The As-prepared version of “Their Exits and Their Entrances: Actors in the .NET Framework”

[title]

Hi, my name is Carter Wickstrom, and hopefully you’re here to learn about Actors in the .NET Framework.

[Shakespeare]

First, a little bit about myself: I’m a jack-of-all-trades software engineer with approximately 20 years of experience in a variety of platforms and environments, the last 2/3s or so of which have been largely C# and .NET.

I currently work for a small consulting startup, iTrellis, and I was first exposed to Actors while working on a project using Scala (a Java-based object/functional language) and Akka (an Actor framework). The concepts behind Actors required some getting used to, but the possibilities that they offer are, I thought, quite intriguing. Since then, I’ve been on one other project (.NET-based) that had some business requirements that seemed to me to be perfect for a small Actor subsystem. The client expressed some polite interest, but it was hard to take it any further than that. I suspect that much of this reticence came from a lack of familiarity with Actors, so my hope is that this talk goes some way towards introducing others to this model and getting them to consider ways in which Actors might be added to their development tool set.

[actors_are]

When I say ‘actor,’ what comes to mind?

[hacker_actors]

How might we apply this concept of actors to software? Any thoughts?

The Actor is a deceptively simple model: it takes in a message, performs an action in response to that message, and sends off a message that its action is completed. Inbox, logic, outbox. Cue, action, cue. It is done asynchronously, fire-and-forget: there’s my cue, here is my action, there’s your cue. What happens next is irrelevant to the individual actor, until and unless it receives another cue.

The Actor was originally proposed as a means of concurrent computation; in other words, having multiple small programs perform mathematical operations during overlapping time frames. It is *not* the same as parallel computation, that is, having programs execute simultaneously, something that is impossible to achieve on a single computer with a single-core processor. Concurrent operations can be achieved on a single-core processor by essentially timesharing the processor, pausing and unpausing the separate programs as they execute.

Actors were first proposed in the early 1970s, around the same time as the Object Oriented Model. Both were efforts to 1.) Take advantage of the sudden increase in computing power becoming available at the time and 2.) Provide metaphors for programming that allow for complex systems that aren’t possible

using purely procedural languages and models. In particular, research into Artificial Intelligence drove interest in Actors.

Grossly simplified, procedural software is exactly what it sounds like, explicitly spelling out steps and the order in which they must be executed: do this, then that, then the other, then check a condition; if true, then do something, else do nothing and so on. What happens when you need to backtrack or start over? Or when the process you're trying to model is complex?

[goto]

Now, this is simplistic and unfair – bad and/or confusing code can be written in any language using any programming approach. Still, this perception drove the creation of other models, such as Object-Oriented, Functional, Parallel, and Actors. None of these are mutually-exclusive; you can (and should) mix and match approaches as appropriate to the problem at hand.

Object-oriented programming seeks to manage complexity by encapsulating data and behavior into classes that can interact independently of one another: A Payroll knows about Employees and Accounts, and if you need to alter your payroll logic, you do it in the Payroll class without affecting the logic of Employees or Accounts.

Functional programming is an established computational model, dating back to the 1930s and lambda calculus proofs; as a programming model, it dates to the 1950s and 1960s and it sets out to minimize complexity by encapsulating data and behavior into functions. The phrase you'll hear is 'functions as a first-class object;' in short, this means being able to treat functions as values, and pass them to other functions as parameter values. Complexity is managed through recursion, and through handling data to 'minimize side effects.' This means that state is rarely if ever exposed between functions, and that the same values passed in to the same function again and again over time will produce the same result every time. A functional payroll could conceivably have a CalculatePayroll function that accepted a CalculateEmployeeHours function as a parameter.

[serial_to_parallel]

Parallel programming deals with complexity by breaking down logic into discrete tasks. To stick with our payroll example, instead of looping again and again through the same CalculatePayroll function for multiple employees, parallel programming would instead execute that function simultaneously for many different employees.

[_to_actors]

If you can look past my terrible editing and artwork, you'll see that Actors attempt to do much the same thing, except their behavior is non-deterministic. They fundamentally do not care about the existence or execution of other actors, except as sources and targets of messages. Like quantum physics, an Actor-based system consists of elements in various indeterminate states. An Actor-based payroll system would likely have multiple instances of Employee Actors telling multiple instances of our Payroll Actor that they needed to be paid. These transactions would occur in no particular order, and eventually everyone would get paid.

[Bullwinkle]

Actors didn't take. The idea influenced theoreticians and designers of various stripes, but Actors themselves proved to be very difficult to implement. Object-Oriented models proved highly successful for all manner of general computing applications, while functional programming stayed largely academic. Parallel models of computation, while themselves also difficult to implement, proved to be far easier to achieve through both hardware and software than Actors with their indeterminate states.

So what changed to move Actors out of academia and into commercial software?

[internet_and_microservices]

Basically, the internet happened. Distributed systems happened. Always-on computing happened. Systems needing to be highly scalable, reliable, and distributed happened. Advances in hardware and software, and the advances in theory that drove them, made it easier to conceptualize of viable Actor implementations. Suddenly, Actors weren't quite so difficult to realize, and, as internet use exploded beyond web sites and email, they seemed to be a natural fit for certain kinds of problems around scalability and reliability. Actors also address one of the harder problems associated with internet programming: state. From the outside, Actors only exchange messages and their data are immutable; state changes happen within the tightly-controlled confines of the Actors themselves, and are not shared between Actors except as status messages. Where RESTful APIs and other such elements of the middle tier are stateless, Actors can maintain their state between messages, making certain types of business logic easier to implement. This ability of Actors to preserve state between messages goes a long way towards boosting the responsiveness of web applications: no extra caching, queues, locking mechanisms, or expensive database calls (with their I/O bottlenecks) are required.

Another trend making Actors attractive was the struggle with Web Services becoming more and more monolithic: instead of services, web applications began to resemble servers or even full-fledged platforms. This trend towards complexity led to discussions about the viability of microservices. Briefly put, microservices are an effort to break up those monoliths: the service performs one task (or relatively small set of related tasks) and that's it. (Noticing a trend here? All of the models we've discussed are all attempts to make applications smaller, more focused, and easier to understand.) There is no standardized definition of the term, but it first began to appear in conference presentations approximately 10 years ago. Actors, by definition, are a great model with which to create microservices: one role, performed independently of other roles. Functional programmers in particular embraced both concepts, and Actor implementations began to appear in F#, Scala, Rust, Erlang, and Haskell, moving both FP and Actors solidly out of academia and into mainstream software development.

[its_complicated]

You'll notice that in that list there is no C# or VB.NET, only F#. The .NET ecosystem's relationship with Actors and similar 'academic' pursuits is a complicated one. It is probably not unfair to say that .NET is first and foremost an environment concerned with Line-of-Business applications (LOB). For much of the time that .NET has been around, that has been the primary (if not sole) focus of Microsoft: getting in-house software developers to create LOB applications with .NET, SQL Server, and Office.

[BillGates_fake_quote]

Maybe you've seen this quote attributed to Bill Gates? It's a fake, but it certainly reflects the perception of many of Microsoft's detractors (and even defenders to some degree) during the mid- to late-1990s.

The internet was just something that Microsoft didn't seem to understand (to be fair, they understood it differently), even as it exploded around them. Microsoft's initial response was to offer tools like ASP.NET Web Forms, InfoPath, and SharePoint – tools oriented towards business users. As such, much of the work on concurrent systems took place outside of Microsoft's commercial purview. Folks at Microsoft Research might have been doing interesting things with concurrency and functional programming, but getting them integrated into Microsoft's commercial offerings was difficult (For instance, F# was never supposed to be a real Visual Studio offering.). Still, attitudes change: Microsoft has debuted Microsoft Azure Reliable Actors, and the larger .NET community as responded with projects such as Akka.NET. These are the two Actor implementations that we'll be comparing.

[MSAzureReliableActors]

Reliable Actors are based heavily upon Microsoft Research's open source Orleans project. The goal behind Orleans was to produce an Actor model and runtime for high-scale cloud computing that didn't require a lot of knowledge of concurrency, resource management, or fault tolerance. How high-scale is it? How about all of the cloud services behind Halo 4? Spurred by such real-world successes, Microsoft decided to offer the Azure-specific Service Fabric as the runtime for cloud-based microservice applications and put forth Reliable Actors as their Actor model.

Microsoft makes the case that their Reliable Actors are suitable when:

1. Your problem set involves many independent units of state and logic

2. You want to deal with single-threaded objects that still scale and maintain consistency

3. Azure Service Fabric controls concurrency and granularity

4. Azure Service Fabric controls the messaging between Actors

Reliable Actors are single-threaded components like regular .NET objects. This single-threaded behavior is part of how concurrency is managed – Reliable Actors are only capable of processing one request at a time, and requests are processed in the order in which they are received. There is no need to explicitly create or destroy Reliable Actors in the Service Fabric; their lifetime is controlled and guaranteed by the Service Fabric application that hosts them, so they effectively appear to be always on. Likewise, when demand rises, the Service Fabric application will automatically create new instances of our Reliable Actors and map message routes to them accordingly. When demand dies down, Service Fabric will eliminate those additional instances automatically; when the last instance of a Reliable Actor is de-referenced and eventually garbage-collected, its state data is persisted to the Service Fabric such that it can be quickly reinstantiated by the Service Fabric application the next time it is required, without the overhead of an associated database call. No attempt is made to recover from a crash – the actor instance is simply destroyed and replaced without any loss of state. This automatic scaling and persistence is the source of the 'Reliable' in the Reliable Actor name.

[Interface_and_ActorProxy]

The only interactions allowed to Reliable Actors by the Service Fabric are asynchronous messages in a request-response pattern, one of the simplest message-based communication patterns. These messages must be defined in an interface as methods; they must not be overloaded, they must not have 'out,' 'ref,' or optional parameters, and they must return Task objects that must be data-contract serializable.

In .NET, data contracts for custom types are created with attributes that indicate which members are to be serialized, and how. Actor clients, that is, the consumers of the Actors' Tasks, must go through an Actor Proxy in order to communicate with an Actor, essentially creating a wrapper using the Actor's interface.

[Akka_and_ReactiveManifesto]

Akka.NET is a line-for-line C# port of the Akka Actor library, written in Scala for the Java Runtime Environment. The only real differences are the result of differences between the two languages and their runtimes. How closely related are they? Close enough that the creators of Akka have thanked the Akka.NET development community for helping uncover bugs in Akka.

Akka describes itself as "[A t]oolkit and runtime for building highly concurrent, distributed, resilient, message-driven applications on the JVM." The design and implementation of Akka was driven by the Reactive Manfesto. Akka.NET sought to directly transplant this design and philosophy from the JVM to the .NET runtime.

[Akka.NET_ActAllTheThings]

Even though Akka.NET is a very high-level abstraction, it requires far more developer input than Azure Reliable Actors. Where Azure Reliable Actors have the Azure Service Fabric to handle messaging, instantiation, Actor lifetimes, and scaling, Akka.NET requires the individual developer to define and or manage how all of these aspects will be handled. This might sound scary (and it can get quite complicated), but we'll see that the differences aren't as great as it might sound.

When you start working with Akka.NET, you're going to learn a whole lot about the details of Actors that are hidden by the ReliableActor framework. First off, forget that whole Service Fabric thing; you need to instantiate an ActorSystem. The ActorSystem is the most expensive and memory-intensive part of an Akka.NET service; you don't want to be creating and destroying these unless you absolutely have to. Your ActorSystem is going to handle the routing of messages, and determine when to stop and start Actors.

Next, instead of having a simple choice of Stateful or Stateless Actor, you're going to have a *lot* more choices: UntypedActor, TypedActor, GuardianActor, SystemGuardianActor, and others.

Actors can manage other Actors. Actors can effectively act as ActorSystem instances, and manage child Actors within their own context. The children can neither send nor receive messages outside of their parent.

The Akka.NET version of the Reliable Actor proxy is reference, handled through instances of IActorRef. All IActorRef offers you is an address pointing to the Actor instance.

You're going to have to manage your own messages. Messages *must* be immutable, which is an easy thing for your typical .NET developer to forget. We love our Plain Old Class Objects with their shifting state. Coding this way in Akka.NET will end in tears. Salty, geeky tears.

[Akka.NET_Code]

There are so many choices that it's a lot more difficult to summarize than the Reliable Actor – but here's a quick and dirty look at an Akka.NET actor and all of its moving parts: the Actor, the IActorRef 'proxy,'

and the message. There are broad similarities between the Reliable Actor code and the Akka.NET code, but some of the differences are immediately apparent.

[ReliableActors_vs_Akka]

If you were building an Actor-based system and were considering the trade-offs of these two tools, what are some of the differences you'd want to take into consideration between Azure Reliable Actors and Akka.NET?

First: ServiceFabric vs ActorSystem: everything managed for you vs. roll your own everything

Second, what's your targeted environment? Azure only vs wherever .NET runs.

Is it proven? You've got the commercial success of Halo built upon Orleans vs a young open-source product.

Product maturity – Service Fabric and Reliable Actors are still pre-release vs. Akka.NET which is 1.0.x *and* has a large Akka community base upon which to draw for resources and inspiration.

But I would imagine that the most compelling trade-off for most folks will be speed of implementation vs. having to really *know* what's going on in your system. Service Fabric is child-proofed, while Akka.NET is working without a net. You can probably spin up a full-fledged system with Reliable Actors faster than with Akka.NET, but with Akka.NET you have much more control. As developers, many of us struggle between the need to get something deliverable done quickly, and the desire to want to know all that there is to know about how a system works. For me, the contrast between Azure Reliable Actors and Akka.NET in many ways comes down to this one issue.

[actor_example]

Before we move on to the Actor example code, are there any questions?

When I started preparing this talk, I really struggled with how to position the code sample. I've watched bad code sessions; they're painful. That said, if they're too simplistic, they don't offer you enough of an overview. Do I go with the canonical "Hello, world!" or something more complex? This being my first stab at a presentation, I opted for something on the simple side, but that will still show, I think, the critical differences between the two Actor systems.

First, let's create an Azure Reliable Actors project. It requires Visual Studio 2015, and the Service Fabric SDK. There's also some PowerShell voodoo that's required, but in the end you'll have an actual instance of a Service Fabric cluster running locally in your development environment, which is, to my way of thinking, really pretty cool. This all means, though, that the setup process will take far longer than actually creating a demo project will. Fortunately, the MSDN 'getting started' documentation is pretty good.

To keep things simple for the purposes of this demo, we're going to create a Stateless Actor.

First off, look at all of those projects: the Actor, its Interface, its Service instance, a deployment project, and client. That's a lot of boilerplate, as with many Visual Studio templates, but fortunately we aren't going to touch most of it.

Let's start by fleshing out our interface. We're going to make an Actor that can perform basic arithmetical calculations: Addition, subtraction, multiplication, and division.

Remember, each one of these needs to be a task:

[code 1]

Then, we need to implement the interface in our Actor:

[code 2]

Now we need to code the client:

[code 3]

Notice the boilerplate that's here; that's the address of our actor.

[expand the services project]

Here's the Service Fabric instance that's going to host the Actor.

Set the actor's service as the startup project, and let's build and run this puppy. Look at the diagnostics that are emitted by our Actor and by Service Fabric.

Now, let's see our Actor in action: [navigate out to Client\bin\debug and run the console app]

Add, subtract, multiply, and divide: anyone want to double-check our math?

Okay, I'm going to leave that up and running. Let's open up another instance of Visual Studio and implement the same calculator in Akka.NET.

Start with a new console project. The setup here is a lot simpler – Akka.NET is available as a nuget package.

Remember, we don't have any boilerplate here, but that shouldn't be too daunting.

First, we need a message, and it has to contain all of the data that we're going to need here.

[code 4]

Does this look right? Can anyone tell me what might be wrong? It isn't immutable: I can change the values of X and Y. Strings are inherently immutable, but I still don't want to expose one in a writable property. In fact, let's take the string out of the equation entirely, and put in an enum that describes what we want to do.

Next, the Actor. We're going to use an UnTyped Actor, and implement its OnReceive method. Remember, Actors work by sending and receiving messages, so this is where all of the work is done.

[code 5]

First, we need our ActorSystem to host our Actor.

Next, we need our proxy.

After our proxy, the messages that we want to send: add, subtract, multiply, divide. Let's make sure we're using the same pairs of numbers, right?

Now we need to send the messages, by using the Actor's Tell method.

Assuming I didn't screw anything up, this should run and we should see some numbers appear in the console.

Are they the same numbers as our other console?

And there you have it: the same Actor implemented in two different frameworks.

Are there any questions?

[thanks]

I'd like to thank you all for showing up to the first session of the morning, and my first presentation of this sort ever; hopefully you found it useful, and, ideally, not terrible. I will try to have my materials, including code samples, references, slides, and image credits all posted at the GitHub URL you see there no later than tomorrow evening.

If the notion of working for a company that is interested in a wide variety of tools and technologies intrigues you, please talk to me, as we (like everyone else, it seems) are hiring.

[me]

Thanks again.