Hootsuite

In Pursuit of Reactive Systems

case study

A DISCUSSION WITH EDWARD STEEL, YANIK BERUBE, JONAS BONÉR, KEN BRITTON, AND TERRY COATTA

Based in Vancouver, Canada, Hootsuite is the most widely used SaaS (software as a service) platform for managing social media. Since its humble beginnings in 2008, Hootsuite has grown into a billion-dollar company with more than 15 million users around the globe.

As Hootsuite evolved over the years, so did the technology stack. A key change was moving from LAMP (Linux, Apache, MySQL, PHP) to microservices. A shift to microservices didn't come without its challenges, however. In this roundtable chat, we discuss how Scala and Lightbend (which offers a reactive application development platform) were an essential part of a successful transition. The exchange includes Jonas Bonér, CTO of Lightbend; Terry Coatta, CTO of Marine Learning Systems; Edward Steel, senior Scala developer at Hootsuite; Yanik Berube, lead software developer at Hootsuite; and Ken Britton, senior director of software development at Hootsuite.

TERRY COATTA I'm curious about the original set of problems Hootsuite was looking to address in the switch to microservices. Can you provide some detail?

EDWARD STEEL Mostly, it had to do with our ability to send out notifications to user mobile devices whenever

something relevant happened on Twitter. By the time we started having some concerns about how we were handling that, we were already servicing several hundreds of thousands of users, each with individual subscriptions tailored to their own specific interests. What was needed was something that could stay connected to Twitter's streaming endpoints.

TC I gather that at about the same time you were making this move, you also took steps to move from PHP to Scala. What drove that?

ES Initially, it had a lot to do with learning about all the success some other organizations had experienced with Scala. This was after Twitter had decided to go with Scala, for example, and that obviously lent a lot of legitimacy to it. Also, the first team here to work with Scala came from quite a varied background. We had some people who were lobbying for a more strongly typed functional language—something on the order of Haskell—and then there were some others with Clojure and Java experience. In taking all that into account, I guess Scala just seemed to check most of the boxes.

JONAS BONÉR What would you say was the principal benefit you saw with Scala? Was it the functional nature of the language itself? Or did it have more to do with the libraries available within that ecosystem?

ES The language itself was the biggest part of it. The main advocate here for Scala was working on a Blackberry client at the time, so he had a lot of JVM (Java virtual machine) knowledge and yet also had become frustrated with Java itself. I guess he was just looking for a better way. Another aspect of our thinking had to do with building a

distributed system that could take advantage of Akka as an available library. That was a big part of the decision as well.

In fact, I think we were able to use some actors right from the start. That was with a very early version of Akka, but it still offered a lot of compelling features we found useful.

JB Were you already thinking in terms of microservices back then, even before that took off as a buzzword? Or were you more drawn by reactive principles having to do with things like a share-nothing architecture, strong isolation, and loose coupling?

ES Microservices were always in the back of our minds. We already had some batch processes written in PHP that were starting to run jobs at that point. That sort of worked, but it was far from an ideal way of doing things. So I think we'd already started to develop an appetite for a system on which we could build a few services, with the thought being that perhaps we could then move toward a service-oriented architecture. The idea of microservices wasn't something that came up until a little later, and that was probably influenced by some of the buzz around the industry at the time.

TC You've already mentioned Akka a couple of times, so can you speak to how that fits in here?

ES At that time, at least, the main appeal of the Akka system for the JVM was that it provided people beyond the Erlang community with access to the actor model. The thing about actors is that they're both message based and highly resilient—which is to say that even when they crash, they can typically be brought back in a useful way. This probably explains why Erlang has so often been used to

develop resilient telecommunications systems. Whenever you're talking about distributed systems or some system where you expect to fire a lot of messages around, you can expect the actor model to really shine.

YANIK BERUBE Just in terms of where this fits into our current infrastructure, we should note that all our microservices are powered by Akka. Internally, we have a server-type library that handles requests and responses via Akka, and we also have at least one, if not more, back-end services that use sets of actors to accomplish work at certain intervals of time.

TC One of the things that comes to mind when I think of the Erlang actor system, beyond the independence you've already mentioned, is that it's quite fine-grained. So I wonder, given your focus on notifications to user mobile devices, whether you might actually require an actor per user just to deal with that?

ES In our case, no. But symptoms of that definitely showed up as we were first building our system. When we were starting out, we learned more about how we should be building the system, as well as about how actors really ought to work. At first, we definitely fell into the trap of putting too much logic into single actors—for example, by putting recovery logic into each actor instead of relying on the supervision hierarchy, which would have allowed us to code less defensively. It turns out it's best just to embrace the "let it crash" philosophy, since that actually offers a lot of robustness.

We also learned the model really shines whenever you separate concerns into single-purpose actors. Besides helping to clarify the design, it opens up a lot of

Whenever you're talking about distributed systems or some system where you expect to fire a lot of messages around, you can expect the actor model to really shine.
—Edward Steel

opportunities in terms of scalability and configuration flexibility at the point of deployment.

JB This applies to microservices as well. That is, there are plenty of opinions about what even defines a microservice. What does that term mean to you? And how does that map conceptually to how you view actors?

YB Internally, we're still trying to define what a microservice is and what the scale of that ought to be. Today, most of our services focus on accomplishing just one set of highly related tasks. Our goal is to have each of these services own some part of our domain model. Data services, for example, would each control their own data, and nothing else would have access to that. To get to that data, you would have to go through the data service itself.

Then we would also have functional services, which are the services that essentially glue business logic onto the data part of the logic. But in terms of the size of these things, I'd say we're still trying to figure that out, and we haven't come up with any hard and fast rules so far.

JB I'm guessing each of your microservices owns its own data store. If so—with these things being stateful—how are you then able to ensure resilience across outages?

ES Each of these services absolutely owns its own data. When it comes to replacing parts of the monolithic system, it generally comes down to dealing with a table or a couple of related tables from the LAMP MySQL database. Generally speaking, the space is pretty minimal in terms of the services that need to be accounted for. It's basically just a matter of retrieving and creating data.

YB I'd say we make fairly heterogeneous use of various

technologies for data storage. And, yes, we do come from a LAMP stack, so there's still a heavy reliance on MySQL, but we also make use of MongoDB and other data-storage technologies.

The services typically each encapsulate some area of the data. In theory, at least, they're each supposed to own their data and rely on data storage dedicated only to them. So, we've recently started looking into storing a bit more data within the services themselves for reasons of efficiency and performance.

TC To be clear, then, there's some separate data-access layer from which the services are able to pull in whatever information they need to manipulate?

ES Yes, but you won't see more than one service accessing the same data store.

JB How do you manage this in terms of rolling out updates? Do you have some mechanism for deploying updates, as well as for taking them down and rolling back? Also, are these services isolated? If so, how did you manage to accomplish that? And, if not, what are you doing to minimize downtime?

YB Right now, every service uses a broker/worker infrastructure. No service can access another service without going through a pool of brokers that then will redistribute requests to multiple workers. This gives us the ability to scale by putting more workers behind the brokers. Then, when it comes to deployment, we can do rolling deploys across the target servers for those workers. In this way we're able to deploy the newest version of a service gradually without affecting the user

experience or the experience of any other services that need to make use of that service as it's being redeployed.

ES Another thing we've recently pulled over from the LAMP side that has proved to be useful for frequent rollouts is feature flagging. That's obviously something that was a lot easier back when we just were working with a bunch of web servers, since we had a central place for handling it. But recently we migrated those same capabilities to HashiCorp's Consul to give ourselves a distributed, strongly consistent store, and that now lets us deploy code on the Scala side with things switched off.

JB Looking back to when you were doing this along with everything else required to maintain a monolith, what do you see as the biggest benefits of having made the move to microservices?

ES In terms of what it takes to scale a team, I think it has proved to be much easier to have well-defined boundaries within the system, since that means you can work with people who have only a general idea of how the overall product works but augment that with a strong, in-depth knowledge of the specific services they're personally responsible for.

I also think the microservices approach gives us a little more control operationally. It becomes much easier to scale and replace specific parts of the system as those needs arise.

YB One clear example of this is the data service my team has been working on. It's a very high-volume service in terms of the amount of data we store, and we knew it would be challenging to scale that, given the storage

technology we're using. The ability to isolate all that data behind a data service makes it a lot easier for us to implement the necessary changes. Basically, this just gives us a lot more control over what it takes to change the persistent technology we're using in the background. So I certainly see this as a big win.

Some of the benefits of moving over to the reactive microservice model supported by the Lightbend stack surfaced almost immediately as the Hootsuite engineering team started discovering opportunities for scaling down on the underlying physical and virtual infrastructure they had run previously on their LAMP stack (where there had been a process for each request). Accordingly, it soon became apparent that operations under the reactive microservices model were going to put much less strain on their resources.

In fact, if anything, the engineers at Hootsuite quickly learned that by continuing to employ some of the practices that had made sense with a LAMP stack, they would actually be denying themselves many of the benefits available by relying to a greater degree on the Lightbend stack. For example, they found there was a real advantage to making greater use of the model classes supported by the Lightbend stack, since those classes come equipped with data-layer knowledge that can prove to be quite useful in a dynamic web-oriented system.

Similarly, they learned that by using individual actors to run substantial portions of their system instead of

decomposing those components into groups of actors, they had been unwittingly depriving themselves of some of the features Akka offers for tuning parts of the system separately, parallelizing them, and then distributing work efficiently among a number of different actors capable of sharing the load.

And then there were also a few other things they learned...

TC So far, we've talked only about general issues. Now I would like to hear about some of your more specific engineering challenges.

JB One thing I'd like to know is whether you're mostly doing reactive scaling, predictive scaling, or some combination of the two to optimize for your hardware.

YB For now at least, our loads don't really change a lot. Or perhaps what I should say is that they change throughout the day, but predictably so from one day to the next. And the way we've designed our services to run behind brokers means we're able just to spin off more workers as necessary. In combination with some great tooling from AWS (Amazon Web Services), we're able to adapt quickly to changing workloads.

ES One thing we did decide to do was to build a framework using ZeroMQ to enable process communication between our various PHP systems. But then we saw later that we could have just as easily pulled all that into Akka.

TC And I'm assuming, with Akka, it would have been easier for you to achieve your goal of adhering to the actor paradigm, while also taking advantage of better recovery mechanisms and finer-grained control.

ES Yes, but I think the key is that because of our ability to change the characteristics of actors by how we configure them, we've been able to adapt this core framework to all types of payloads and traffic profiles for the various services. We can say, "This service is using a blocking database," at which point a large thread pool will be supplied. If the service happens to be handling two different kinds of jobs, we can separate them into different execution tracks.

More recently, we've also had a fair amount of success using circuit breakers in situations where we've had a number of progressive timeouts as a consequence of some third party getting involved. But now we can just cut the connection and carry on. Much of this comes for free just because of all the tools Akka provides. We've learned that we can take much better advantage of those tools by keeping our designs as simple as possible.

JB Something I also find very interesting is that Akka and Erlang appear to be the only platforms or libraries that put an emphasis on embracing and managing failure—which is to say they're basically designed for resilience. The best way to get the fullest benefit of that is if they're part of your application from day 1. That way, you can fully embrace failure right at the core of your architecture.

But, with that being said, how did this newfound embrace of failure work out in practice for those of your developers who had come from other environments with very different mindsets? Was this something they were able to accept and start feeling natural about in fairly short order?

YB For some, it actually required a pretty substantial mindset shift. But I think Akka—or at least the actor model—makes it easier to understand the benefits of that, since you have to be fairly explicit about how you handle failures as a supervisor. That is, as an actor that spins off other actors, you must have rules in place as to what ought to happen should one of your child actors end up failing. But, yes, people had to be educated about that. And even then, it still took a bit of getting used to.

Now, as new developers come in, we see them resort to the more traditional patterns of handling exceptions. But once you get some exposure to how much saner it is just to leave that to the supervisor hierarchy, there's generally no turning back. Your code just becomes a lot simpler that way, meaning you can turn your focus instead to figuring out what each actor ought to be responsible for.

TC Talking about actor frameworks in the abstract is one thing, but what does this look like once the rubber actually hits the road? For example, how do you deal with failure cases?

YB You can just let the actor fail, which means it will essentially die, with a notification of that then being sent off to the supervisor. Then the supervisor can decide, based on the severity or the nature of the failure, how to deal with the situation—whether that means spinning off a new actor or simply ignoring the failure. If it seems the problem is something the system actually ought to be able to handle, it will just use a new actor essentially to send the same message again.

But the point is that the actor model allows you to focus all the logic related to the handling of specific failure cases

in one place. Because actor systems are hierarchies, one possibility is that you'll end up deciding the problem isn't really the original actor's responsibility but instead should be handled by that actor's supervisor. That's because the logic behind the creation of these hierarchies determines not only where the processing is to be done, but also where the failures are to be handled—which is not only a natural way to organize code, but also an approach that very clearly separates concerns.

I started to think about how we should handle the communication between services around the way we handle failures. So now that's something we always think about.
—Yanik Berube

JB I think that really hits the nail on the head. It comes down to distinguishing between what we call *errors*—which are things that the user is responsible for dealing with—and *failures*. This creates a model that is easier to reason about, rather than littering your code with try/catch statements wherever failures might happen—since failures can, and will, happen *anywhere* in a distributed system.

YB One of the fantastic lessons that's come out of all this is that it has allowed me to start thinking about how the system actually works in terms of handling failures and dealing with the external agents we communicate with via messages. Basically, I started to think about how we should handle the communication between services around the way we handle failures. So now that's something we always think about.

The reality is that any time we talk to external services, we should expect some failures. They're just going to happen. This means we shouldn't be banking on some external service responding in a short amount of time. We want to explicitly set timeouts. Then if we see that the service is starting to fail very quickly or with some high

frequency, we'll know it's time to trip a circuit breaker to ease the pressure on that service and not have those failures echo across all services. I have to say that came as a bonus benefit I certainly wasn't expecting when we first started working with Akka.

Providing for greater efficiency in the utilization of system resources by resorting to a distributed microservices architecture is one thing. But to what degree is that liable to end up shifting additional burdens to your programmers? After all, coding for asynchronous distributed systems has long been considered ground that only the most highly trained Jedi should dare to tread.

What can be done to ease the transition to a reactive microservices environment for programmers more accustomed to working within the confines of synchronous environments? Won't all the assumptions they typically make regarding the state of resources be regularly violated? And how to get a large team of coders up the concurrency learning curve in reasonably short order?

Here's what the Hootsuite team learned...

TC Let's talk a little about the impact the move to microservices has had on your developers. In particular, I would think this means you're throwing a lot more asynchrony at them than most developers are accustomed to. I imagine they probably also have a lot more data consistency issues to worry about now.

YB Although the asynchrony problem hasn't been fully

addressed, Scala Futures (data structures used to retrieve the results of concurrent operations) actually make it really easy to work with asynchronous computation. I mean, it still takes some time to adjust to the fact that anything and everything can and will fail. But, with Scala Futures, it's actually quite easy for relatively uninitiated programmers to learn how to express themselves in an asynchronous world.

ES If you're coming at this from the perspective of thread-based concurrency, it's going to seem much scarier for a lot of use cases than if you're coming at it from a Futures point of view. Also, when you're working directly inside actors, even though messages are flying around asynchronously and the system is doing a thousand things at the same time, you're insulated from what it takes to synchronize any state modifications, since an actor will process only one message at a time.

KEN BRITTON I've noticed when developers first start writing Scala, they end up with these deeply nested, control-flow-style programs. You see a lot of that in imperative languages, but there's no penalty for it. In a strongly typed functional language, however, it's much more difficult to line up your types through a complex hierarchy. Developers learn quickly that they're better served by writing small function blocks and then composing programs out of those.

Akka takes this one step further by encouraging you to break up your logic with messages. I've observed a common evolution pattern where developers will start off with these very bulky, complex actors, only to discover later that they could have instead piped a Future to their

own actor or any other actor. In fact, I've witnessed a number of aha moments where developers hit upon the realization that these tools actually encourage them to build smaller composable units of software.

JB That matches my experience as well. Actors are very object oriented, and they encapsulate state and behavior—all of which I think of as mapping well to a traditional approach. Futures, on the other hand, lend themselves to functional thinking—with all these small, stateless, one-off things you can compose easily. But have you found you can actually make these things derived from two very different universes work well together? Do you blend them or keep them separate?

YB We've used them together in parallel, and I think they work well that way. Ken mentioned this idea of generating Futures and then piping them either to yourself as an actor or to some other actors. I think that pattern works quite well. It's both simple and elegant.

ES I have to admit I stumbled over that mental shift a bit early on. But, yes, I'd say we've been able to blend actors and Futures successfully for the most part.

JB Do you feel that certain types of problems lend themselves better to one or the other?

ES In our simpler services, the routing of a request to the code is all actor based, and then the actual business logic is generally written as calls to other things that produce Futures. I suppose that when you're thinking about infrastructure and piping things around, it's very natural to think of that in terms of actors. Business logic, on the other hand, perhaps maps a little more readily to the functional point of view.

've witnessed a number of *aha* moments where developers hit upon the realization that these tools actually encourage them to build smaller composable units of sofware.
—Ken Britton

KB We're also finding that a rich object-oriented model is helpful in our messaging. For example, we've started defining richer success and failure messages containing enough detail to let an actor know exactly how to respond. So, now our message hierarchy has expanded to encapsulate a lot of information, which we think nicely aligns object-oriented concepts with the actor model.

TC One thing that occurred to me as you talked about your environment is that it seems you've moved not only from a monolithic architecture, but also, in some sense, from a monolithic technology to a much wider array of technologies. So I wonder if you now find it more difficult to operate in that environment, as well as to train people to work in it. Whereas before maybe it was sufficient just to find some new hires that were proficient in PHP, now you've got ZeroMQ and actors and Futures and any number of other things for them to wrap their heads around. Without question, your environment has become more complex. But is it now in some respects also actually an easier place in which to operate?

YB I think the act of dividing the logic into a lot of different self-contained services has made it easier at some level to reason about how the system works. But we're not finished yet. There's still plenty of work to do and lots of challenging areas to continue reasoning about.

And, yes, of course, the environment has become a bit more complex. I have to agree with that. But the benefits outweigh the drawbacks of rolling in all this technology, since we now have more layers of abstraction to take advantage of. We have teams that are generally aware of the big picture but are mostly focused on just a few

microservices they understand really well. That's an approach that will have huge benefits for our operations as we scale them moving forward.

KB It has become apparent how critical frameworks and standards are for development teams when using microservices. People often mistake the flexibility microservices provide with a requirement to use different technologies for each service. Like all development teams, we still need to keep the number of technologies we use to a minimum so we can easily train new people, maintain our code, support moves between teams, and the like.

We've also seen a trend toward smaller services. Our initial microservices were actually more like loosely coupled macroservices. Over time, though, we've pushed more of the deployment, runtime, and so forth into shared libraries, tooling, and the like. This ensures the services are focused on logic rather than plumbing, while also sticking to team standards.

CONTENTS