Actors in Scala

PrePrint™ Edition

Excerpt

artima

ARTIMA PRESS

MOUNTAIN VIEW, CALIFORNIA

Chapter 1

Concurrency Everywhere

The actor model of concurrency was born of a practical need: When Carl Hewitt and his team at MIT first described actors in the 1970s, computers were relatively slow.[1] While it was already possible to divide up work among several computers and to compute in parallel, Hewitt's team wanted a model that would not only simplify building such concurrent systems, but would also let them reason about concurrent programs in general. Such reasoning, it was believed, would allow developers to be more certain that their concurrent programs worked as intended.

Although actor-based concurrency has been an important concept ever since, it is only now gaining wide-spread acceptance. That is in part because until recently no widely used programming language offered first-class support for actors. An effective actors implementation places a great burden on a host language, and few mainstream languages were up to the task. Scala rises to that challenge, and offers a full-featured implementation of actor-based concurrency on the Java virtual machine (JVM).[2] Because Scala code seamlessly interoperates with code and libraries written in Java, or any other JVM language, Scala actors offer an exciting and practical way to build scalable and reliable concurrent programs.

Like many powerful concepts, the actor model can be understood and used on several levels. On one level, actor-based programming provides an easy way to exchange messages between independently running threads or processes. On another level, actors make concurrent programming gener-

[1]Hewitt *et al.*, *A Universal Modular ACTOR Formalism for Artificial Intelligence* [**?**]

[2]Haller and Odersky, *Scala Actors: Unifying thread-based and event-based programming* [**?**]

ally simpler, because actors let developers focus on high-level concurrency abstractions and shield programmers from intricacies that can easily lead to errors. On an even broader level, actors are about building reliable programs in a world where concurrency is the norm, not the exception—a world that is fast approaching.

This book aims to explain actor-based programming with Scala on all those levels. Before diving into the details of Scala actors, it helps to take a step back and place actors in the context of other approaches to concurrent programming, some of which may already be familiar to you.

1.1 Scaling with concurrency

The mainstream computing architectures of the past decades focused on making the execution of a single thread of sequential instructions faster. That led to an application of Moore's Law to computing performance: Processor performance per unit cost has doubled roughly every eighteen months for the last twenty years, and developers counted on that trend to ensure that their increasingly complex programs performed well.[3]

Moore's Law has been remarkably accurate in predicting processor performance, and it is reasonable to expect processor computing capacity to double every one-and-a-half years for at least another decade. To make that increase practical, however, chip designers had to implement a major shift in their design focus in recent years. Instead of trying to improve the clock cycles dedicated to executing a single thread of instructions, new processor designs make it possible to execute many concurrent instruction threads on a single chip. While the clock speed of each computing core on a chip is expected to improve only marginally over the next few years, processors with dozens of cores are already showing up in commodity servers, and multicore chips are the norm even in inexpensive desktops and notebooks.

This shift in the design of high-volume, commodity processor architectures, such as the Intel x86, has at least two ramifications for developers. First, because individual core clock cycles will increase only modestly, we will need to pay renewed attention to the algorithmic efficiency of sequential code. Second, and more important in the context of actors, we will need to design programs such that they take maximum advantage of available processor cores. In other words, we not only need to write programs that work

[3] Sutter, *The free lunch is over: A fundamental turn toward concurrency* [**?**]

correctly on concurrent hardware, but also design programs that opportunistically scale to all available processing units or cores.

1.2 Actors versus threads

In a concurrent program, many independently executing threads, or sequential processes, work together to fulfill an application's requirements. Investigation into concurrent programming has mostly focused on defining how concurrently executing sequential processes can communicate such that a larger process—for example, a program that executed those processes—can proceed predictably.

The two most common ways of communication among concurrent threads are synchronization on shared state, and message passing. Many familiar programming constructs, such as semaphores and monitors, are based on shared-state synchronization. Developers of concurrent programs are familiar with those structures. For example, Java programmers can find these structures in the `java.util.concurrent` library.[4] Among the biggest challenges for anyone using shared-state concurrency are avoiding concurrency hazards, such as data races and deadlocks, and scalability.[5]

Message passing is an alternative way of communication among cooperating threads. There are two important categories of systems based on message passing. In channel-based systems, messages are sent to *channels* (or *ports*) that processes can share. Several processes can then receive messages from the same shared channels. Examples of channel-based systems are MPI[6] and systems based on the CSP paradigm[7], such as the Go language. Systems based on actors (or agents, or Erlang-style processes[8]) are in the second category of message-passing concurrency. In these systems, messages are sent directly to actors; it is not necessary to create intermediary channels between processes.

[4]Goetz *et al.*, *Java Concurrency in Practice* [**?**]

[5]One of the reasons why scalability is hard to achieve using locks (or Java-style synchronization) is the fact that coarse-grained locking increases the amount of code that is executed sequentially. Moreover, accessing a small number of locks (or, in the extreme case, a single global lock) from several threads may increase the cost of synchronization significantly.

[6]Gropp *et al.*, *Using MPI: Portable Parallel Programming with the Message–Passing Interface* [**?**]

[7]Hoare, *Communicating sequential processes* [**?**]

[8]Armstrong *et al.*, *Concurrent Programming in Erlang* [**?**]

An important advantage of message passing over shared-state concurrency is that it makes it easier to avoid data races. If processes communicate only by passing messages, and those messages are immutable, then race conditions are avoided by design. Moreover, anecdotal evidence suggests that this approach in practice also reduces the risk of deadlock. A potential disadvantage of message passing is that the communication overhead may be high. To communicate, processes have to create and send messages, and these messages are often buffered in queues before they can be received to support asynchronous communication. In contrast, shared-state concurrency enables direct access to shared memory, as long as it is properly synchronized. To reduce the communication overhead of message passing, large messages should not be transferred by copying the message state; instead, only a reference to the message should be sent. However, this reintroduces the risk for data races when several processes have access to the same mutable (message) data. It is an ongoing research effort to provide static checkers (for instance, the Scala compiler plug-in for uniqueness types[9] that can verify that programs passing mutable messages by reference do not contain data races.

Let's take a step back, and look at actor-based programming from a higher-level perspective. To appreciate the difference, and the relationship, between more traditional concurrency constructs and actors, it helps to pay a brief visit to the local railroad yard.

Imagine yourself standing on a bridge overlooking the multitude of individual tracks entering the rail yard. You can observe many seemingly independent activities taking place, such as trains arriving and leaving, cars being loaded and unloaded and so on.

Suppose, then, that your job was to design such a railroad yard. Thinking in terms of threads, locks, monitors, and so on, is similar to the problem of figuring out how to make sure that trains running on parallel tracks don't collide. It is a very important requirement; without that, the rail yard would be a dangerous place, indeed. To accomplish that task, you would employ specialized artifacts, such as semaphores, monitors, switches.

Actors illuminate the same rail yard from the higher perspective of ensuring that all the concurrent activities taking place at the rail yard progress smoothly: that all the delivery vehicles find their ways to train cars, all the trains can make their progress through the tracks, and all the activities are

[9]Haller and Odersky, *Capabilities for Uniqueness and Borrowing* [**?**]

properly coordinated.

You will need both perspectives when designing a rail yard: Thinking from the relatively low-level perspective of individual tracks ensures that trains don't inadvertently cross paths; thinking from the perspective of the entire facility helps ensure that your design faciliates smooth overall operation, and that your rail yard can scale, if needed, to accommodate increased traffic. Simply adding new rail tracks only goes so far: you need some overall design principles to ensure that the whole rail yard can grow to handle increased traffic, and that greater traffic can scale to the full capacity of the tracks.

Working on the relatively low-level details of individual tracks (or problems associated with interleaving threads), on the one hand, and the higher-level perspective of the entire facility (actors) on the other, require somewhat different skills and experience. An actor-based system is often implemented in terms of threads, locks, monitors, and the like, but actors hide those low-level details, and allow you to think of concurrent programs from a higher vantage point.

1.3 The indeterministic soda machine

In addition to allowing you to focus on the scalability aspect of concurrent applications, actors' higher-level perspective on concurrency is helpful because it provides a more realistic abstraction for understanding how concurrent applications work. Specifically, concurrent programs exhibit two characteristics that, while also present in sequential applications, are especially pronounced when a program is designed from the ground up to take advantage of concurrency. To see what these are, we need only to stop by the office soda machine.

A soda machine is convenient not only to quench our thirst on a hot summer day, but also because it's a good metaphor for a kind of program that moves from one well-defined state to another. To start out, a soda machine awaits input from the user, perhaps prompting the user to insert some coins. Inserting those coins causes the soda machine to enter a state where it can now ask the user to make a selection of the desired drink. As soon as the user makes that selection, the soda machine dispenses a can and moves back into its initial state. On occasion, it may also run out of soda cans—that would place it in an "out of service" state.

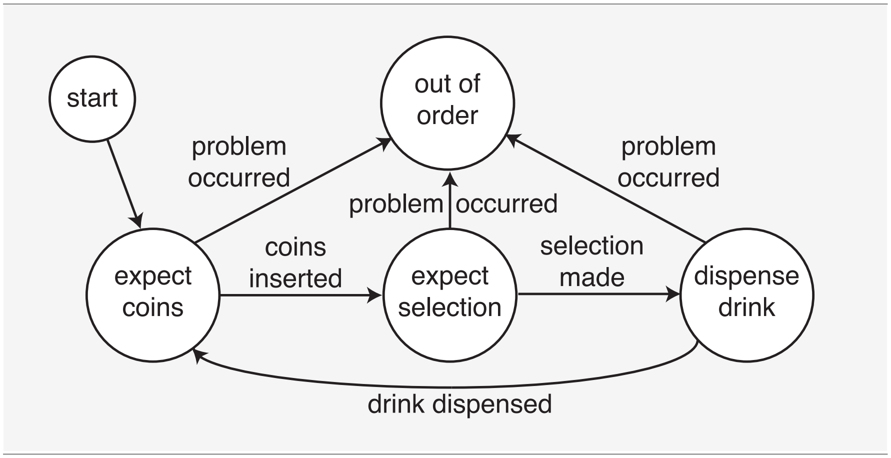

Figure 1.1 · Global state in a soda machine and state transitions.

At any point in time, a soda machine is aware of only one state. That state is also global to the machine: Each component—the coin input device, the display unit, the selection entry keypad, the can dispenser, and so on—must consult that global state to determine what action to take next: For instance, if the machine is in the state where the user has already made his selection, the can dispenser component is allowed to release a soda can into the output tray.

In addition to always being in a well-defined state, our simple abstraction suggests two further characteristics of a soda machine: First, that the number of possible states the machine can enter is finite and, second, that given any one of those possible states, we can determine in advance what the next state will be. For instance, if you inserted a sufficient amount of coins, you would expect to be prompted for the choice of drink. And having made that choice, you expect the selected drink to be dispensed.

Of course, you've probably had occasions to experience soda machines that did not exactly behave in such a predictable, deterministic, way. You may have inserted plenty of coins, but instead of being prompted for your choice, you were presented with an unwelcoming "OUT OF ORDER" message. Or you may not have received any message at all—but also did not receive your frosty refreshment, no matter how hard you pounded the machine. Real-world experience teaches us that soda machines, like most physical objects, are not entirely deterministic. Most of the time they move from one well-defined state to another in an expected, predetermined fashion; but on occasion they move from one state to another—to an error state, for instance—in a way that could not be predicted in advance.

A more realistic model of a soda machine, therefore, should include the property of some indeterminism: A model that readily admits a soda machine's ability to shift from one state to another in a way that could not be determined in advance with certainty.

Although we are generally adept at dealing with such indeterminism in physical objects—as well as when dealing with people—when we encounter such indeterminism in software, we tend to consider that behavior a bug. Examining such "bugs" may reveal that they crept into our code because some aspect of our program was not sufficiently specified.

Naturally, as developers we desire to create programs that are well-specified and, therefore, behave as expected—programs that act exactly in accord with detailed and exhaustive specifications. Indeed, one way to provide more or less exact specifications for code is by writing test cases for it.

Concurrent programs, however, are a bit more like soda machines than deterministic sequential code. That's because concurrent programs gain many of their benefits due to some aspects of a concurrent system intentionally being left unspecified.

The reason for that is easy to understand intuitively when considering a processor with four cores: Suppose that code running on the first core sends messages to code running on the three other cores, and then awaits replies back from each. Upon receiving a reply, the first core performs further processing on the response message.

In practice, the order in which cores 2, 3, and 4 send back their replies is determined by the order in which the three cores finish their computations. If that reply order is left unspecified, then core 1 can start processing a reply as soon as it receives one: it does not have to wait for the slowest core to finish its work.

In this example, leaving the reply order from cores 2, 3, and 4 unspecified helps to best utilize the available computing resources. At the same time, your program can no longer rely on any specific message ordering. Instead, your application must function deterministically even though its component computations, or how those components interact, may not be fully specified.

One application of building deterministic systems out of indeterministic component computations are data centers constructed of commodity, off-the-shelf (COTS) components. Many well-known Web services companies have

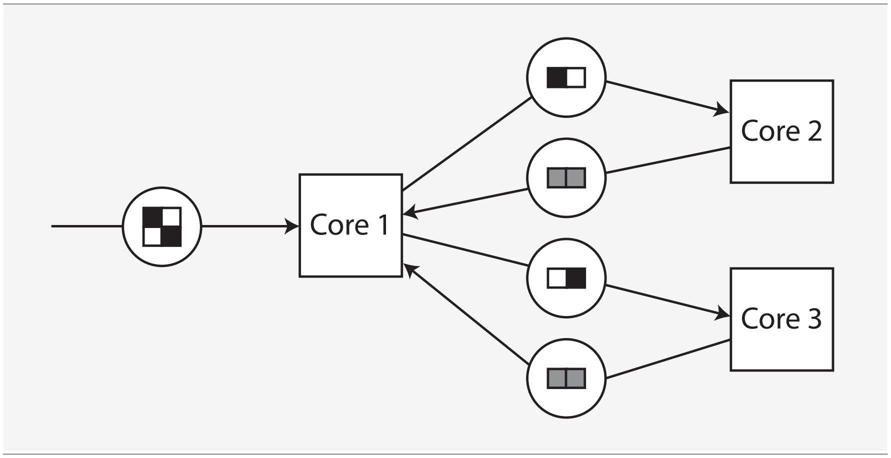

Figure 1.2 · Messages between cores.

proven the economic advantages of using COTS hardware as basic building blocks for highly reliable data centers. Such an environment becomes practical when infrastructure software alleviates the need for developers to concern themselves with the intricacies of how such a data center partitions work between the various hardware components. Instead, application developers can focus on higher-level concerns, such as specifying the algorithms to use when servicing an incoming request.

A popular example of an infrastructure that makes programming on COTS clusters easier is MapReduce.[10] With MapReduce, a user provides some data, as well as some algorithms to operate on that data, and submits that as a request to the MapReduce infrastructure software. The MapReduce software, in turn, distributes the workload required to compute the specified request across available cluster nodes and returns a result to the user.

An important aspect of MapReduce is that, upon submitting a job, a user can reasonably expect some result back. For instance, should a node executing parts of a MapReduce job fail to return results within a specified time period, the MapReduce software restarts that component job on another node. Because of its guarantee of returning a result, MapReduce not only allows an infrastructure to scale a compute-intensive job to a cluster of nodes, but more significantly, MapReduce lends reliability guarantees to the

[10]Dean and Ghemawat, *MapReduce: Simplified data processing on large clusters* [**?**]

computation. It is that reliability aspect that makes MapReduce suitable for COTS-based compute clusters.

While a developer using MapReduce can expect to receive a result back, exactly when the result will arrive cannot be known prior to submitting the job: The user knows only that a result will be received, but he cannot, in advance, know when that will be. More generally, the system provides a guarantee that at some point a computation is brought to completion, but a developer using the system cannot in advance put a time bound on the length of time a computation would run.

Intuitively, it is easy to understand the reason for that: As the infrastructure software partitions the computation, it must communicate with other system components—it must send messages and await replies from individual cluster nodes, for instance. Such communication can incur various latencies, and those communication latencies impact the time it takes to return a result. You can't tell, in advance of submitting a job, how large those latencies will be.

Although some MapReduce implementations aim to ensure that a job returns some results—albeit perhaps incomplete results—in a specified amount of time, the actors model of concurrent computation is more general: It acknowledges that we may not know in advance just how long a concurrent computation would take. Put another way, you cannot place a time bound in advance on the length a concurrent computation would run. That's in contrast to traditional, sequential algorithms that model computations with well-defined execution times on a given input.

By acknowledging the property of unbounded computational times, actors aim to provide a more realistic model of concurrent computing. While varying communication latencies is easy to grasp in the case of distributed systems or clusters, it is also not possible in a four-core processor to tell in advance how long before cores 2, 3, and 4 will send their replies back to core 1. All we can say is that the replies will eventually arrive.

At the same time, unboundedness does not imply infinite times: While infinity is an intriguing concept, it lends but limited usefulness to realistically modeling computations. The actor model, indeed, requires that a concurrent computation terminate in finite time, but it also acknowledges that it may not be possible to tell, in advance, just how long that time will be.

In the actor model, unboundedness and indeterminism—or, *unbounded indeterminism*—are key attributes of concurrent computing. While also present in primarily sequential systems, these are pervasive attributes of concurrent programs. Acknowledging these attributes of concurrency and providing a model that allows a developer to reason about a concurrent program in the face of those attributes are the prime goals of actors. The actor model accomplishes that by providing a surprisingly simple abstraction that can express program control structures you are familiar with from sequential programs—such as `if`, `while`, `for`, and so on—and make those control structures work predictably in a system that can opportunistically scale in a concurrent environment.