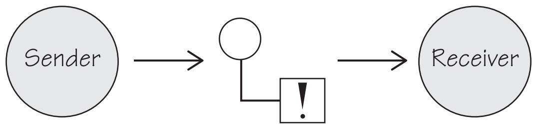


Figure 6.1 The `Sender`, by means of a Command Message, tells the `Receiver` to do something.

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:

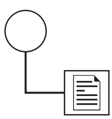
```
case class ExecuteBuyOrder(
    portfolioId: String,
    symbol: String,
    quantity: Int,
    price: Money)

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`.

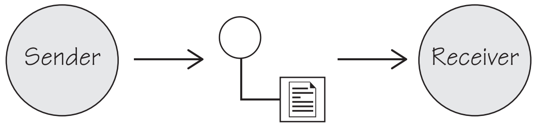


Figure 6.2 The `Sender`, by means of a Document Message, provides the `Receiver` with information but without indicating how it should be used.

The following is a Document Message that conveys data about a quotation fulfillment:

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

The information provided by the `QuoteFulfillment` 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 `QuoteFulfillment` Document Message:

```
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 `QuoteFulfillment` Document Message:

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

```
    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(
```

```
        rfqId,
        quotesRequested,
        priceQuotes :+ fulfilledPriceQuote,
        requester)
  }
}
```

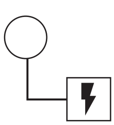
The Document Message itself may contain some data describing how each processing step actor is to dispatch to the next step. This might be handled by placing the actor address+name of each step inside the original document in the order in which the steps should occur. As each step completes, it simply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

```
val quotationFulfillment =
       quotationFulfillment.appendWith(newPriceQuote)
quotationFulfillment.stepFollowing(name) ! quotationFulfillment
```

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

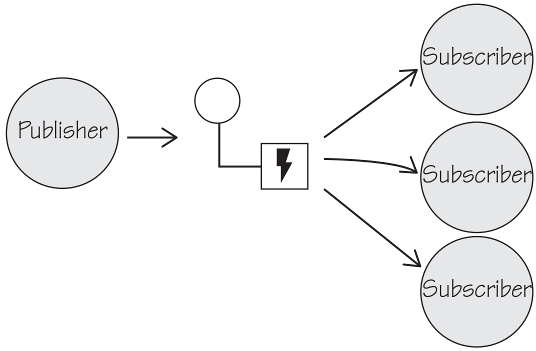


Figure 6.3 Using an Event Message, a `Publisher` may notify multiple `Subscriber` actors about something that happened in the domain model.

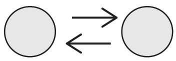
interested parties about a given event. Yet, sometimes it may be appropriate to tell a specific actor about an event or tell the specific actor and also publish to an abstract set of subscribers. See also Domain Events as discussed in *Implementing Domain-Driven Design* [IDDD].

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



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].

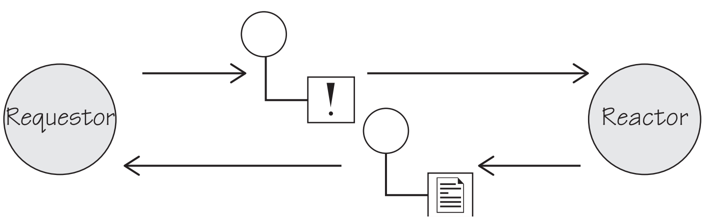


Figure 6.4 A `Requestor` and a `Reactor` collaborate with each other using Request-Reply.

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.

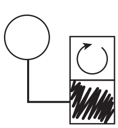
The first message, `StartWith`, is sent to the `Client` to tell it to start the Request-Reply scenario. Although `StartWith` is a *Command Message (202)* request, note that the `Client` does not produce a reply to the `App`. The `StartWith` message takes one parameter, which is the instance of the `Server` 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.

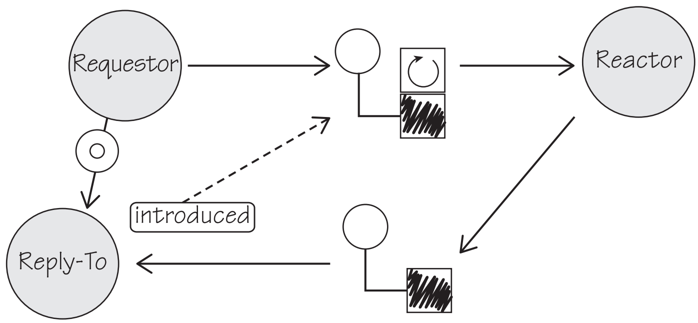


Figure 6.5 A `Requestor` uses a Return Address to tell the `Reactor` to reply to a third party.

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 (209)* 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.

```
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:

```
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.

```
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")
  }
}
```

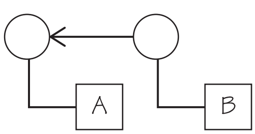
This is the output produced by the Return Address example:

```
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



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

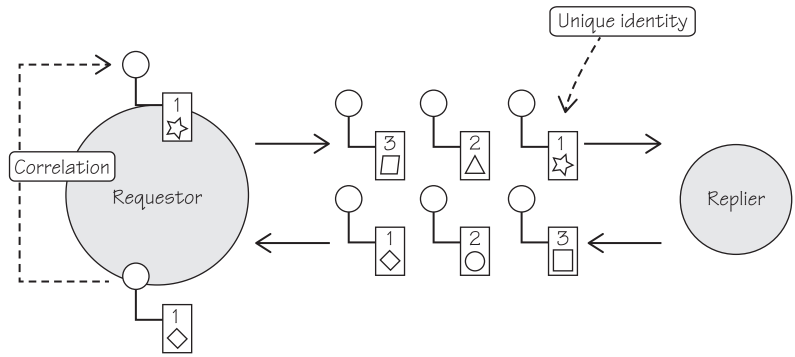


Figure 6.6 A `Requestor` attaches a Correlation Identifier to outgoing *Messages (130)* in order for the `Replier` to associate its replies with the originating Message.

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.

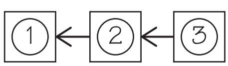
Each of the following message types are correlated using the `rfqId` (request for quotation ID):

```
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)
```

```
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 hock 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:

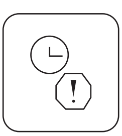
1. If M1 is delivered, it must be delivered before M2 and M3.
2. If M2 is delivered, it must be delivered before M3.
3. Since there is no (default) guaranteed delivery, any of the messages M1, M2, and/or M3, may be dropped, in other words, not arrive at Batch-Receiver.

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

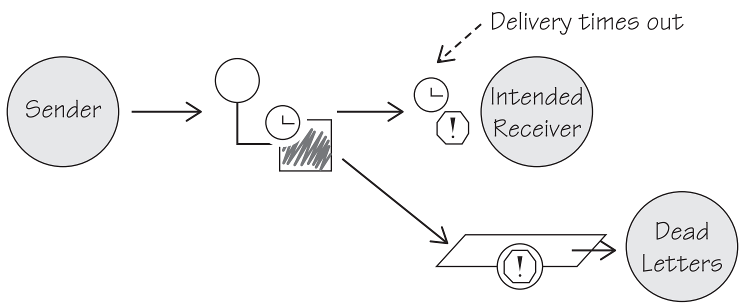


Figure 6.7 A Message Expiration is attached to a Message that may become stale.

(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.

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

  def isExpired(): Boolean = {
    val elapsed = System.currentTimeMillis() - occurredOn

    elapsed > timeToLive
  }
}
```

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)*:

```
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")
```

```
  purchaseRouter ! PlaceOrder("1", "11", 50.00, 1000)
  purchaseRouter ! PlaceOrder("2", "22", 250.00, 100)
  purchaseRouter ! PlaceOrder("3", "33", 32.95, 10)

  awaitCompletion
  println("MessageExpiration: is completed.")
}
```

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.

```
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:

```
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↵
```

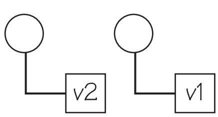
```
$placeOrder")
      }

      MessageExpiration.completedStep()

    case message: Any =>
      println(s"PurchaseAgent: received unexpected:↵
$message")
  }
}
```

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.

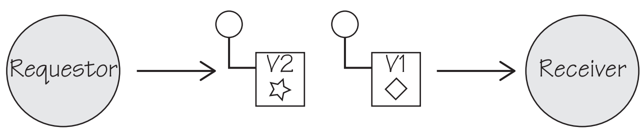


Figure 6.8 Use a Format Indicator to specify the current compositional definition of a given *Message (130)* type.

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[1] 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.

sage 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)*:

```
// 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.

```
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 sent 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

D