Akka Concurrency

PrePrint™ Edition
Excerpt

artima
Artima Press
Mountain View, California

Buy the Book · Discuss

Chapter 4

Akka Does Concurrency

Akka is a domain-neutral concurrency toolkit designed for the purposes of building scalable, fault-tolerant applications on the JVM. It provides many tools to help you achieve your goals, and in this chapter we'll start to understand how to work with those tools so you can start building your high-quality, highly concurrent applications. At the end of this chapter, you should be in a better position to begin *thinking* in the Akka paradigm.

4.1 The actor

When we get past what Akka *is* and start looking at what it *contains*, it's the actor that pops its head up first. The actor does most of the heavy lifting in our applications due to its flexibility, its location independence, and its fault-tolerant behaviour. But even beyond these features, there's an interesting consequence of the actor design—it helps make concurrency development more *intuitive*.

Your day-to-day world is full of concurrency. You impose it on yourself as well as the people around you, and they impose it on you. The real-world equivalents of critical sections and locks as well as synchronized methods and data are all naturally handled by yourself and the people in your world. People manage this by literally doing only *one thing at a time*. We like to pretend that we can multi-task, but it's simply not true. Anything meaningful that we do requires that we do just that one thing. We can pause that task and resume it later, switch it out for something else to work on and then return to it, but actually doing more than one thing at a time just isn't in our wheelhouse.

Buy the Book · Discuss

So what if we want to do more than one thing at a time? The answer is pretty obvious: we just use more than one person. There's not much in the world that we've benefited from that wasn't created by a gaggle of talented people.

This is why actors make our application development more intuitive and our application designs easier to reason about: they're modeled after our day-to-day lives.

Concurrency through messaging

If you want a coworker to do something for you (such as write a bunch of tests for your code because you're simply too busy playing NetHack[1] to engage in such trivialities), what do you do? You send the poor sod an email, of course.

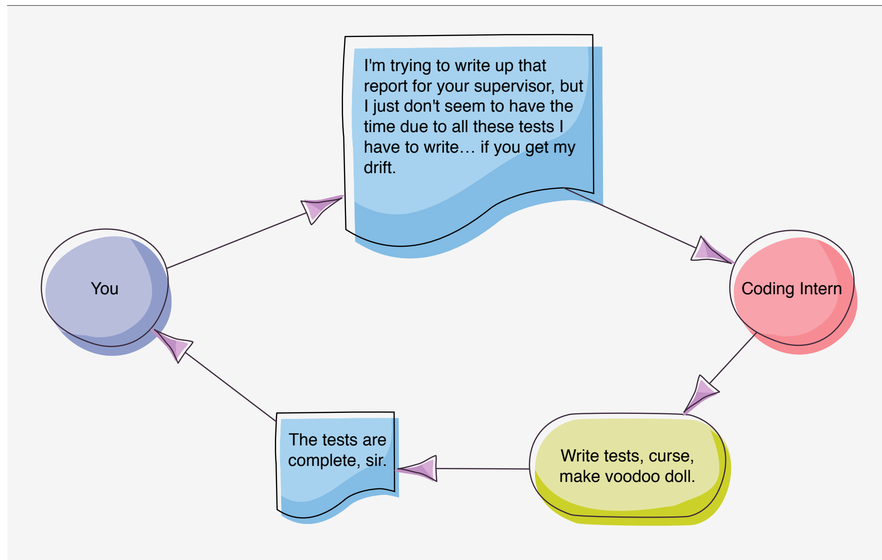

Figure 4.1 · The most productive way to abuse a coding intern.

It's just that simple. Get the right people in place, have a decent mechanism for shunting messages around (bonus points if they're durable), and you're good to go. Hell, if you could spawn enough interns you may be able to play NetHack all day, every day, and even get paid to do it.

[1] http://www.nethack.org/

Actors follow this model. You send an actor a message that tells it to do something, which it does presumably quickly and well, and then it tells you what it did. You can scale this model out to thousands or millions (or billions?) of actors and many orders of magnitude more messages and your applications are still reasonable, not to mention huge and fast.

Concurrency through delegation

Given what happened in Figure 4.1, it would seem pretty obvious that we can delegate work from one actor to another, but you can take this simple idea even farther to achieve your goals. Since interns are so wonderfully cheap, there's no reason we can't have a ton of interns chained to desks in a dark room somewhere churning out whatever it is they are supposed to churn out.

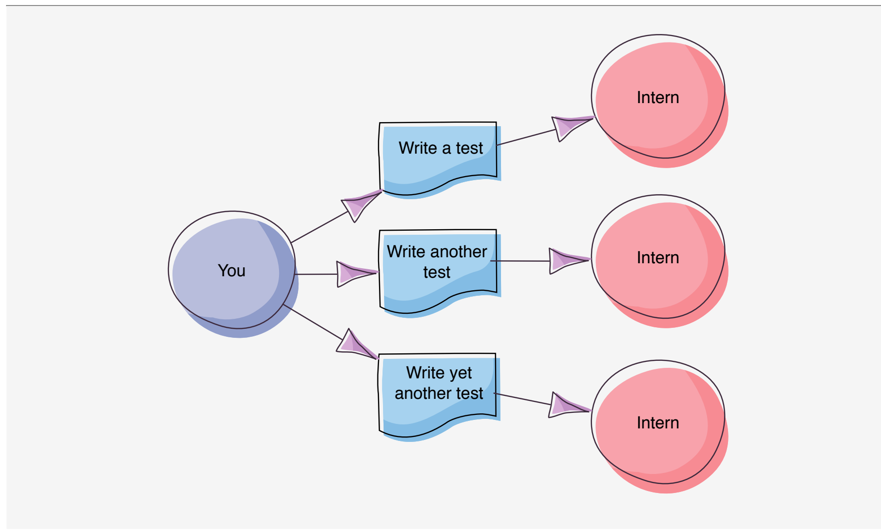

Figure 4.2 · Three interns mean three times the work (and abuse).

But Figure 4.2 is a pretty ineffective use of such a cheap resource. It might even be better to have a single goal in mind and set a bunch of interns to the task. They can each do it the exact same way, or they can all use a different method for achieving the goal. You don't need to care *how* they get it done, just that someone gets it done before the rest. The intern who wins gets a decent report to his or her supervisor, and maybe even a job offer

(although, seriously, you're pretty mean) while the other interns get a rather unfavorable letter sent to their supervisors.

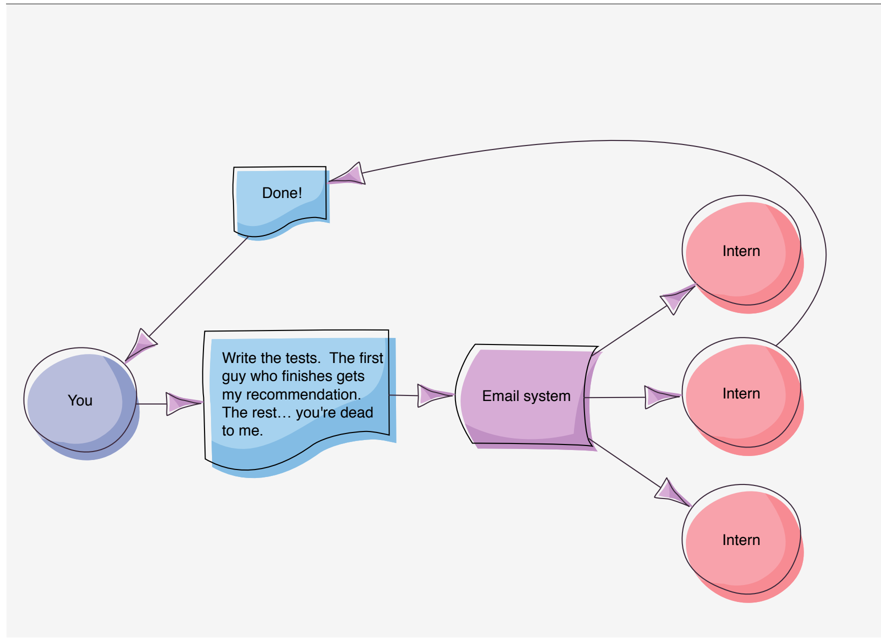

Figure 4.3 · Competition between interns is never a bad thing.

Figure 4.3 gets a particular job done quicker by burning resources with wild abandon. If you've got the people and they're not doing anything else, then why not give them some work to do? Sure, you might throw the results of their efforts right in the trash, but who cares? OK, maybe they'll care, but who cares about that?[2]

So what if your interns realize what you're doing and one of them decides to learn from your example? If he's got the resources available to him, then he's probably going to win if he follows your tactics. There's nothing to stop him from doing something like what's in Figure 4.4.

Interestingly enough, the guy who won in Figure 4.4 would probably be the guy who you hired, but he'd also be the guy that you fired because his friends would eventually get tired of working for free, and he'd be exposed as the lazy, slack-jawed worker he really is. Too bad for you.

[2] Jeez, you're *really* mean.

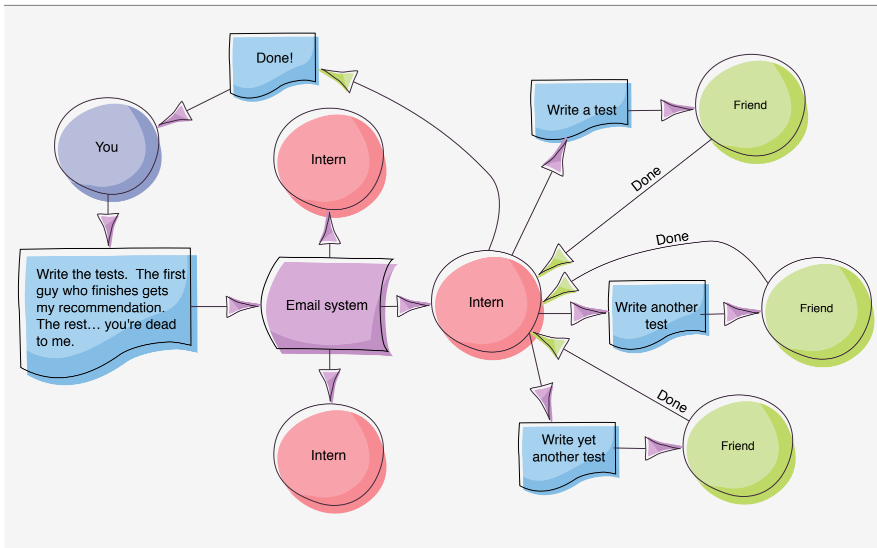

Figure 4.4 · Sneaky (or smart) interns are guaranteed to win every time.

Delegation for safety

While we're on the subject of delegation, we should probably talk about one of the other advantages it provides: *safety*. When was the last time you heard of a sitting U.S. president heading out on a mission with a Navy Seal team to rescue one of his constituents from a group of terrorists? OK, maybe it's because the guy's seriously out of shape, or couldn't hit the broadside of a barn with a bullet the size of a fist from ten paces out, but let's assume he's *awesome*. He still wouldn't go on that mission. Why not? He's just too damn important. There are times when actors are too important to go on dangerous missions, and when that's the case, we delegate the mission to someone else.

You're happy and safe in Figure 4.5 because you can delegate the dangerous work to others. You may be mean, but you're certainly no fool! All of that cool information that you hold—the nuclear launch codes, the itinerary for that policy summit, your spouse's birthday, and all of that other important stuff—is safely locked away in your brain. Unfortunately, Joe didn't make it, but truth be told, his brain was full of quotes from episodes of Family Guy. Cool as that is, it's just not vital stuff.

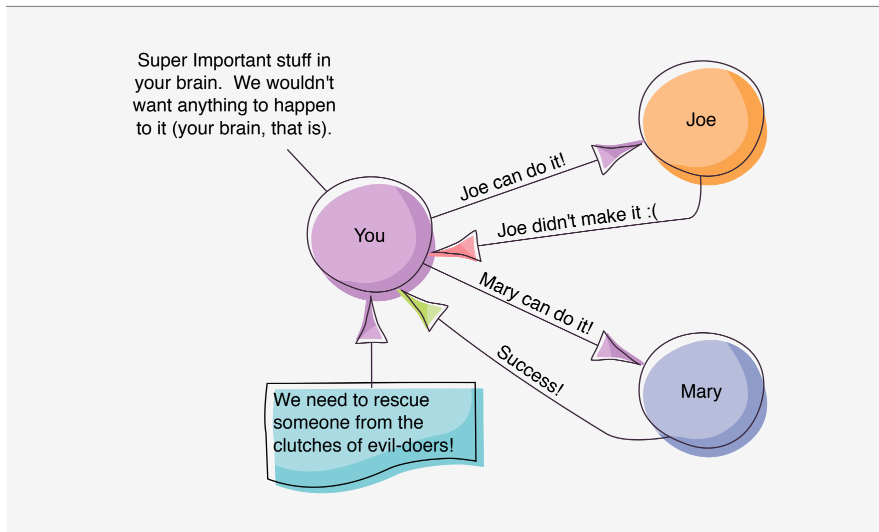

Figure 4.5 · Putting certain actors in harm's way keeps other actors safe.

While we're on the subject of death

We weren't explicit with Figure 4.5, but let's be perfectly frank about it—Joe died. It's unfortunate, but it happens. An actor's life isn't always an easy or safe one but the point is that the actor does have a life and along with it, a *life cycle*. We're going to see much more about the actor life cycle later, and find ways in which we can hook into its life cycle, as well as the life cycles of others. What's interesting at the moment is that there is a life cycle and that it (sort of) matches what we're used to in real life. The people you work with, the interns you are continuously beating on (metaphorically speaking, of course) had to be born at some point, and there will come a day, sooner or later, when they're going to give up their ghost.

But the death of an actor is nothing to get upset about. Actor death can be a very good thing. In an Akka application, there's always someone looking out for actors; someone's always got their back. It's not really the fact that they die that is so great, it's the fact that someone is there to watch it and do something about it. There is, at most, one guy around to clean him up, resurrect him, ignore him and let someone else figure it out, or just ignore him altogether. We can literally just pretend that nothing bad actually happened.

When death occurs, there's only one guy who manages to do something with the deceased, but there are many guys who can react to that death and take action upon notification of it. Presumably that notification is something along the lines of what we saw in Figure 4.5. The notification in that case was the unfortunate message: *Joe didn't make it*. You were able to understand the implications of that message and send Mary to take care of it. If you had sent her first, Joe would probably still be with us, but hey, you can't always make the solid decisions.

There's nothing wrong with creating child actors for the sole purpose of putting them in harm's way. In fact, it's a very good thing. So don't be afraid of giving birth to an actor only to have him meet his ultimate demise micro-seconds later. He's more than happy to give his life in the service of his parent's good.

You also shouldn't be afraid to use death to your advantage. Very often, an actor can self-terminate when its work is completed and that death can be a signal to anyone watching that the time has come to move on to the next operation.

Doing one thing at a time

Actors only do one thing at a time; that's the model of concurrency. If you want to have more than one thing happen simultaneously, then you need to create more than one actor to do that work. This makes pretty good sense, right? We've been saying all along that actor programming draws a lot on your day-to-day life experiences. If you want work done faster, put more people on the job.[3]

Figure 4.6 provides a taste of what the actor structure looks like with respect to processing things.

1. Messages come into a mailbox through (unless you want otherwise) a non-blocking *enqueue* operation.

    - This allows the caller to go about his business and doesn't tie up a waiting thread.

2. The enqueue operation wakes up the dispatcher who sees that there's a new message for the actor to process.

[3] Those of you who are thinking of the *Mythical Man Month* have earned a cookie, but forget about it. Actors are not bound by such trivialities.

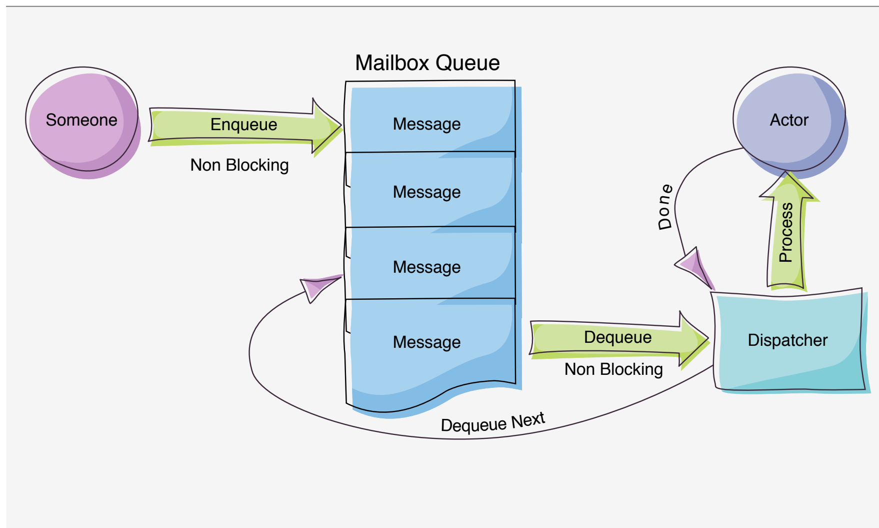

Figure 4.6 · A simplified view of actor message processing.

- In the case of Figure 4.6, we can see that the actor is already processing a message, so there's really nothing for the dispatcher to do in this case, but if it were not processing anything at the moment it would be scheduled for execution on a thread and given the message to process.

3. The dispatcher sends the message to the actor and the actor processes it on whatever thread it was put on to do the work.

   - During the time when the actor is processing the message, it's in its own little world.
   - It can't see other messages being queued and it can't be affected by anything else that's happening elsewhere.
   - The actor is just head-down doing what it needs to do. If it takes a long time, then it's going to tie up that thread for a long time. It's just that simple.

4. Eventually, the actor will finish processing the message.

- The mere fact that it's complete will signal the dispatcher, which can then pull the next message off the queue and give it to the actor to start the cycle all over again.

Details of the Akka implementation are subject to change, so that may not be 100% accurate, of course, but the basic notion is correct. The whole point is that it's the messages that matter and the processing of those messages happens one at a time. The act of queueing and dispatching them is entirely non-blocking by default, which allows threads to be truly dedicated to doing work. Akka does a good job of staying out of your way so that when you have scalability problems or bottlenecks in performance, it's *your fault*. And that's the great news: if it's your fault, then you're in complete control of fixing it.

What's more is that the processing of those messages happens in complete isolation from other work. It's simply not possible for anything to happen that can screw with what the actor is doing right now. You don't need to lock the actor's private data. You don't have to synchronize a set of internal operations that must be atomic. All you have to do is write your code.

The *message* is the message

Have you ever heard the phrase, "The medium is the message?"[4] I'm sure it made great sense to Marshall McLuhan when he said it and I'm sure that it resonates with a bunch of other people, but it always seemed pretty silly to me. You know what really makes a good message? *A message.* Thankfully, actor programming is really all about the message. It's the *message* that travels from place to place, and it's the message that carries the really interesting state. For our purposes, it's also the message that carries the type. A strongly typed message allows us to write code that makes sense to the compiler, and if we can make the compiler happy then we're probably going to be pretty happy ourselves.

But let's step back for a second. What does it mean to say that a message carries the interesting state? Aren't actors the important mechanism here? Isn't it actors that *do things*? Of course it is, but if you remember back to Section 2.3 you might recall that objects that *change* can be a bit unwieldy. If the entire state of a running algorithm is contained inside the messages used

[4]Wikipedia, *s.v.* "The medium is the message," accessed Feb 15, 2013, http://en.wikipedia.org/wiki/The_medium_is_the_message

to execute that algorithm, then we are free to give that work to any actor with code that can process that state. What's more is that the actor, which may be processing the algorithm at any given moment, isn't burdened by weird internal data that it has to keep alive during complex message processing.

Aggregating RSS feeds

To illustrate, let's say you want to collect data from several RSS feeds, aggregate them into one interesting content feed, and then send them off to somewhere else. Moreover, you want to make sure that you can scale the problem to multiple threads when you become successful and have to do this for a thousand users simultaneously. You don't care about making a single user's requests go quickly, you care about increasing your capacity for the number of users on a given machine and their given requests, so we're going to do an individual user's set of requests sequentially.

Figure 4.7 shows us what an algorithm would look like that behaves this way. Note that the messages that travel between different invocations of the actor have two separate sets of data in them: the list of sites to pull data from, and the results of pulling that data. Initially, the list of sites is "full" (*i.e.*, has N things in it) and the list of results is empty. As the algorithm progresses, the list of sites to visit becomes smaller and the list of results becomes proportionally larger. Eventually the actor gets a message where there's nothing to do; the list of sites to visit is empty. When it gets this message, it triggers different behaviour that collects the results into a single aggregated feed and then publishes that forward to someone else (which we don't illustrate).

The fact that we've broken the problem up into individual messages ensures that we give back the executing thread at semi-regular intervals. This keeps the system responsive and lets it handle a greater capacity of users. The actor that's doing the processing could even manage a bunch of different users for us if we want, because it's clueless about what's happening between invocations; all of the state is held inside the messages themselves.

Message immutability

The messages that are used in the RSS aggregation algorithm from Figure 4.7 are immutable. This will keep coming up—it came up before in Section 2.3 and it will come up again. In order for Akka to be able to do the cool things

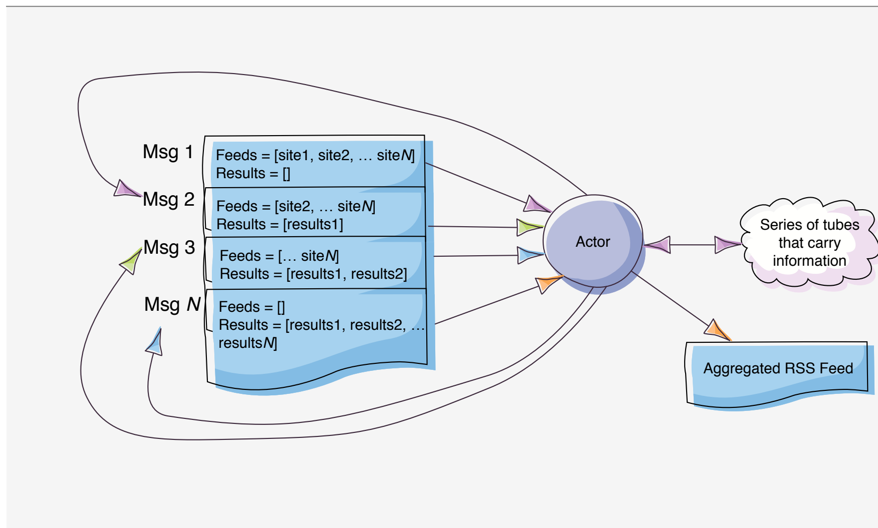

Figure 4.7 · Using actor messages to aggregate RSS feeds in a stateful way.

that it does, and to work quickly and deterministically, it needs you to make sure that your messages are immutable. It's not just a good idea, it's the Akka law. Break it and you break yourself. Don't break yourself.

Strongly typed messages, loosely typed endpoints

Just because the message is so important don't start thinking that the actor isn't; the two are simply different things with important individual responsibilities. Messages are key to the model of the actor application, but it's the actors that facilitate messaging.

One of the things that really helps actors deliver on power is the fact that they're loosely typed; every actor looks like every other actor. They may behave differently or may accept a different set of messages, but until someone sends those messages, you'll never know the difference.

Now, before we extol the virtues of the *untyped* actor, we've got to get something out of the way: Akka has a *typed* actor as well. We're going to ignore it in this book because, while the typed actor has its purpose, it's the flexibility of the untyped actor that drives a lot of power into an actor program. To further explore this idea, let's look at how the actor and the messages interact.

An actor is behaviour

One of the ways to view an actor/message pair is to see them together as loosely equivalent to a function. Figure 4.8 shows one side of how you can picture this; the actor contains the behaviour that is driven forward by the message. The message is the symbol we use to describe the particular behaviour that the actor will execute (such as "Buy from the Grocery Store"), and in order to execute that behaviour, the actor will probably need some data (although not necessarily). This data is held in the body of the message.

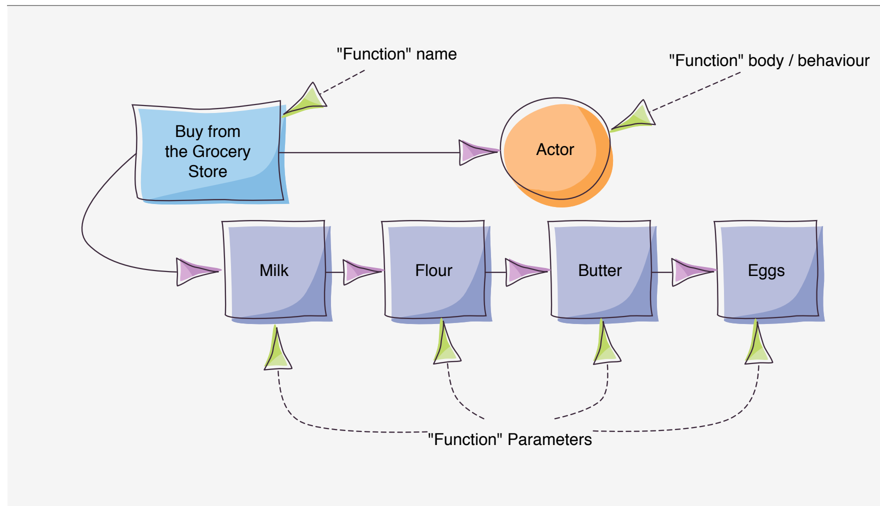

Figure 4.8 · The actor and message can be viewed as a function.

This decoupling of behaviour from the invocation definition is also not unlike a polymorphic function call. An interface can declare the method signature but you can use any number of implementations of that interface to implement the method signature in whatever manner is reasonable for those implementations. However, with an untyped actor you have more flexibility due to the fact that the actor does not need to implement the strongly typed interface. The actor must only be able to process a message and that message is the strongly typed entity.

Now, as we said above, this is only part of the story. We can't just throw the word *function* around as though its meaning were so easy to tailor to our needs. Functions, in most people's definition, evaluate their input data to output data. Actors almost by definition have side effects. To truly view them

as functions instead of `void` procedures, we need to complete the picture. The next step is realizing that actors can send messages as well. Figure 4.9 shows the obviousness of that idea.

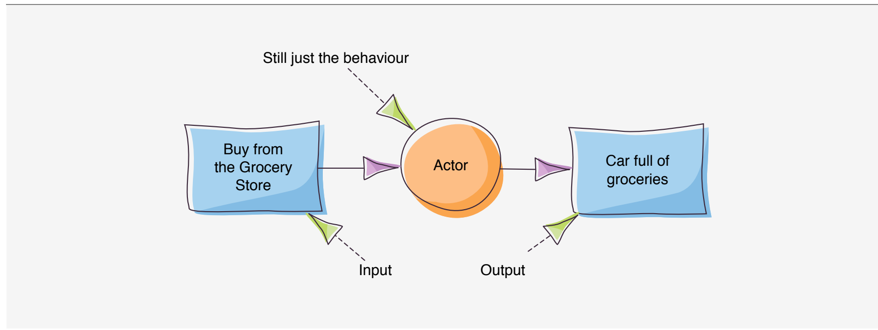

Figure 4.9 · The actor is now an input/output function. . . mostly.

We're really stretching the analogy now. We can think of the actor as returning the new message, if that helps you wrap your head around some of the concepts of actors. However, we must recognize that it only really works as an analogy when the entity that receives the returned message is the same one that sent the request, as in Figure 4.10.

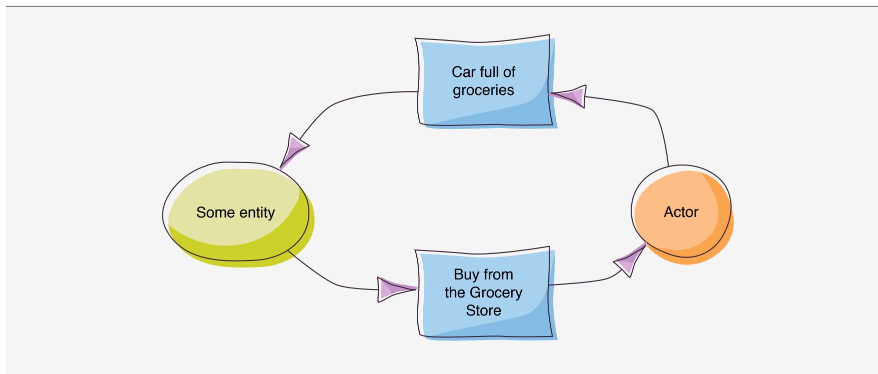

Figure 4.10 · When the response is received, the "function" is complete.

In truth, the actor isn't necessarily *returning* the message; it's really just *sending* the message to some other entity, which is possibly another actor. The entity to which he's sending it may be the initial actor that made the

request or it might be something else entirely. The actor itself doesn't really need to know anything about who sent what or who he's sending things to. All of this plumbing can be set up on-the-fly by anyone who interacts with the actor. For example, let's have someone tell an actor to get some groceries, but to deliver them to someone else, as depicted in Figure 4.11.

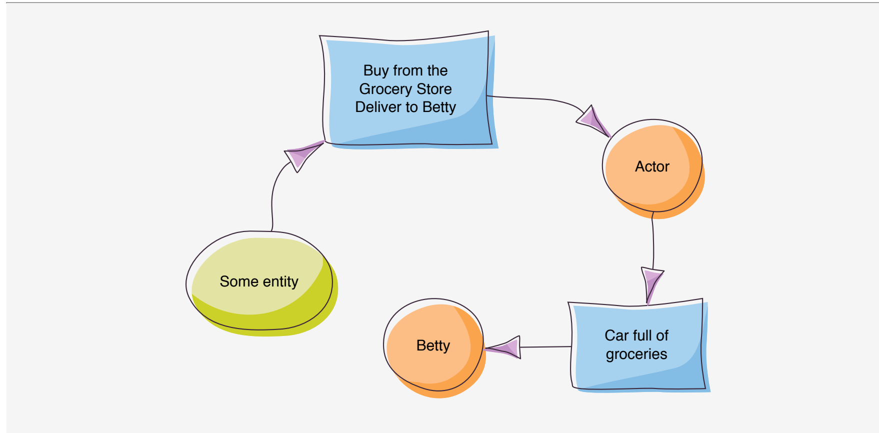

Figure 4.11 · Actors need not send responses to the initiator.

Because we've spent most of our programming lives writing functions, it's important to try and draw a parallel to them. To a certain extent, there is a relationship there, but it breaks down fairly quickly, as you can see. An actor is *behaviour* and we can wire up that behaviour however we see fit. This wiring can be simple, as in the case of Figure 4.9, or it can be far more complicated than anything we've seen thus far. Not only that, it can be entirely determined at runtime. You can dynamically create new actors to handle work that wasn't able to be statically constructed in your editor. This is part of the actor paradigm; we need to get your brain to move beyond the analogy of the function and start thinking in terms of actors. That's part of what this chapter's all about.

Add behaviour by adding actors

One of the excellent things you can do with actors is to add behaviour to an algorithm by inserting actors into the message flow. For example, let's say you've got a system that distributes a bunch of events to actors and you want to start recording those events to disk. Rather than mixing behaviour

into a single class or inheriting functionality, in the style of OO, we have a different alternative. With untyped actors, you can get away with putting a `tee`[5] in between the source and destination actor, as depicted in Figure 4.12.

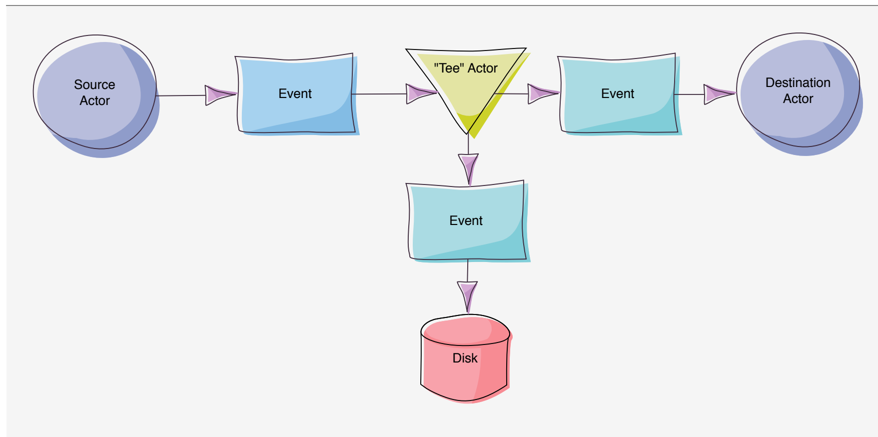

Figure 4.12 · Constructing message pipelines is easy with actors.

This sort of thing happens all the time in actor programming. When a problem presents itself, you tend to solve it by way of creating more actors with discrete behaviour than by adding functionality to existing actors. It's the fact that the actors are untyped and that the real information is contained within the messages that makes this sort of flexibility possible.

Rather than modifying *N* classes or functions by putting in a callout (*e.g.*, to a logging function) or refactoring to a new and very specific class hierarchy, it may be quite natural to slide a new actor into the message flow to let it intercept certain messages, reroute them, duplicate them, transform them, or whatever else is required by your situation.

The separation of typed-ness between the strongly typed message and the loosely typed actor brings power to your designs and your code.

Don't be scared

You're a type junky. I get it. I'm a type junky too. One of the major reasons I write in Scala is because it gives me a strong type system, and that lets me know that my programs are sane when the compiler spits them out. How can

[5] As in the good ol' UNIX `tee` program.

type junkies such as ourselves live in the untyped world of actors and still manage to sleep at night?

Web servers are untyped as well, and when we write a web service we're sending messages to an untyped endpoint. This doesn't make us cringe because there are so few of them. I only have a few URLs that I code against so it's easy to keep it straight in my head, and I can be quite sure that messages aren't going to the wrong spots. But when you have an actor system, you don't have a few endpoints, you have tens to thousands to *millions*. Millions of untyped endpoints can give type junkies the shakes.

Don't sweat it. I have no numbers or theory to convince you that no sweat should be shed over this lack of type safety; all I can say is that I've never sent the wrong message to an actor. This is probably due to the fact that actor programs are so easy to reason about; when things are clear, confusion doesn't exist, and it's confusion that makes us mess stuff up.

But, even if we do send the wrong message to an actor from time to time, it's going to be worth it; so worth it, you won't even think about it. If you're a type junkie, let it go. You'll still use the type system for a ton of stuff and it will be the sweet safety net that it's always been. But when you leave it behind for this one type of object, the actor, that will free you up to do some incredible things.

Reactive programming

Actor programming is *reactive programming*. Another way to say this is that it's *event-driven programming*. Event-driven programming has been with us for a long time, but it's arguably never been epitomized as much as with actor programming. The reason for this is that actors naturally sit there just waiting for something to happen (*i.e.*, waiting for a message).[6] It's not the act of *sending* a message that's important; it's the act of *receiving* one that really matters.

There are two major reasons for this:

1. People like to think in terms of timing. They want to know how long it takes for something to happen after a message is sent.

    - This is a very natural expectation. But in actor programming, you have to put this into context.

[6] OK, they don't "wait" in the traditional sense; that would tie up threads needlessly, and that would be downright dumb.

- What does it mean for the message to be sent in the first place?
- Is it in the actor's mailbox? Is it on a queue ready to be sent to the mailbox? Is it traversing a network, and is there a store-and-forward system that it's been handed off to? Is the queuing of a message a synchronous or asynchronous function?
- Once it's in the mailbox, what does that mean? Is it one of twenty thousand other messages waiting to be processed, or would the mailbox be empty otherwise? Is it in a priority mailbox, and is it so low that it's going to be trumped for the next little while?

Clearly, the act of sending something isn't really all that deterministic. So when you start trying to put bounds or meaning on it with respect to timing, things get very murky very quickly.

2. People also like to attach significance to the sending of the message much like they would a function call.

   - If we say `Math.exp(-5.0)`, then the act of invoking that function has meaning. The code that underlies the `exp` function is executed on the current thread. Dead simple.
   - But, due to all of the reasons discussed above, we can't say the same about queueing a message in an actor's mailbox.

   The act of sending a message is important, since without it nothing would happen, but it's the reception of that message that carries true meaning in actor programming. When the actor pulls that message out of its mailbox and begins processing it, then it has truly *received* that message. It's at this time when meaning is applied in the sense of execution.

These reasons illustrate why *reception* is the important part of message passing in an actor system, but it doesn't make the reactive programming argument completely solid.

Well, you won't get a completely solid argument for it, since nothing is black and white in our complex world of software development, at least nothing at this level of complexity. What's important right now is that you start thinking along those lines, especially if you're not used to it. It's perfectly reasonable to code your actors to react to events that occur in the system,

which is something that isn't necessarily common in standard OO code (for example). It can be as simple as the difference between these two statements:

- Turn the car left.
- The steering wheel on the car has turned to the left.

In the first, someone issued a *command* or a directive that says to do something. In the second, someone posted an *event* that indicates a change to the state of the world. This change to the state of the world would result in the car turning to the left (we hope), which may cause another change to the state of the world, and so forth.

The difference between the two is subtle, but important. Actor programming isn't just about a set of tools, but about *thinking* differently about how you design and write your software. While you aren't going to spend all of your time writing reactionary code, there is some great potential for improving your designs by thinking in a more reactionary style in many cases.

4.2 The future

In the early days of Akka, the actor was the true headliner of the production, and the *future* was mostly there to support the actor. As time progressed, the Akka team built out the future concept more and more, and now in 2.x the future has come into its own. It has grown up into a full-fledged paradigm of concurrent programming that helps you solve tons of interesting problems with speed and grace.

Unlike the actor, the future should be much more familiar to most, so we'll be blasting past it a bit quicker than we did the actor. But fear not, these are the early stages only; we'll be covering much more of the future in later chapters.

Contrasting with the actor

The actor is not a silver bullet. There are many times when the problem with which you're faced isn't solved well with actors. One of the easiest examples I find is the idea of multiplying a bunch of matrices together. It looks like what we've got in Figure 4.13.

We would like to parallelize this computation in order to saturate all of our cores and/or all of our machines. To break the problem up, we can group

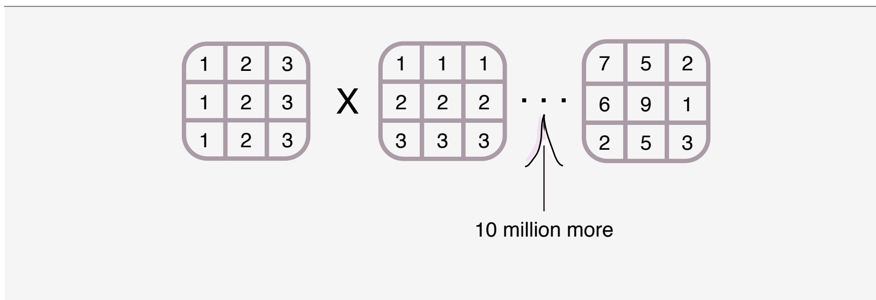

Figure 4.13 · 10,000,003 matrices, from which we want the product.

the multiplications, evaluate them in parallel, then multiply the results together to get one final matrix. Figure 4.14 shows us a specific case of two groups of matrices being processed, but we can generalize the idea to as many groups as we need.

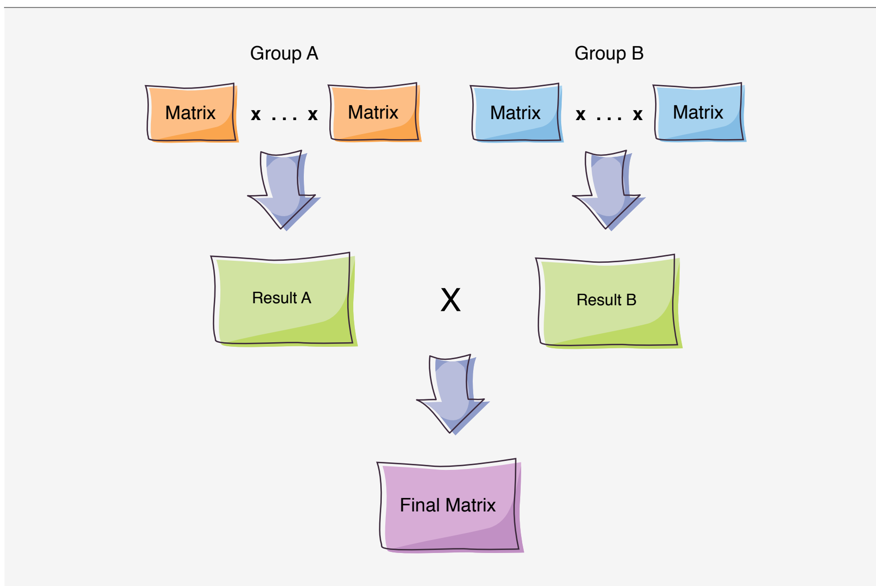

Figure 4.14 · Grouping and parallelizing matrix multiplication.

There's a subtlety to Figure 4.14 that might not be obvious to you if you've never done this before. When multiplying matrices together, *order matters*. It's not the same as multiplying N numbers together, which you can

do in any order you'd like ($5 \times 2 \times 7$ is the same as $7 \times 5 \times 2$). The dimensions of the matrices have to line up properly, and if you start shuffling the order around, you're going to find that the dimensions won't line up anymore, or if they do, you're not going to get the right answer.

The challenge here isn't the grouping and multiplying together of those groups since their ordering is already set for us. What's harder is taking the results and keeping them in the right order. You must multiply $A_{result} \times B_{result} \times C_{result} \ldots$ and so on. If we model this problem with actors, then keeping the results in the right sequence is non-trivial. It's not brutal, but it's a pain. Figure 4.15 shows the core of why it's a problem.

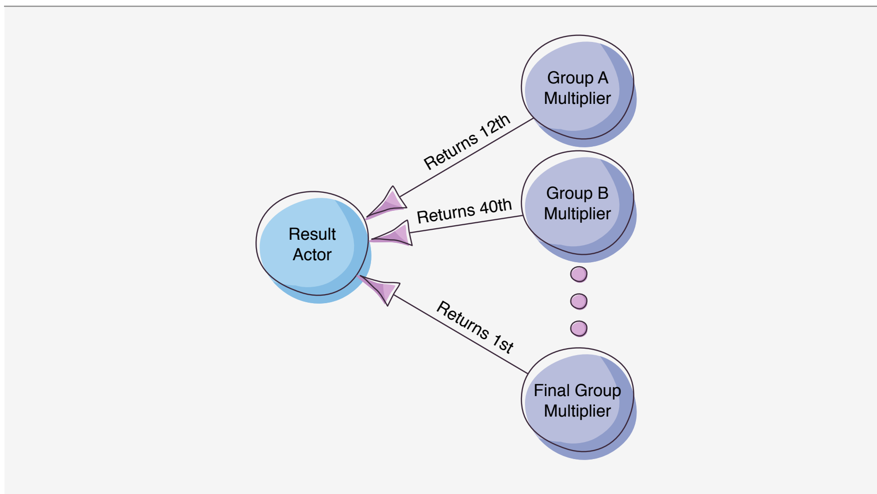

Figure 4.15 · Actors don't respond in any deterministic order.

You can have a single actor receive all of the matrices to be multiplied; it can then group them and spawn new actors to multiply the groups. As each group completes, it can send the result back to the original actor and it can store that result while it waits for the rest. But it can't just store it without thinking about where it needs to go. So you end up having to pass a group to an actor and give it some sort of sequence number as well. When the result comes back, it must return the same sequence number so that the original actor can slot it into the right spot. In addition, as each result comes in, the actor must check to see if the latest result is actually the last result and, if so, it can then multiply the results together and then pass the final result off to someone else.

Whew! That's a lot of work. It's certainly doable, but it's way more of a bother than you'd like. Fortunately, the Akka future implementation makes this problem much easier for us.

Futures are great at being context-free

One thing that futures are great at is accelerating "raw computation," which is why we've started by looking at matrix multiplication. The information required to multiply N matrices together is simply the matrices themselves and their ordering. We don't need anything from a disk, or the network, or a user, or anything of that sort. All we have to do is just plow through N matrices, multiplying them together. If you want to parallelize a very deterministic algorithm, futures are the way to go.

So, how would futures help us solve the matrix multiplication problem better than actors? They solve the two biggest problems we have: maintaining the sequence and knowing when everything's done (see Figure 4.16).

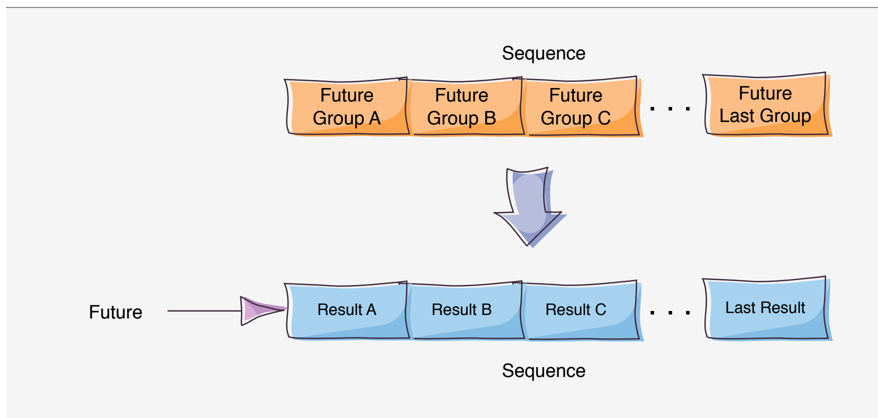

Figure 4.16 · Converting a list of futures into a future of results.

All we need to do is transform our list of matrices into a list of a groups of matrices, and then transform that into a list of futures that compute the multiplications.

```
val list = ... list of matrices ...
val grouped = list.grouped(5000) // 5000, just for fun
val futures = grouped.map { m => Future { ... multiply them ... } }
```

We've obviously left out some details, which aren't really important for us right now. The bottom line is that we've converted our list of matrices to a list of futures, and the only Akka-like thing in that code snippet is the construction of the future with a closure that multiplies the group.

Now we need to collect things, which was the same problem we had to solve with the actor-based approach. We don't have the sequencing problem since the list of futures is in the same order as the groups, but how do we know when all of the futures complete? We're not going to go into any detail about what you'll see because we're not ready for it, but the simplicity of it should get you thinking in the right mode.

```
val results = Future.sequence(futures)
// results is now a Future whose value is the list of resulting group
// multiplications.
val finalResult = ... multiply the last list of matrices together ...
```

Again, we've left some details out, but that's the bulk of it. Not bad for half a dozen lines of code, eh?

Futures compose, actors don't

Actors are great at many things, as we've seen, and what we've seen is merely a glimpse into their potential. But one of the things that actors don't do well is *compose*.

This is rather significant, and if you're a devotee of functional programming, or you've worked with OO patterns such as the Decorator[7] or Chain of Responsibility,[8] then you understand that significance.[9] Functional composition, in particular, gives us a level of expressiveness that brings a large amount of power and flexibility to our daily coding. What if we could bring that level of expressiveness to our daily coding while at the same time mixing in concurrency? If the picture of a Tyrannosaurus Rex slam-dunking a basketball during the final moments of an inter-galactic game of hoops against

[7] Wikipedia, *s.v.* "Decorator pattern," accessed Feb 15, 2013, http://en.wikipedia.org/wiki/Decorator_pattern

[8] Wikipedia, *s.v.* "Chain-of-responsibility pattern," accessed Feb 15, 2013, http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern

[9] I do apologize for lumping the OO composition in the same league as functional composition. While I'm quite aware that they don't belong together, for the sake of establishing familiarity I hope you'll forgive me.

the backdrop of 1.5 billion simultaneous supernovas just popped into your head, then you're getting the idea.

Akka's futures implementation allows us to set up sequential pipelines of code that run asynchronously to other pipelines, but also allows us to create an awesome interplay of parallel and sequential pipelines that run together, are still very easy to reason about, are concise, and still very functional.

Futures work with actors

Futures are designed to work with actors. The converse isn't really true, but that's simply because there's no reason for it to be. A long time ago, Akka had a whole bunch of ways to send a message to an actor. The actor itself had three methods declared on it: `!`, `!!`, and `!!!`. No, that's not a stutter. The different methods signified that the call could be non-blocking, blocking, or future-based, respectively. This was a decent model for learning how to write the API, and the Akka team learned a lot from it; they learned that it wasn't great. Since then things have been changed, and only the non-blocking version is used. The actor itself doesn't know anything about futures.

The actor and the future bind together using an external pattern and the future is the one that understands what an actor is (for all intents and purposes). It's really as simple as Figure 4.17.

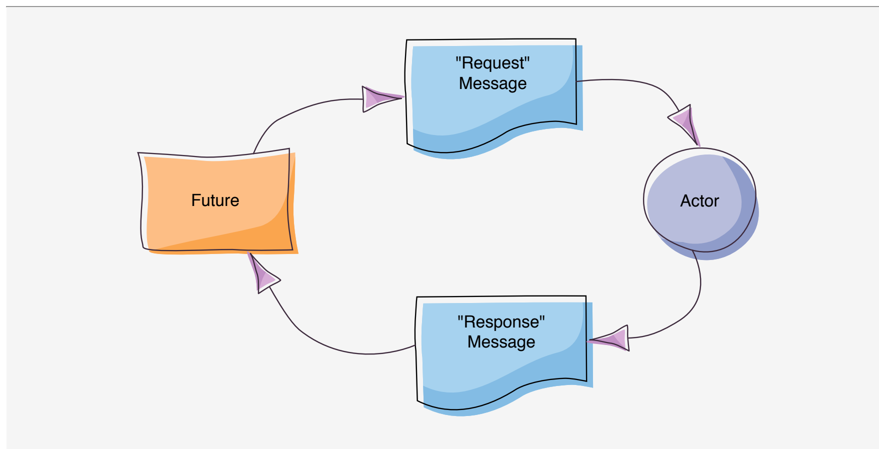

Figure 4.17 · Non-blocking interplay between actors and futures.

One of the many things that this allows is the continuation of the pipelin-

ing concept. A future can be used to coordinate responses, and then *pipe* that response message to another actor instance. And of course all of the usual transformations you'd like to apply to the resulting message can be applied before piping it to that actor. The amount of flexibility provided by the future-to-actor relationship creates a partnership in the Akka toolkit that is greater than the sum of its parts.

Thinking in the future

As with everything else in Akka, using futures isn't just about tossing another tool in the chest, it's about allowing you to think about your code differently. We don't need to worry about "running this in parallel" or "waiting for that to complete" or "putting that other thing on a thread." We don't need to worry about building out the future by hand (*i.e.*, by creating work to go on threads, or any of that nonsense). We simply construct our algorithms and let the "future" happen for itself.

For example, one thing we might do in our day-to-day work may be to create a couple of queues for different threads to use for communication. One side may pull work out from their queue while the other side polls, or otherwise waits, on their own queue for results. With futures, we don't need to think in that manner anymore. We would create a future to do the work, and then compose another future on top of it to process the results. Everything can be done from the call site directly and we don't concern ourselves with queues, messages, protocols, or even threads. The future just unfolds as we've defined, without our having to construct any scaffolding to realize that future.

4.3 The other stuff

There are many other tools in the Akka tool chest but most of them dovetail with either the actor or the future, so you've been introduced to the most important concepts you need in order to understand the rest.

So what's the rest?

The EventBus

The `EventBus` is a nifty little publish/subscribe abstraction that evolved out of an internal Akka implementation the team thought the world might just

make some decent use out of. You're going to find out that they were right.

As we work with messages and events, the idea of distributing certain types of event classes to various types of endpoints just naturally becomes desirable. This happens on a micro level all the way up to a macro level. You might want to have your actor send certain messages to a few friends that have an interest in what it has to say, or you might want to broadcast a small amount of events across your entire super-computing cluster of actors that spans the entire Northern Hemisphere.

The scheduler

Concurrent programming, especially coupled with the concepts of events, has always needed timed or future-based events. Akka provides you with a scheduler that executes functions at timed intervals or single operations at some point in the future. It even provides a special construct for sending messages to specific actors in the future.

Not much is alien to us in the world of the scheduler, so you should be pretty familiar with the concept. We'll see and use it extensively, so if you're not familiar with it now, you will be.

Dataflow concurrency

Dataflow concurrency builds on futures and allows you to look at your application's concurrency from the point of view of the data it uses.

Instead of creating your application as a set of operations that happen in parallel, you can think of it more as algorithms that operate on data. At some point, a piece of data acquires a value, which allows other parts of the application that are waiting on that data to move forward. It operates more like an application that's using locks and condition variables than one that's using futures, except that it's much more deterministic and it doesn't block threads. Figure 4.18 shows the difference.

With futures, our goal is to run functions concurrently with other functions and rendezvous on the results of those functions if we need to. When we employ dataflow, we're getting concurrency more intrusively than that. Pieces of our functions run in parallel with pieces of other functions and they rendezvous on any shared data on which they might be working.

"So, they're sharing mutable data? Isn't that a bad thing?" Well, it's not actually mutable in the traditional sense, so the sharing isn't quite the same

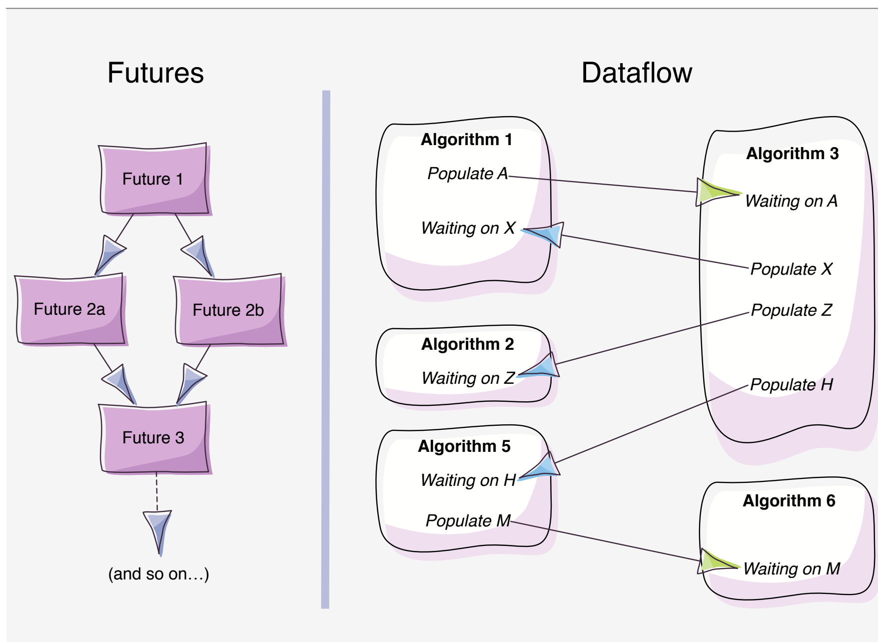

Figure 4.18 · Dataflow provides a granular imperative feel to concurrency.

as we're used to in shared-state concurrency. These aren't *variables* but are *values* and thus immutable. The only difference between dataflow values and standard values is that dataflow values exist in a future context, whereas standard (non-lazy) values are, effectively, set at the time of access.

"But those (hidden) locks and condition variables are a bad thing, right?" They would be if the data were more promiscuous than it is, but it isn't.

We'll be getting into dataflow concurrency later, but any notions you might have about it bringing back the paradigm of shared-state concurrency that we're (somewhat) trying to leave behind shouldn't bother you. The nice thing about dataflow concurrency is that, while things could go horribly wrong (*e.g.*, you could get a deadlock), they're *guaranteed to go wrong all the time*. So you're not stuck looking for Heisenbugs in production because the first time you run the code, it's going to go bad on you.

Message routing

Passing messages to untyped endpoints provides you with a ton of flexibility, and one of those points of flexibility is embodied in the routing feature of Akka. You can send messages anywhere you'd like, of course, but what good is that? Well, if you think back to Figures 4.2, 4.3, and 4.4 you might recall that sending the same message to multiple endpoints can get us greater levels of concurrency, and Figure 4.5 tells us that we can use routing to get us some safety.

Akka provides routing right down to the configuration level of your application. We can use routing to make our applications faster, more scalable, more fault tolerant, and a lot more flexible. The fact that actors can only do one thing at a time will never be a problem for us.

Agents

Agents are inspired by a like-named feature in Clojure[10] and might look a bit like the atomic classes that are part of the `java.util.concurrent.atomic` package, but they're much more than that. Agents are effectively actors and thus provide the same single-threaded guarantees that actors provide, but without the need to send messages to them in order to obtain their values.

You can use agents to provide deterministic locations in memory that are guaranteed to be safe places to store and read data that can change across entities. Agents can be waited on, while other entities play with them, and can also participate in transactions. This makes them much more interesting than the atomic family of classes from `java.util.concurrent.atomic`.

And others...

You've become acquainted with the core philosophies and classes that Akka provides. As you'll see as you continue to read, Akka provides more tools that we can use, including non-blocking IO, interaction with Akka deployments on remote hosts, distributed transactions, finite-state-machines, fault-tolerance, performance tuning, and others.

[10] http://clojure.org/

4.4 You grabbed the right toolkit

In summary, welcome aboard! You've just received a whirlwind tour of the high points of Akka and should have some clue as to why it will be the awesome toolkit that you've heard about. When it comes to building highly concurrent and fault-tolerant applications on the JVM, Akka is a solid choice.

As we progress, you'll learn how to apply the tools we've already discussed to your application design and development. You'll also start seeing *a lot* more code than we've seen thus far that will help establish a set of patterns for coding in Akka. Later on, we'll establish a set of anti-patterns, because there certainly are a fair number of those. Like any decent power tool, if you point it straight at your eye and then run forward, bad things will happen. There are great ways to use Akka and there are also the power-tool-to-the-eye ways, and we're going to favour the former.

You've learned a ton so far, and you should feel pretty awesome about that, but before you run out into the street naked declaring your superiority over the mere machine that you sit in front of, let's cover some more of the nuts and bolts.

OK, take a deep breath and a stretch and let's dive in!