TECHNICAL WHITE PAPER

# Akka A to Z

## An Architect's Guide To Designing, Building, And Running Reactive Systems

**By Hugh McKee & Oliver White | Lightbend, Inc.**

Lightbend

## Table Of Contents

# Executive Summary

By now, you've probably heard of Akka, the JVM toolkit for building scalable, resilient and resource efficient applications in Java or Scala. With 16 open-source and commercial modules in the toolkit, Akka takes developers from actors on a single JVM, all the way out to network partition healing and clusters of servers distributed across fleets of JVMs. But with such a broad range of features, how can Architects and Developers understand Akka from a high-level perspective?

At the end of this white paper, you will better understand Akka "from A to Z", starting with a tour from the humble actor and finishing all the way at the clustered systems level. Specifically, you will learn about:

- How Akka Actors function, from creating systems to managing supervision and routing.
- The way Akka embraces Reactive Streams with Akka Streams and Alpakka
- How to build distributed, clustered systems with Akka, even incorporating clusters within clusters.
- How various components of the Akka toolkit provide out-of-the-box solutions for distributed data, distributed persistence, pub-sub, and ES/CQRS
- Which commercial technologies are available for Akka, and how they work, as part of a Lightbend subscription.

# A Little History of Akka

Believe it or not, Akka's roots begin back in 1973 (approximately 8000 years ago in the IT world), in a laboratory in MIT, where a mathematics Ph.D. named Carl Hewitt co-authored a paper called *A Universal Modular Actor Formalism for Artificial Intelligence*. In it, he introduced a mathematical model that treats "actors" as the universal primitives of concurrent computation[1].

*"[The Actor Model is] motivated by the prospect of highly parallel computing machines consisting of dozens, hundreds, or even thousands of independent microprocessors, each with its own local memory and communications processor, communicating via a high-performance communications network."*

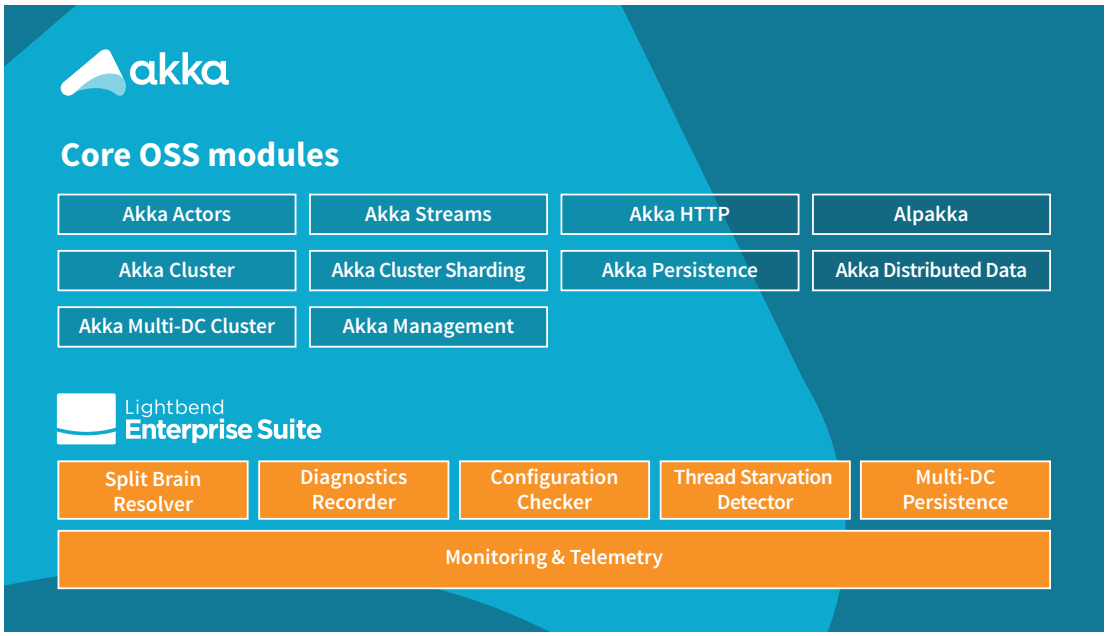
Carl Hewitt

At that time, low-latency, concurrent computations across a broad spectrum of cloud infrastructure was not truly viable in a commercial setting. It wasn't until much later that enterprise-grade cloud computing came along.

In 2009, Lightbend co-founder and CTO, Jonas Bonér, created the Akka project as a way to bring a distributed, highly concurrent and event driven implementation of Hewitt's Actor Model–as well as Erlang's implementation of it–to the JVM.

Now in 2018–just a year away from Akka's 10th anniversary–we continue to support the goal of bringing the power of multicore, asynchronous, self-healing and highly scalable systems to Java and Scala architects and developers–without having to know about all the low-level internal plumbing behind it all.

1 https://en.wikipedia.org/wiki/Actor_model




*The Akka toolkit contains a growing selection of open source modules, as well as commercial modules to support the needs of enterprise*

It's important to note that Akka is a JVM toolkit/library, rather than a framework (e.g. like Play and Lagom). While there are a selection of both open source and commercial components (part of Lightbend Enterprise Suite), the focus of this paper will be on actors, streams and HTTP, clustering, persistence and deployment concepts.

This journey will go from individual actors and how they behave through supervision and self healing, through to the Reactive Streams initiative with Akka Streams, Akka HTTP and Alpakka, then into the world of distributed persistence of event sourcing and CQRS, microservices, and finally distributed clusters and how to manage, monitor and orchestrate them.

With this in mind, let's start at the very beginning: Akka Actors.

# Part 1: Akka Actors And The "Lives" They Lead

## The Actor Model

At the atomic or molecular level of Akka, we begin by taking a look at actors, which present a different way of programming than what many of developers have grown up with. Most developers have learned object-oriented programming, or maybe even functional programming, but for the most part, most of it has been imperative and synchronous programming involving lots of threads and related elements.

The actor model brings in something that's very different. An actor itself just a class, which can be implemented in Java or Scala. An actor has methods, but the big difference between an actor and what we're normally used to as software developers is that the methods on an actor are not directly accessible from outside the actor.

Methods are not invoked on an actor directly, which takes some getting used to. The reason for this is that the only way to talk to an actor is to send it a message, which is sent through the actor system that Akka provides.

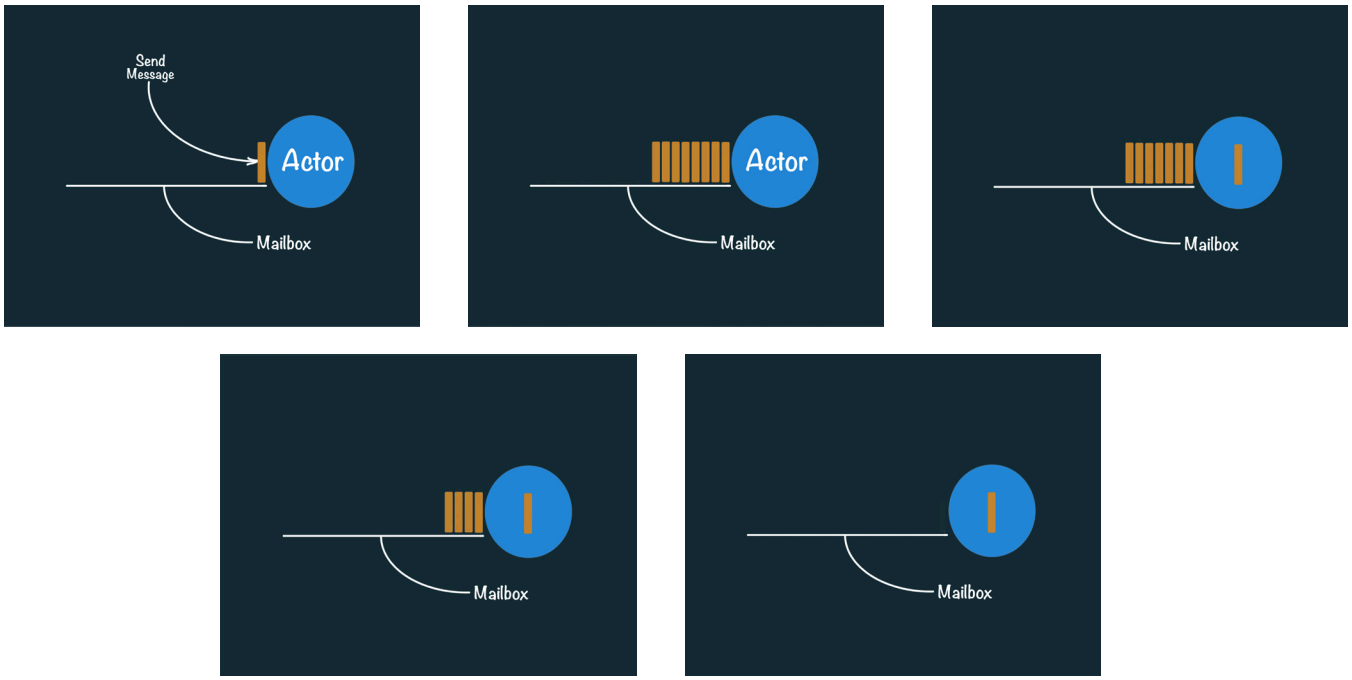



*Figure 1*

With actors, messages are sent asynchronously, not synchronously, between actors. As shown in Figure 1, messages are first deposited into a mailbox of pending messages for the actor to process. The actor works through these messages one at a time, message after message after message, processing them all.

That's all there is to basic mechanics of an actor. The end result is that an Akka system consists of a bunch of actors that communicate with each other by sending each other messages.


*Figure 2*

Here's how it works: in Figure 2, actor A is sending a message to actor B. This is an asynchronous message, just like two people sending each other a text message. A sends B a text message, which is received, letting A free to continue with other business.

A is not sitting in suspended animation waiting for a response from B, as may be seen in a typical method invocation with object-oriented programming, where the caller waits for a response. In this case, A sends a messages and it's free to move on, do something else, or it can just stop doing anything.


*Figure 3*

In Figure 3, B gets the message from A that triggers some kind of a state change; perhaps actor B represents a bank account, and the message signalling a positive withdrawal arrived, meaning the state change is the updated balance of that particular account. If desired, it's possible to program B to send a message back to A saying, "Hey I did what you asked me to do."

In reality, actors A and B may be running in a distributed environment, so B could be on another machine and unable to respond. Right from the very beginning, this raises the question about what to do; B needs to get the message from A, complete its task, and maybe send a message back, but that might not happen.

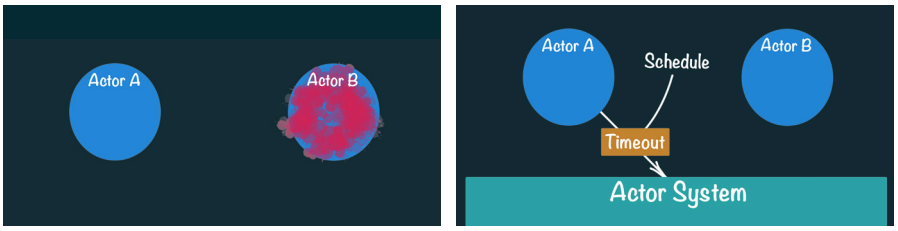



*Figure 4*

Figure 4 shows how A is able to report directly to Akka's actor system with a request: "Hey I would like you to schedule a message to be sent to me at some point in the future." This could be milliseconds, seconds, or some minutes. But a plan is in place for when A sends a message to B that it cannot respond to, a time-out message will be sent.

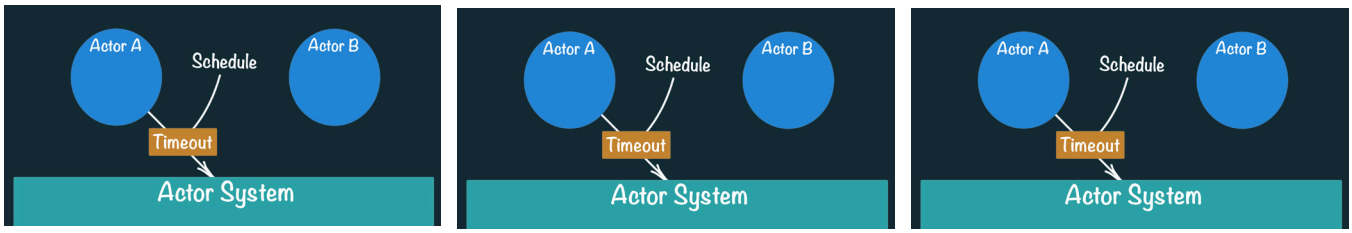



*Figure 5*

In Figure 5, A gets back this timeout message, which is one of two types of messages that A was created to process: a response from B or a timeout message. In either case, A knows what to do when either one of those two situations happen. This is different than exception handling, where most exceptions just get thrown. In Akka, resilience is an architectural feature, not an afterthought.

## Actor Supervision (Self-Healing)

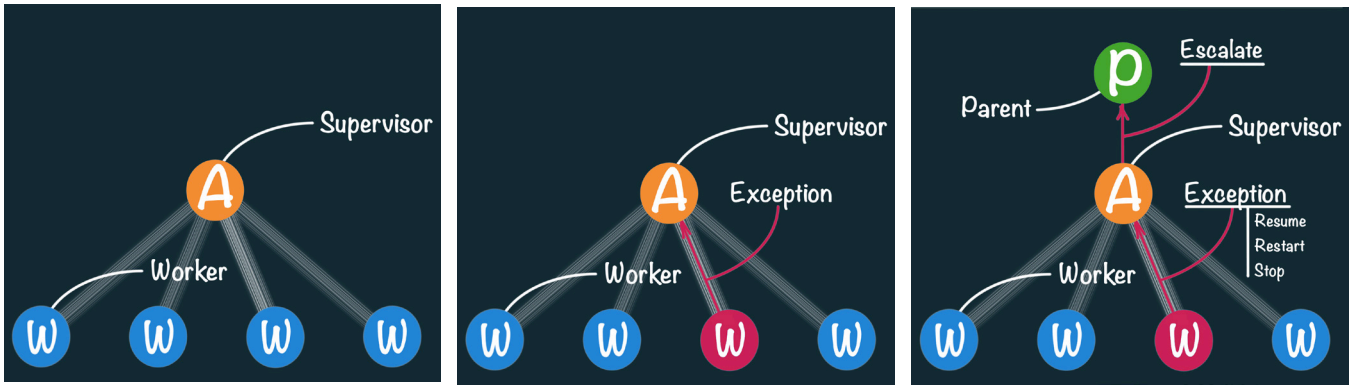



*Figure 6*

With resilience built into Akka as part of its design, now it's time to look at Actor Supervision. Actors are able to create other actors, forming actor hierarchies, as we can see in Figure 6. In this example, actor A, which is considered to be a supervisor, is creating these multiple worker actors (W). This produces a "parent-child" or "supervisor-worker" relationship between A and the actors that it creates.

The relationship between the supervisor and its workers is a supervision strategy, so that if one of the worker actors runs into a problem when processing a message it's not the worker itself that handles exceptions, it's the supervisor that takes over. This supervision strategy is well-defined, but also customizable–there will always be a plan for what do you do in the case of an exception within a worker actor that the supervisor reacts to.

The available responses for supervisor A that can be set up are to resume the worker actor if the exception really isn't that dangerous, or reset the actor by restarting it, or it could say, "This is really a pretty serious problem, I'm just going to stop this actor." Furthermore, supervisor A can look at the exception and say, "This is really bad. I'm not set up to handle this kind of a problem. I need to escalate this problem up to my supervisor."

So there's this whole escalation hierarchy here that is available for problems that bubble up, depending on the severity of it. All this is under the developer's control as the implementer, and the thing to remember here is that there's this well-defined way for handling problems.

A natural inclination for beginners to the actor system is to implement a hierarchy with actors that have a lot of code in them, but as familiarity with the actor model developers over time, it becomes easier to divide and conquer.

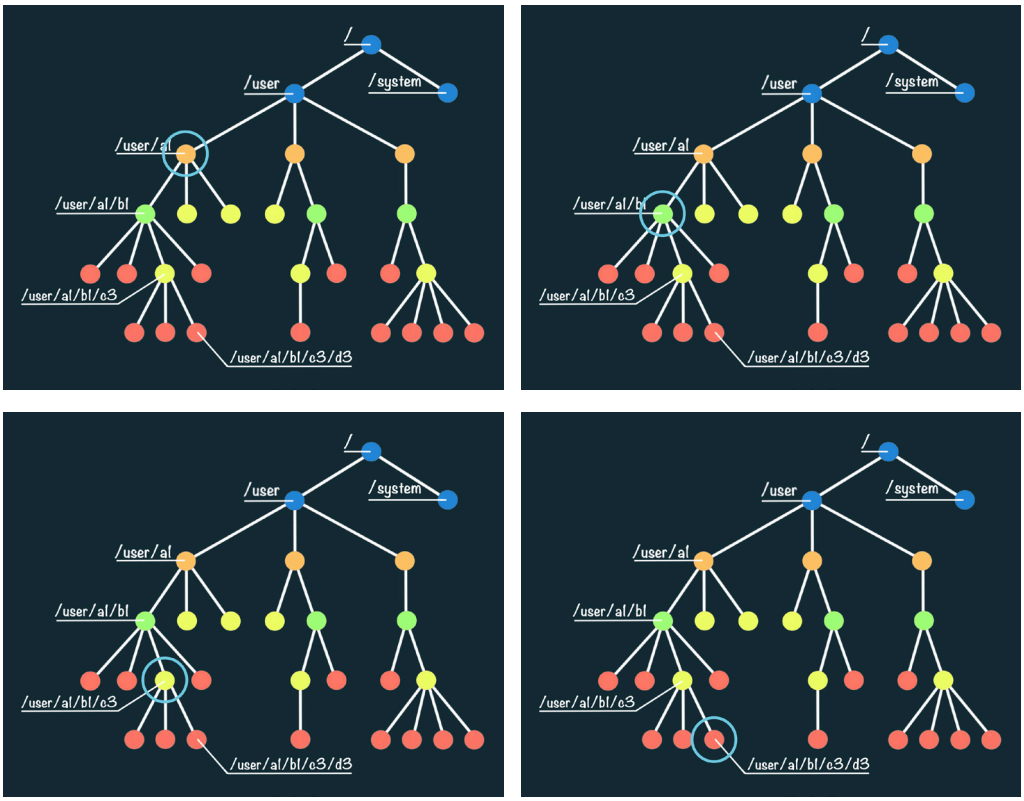



*Figure 7*

One reason for dividing and conquering has to do with reducing risk. The actor hierarchy is built to delegate "risky" operations off to the edge.

In Figure 7 we have user A1, B1 and C3, with C3 is pushing down some risky behavior down into these D actors (i.e. D3). The edge nodes here are red, signifying that these actors are doing risky things, for example talking over the network to a database. This is risky because the network can go down, the database could go down, and many other things could go wrong.

By pushing that risky behavior off into worker actors, the actors above it are in a way somewhat shielded. With actor hierarchies, work is delegated to provide more fine-grained specialization of actors, such as pushing risky behavior off into the leaves of the tree.

## Routing and Concurrency

Akka specializes in multi-threaded, multicore concurrency, making it simple to handle operations that we would traditionally set up with multiple threads by hand. Things here can easily get complicated, so Akka enables out-of-the-box concurrency without getting into a lot of the technical challenges of multi-threaded programming in Java and Scala.

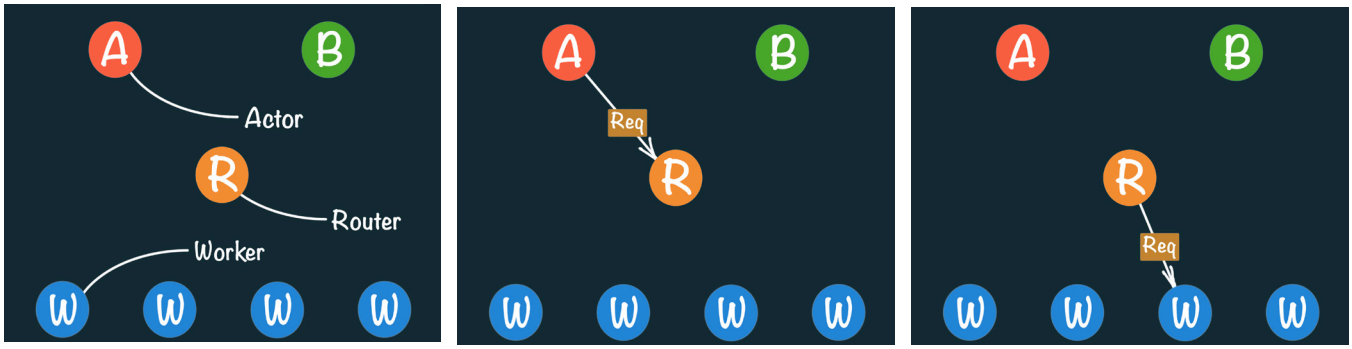



*Figure 8*

In Figure 8, we have actors A and B, and a router actor R, which has has created a bunch of workers so A and B can get some work done. A sends a message to the router actor asking it to do something. The router actor doesn't actually do the work, but delegates it off to a worker actor.

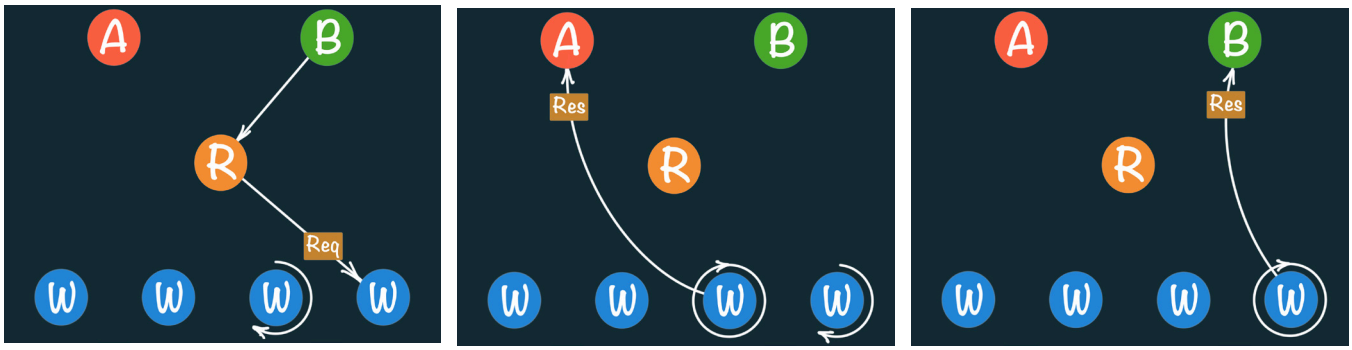



*Figure 9*

While that worker is performing the task for A, now B sends a message to the router actor asking it to do some work. In Figure 9, the router actor forwards that message off to one of its other workers and now we have multi-threaded concurrency without any manual coding. From the perspective of A and B, R is doing a lot of work and it's doing a lot of work at the same time, but in reality R has, behind the scenes, delegated that work off to these actors.

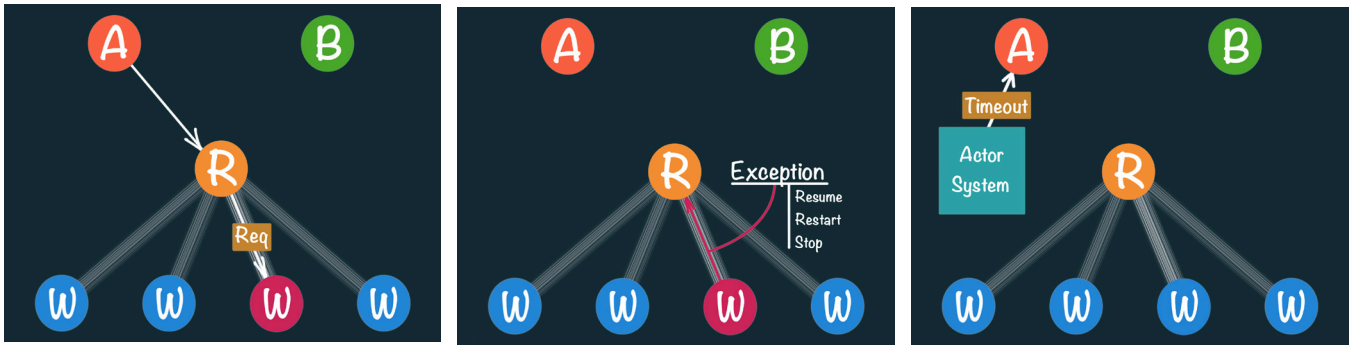



*Figure 10*

In Figure 10, we look again at Akka's supervision strategy in this example. Let's imagine that A sends a message to the router, which forwards that message off to a worker, but the worker runs into a problem.

In normal programming with the "try-catch" method, the caught exception would normally be sent to A; however, with the actor system, it's the supervisor that sees the exception. So A initially sent this message to to R, which forwarded it off to the worker. The worker ran into a problem. It's the supervisor actor R that sees the exception, so A doesn't get back the response back that it's expecting.

This is where the timeout mechanism can be implemented, so that R can send a response back to A saying, "Hey there's a problem and I can't do what you asked me to do". Regardless, A is not seeing the exception and the caller is not having to deal with an issue that may not best handled by the clients of the service.

With the actor model at the core of Akka, it's clear that there is a lot of powerful built-in plumbing to manage concurrency and resiliency. While this represents a different way of thinking, it's a highly productive and safe way to build distributed systems.

## To Sum Up

In short, here are the key takeaways from this section:

- Actors are message-driven, stateful building blocks that pass messages asynchronously.
- Actors are lightweight and do not hold threads, unlike traditional synchronous and blocking technologies
- Actors may create other actors, and use native supervision hierarchies to support self-healing (resilience) in the face of errors.

# Part 2: Streaming Workloads with Reactive Streams, Akka Streams, Akka HTTP, and Alpakka

One of the most popular OSS modules of Akka came out just a few years ago: Akka Streams. Based on the Reactive Streams initiative, which Lightbend spearheaded along with engineers from companies like Netflix, Oracle, Pivotal and more, Reactive Streams enables processing of data streams with backpressure. Like TCP, backpressure prevents a producer from overloading a downstream consumer that can't handle anymore flow.

## Reactive Streams and Akka Streams

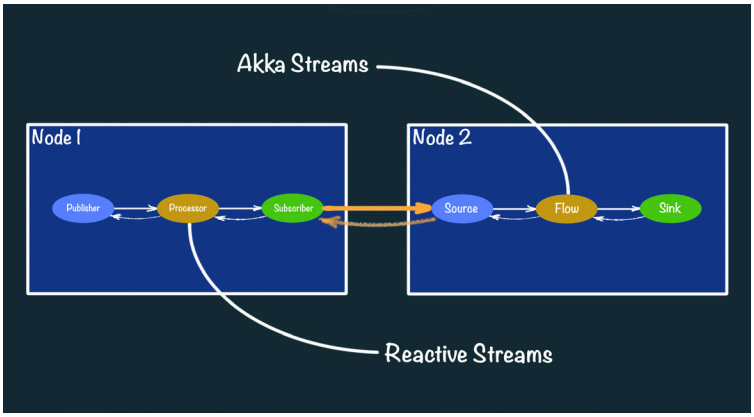



*Figure 11*

In Figure 11, we see on the left (the Reactive Streams Java 9 terminology) a publisher, a subscriber, and a processor, which are the main components of a stream. In JDK 9, this implementation is not intended to be used directly, but rather as a foundational piece for streaming. Akka Streams is built on top of the Reactive Streams standard, plus adding more functionality beyond that.

On the right side in Figure 11, representing the terminology used in Akka Streams, there are sources, sinks, and flows, versus with Java there was a publisher, subscriber, and processor. It's the same functionality, and Akka Streams brings the experience of Lightbend's Akka team and the OSS community to add functionality in the flow area. Akka Streams provides things that would be familiar to any functional programmer, like filter, and fold, and map, which are methods supplied with the flow for transforming data as it flows through the stream.

## Akka Streams, Akka HTTP, and Alpakka

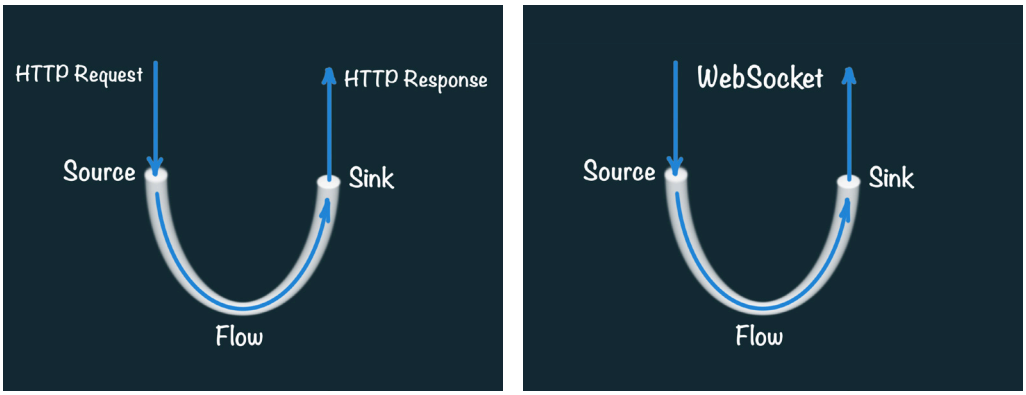



*Figure 12*

Akka HTTP is built on top of Akka Streams, which is a natural fit for an HTTP request which should be transformed into an HTTP response (Figure 12). The flow is the transformation that occurs within the stream, which also works with websockets to process messages as they're coming in, or going out, using Akka Streams and/or Akka HTTP.

For enterprise integrations, there is another project called Alpakka, which brings a Reactive, Akka Streams-based alternative to Apache Camel. Similar to the Camel community, the open source community has heavily embraced building out different connectors (to the right) in order to create a backpressure capable upgrade to traditional Camel endpoints.

For example, taking a source of Wikipedia images Alpakka can "enrich the data" and push it out to various connectors, such as Amazon S3, Apache Kafka, or Microsoft Azure Storage Queue.

### Connectors

- AMQP Connector
- Apache Geode connector
- Apache Solr Connector
- AWS DynamoDB Connector
- AWS Kinesis Connector
- AWS Lambda Connector
- AWS S3 Connector
- AWS SNS Connector
- AWS SQS Connector
- Azure Storage Queue Connector
- Cassandra Connector
- Elasticsearch Connector
- File Connectors
- FTP Connector
- HBase connector
- IronMq Connector
- JMS Connector
- MongoDB Connector
- MQTT Connector
- OrientDB Connector
- Server-sent Events (SSE) Connector
- Slick (JDBC) Connector
- Spring Web
- Unix Domain Socket Connector

Alpakka supplies a lot of power in fairly concise code, using Akka Streams and Akka actors under the covers to avoid dealing directly with low-level internals. With Akka Streams there is a more functional type of programming to deal with flows of things, and in general, the data is passing through various functions to perform different transformation operations.

## To Sum Up

In short, here are the key takeaways from this section:

- The Reactive Streams initiative seeks to provide backpressure to streaming workloads to avoid overloading producers or consumers.
- Akka Streams is a Reactive Streams implementation that provides a rich set of flow processing transformations.
- Built on Akka Streams, Akka HTTP and Alpakka are backpressure-enabled modules that provide resilient HTTP server and enterprise integration functionality as part of the Akka ecosystem.

# Part 3: Building a Cluster with Akka Cluster and Clustering Sharding

One of the most useful features of Akka for enterprise users is the creation of clusters. For those who have been developing in Java for a long time, Akka opens up a universe where development is no longer constrained to a single JVM: Akka-based applications behave as a single system running in a distributed environment with multiple JVMs. Here is how that works...

## Creating A Cluster

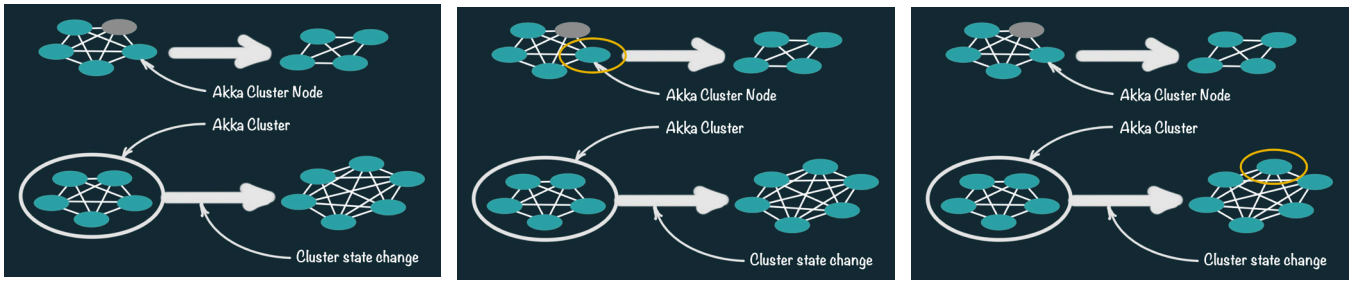



*Figure 13*

In Figure 13, there is a collection of JVMs that are running together as a single Akka cluster. They are aware of each other, and monitor each other through a gossip protocol of heartbeats that keep an eye on everything, described in greater detail in Figure 14.

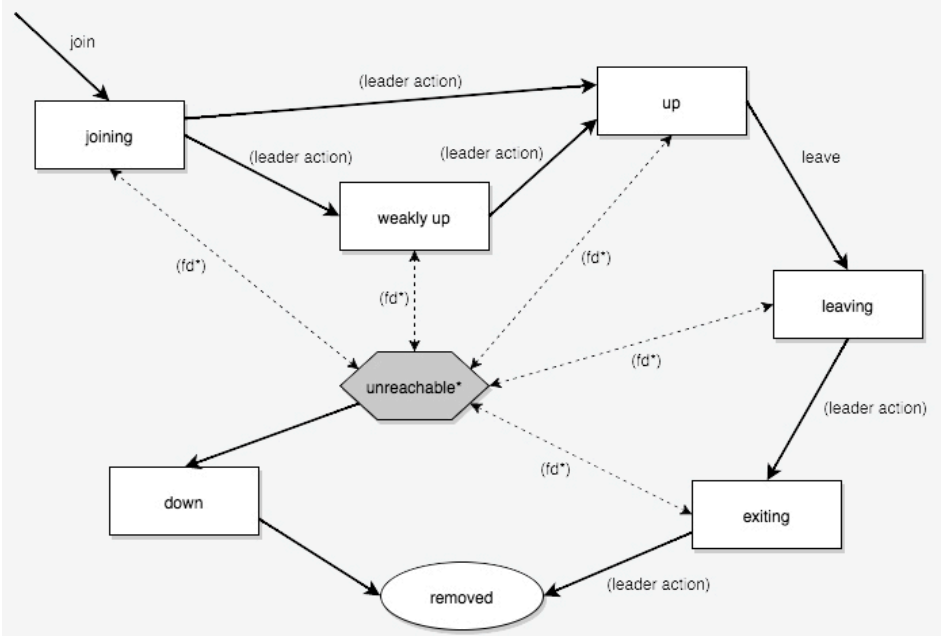



*Figure 14*

Here, the management of all these nodes within the cluster is handled by Akka itself. Akka knows when nodes join or leave the cluster, and there's a well-defined way for Akka to keep the system running when the topology of the cluster changes. Developers and architects are participants in this methodology, but the real use case is focused on how actors react to changes in the topology of the cluster without having to worry about anything.

## Cluster Sharding

Another interesting pattern of Akka actors that has been set up out of the box in Akka is *cluster sharding*. Cluster sharding is used when distributing actors across several nodes in the cluster with the ability to interact with them using their logical identifier, but without having to care about their physical location in the cluster (which might also change over time).

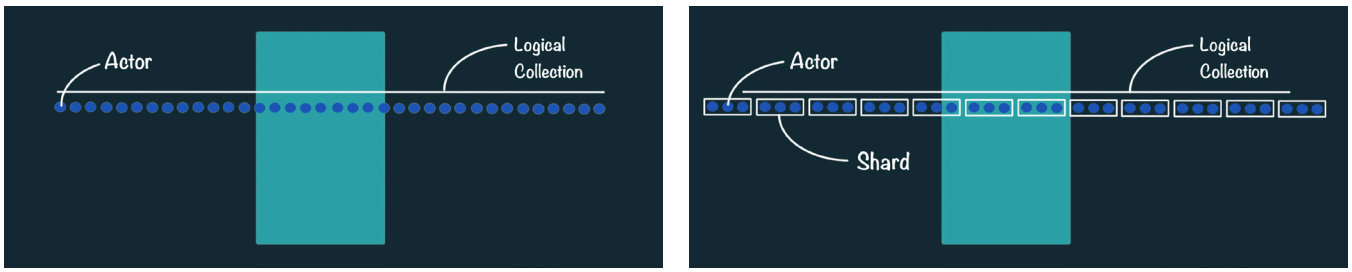



*Figure 15*

In Figure 15 (left), imagine an IoT use case where each actor represents the state of a device or entity. There may be a large collection of entities and a subset of those entities are currently in use. It would be best to have the state of each device stored in memory, but they don't fit on one machine. In Figure 15 (right) the rectangular box represents a single node that's too small for this large collection of actors to fit on.

It's also likely that putting all these actors on a single machine is a poor choice--it would be better for the system's resilience and elasticity to be running across multiple machines. That way, if a node is lost, there are other nodes that can pick up the slack while the system is running, with perhaps a momentary burp while reacting to the loss of a node (as opposed to a total system failure).

Simultaneously, the actors on that node are recovering while the system as a whole keeps running without a complete failure out of a monolith horror story.

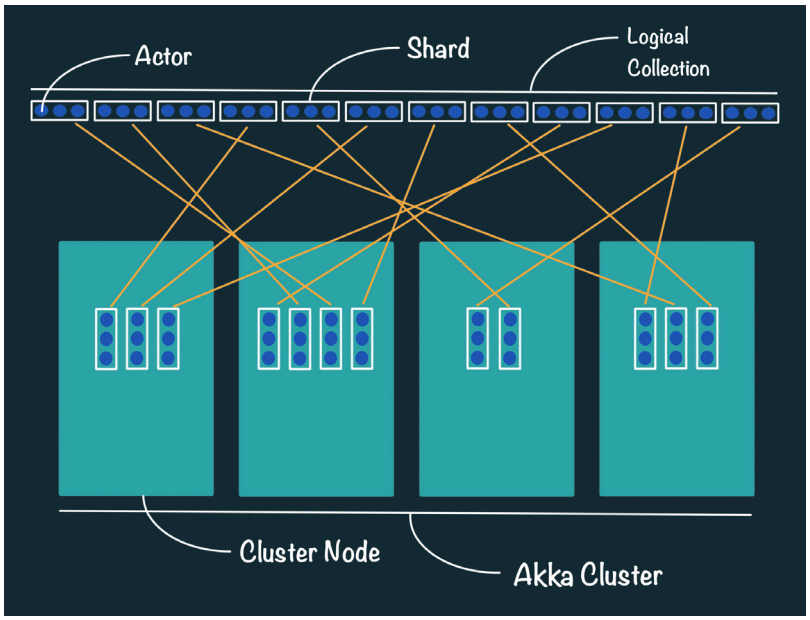



*Figure 16*

As seen in Figure 16, cluster sharding enables a large collection of actors to be broken up into logical shards. To use an IoT example, image that there's some kind of device identifier from which a hash code of it is taken. This then lets you do a modulo divide and that comes up with the shard ID for each one of the entities. Cluster sharding uses a well-defined sharding strategy to take that logical collection and distribute those shards across a cluster.

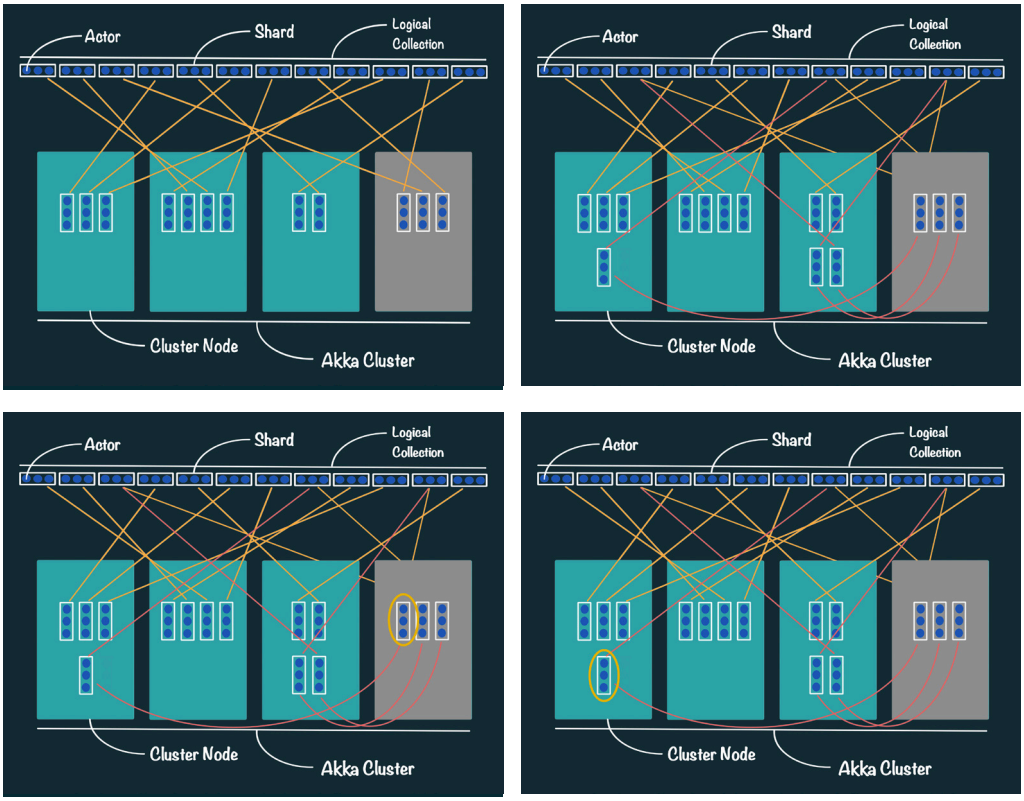



*Figure 17*

In Figure 17, there is a cluster of four nodes where cluster sharding is handling where each shard is distributed across the cluster. It handles the routing of messages to each one of the individual actors (the small blue dots) where an actor represents a particular entity, like a bank account or an IoT device

It really starts to get interesting when these shards that are distributed across the cluster and a node goes down. When a node goes down, cluster sharding knows which shards were on that node and it helps in the resurrection of the actors that were taken out by the loss of this node by redistributing those actors to other nodes in the cluster.

Because cluster sharding manages of all of these shards and their distribution, developers don't have to worry about any of that. This leaves the focus on the recovery of the state of these actors–which is definable depending on the use case at hand. For example, if these actors each represent bank accounts, they should be recreated on other nodes in the cluster its state (i.e. the current balance of this account) has to be recovered. This is handled in part by Akka Persistence, which will be described later on in this paper.

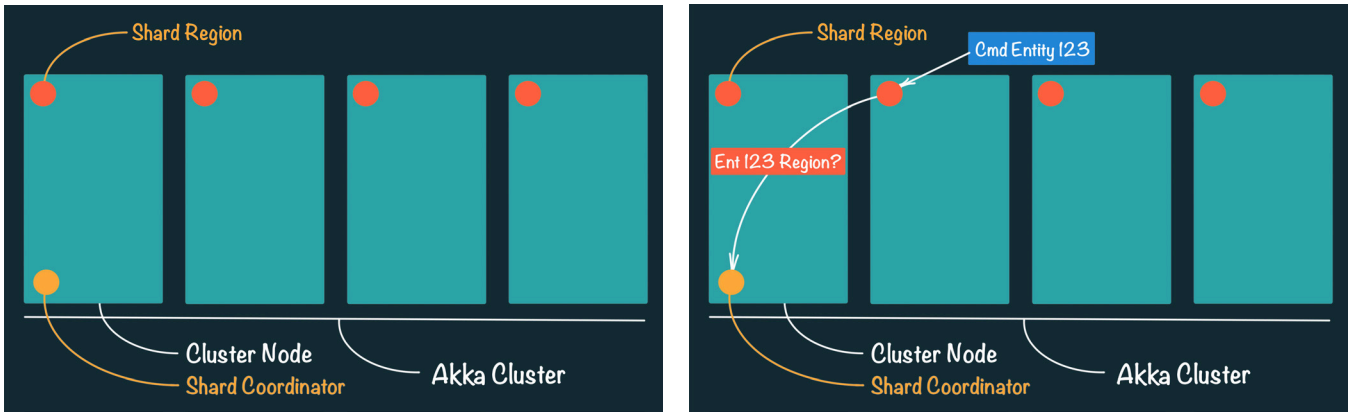



*Figure 18*

In Figure 18, we see how cluster sharding works by using a Shard Coordinator actor, shown in the bottom-left of the image. There is one Shard Coordinator for the entire cluster, which works with a Shard Region Actor that exists on each node in the cluster.

This allows us to disregard where the specific entity actor is in the cluster when a message is sent it. When a request comes in, there's a load balancer in front of it that round-robins messages across the cluster. When a message comes in for Entity 123 and it just happens to fall on node two, the Shard Region actor for node two gets it and responds "I don't know where this actor is. Where is it in the Cluster?"

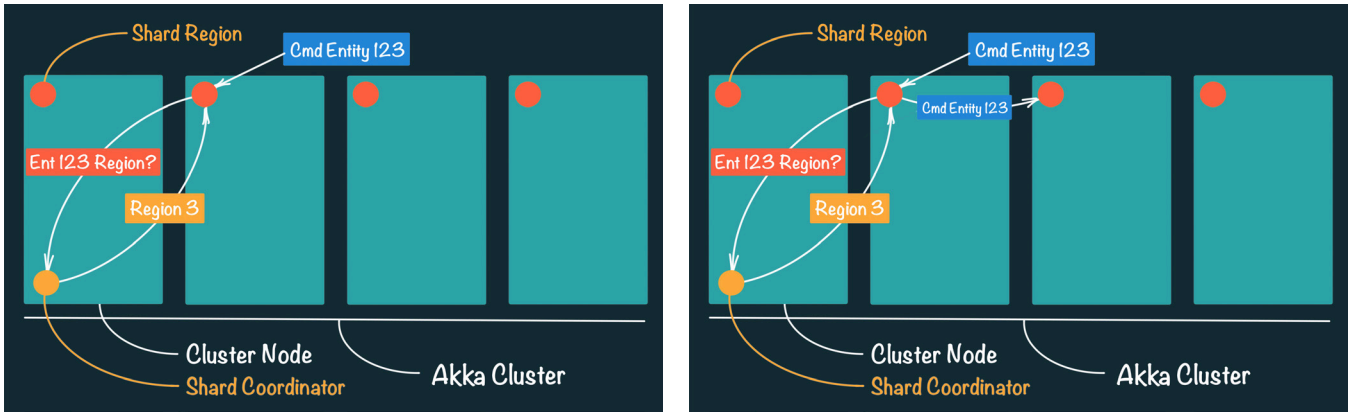



*Figure 19*

In Figure 19, the Shard Region actor for node two then sends a message to the Cluster Singleton Shard Coordinator asking, "Where is Entity 123 supposed to be distributed across all the Shards in the Cluster?" The lookup process, determining where a given entity is located in a cluster, is done using a shard Id that is computed from each incoming entity message. The ShardCoordinator decides which ShardRegion shall own the Shard and informs that ShardRegion.

From here, the Shard Coordinator sends a message back saying, “Oh, it’s in Region Three”, allowing the Shard Region actor on node two to forward the messages saying, “Hey, here’s a message for Entity 123, off to Shard Region Three.”

Now, the Shard Region actor on node three will have to go through the same cycle, because it doesn’t know either. So, it asks the Shard Coordinator “Where is Entity 123?”. When the Shard Coordinator replies that Entity 123 is on node three, then node three is updated with the fact that Entity 123 belongs on its node. The beauty of this, again, is that developers only need to write the code to send a message to the Shard Region, and that’s it.

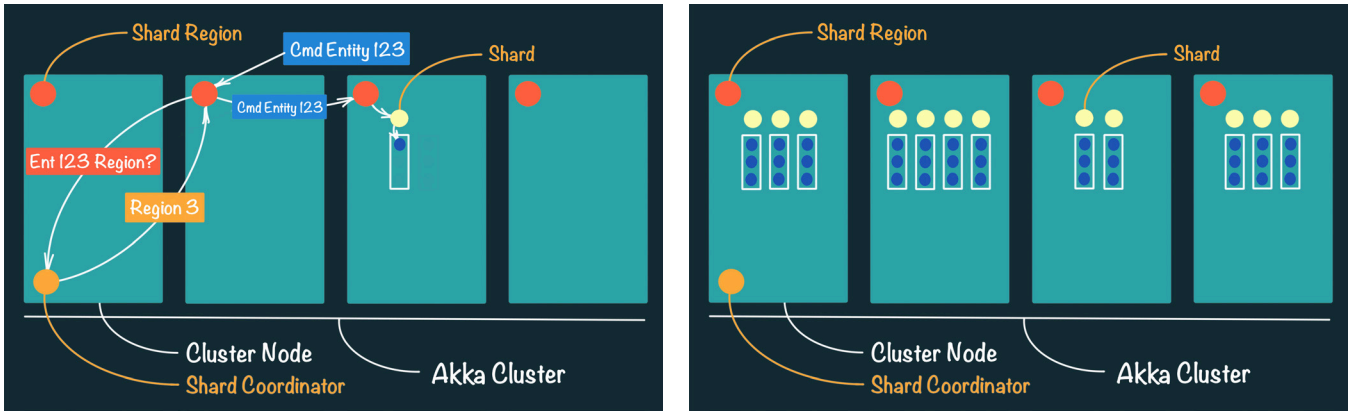



*Figure 20*

As seen in Figure 20, Cluster Sharding handles the rest of it with intelligence built in to allow each Shard Region to remember where the Shards are, so it doesn’t have to continually ask the Shard Coordinator where a particular Shard is. Once it’s been told where a Shard is, all the Shard Regions remember.

The complexity of these interactions are in reality handled by just four different elements: the Shard Coordinator, the Shard Region, we’ve got the Shard, and we have the Actors themselves. These represent the Cluster Singleton actor, the Node Singleton actor, and then the Sharded actors themselves.

## To Sum Up

In short, here are the key takeaways from this section:

- Akka’s Cluster features provide a host of functions for scaling your Reactive systems in a safe, and supervised way.
- Cluster Sharding is implemented for different use cases as a way to ensure state is maintained when messages travel across various asynchronous boundaries.
- Developers do not have to worry about the low level implementation of these interactions, since Akka takes care of it natively.

# Part 4: Akka Cluster with Event Sourcing & CQRS, Publish/Subscribe, and Distributed Data

Cluster sharding is a powerful and important tool for maintaining state across actor systems, but what happens when you add data persistence and storage to the story? This is where Event Sourcing and CQRS (Command Query Responsibility Segregation) enter the scene. These features have been gaining in popularity recently as distributed systems increase in adoption, and are provided in Akka as a different way of working with persistence and microservices.

## Event Sourcing and CQRS

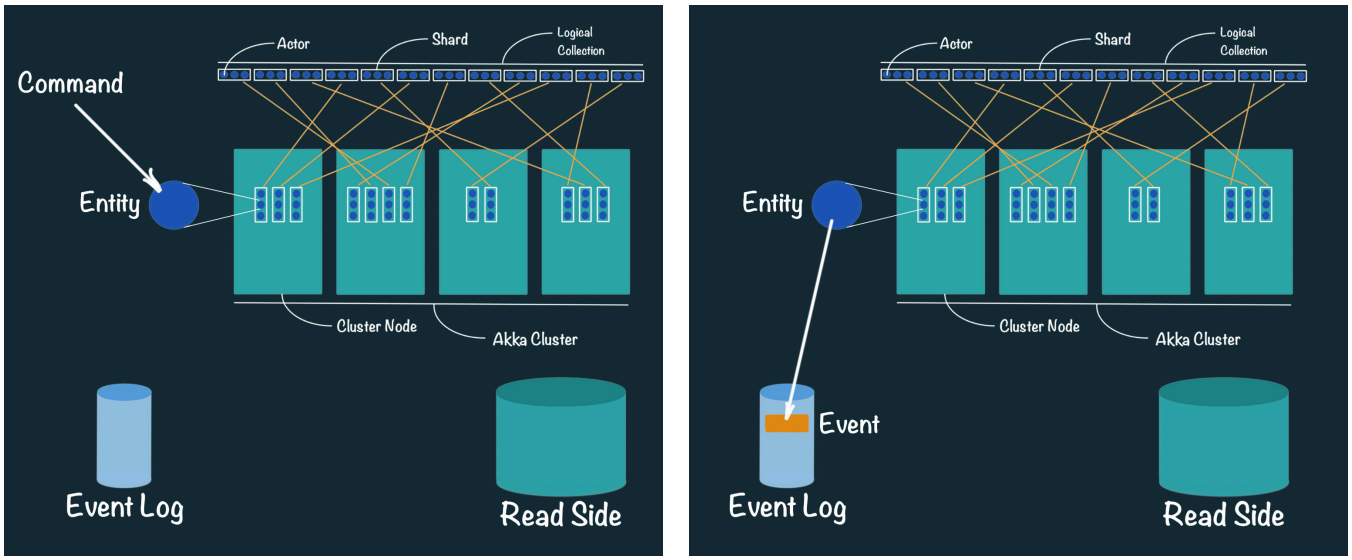



*Figure 21*

Continuing with the Cluster Sharding example, in Figure 21 Akka Persistence is used to ensure that if we lose a node and the actors have to be recreated, it's possible to recover state with some kind of persistence store. Akka Persistence uses Event Sourcing–with an event log–to accomplish this.

When a command comes in, it gets routed to the Entity, by the Entity ID, through Cluster Sharding and then when the message arrives at that actor, the actor is implementing Akka Persistence. Let's return to the bank account example where strong consistency is needed: in the Akka Persistence pattern there's a persister and in this case, we're actually persisting the event that says "Here's a deposit".

A command is a request to do something in the future, whereas an event is saying, "yeah, it happened". What is being stored is the fact that it was done–the deposit was accepted, and the withdrawal from somewhere else was accepted. Once the event is persisted in the event log, the entity itself updates its state and, either incrementing or decrementing the balance.

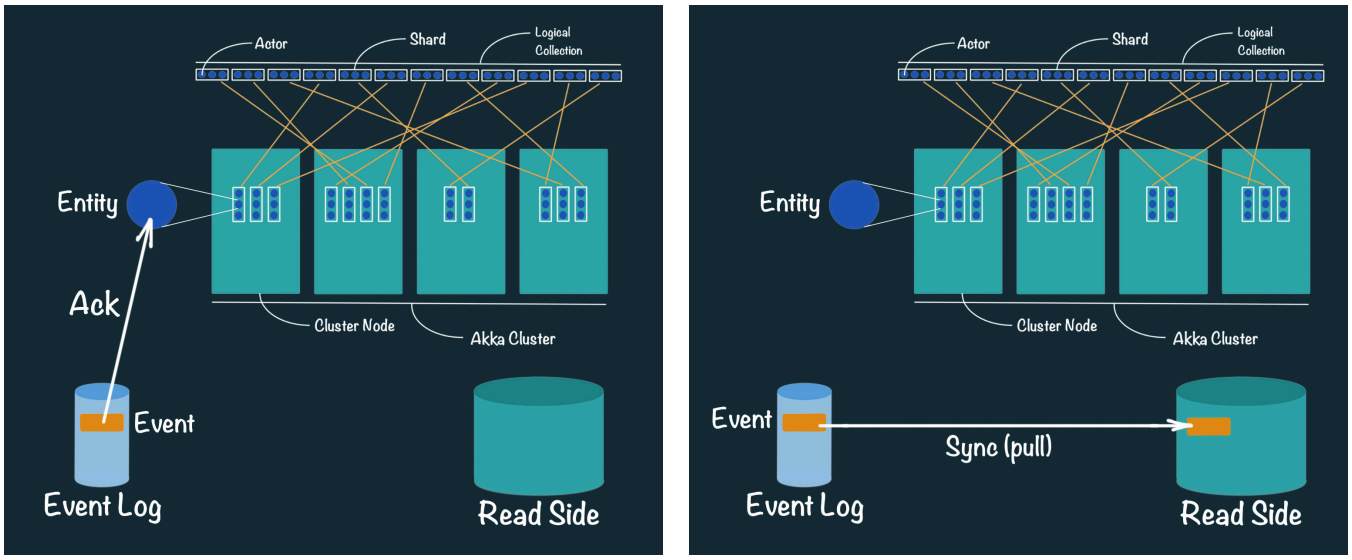



*Figure 22*

In Figure 22, another feature called Akka Persistence query is utilized, which looks at the query responsibility of CQRS–this is also known as the *write side* and the *read side*. The write side is the event log, the read side is where data is stored in a more queryable way. Because an event log isn't very queryable, what's happening here is that the events are being propagated from the event log into the read side. This is non-transactional, meaning there isn't a single database transaction, so there has to be some things done to guarantee that events aren't missed. This important feature is built into Akka Persistence query, where the read side is pulling events from the event log, rather than the more brittle approach of pushing events from the event log to the read side.

## Publish and Subscribe

Akka has built-in features to ensure that commands and events are handled appropriately, called publish and subscribe. In this case, more actors are added to the mix in order to handle this, including mediator actors, published actors, and subscriber actors.

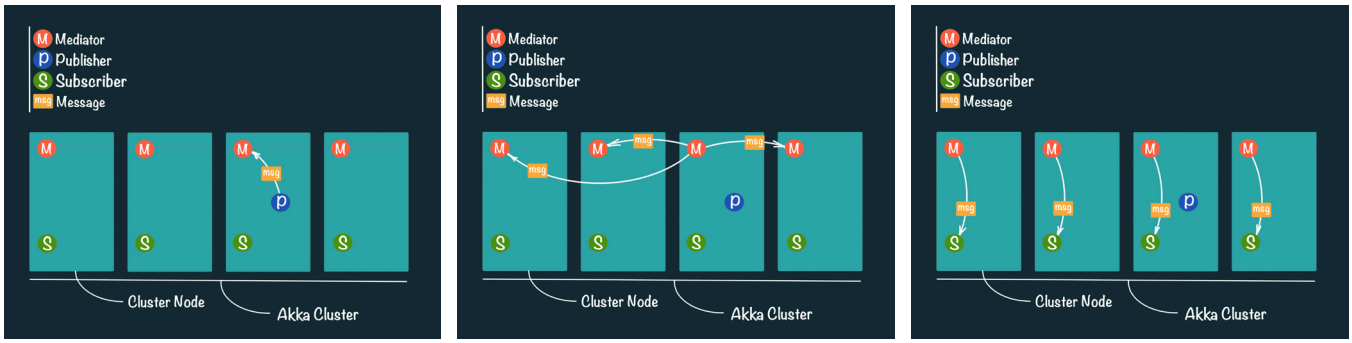



*Figure 23*

If Figure 23, there is a mediator actor in each node in the cluster. There are N number of subscriber actors that have registered some interest in a kind of event. Then there are publisher actors that publish an event (i.e. send a message). The publisher simply sends a message to its local mediator actor on its local node, and it's done. These mediator actors know that there's a cluster here, and they know that they've got counterparts on other nodes within the cluster.

The mediator actor that first gets the message fires off that message off to its teammates across the cluster and says, "Hey, here's an event that was published and someone needs to deal with that". In the same round-robin way as described in cluster sharding, each mediator actor then sends that message onto whatever subscribers are available. This happens as a cascading series of messages across the cluster using the actors designed for publish and subscribe, but developers don't have to worry about any of this since the local mediator actors already know what to do.

## Distributed Data

Whereas Akka Persistence focuses on maintaining state for strong consistency, Akka Distributed Data enables eventually consistent messages to be sent from actors on any node in the cluster without much coordination. This is handled through the concept of Conflict-Free Replicated Data Types (CRDTs), which are useful for things like counters, sets, maps and registers, where high read and write availability (partition tolerance) with low latency is needed.

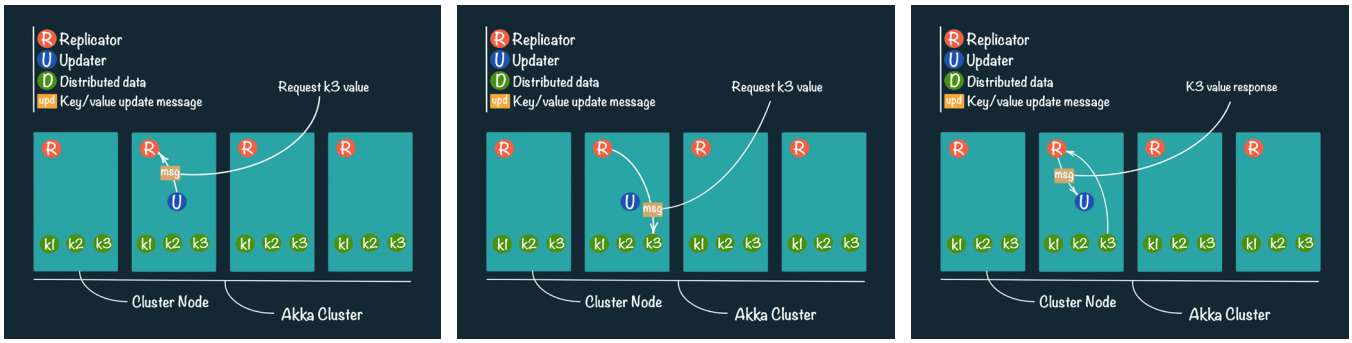



*Figure 24*

There are some restrictions on the kinds of data that you can use with Akka Distributed Data; for instance, in an eventually consistent system a read may return an out-of-date value. Akka handles this by using other types of actors that are distributed across the cluster. In Figure 24, there are replicator actors that have been created on each node across the cluster.

In this image, there is an actor for each K value (K1, K2, K3) replicated across the cluster, along with an updater actor on node 2. If the updater actor wants to know the value of K3, so it sends a message to the replicator. The updater actor doesn't know anything else in this environment, just that it's sent a message to its local replicator actor. The replicator actor retrieves the value from K3 and sends a message back to the updater with the value.

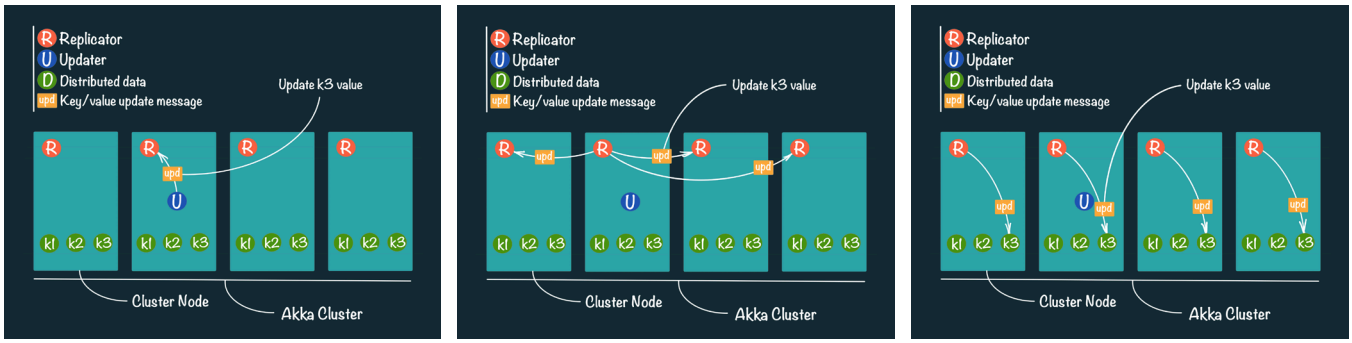



*Figure 25*

In Figure 25, the updater actor requests a change in the value of K3. Now the situation has changed, because the new value of K3 must be replicated across the cluster for all instances of K3. The replicator actor is cluster aware, so it knows that it has teammates on other nodes in the cluster. It forwards that update message off to the other replicator actors, which are responsible for changing their local copy of K3.

From the developer perspective, it's simply a message sent to the local replicator actor; the magic is happening automatically through the actions of the other actors, which is something that doesn't have to be coded for manually.

## To Sum Up

In short, here are the key takeaways from this section:

- Akka supports Event Sourcing and CQRS techniques for distributed data layers operating in Reactive systems
- Modules such as Akka Persistence and Akka Distributed Data provide mechanisms for both strong and eventual consistency
- Akka Cluster awareness allows for a “fire and forget” publish/subscribe paradigm that abstracts away the low-level plumbing needed to support distributed systems.

# Part 5: Monoliths, Microliths, Lambdas, and Reactive Systems In Production

Taking a step back up to the systems level, it's clear that Akka is designed to assist enterprises in moving away from building Monoliths when this design pattern is no longer advantageous. Monoliths are what most developers have been doing for a long time, and are well understood. This comfort level has produced a sort of hybrid approach between monoliths and microservices, which some are calling "microliths".

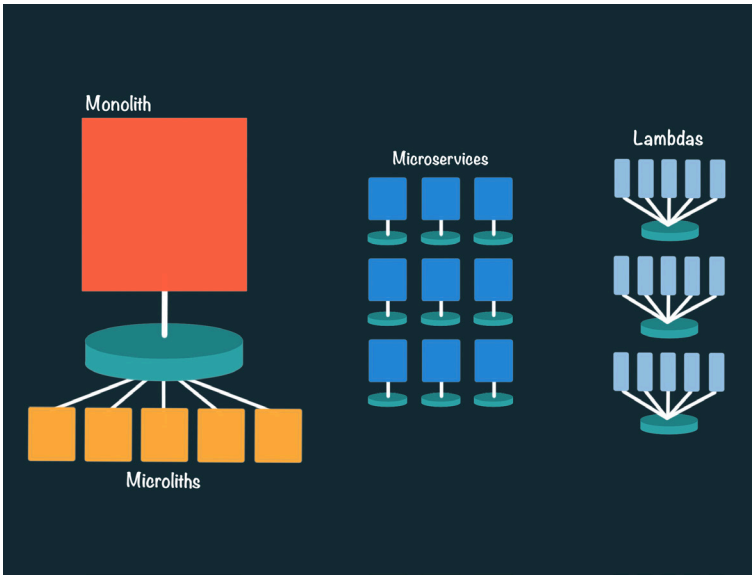



*Figure 26*

As seen in Figure 26, a microlith shares some of the qualities of a microservice, but not quite; they are microliths because they are all using the same database.

The whole point of a microservice is to have loose coupling between services, where coupling only happens at the API level. So while a microlith may be independently deployable, true microservices don't share databases and instead rely on concepts like event sourcing, CQRS and CRDTs, as mentioned earlier.

Even more exciting are Amazon's introduction of Lambdas, which are server-less functions that Akka is ideal for: for a function to perform very quickly, Akka takes care of the multi-threading and concurrency to ensure performance.

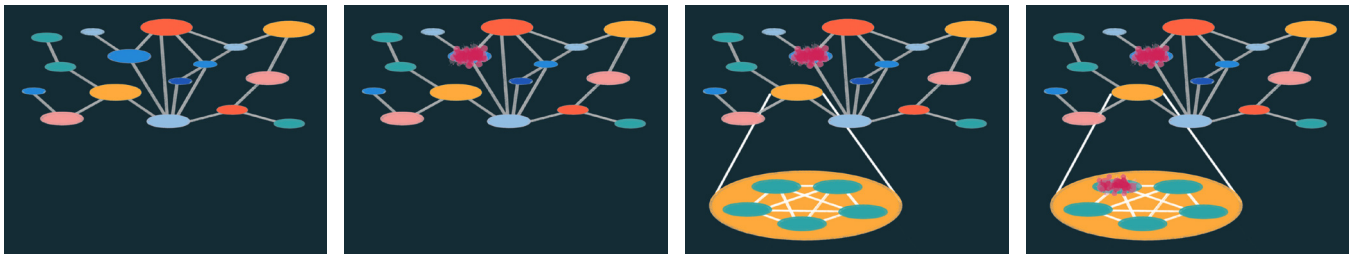


*Figure 27*

As illustrated in Figure 27, Akka ensures that a distributed system running on multiple nodes can take a hit and keep on going, as well as providing the ability to increase capacity whenever needed.

This is especially important these days, where every time some system goes down (think airlines, for example) the story is reported on the news. Even Amazon cannot be immune to these outages, but the point here is that no one wants to be in the news when the inevitable happens.

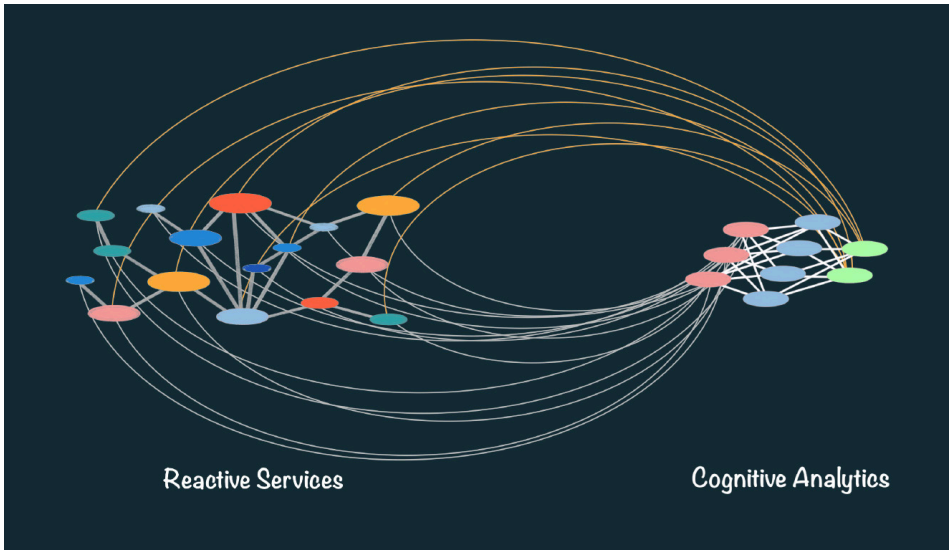



*Figure 28*

This brings application development into a new realm, where Reactive Services are producing data, and Cognitive Analytics based on machine learning, AI, etc. are consuming the data produced (Figure 28).

With data as the "new gold", these systems work together to feed analysis back into the application to make those applications better, smarter, and more compelling to their users. Akka helps developers build these systems in a Reactive way, so that these systems are responsive no matter what the load is, they're resilient to network outages or nodes going down, and they are elastic so that they can expand, or contract, depending on the traffic load, saving money.

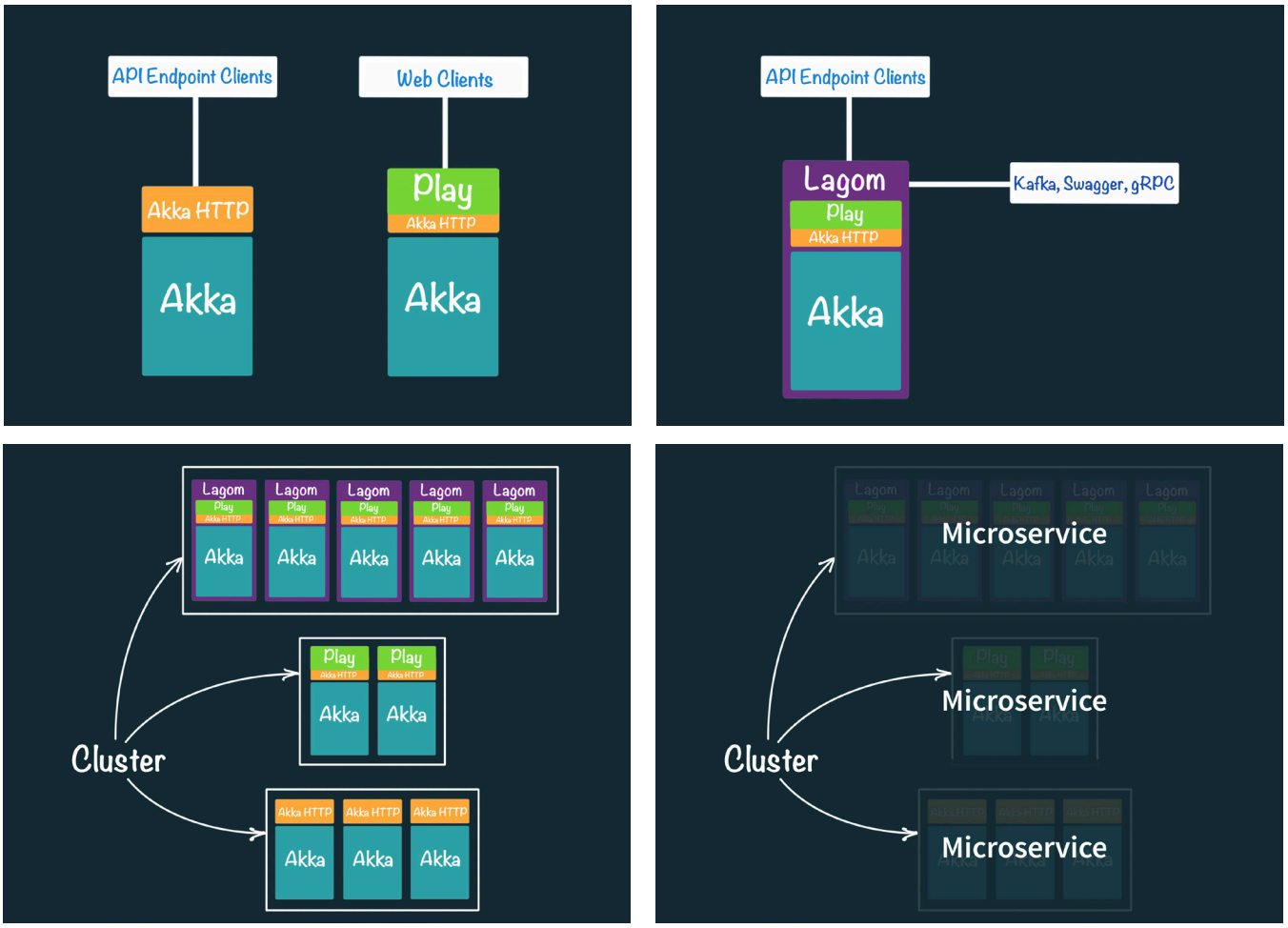
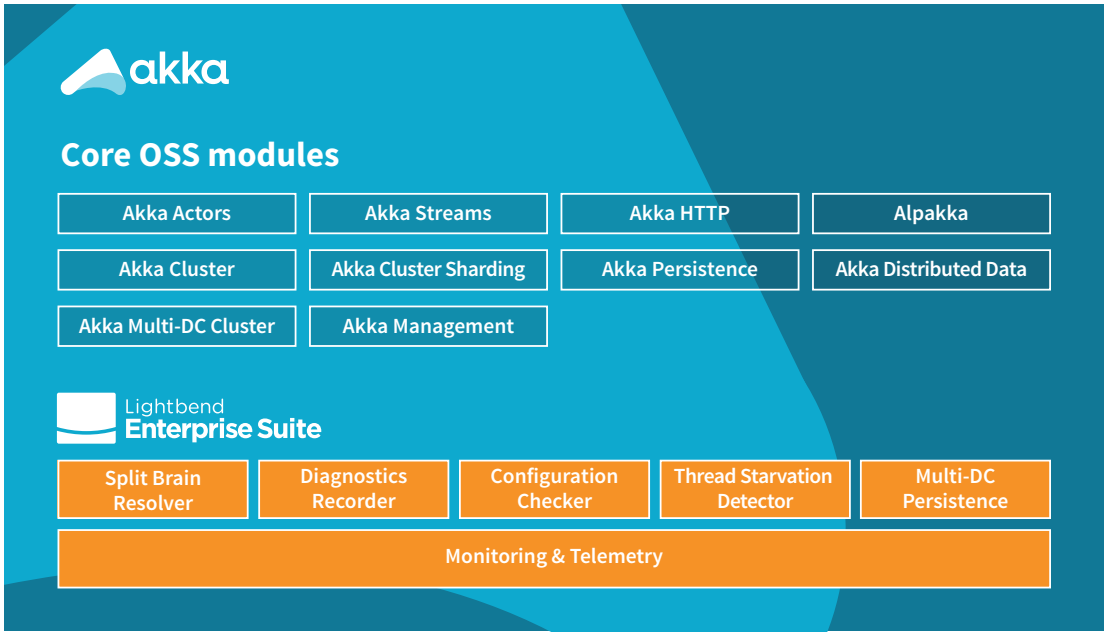



*Figure 29*

In Figure 29, we see how Akka represents the core of other Lightbend technologies like Akka HTTP for RESTful endpoints, Play Framework for web clients, and Lagom for building microservices in an opinionated way using event sourcing and CQRS. These configurations are represented as clusters, and for developers building microservices, it's easy to create them in a way so that they are clusters in themselves.

## Commercial Features For Serious Enterprises

Beyond the open source Akka modules presented in this white paper, Lightbend offers a selection of commercial technologies to support the needs of enterprises. From monitoring and telemetry to pre-defined techniques for resolving network partitions and configuration hints, these tools are part of the Lightbend Enterprise Suite, which is included as part of a **Lightbend subscription**.




## Akka Monitoring and Telemetry

*The Industry's Most Advanced And Resource Efficient Telemetry For Reactive Systems*

- Traditional monolithic systems, unlike Reactive systems, operate within a tightly coupled stack, which makes them straightforward to trace, visualize, and troubleshoot.
- Reactive systems, however, are distributed across various boundaries (i.e. microservices, DBs, clusters and data centers), which presents a whole new challenge to effective monitoring and troubleshooting; traditional APM technologies were created to monitor traditional monolithic architectures and are not set up to monitor Reactive systems.
- Monitoring and Telemetry from Lightbend provide users with a deep, detailed view into the health, availability and performance of their Akka, Play, Lagom and Scala-based distributed, async applications, and allows users to reduce costs by customizing what they track to avoid clogging systems with huge amounts of unnecessary data.

## Akka Split Brain Resolver

*The Intelligent & Automated Way To Resolve Network Partitions Fast*

- In distributed systems, network partitions are unavoidable and occur regularly; the risk is that this often results in data inconsistencies that can affect other system components and end users.
- While eliminating them completely is not possible, it IS possible to resolve them quickly without contaminating data.
- Akka Split Brain Resolver  provides various predefined strategies for mitigating the risk of data inconsistencies by quickly recovering unreachable nodes in an Akka Cluster. Using multiple pre-defined resolution strategies for different use cases, requirements, and system architectures, it intelligently automates the recovery of failed nodes–**in about 1 minute for a 1000-node cluster**.

## Akka Thread Starvation Detector

*The Fast Way To Identify Bottlenecks In Your Legacy Synchronous/Blocking Technologies*

- In most cases, enterprises do not have the luxury of using only fully Reactive technologies; legacy tools that are synchronous and blocking (i.e. databases like JDBC and Oracle DB) are bound to remain in place for some time for different reasons.
- When using these system with Akka, which is asynchronous and nonblocking, it's necessary to find out where blocked messages outside of Akka are building up, as this will slow down system performance and eventually cascade into failure.
- Akka Thread Starvation Detector identifies blocked threads in synchronous and blocking technologies that connect to Akka applications, and provides information to development & DevOps teams to track it down.

## Akka Configuration Checker

*The Automated Way To Leverage Akka Configuration Best Practices For Optimal Performance*

- In order to remain flexible and customizable for various system environments, Akka comes with a lot of configuration settings that can be altered.
- It's difficult to know what and how to tweak configuration settings, and when to utilize "power user" settings. Worse still, it's risky when something that was misconfigured backfires, resulting in terrible stability and performance.
- Akka Configuration Checker is a developer productivity tool that performs in-depth reviews of various Akka configuration settings and provides codified best practice recommendations to ensure optimal performance in production.

## Akka Diagnostics Recorder

*The Direct Way To Deliver Critical Diagnostic Information To Lightbend Support For Rapid Issue Resolution*

- When something goes wrong in a production system, it's vital to have easy access to the information behind the problem for rapid troubleshooting (i.e to the Lightbend support team).
- When reporting Akka issues, it can be tedious and error-prone for users to discover and collect all of the information the Akka team really needs to understand and solve their issues promptly. This makes support cases take longer and no doubt adds a level of frustration to the customer experience.
- Akka Diagnostics Recorder automatically collects and reports relevant runtime and configuration information (i.e. from Akka Configuration Checker) when an error or issue occurs, making it easy to send to the Lightbend support team for rapid assistance.

## Akka Multi-DC Tooling (Cluster and Persistence)

*Bringing The Resilience And Scalability Of Akka Cluster To Power Users Deploying To Multiple Data Centers*

- For especially large and performant distributed systems, there are benefits to operating clusters between multiple data centers to ensure responsiveness no matter what's happening.
- Until now, however, it was not possible to move an actor from one DC to another without it losing state—in addition, handling network partitions at the DC level is considerably more risky.
- To conquer this, Akka Multi-DC Persistence (on top of Akka Multi-DC Cluster) provides the capability of deploying  the same scalable and resilient Akka applications across different data centers around the globe without the need to manually wire and maintain separate infrastructure.

## Additional Resources

**Lightbend Tech Hub** - Empowering enterprises with guides and documentation on commercial tooling, sample projects, quickstarts, and more

**Akka.io** - The OSS Akka website for documentation and community involvement

**Free O'Reilly eBook** - *Designing Reactive Systems: The Role Of Actors In Distributed Architecture*, by Hugh McKee of Lightbend, Inc.

**Case Studies** - How Akka powers these 24 real-life Lightbend customers

## Modernize Your Enterprise With Lightbend

The architecture you've used to run your enterprise for the last 15 years can advance dramatically thanks to significant advances in distributed computing and the cloud. Your greenfield or modernized applications should be designed appropriately to take advantage of these advances. This will enable you to unleash your revenue, and minimize your cost of developing and maintaining software...Put another way, if you adopt Lightbend Reactive Platform or Lightbend Fast Data Platform as the core of your infrastructure, you will run more profitably.

That's where Lightbend comes in. Lightbend provides the leading JVM application development platform for building microservices and fast data applications on a message-driven runtime to deliver the dramatic benefits of multi-core and cloud computing architectures. We've helped the most admired brands around the globe modernize their enterprise at startup speed.

Lightbend

Lightbend (Twitter: **@Lightbend**) provides the leading Reactive application development platform for building distributed systems. Based on a message-driven runtime, these distributed systems, which include microservices and fast data applications, can scale effortlessly on multi-core and cloud computing architectures. Many of the most admired brands around the globe are transforming their businesses with our platform, engaging billions of users every day through software that is changing the world.

Lightbend, Inc. 625 Market Street, 10th Floor, San Francisco, CA 94105 | **www.lightbend.com**