GUIDE

Build Stateful Cloud Native Applications

Tips and Best Practices for a Reliable System Architecture

Lightbend

TOWARDS SERVERLESS 2.0

Distributed state management for cloud-native applications.

Visit **cloudstate.io** to learn more about this new addition to the Lightbend OSS Family!

Table Of Contents

CHAPTER 1

Serverless: Coming To A Town Near You

By Jonas Bonér
CTO and Co-Founder of Lightbend

At an obscure grid computing event—GlobusWorld, in 2005—then Network World editor-in-chief John Dix moderated spokespersons from Cisco, HP, IBM, Intel, Nortel, and SAP through a conversation about Grid and the **Future of the Networked Machine**. His opening remarks:

> *"By many accounts, average system utilization across organizations is 15% to 20% today, while obviously, the ideal would be around 80%. What's more, some 20% of IT budgets go to operations, marginally less than the 25% earmarked for capital investments. We've created large, underutilized, complex environments that are costly to maintain. So there is a huge need to do this better, and the prevailing thinking at this point seems to be that grid is the answer."*

The rest of the conversation that followed is an interesting time capsule of predictions from 15 years ago about what distributed computing and the cloud's impact would be on the enterprise stack, and a lot of it is accurate.

Fifteen years ago it was about resource utilization, and companies moving away from huge capital expenditures on servers and datacenters, onto models where pay-as-you-go was possible—that was what the market was demanding, and that's what shaped cloud computing as we know it today.

Now, that new consensus cry for the next computing wave of the future is *speed*—getting developers what they need faster, getting value from data faster, getting IT out of the way, and shipping software faster, all while maintaining reliability and predictability guarantees.

We're at another juncture in enterprise computing where there is a large push behind a big vision of the future, the push towards *serverless* architectures—a world where less human oversight and participation is required in operations.

But While We're Imagining This Future, How Did the Serverless Vision Get Limited to Functions?

I strongly believe in the serverless movement. In the last year there's been a lot of interesting work (for example around **Knative** project) expanding on the serverless UX to cover the whole software lifecycle, from build (source to image), to CI/CD pipelines, to deployment, to runtime management (autoscaling, scale to 0, automatic failover, etc.). However, the programming model is still mainly limited to stateless functions—the so-called Function-as-a-Service (FaaS) model—which limits the range of use-cases supported.

At the moment I'm seeing a lot of the conversation confusing FaaS with serverless. Similar to blockchain being used (mistakenly) interchangeably with Bitcoin (Bitcoin is an implementation of blockchain and not equivalent), FaaS is an *implementation of serverless*. While I think that FaaS is a great piece of technology, it is selling the promise of serverless short. Serverless is all about the UX, a UX that can address many implementations, with FaaS being the first.

But why all the fuss about functions in the first place? Functions are extremely useful as low-level building blocks for software. Functional programming is all about programming with functions—working with functions as first-class values that can be sent around, composed, and reused. It's a great abstraction to use. I would define functions as essentially simple Lego blocks, that have well-defined input and well-defined output—taking data in, processing it, and emitting new data as output. A pure function is stateless, which makes it predictable, you can trust that given a certain input it will always produce the same output.

The output of one function can become the input of the other, making it possible to string them together just like Lego blocks, composing larger pieces of functionality from small reusable parts. Individual functions are by themselves not that useful because they (should) have a limited scope, providing a single well-defined piece of functionality. But by composing them into larger functions you can build rich and powerful components or services. Scala is one example of a mainstream language with a powerful functional side (the other side being OO).

FaaS is building on the idea of composing these small well-defined pieces of functionality into larger workflows, all driven by the production and consumption of events (of data). This data-shipping architecture is great for data processing oriented use-cases where (so-called **embarrassingly parallel**) functions are composed into workflows processing data downstream and eventually producing a result to be emitted as an event to be consumed by the user or some other service or system.

But it is not a general platform for building modern real-time data-centric applications and systems. As with all targeted solutions designed to solve a narrow and specific problem well, it suffers from painful constraints and limitations when used beyond its intended scope.

One such limitation is that FaaS functions are ephemeral, stateless, and short-lived (for example, Amazon Lambda caps their lifespan to 15 minutes). This makes it problematic to build general-purpose data-centric cloud-native applications since it is simply too costly—in terms of performance, latency, and throughput—to lose the computational context (locality of reference) and being forced to load and store the state from the backend storage over and over again.

Another limitation is that functions have no direct addressability, which means that they can't communicate directly with each other using point-to-point communication but always need to resort to publish-subscribe, passing all data over some slow and expensive storage medium. A model that can work well for event-driven use-cases but yields too high latency for addressing general-purpose distributed computing problems. For a detailed discussion on this, and other limitations and problems with FaaS read the paper **"Serverless Computing: One Step Forward, Two Steps Back"** by Joe Hellerstein, et al.

Functions is a great tool that has its place in the cloud computing toolkit, but for serverless to reach the grand vision that the industry is demanding of an ops-less world while allowing us to build modern data-centric real-time applications, we can't continue to ignore the hardest problem in distributed systems: managing *state*—your data.

State Is the Hardest Part and the Most Interesting Opportunity for Serverless

In the cloud-native world of application development, I'm still seeing a strong reliance on stateless, and often synchronous, protocols and design. People embrace containers but too often hold on to old architecture, design, habits, patterns, practices, and tools—made for a world of monolithic single node systems running on top of the almighty SQL database.

The serverless movement today is very focused on the automation of the underlying infrastructure, but it has to some extent ignored the equally complicated requirements at the application layer, where the move towards fast data and event-driven stateful architectures creates all sorts of new challenges for operating systems in production.

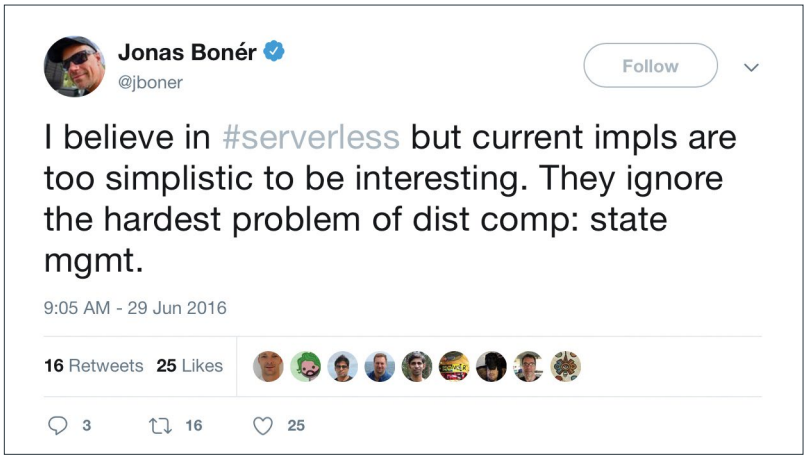
It might sound like a good idea to ignore the hardest part (the state) and push its responsibility out of the application layer—and sometimes it is. However, as applications today are becoming increasingly data-centric and data-driven, taking ownership of your data by having an efficient, performant, and reliable way of managing, processing, transforming, and enriching data close to the application itself, is more important than ever.

Many applications can't afford the round-trip to the database for each data access or storage—as we have been used to do in traditional 3-tier architectures—but need to continuously process data in close to real-time, mining knowledge from never-ending streams of data as it "flies by". Data that also often needs to be processed in a distributed way—for scalability, low-latency, and throughput—before it is ready to be stored.

This shift from "data at rest" to "data in motion" has forced many companies to fast data architectures with distributed stream processing and event-driven microservices—putting stateful state and data management at the center of application design.

This is just the starting point for the many concerns of managing state in distributed applications, and a domain that has to be conquered for the serverless movement to keep making interesting progress against its objectives to raise the abstraction level and reduce the human interaction with operations of systems in production.

Serverless Without State Is Just Not That Interesting



I tweeted that almost three years ago, and I think it's still largely true (even though a lot of progress has been made improving and expanding on the overall serverless UX).

While the 1.0 version of serverless was all about stateless functions, the 2.0 version will focus largely on state—allowing us to build general-purpose distributed applications while enjoying the UX of serverless.

If serverless is conceptually about how to remove humans from the equation and solve developers' hardest problems with reasoning about systems in production, then they need declarative APIs and high-

level abstractions with rich and easily understood semantics (beyond low-level primitives like functions) for working with never-ending streams of data, manage complex distributed data workflows, and managing distributed state in a reliable, resilient, scalable, and performant way.

Examples include support for:

- Stateful long-lived virtual addressable components[1]
- A wider range of options for coordination and communication patterns (beyond event-based pub-sub over a broker), including fine-grained sharing of state using common patterns like point-to-point, broadcast, aggregation, merging, shuffling, etc.
- Tools for managing distributed state reliably at scale—in a durable or ephemeral fashion—with options for consistency ranging from strong to eventual/causal consistency[2], and ways to physically co-locate code and data while remaining logically separate.
- Ways to physically co-locate code and data while remaining logically separate, and being able to reason about streaming pipelines and the properties and guarantees the pipeline has as a whole[3].
- Predictable performance, latency, and throughput—in startup time, communication, coordination, and durable storage/access of data.

Conclusion: The Requirements of Serverless Are Evolving from the Infrastructure, up to the Application Logic

The enterprise is broadly replatforming datacenters on containers, Kubernetes, and cloud-native frameworks in its orbit, and the adoption and momentum have been remarkable.

One question that I frequently hear is: "*Now that my application is containerized, do I still need to worry about all that hard distributed systems stuff? Won't Kubernetes solve all my problems around cloud resilience, scalability, stability, and safety?*" Unfortunately, the answer is "*No, definitely no*"—it is just not that simple.

While I believe that Kubernetes is the best way to manage and orchestrate containers in the cloud, it's not a cure-all for programming challenges at the application level, such as:

---

1 All the terms here are important, so let me clarify them. Stateful: in-memory yet durable and resilient state; Long-lived: life-cycle is not bound to a specific session, context available until explicitly destroyed; Virtual: location transparent and mobile, not bound to a physical location; Addressable: referenced through a stable address. One example of a component with these traits would be **Actors** (the Actor Model).

2 For example, disorderly programming constructs like **CRDTs** (discussed more below).

3 Properties such as backpressure, windowing, completeness vs correctness, etc.

- The underlying business logic and operational semantics of the application.
- Managing distributed application data consistency and integrity.
- Managing distributed and local workflow and communication.
- Integration with other systems.

In the now classic paper "**End-To-End Arguments In System Design**" from 1984, Saltzer, Reed, and Clark discuss the problem that many functions in the underlying infrastructure (the paper talks about communication systems) can only be completely and correctly implemented with the help of the application at the endpoints.

This is not an argument against the use of low-level infrastructure tools like Kubernetes and Istio—they clearly bring a ton of value—but a call for closer collaboration between the infrastructure and application layers in maintaining holistic correctness and safety guarantees.

End-to-end correctness, consistency, and safety mean different things for different services. It's totally dependent on the use-case, and can't be outsourced completely to the infrastructure. To **quote Pat Helland**: "*The management of uncertainty must be implemented in the business logic.*"

In other words, a holistically stable system is still the responsibility of the application, not the infrastructure tooling used—and the next generation serverless implementations need to provide programming models and a holistic UX working in concert with the underlying infrastructure maintaining these properties, without continuing to ignore the hardest, and most important problem: how to manage your data in the cloud—reliably at scale.

Coming up next...

In Chapter 2, we take a deeper look at the differences between stateful and stateless application design patterns, and why a stateful approach can often make sense for applications that are distributed, clustered, and managing real-time streaming data.

CHAPTER 2

Stateless VS Stateful Application Design

By Hugh McKee
Developer Advocate at Lightbend

How to Build Stateful Cloud-Native Applications

How to Build Stateful Cloud-Native Applications In Chapter 1, we discussed the role of state in serverless and cloud-native applications, and where the challenges and benefits lie in taking this approach. In this chapter, we get into more of the details behind what it looks like to build a stateful, cloud-native application.

For a long time, stateless services have been the primary choice for developers. One of the reasons for using a **stateless protocol** is that it provides for resiliency from failures, recovery strategies in the event of failures, and the option to scale processing capacity up and down to handle variances in traffic.

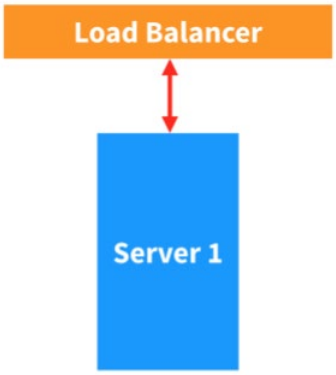


Figure 1. Stateless Protocol

A common pattern is to stand up stateless servers behind a **load balancer**, as shown above. The load balancer provides a single point of contact for clients. Clients connect to the load balancer, send requests to it, and the load balancer routes these requests to one of the available servers that are connected to the load balancer.

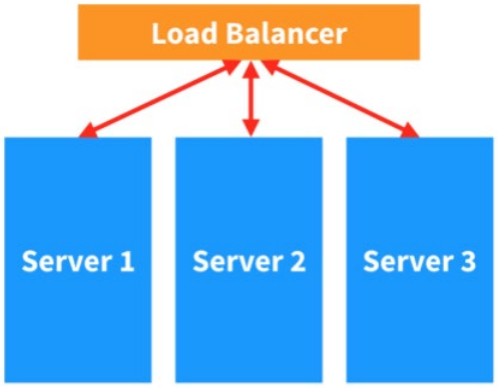


Figure 2. Stateless Protocol

When the client traffic increases more servers can be started and connected to the load balancer. The load balancer then routes traffic to each of the servers using a request distribution strategy, such as **round-robin** distribution strategy.

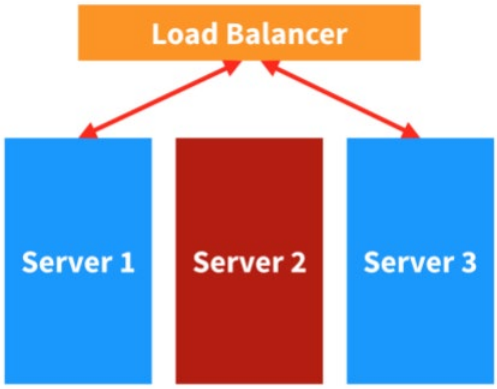


Figure 3. Stateless Protocol

When a server goes offline, as shown above, the load balancer detects this and routes traffic to the remaining online servers.

While on the surface building a stateless web app or microservice may be the right way to go, in some cases it is not necessarily the best approach for cloud-native services.

When Does It Makes Sense to Go Stateless?

Wikipedia defines stateless as:

> *"In computing, a stateless protocol is a communications protocol in which no session information is retained by the receiver, usually a server. Relevant session data is sent to the receiver by the client in such a way that every packet of information transferred can be understood in isolation, without context information from previous packets in the session. This property of stateless protocols makes them ideal in high volume applications, increasing performance by removing server load caused by retention of session information.*

There are two key advantages when building stateless systems. First, the programming complexity is reduced. Incoming requests are received, processed, and forgotten. Second, there is no need to maintain state, and often the complexity revolves around maintaining session state, which typically involves replicating the session state across the cluster. This replication approach is used to maintain the session state when one of the servers goes offline.

In the last sentence of the Wikipedia definition of a stateless protocol, it states - "This property of stateless protocols makes them ideal in high volume applications, increasing performance by removing server load caused by retention of session information." While it is true that the stateless approach does not have the overhead of maintaining session state it does introduce processing patterns that have their own overhead and performance costs.

The term stateless is somewhat misleading. Applications by their very nature deal with the state of things that is what they do, they create, read, update, and delete stateful items. The typical processing flow of a stateless process is to receive a request, retrieve the state from a persistence store, such as a relational database, make the requested state changes, store the changed state back into the persistence stores, and then forget that anything happened.

While there may be reductions in overhead related to not maintaining session state on the servers there may be costs associated with delegating state management outside of the application, such as delegating the sole responsibility for state management to the persistence layer.

This cost often is seen when the persistence layer slows down due to high contention. It may be true that it is possible to scale processing capacity at the application layer with the ability to increase or decrease the number of stateless servers, it is also true that the persistence layer does not have unlimited processing capacity. Once the persistence processing capacity is exceeded the application often cannot go any faster.

It is also important to understand that the decision to use the stateless approach has contributed to the persistence capacity limits by delegating state management from the application layer to the persistence layer.

When to Use a Stateful Approach?

For many developers and architects working with cloud-native applications, our intuition tells us that on the surface a stateful approach has advantages. The most obvious benefit is the potential for a reduction in the overhead associated with retrieving state on every request. However, our intuition also tells us that maintaining state has an associated cost with the potential for increased complexity.

Often, however, this perception of increased complexity is because we are looking at the problem from the perspective of our current way of doing things. That is our current approaches for maintaining state across a cluster and our current relational CRUD based ways for handling persistence.

There are stateful alternatives. Here we will look at two of the stateful alternatives:

- Clustered state management
- Event based state persistence

These two stateful alternatives share an events first way of processing and persisting state changes.

First, we will look at the stateful persistence approach. As mentioned this approach is focused on persisting events. Using the classic shopping cart scenario, each change to the state of a shopping cart is persisted as a sequence of events.

User	ID	Time	Event
You	64	08:11	Add Item 1567
Me	17	08:15	Add Item 3254
Other	76	08:16	Add Item 8359
You	64	08:20	Add Item 2632
Me	17	08:20	Add Item 4983
Other	76	08:24	Change Item 8359
You	64	08:25	Remove Item 1567
You	64	08:26	Add shipping address
Other	76	08:30	Add Item 2438
Me	17	08:33	Add shipping address
Me	17	08:33	Add billing address
You	64	08:35	Add billing address

Figure 4. Persisted Events

Shown in Figure 4 is a series of shopping cart state change events. This is an example of an event log. The events are persisted to an event log stored in a database as each event occurs.

Events are statements of fact, a log of things that happened at some point in the past, a historical record. In the above event log, the events aggregate to show that your shopping cart contains one item, item 1567 and a shipping and billing address. My shopping cart contains two items along with shipping and billing addresses. Finally, there is the other shopping cart that contains two items.

Also, note that the event log has recorded various shopping cart changes. For example, you removed an item from your cart. The other user changed one of the cart items. This is an example of types of historical data that is typically lost when using the traditional CRUD based persistence approach.

At any point in time, it is possible to determine the state of a shopping cart by replaying the events up until that time. Of course, this makes it possible to recover the current state of any shopping cart at the current time. It also makes it possible to view the state for a cart at a time in the past. For example, your cart contained two items at 08:20.

One of the advantages of persisting data using events is that it is now possible to record all of the interesting events that happened over time that resulted in the current state of each shopping cart. This event data, such as removing or changing items, can be extremely interesting for downstream data analytics.

Another event log advantage is that the persistence data structure is a simple key and value pair. In this example case, the key is user Id, item Id, and time and the value is the event data. The event log is also idempotent; events are insert only; there are no updates and no deletes. The insert only approach reduces the load and contention of the persistence layer.

Next, we will look at clustered state management for cloud-native applications.

Managing State in Clusters

In the case of a clustered environment, the state of each shopping cart is managed by an actor that is based on the **actor model**. In the shopping cart example, there is a unique actor instance that is responsible for handling cart state changes. Going with the above example there are three actors in use, one to handle your shopping cart, another actor to handle my shopping cart, and a third actor to handle the other shopping cart. The example actor system diagram also shows many other shopping cart actors.

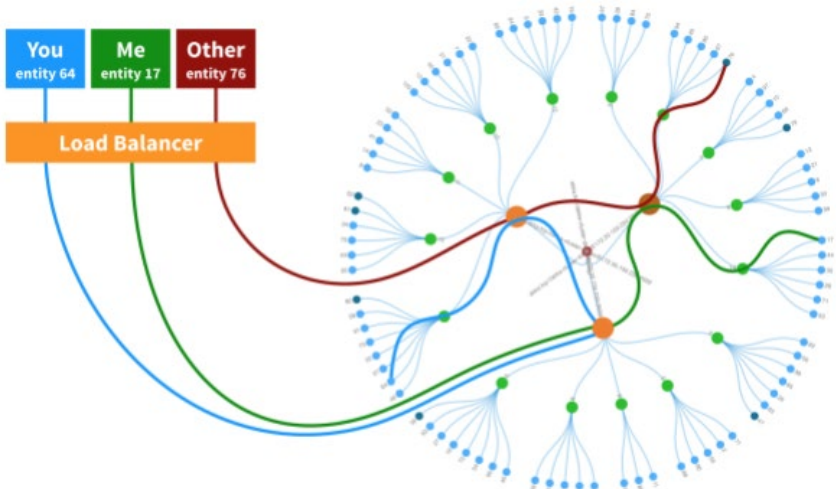


Figure 5. Shopping Cart Actor System

Shown in Figure 4 is a series of shopping cart state change events. This is an example of an event log. The events are persisted to an event log stored in a database as each event occurs.

Shown in the above diagram is an actor-based, clustered shopping cart system. This is also an example of a shopping cart microservice. On the left are three clients, you, me, and the other. Each of us is building our shopping carts vis some web based device. Access to the system is handled via a load balancer. The load balancer routes incoming requests to service endpoints.

The “crop circle” on the right is a visual depiction of the more interesting actors in the shopping cart system. The blue leaf circles on the perimeter of the diagram represent stateful shopping cart actors. The green circles that connect to the outer leaf circles represent shards. The shards are used to distribute the shopping cart actors across a cluster. Note that in this example there are 15 shards.

The shards connect up to what is called a shard region actor. There is one shard region actor per server node in the cluster. So in the above diagram, this cluster is currently composed of three serves and each server hosts a shard region actor.

The lines connecting the web clients to the actors provide an example of how incoming client requests are routed to each shopping cart actor instance. As an example, your client request was routed by the load balancer to the HTTP endpoint hosted on the bottom server circle.

From there the HTTP request is used to create a message that contains your request, such as add an item to the shopping cart actor. Initially, the message is sent to the local shard region actor. It is the responsibility of the shard region actor and other sharding related actor (see **Akka Cluster Sharding** for more details) to determine how the message should be routed. In the above example, your shopping cart actor is located on another server. The shard region actor will forward the message across the network to the shard region actor on the other server. This second shard region actor will then forward the message to the shard actor that maps to your shopping cart Id. The shard actor then forwards the message to your shopping cart actor.

When a shopping cart actor receives a message, it first validates that the message is ok to process. Validation is use case specific, for example, when a request is received to add an item to a shopping cart, the validation might be a verification that the item is a valid catalog item and it is currently available. If the request is valid the next step is an event is created, and it is then persisted to the event log.

Once the event has been inserted into the event log the shopping cart actor then updates the state of the shopping cart. Again, each shopping cart actor's state is the current state of the specific shopping cart.

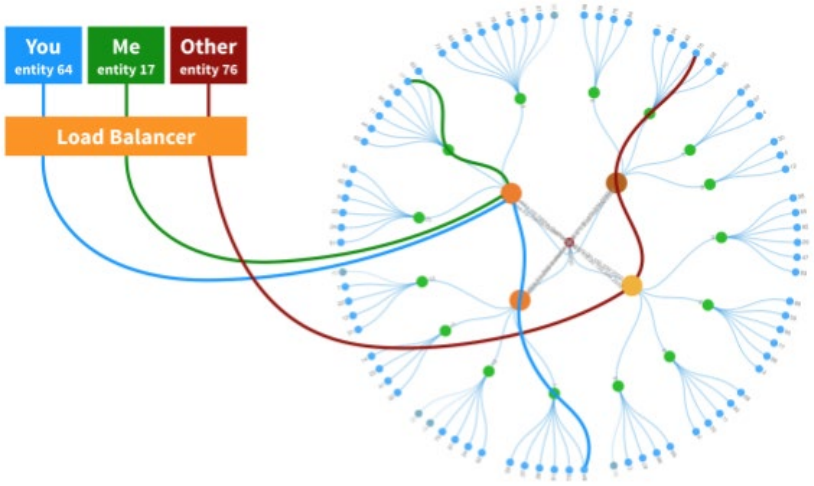


Figure 6. Shopping Cart Actor System

As a second example, Figure 6 shows a fourth server was added to the cluster. Note that there are still 15 green shard actors and that some of the shard actors have moved to the new fourth server. This is an example of how the Akka Cluster sharding is capable of recognizing that the cluster has changed and how it is designed to distribute work across the cluster as it scales up and down.

Following along with the example with your shopping cart. Say you are adding the next item to the cart. In this case, the client HTTP request lands on another server, not the same server that received the first request. Also, your shopping cart actor has moved to the new server that just joined the cluster.

When a shard is moved to another server, the old shard actor and its associated shopping cart actors are stopped on the old server. Then on the new server new instances of the actors are started. Shopping

cart actors are started as needed. That is they are only started when a message is sent to the actor in this case of your shopping cart actor. The old instance was stopped.

When you submitted the request to add the second item the message was routed to the shard actor that maps to your shopping cart Id. The shard actor checks and sees that your shopping cart actor is not running. The shard actor starts your actor. When the shopping cart actor is started it first recovers its state by reading and re-playing all of the persisted events. In this scenario, one item was previously added to the cart. Once the shopping cart state is recovered, then the incoming message is forwarded to the actor.

In this scenario, the actor system is doing most of the heavy lifting for us. In contrast, the application code is relatively simple. At the HTTP endpoint, custom code is used to handle incoming requests, transform them to message objects, and then send the messages to the local shard region actor. On the other end of the message flow custom code is created to handle incoming messages, validate, persist, and then update the state of the shopping cart. Custom code is also required to recover existing events when each instance of a shopping cart actor is restarted. In between, the actor system handles all of the routing and messaging both within each JVM and across the network, from private cloud to hybrid to on-premise.

Stateless for "Good Enough"; Stateful for Real-Time Streaming Data

As usual, when it comes to cloud-native software systems, determining the best approach depends on the specific circumstances. This certainly applies when considering stateless versus stateful systems. In many cases, the stateless approach is an acceptable solution; however, there is a growing number of scenarios where using a stateful approach will be a better alternative. This is undoubtedly true for the ever-increasing demand for high-performance near real-time and stream based systems.

CHAPTER 3

The State Of Distributed State

By Sean Walsh
Field CTO at Lightbend

In Chapter 3, we look deeper at distributed state, CRUD vs Event Sourcing, and how Domain Driven Design (DDD) and an events-first approach to distributed, cloud-native and real-time streaming systems can be helpful.

Some Quick Words On Distributed State

In Chapter 2, we looked the details behind what it looks like to build a stateful, cloud-native application from a high-level perspective. In this part, we take a look at application state itself, and why we should do everything we can to successfully distribute it. Let's start with defining the word *state*.

Merriam-Webster defines it as "a condition or stage in the physical being of something". Using the insect kingdom as an example, there would be an egg, larva, pupa, and adult. Virtually all things have state, and that is also true of computing. State is useful in order to make decisions or take actions. An insect in the egg state would have very different behavior than one in the adult state; in the egg state, the insect can

either die or metamorphosize into a larva, while the adult can only die. This is just one example of what I'll call *contextual-state*, where application state, as with beauty, is also in the eye of the beholder. What is application state?

Just as in the insect example, objects in your application also have current state and behavior over time. For example, the accumulated movements of an electric car might result in a geographic position of 40.712776 latitude and -74.005974 longitude at 12 PM, on Monday, March 11, 2019. That snapshot of the object's state at this point is very important when it is desired to move the car in a certain direction; yet once that decision is made, it is then reflected in a new state of the vehicle.

What Is Subjective State?

State needs to exist within a certain context, for example within a *bounded-context* if we are to consider **domain-driven-design** (DDD).

Let's take for example an airport model and look at two bounded-contexts: ground control and departures. Both of these contexts have very different interests, ground control moves planes and other vehicles around the tarmac, while departures are interested in getting a plane flown safely out of the local airspace.

The **SOA** architect might see the aircraft as the common item and attempt to model it centrally to be shared among these two contexts; however, as you can imagine, this just leads to a functional bottleneck for no good reason. In fact, with these two example contexts, most likely the only shared attribute would be the aircraft's identity or callsign. **All the other attributes would be of unique interest to the given context**. For example, ground control would not care about altitude, destination or passenger manifest, while departures probably would.

We can introduce the term s*ubjective-state* for cases when you have a view on something that is specific to your own needs, or even at a certain time granularity or summarization. Any attempt to share state across bounded contexts flies in the face of DDD. State should be kept private and both **highly available** and **highly accurate** in order for the domain to make quick and accurate decisions. We'll explain how to accomplish the lofty goals of availability and accuracy a bit later.

What About Events?

Any discussion of application state will likely include the topic of events and **event-driven architecture** (EDA). Now that we've established that–in most cases–the proper use of state makes it meaningless to share with the outside world, what do we do?

Events just happen to already exist in your business, whether you are capitalizing on them or not. The world is made up of events and so are our systems. If you know enough about your business you can lock yourself in a room and capture all of your business events on sticky notes, this is called *event storming*.

Once you capture all of the events it is a matter of grouping them according to the bounded context in which they belong and directing your development teams to have at it, roughly translating a bound context to an individual set of microservices to be developed.

This is where *event-sourcing*, an alternative to the *CRUD* (create/read/update/delete) model of storing current state, comes into play. **Event sourcing** makes use of immutable facts that are stored, in order, in a durable event log, representing the history of the state of the domain over time. State is then relegated to any observer of "order" in the system, each having its own unique pivot on the data. For example, *OrderCreated*, *OrderShipped* or *OrderBilled* are different event states previously visible with CRUD only as a snapshot; with event sourcing, this is now derived by taking into account all of the events in the order they occurred that lead to that state.

Where CRUD vs Event Sourcing Matters

Event sourcing is a mature concept and has been around for decades for the likes of banking and telemetry-based applications, such as within the energy or manufacturing industries. Martin Fowler was writing about it as early as 2005 **here**. Event sourcing just so happens to map perfectly into our "sharing stuff" challenge. The world is moving to event sourcing more now than ever, due to the growth of streaming analytics and systems of microservices.

The following example shows event sourcing in use in an orders system and what it looks like compared to the CRUD (create/read/update/delete) current state model we are so used to building.

CURRENT STATE STORAGE MODEL. I.E. C R U D

order id	account id	billing adress	shipping address	date	order total
1234	9876	1 Elm St…	1 Elm St…	2/15/19	$4000.00
2345	8765	20 Oak St…	20 Oak St…	2/01/19	$2500.00
3456	7654	300 Maple ..	300 Maple..	1/15/19	$1000.00
…	…	…	…	…	…

EVENT SOURCING STORAGE MODEL

event id	event type	entity id	event payload	timestamp
9a5ddc26-6ffd-4c92-8fba-921ba3d42dc6	OrderCreated	1234	{"accountId": 9876, …}	2019-02-15T15:53:00
f29b3a2e-7d11-4118-b616-90e0f2d3b93c	OrderShipped	1234	{"shippedBy":"Ralph", …}	2019-02-16T12:02:00
…	…	…	….	…

Current State vs Events

You'll notice in the example above that the event-sourced domain storage is made up of things that have occurred upon that piece of the domain, the order, rather than just the current state of the order. In one fell swoop, we have gained insight into behavior over time and utilize concrete events, understood by the business to be our system of record.

This is rather an arbitrary use of a current state model. The domain storage is also vastly simplified and uniform across domains since all events look the same, some id, event-type, JSON or other payload and a timestamp. No compound keys, or guesswork, and there is another great benefit too: these arbitrary event IDs can be used to uniformly shard the data across clustered storage, or uniformly shard your domain across your application cluster! Much more on this later.

So behavior determines state, but state also affects behavior. An injured soccer player is not someone who kicks a ball or throws it, but an injured soccer (football for our non-US readers) player is really a patient. When the soccer player is no longer injured he can become available for play and join a game and so it goes. Going back to the aircraft example, behavior determining state would be an airplane with its doors open and still available for booking. When the doors close, no further bookings are available and the airliners take on the states of leaving the gate, entering the air, etc. While in those different states, the airliner has different characteristics, such as in the example of the injured soccer player.

On Distributed State, or Better Yet, Distributed Domain

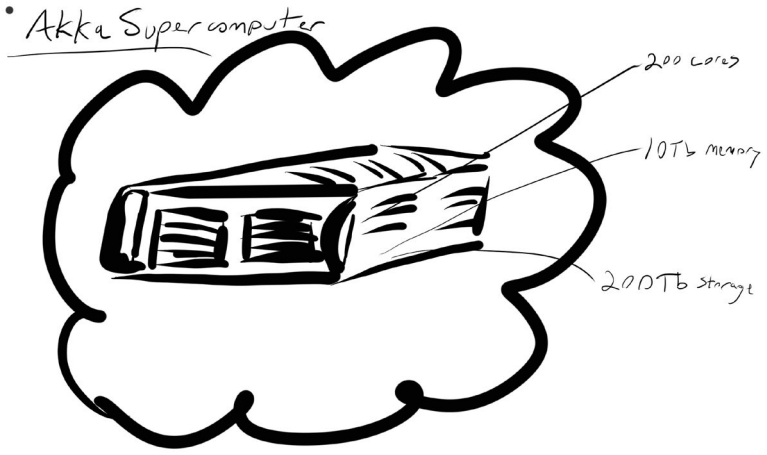
The term *distributed state* really addresses what I'd consider the old way of dealing with scale on an N-tier *stateless architecture*. Why did I just type that? Didn't I already mention that pretty much all applications have state?

What happens in a stateless architecture is that the design is bound by the lack of a properly clustered application (or business) tier. In the absence of a clustered application tier, we're forced to distributed our state, using the only thing available to us and capable of such things–our database. Now we've removed the ability of the domain to make real-time decisions. Since the state is not private, nor held by the domain model the application nodes are forced to query the state from the database before making a decision, and finally writing the new state back to the database.

While this is happening you squint your eyes real tight and look away, hoping that another "stateless" node isn't making the same sort of decision at the same point in time, duplicating the data writes and polluting your data. We see this happen all the time, and even to people who understand the problem well enough to fix it will soon learn that the only answer is to utilize multiple-keyed collections in your database to make it impossible to write the same thing twice.

This however now leaks domain behavior back into your data store, just like the old days of triggers and stored procedures.

So what do we do? Luckily we no longer live in the old days and distributed, clustered, cloud-native application software does exist in the form of **Akka**. **Akka harnesses your entire cloud and presents it as a single, virtual supercomputer–and a very resilient one at that.**



For the first time, our entire application can fit in memory as needed. We no longer need to be "request bound". A single request can trigger any number of asynchronous behaviors across your microservices, which lets developers avoid thinking about doing a chunk of work and then offloading to a message bus so another node can help do the work.

Akka uses the tried and true **actor model** for addressable application components across your cloud. Akka also provides an *illusion of now*, due to the actor interface being a mailbox. While the actor is processing a message from its mailbox, it will not be interrupted until that message has been fully processed, and its internal state has been changed as a result.

This illusion of now also exists in domain-driven design in the form of an *aggregate root*. An aggregate root represents an encapsulation of behavior with strongly consistent data within. This maps neatly into Akka actors, so much so that Akka has the concept of the *persistent entity*. The persistent entity maintains its state privately and makes decisions upon that state. These decisions take the form of events, which are written to the event store database of your choice. Since the events fully tell the story of that piece of the domain, such as an order, it is a simple matter of instantiating the persistent entity actor on a healthy node of the cluster and replaying the events and derive current state and then being ready to do business.

Akka Actors are resilient in that they may be kept in memory indefinitely or set to time-out after periods of nonuse. They are uniformly sharded across your application cluster and moved to different nodes as needed. These persistent, sharded actors are singletons, meaning there will ever only be one instantiated somewhere. This even goes for a failed network where a **network partition (aka split brain scenario)** occurs. With Akka at its core, Lightbend Platform will use a chosen strategy to decide which side of the partitions cluster is the "good side" and which nodes to shut down, guarding against data contamination that would result in multiple singletons existing at the same time.

This singleton safety also applies to **clustering and persistence across multiple data centers** as the events happen to be a convenient thing to share for the purposes of failover, redundancy or geo-servicing of your users.

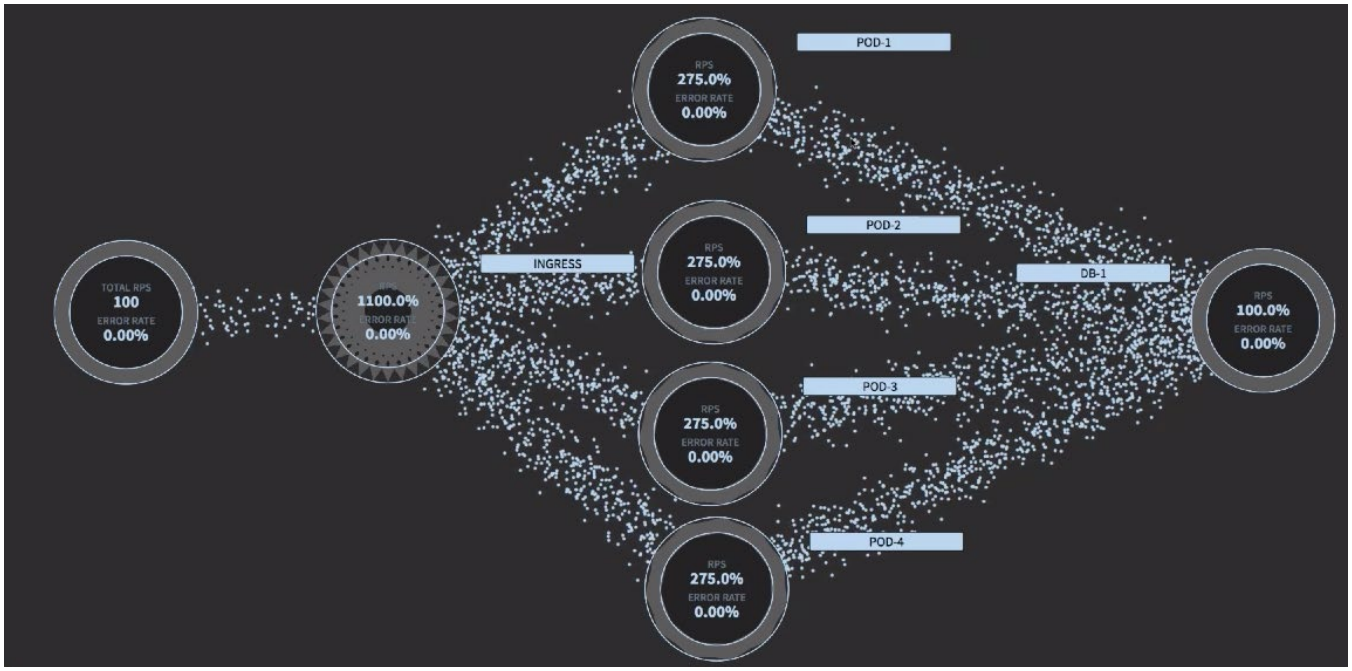
The same guarantee exists to enforce the single writer principle, where, though hot replicas exist across the data centers, only a single persistent actor is ever effectively doing business at any given time, achieved by applying conflicting update resolution. Read more about that **here**.

Availability and Accuracy

Now that we have an idea of how distributed, persistent entities work, it's not a stretch to figure out how we accomplish availability and accuracy. Entities in a well-tuned cluster will have the utmost availability since they are already in memory (this is tunable), and are self-sufficient in that they do not need to rely on real-time database reads to make decisions.

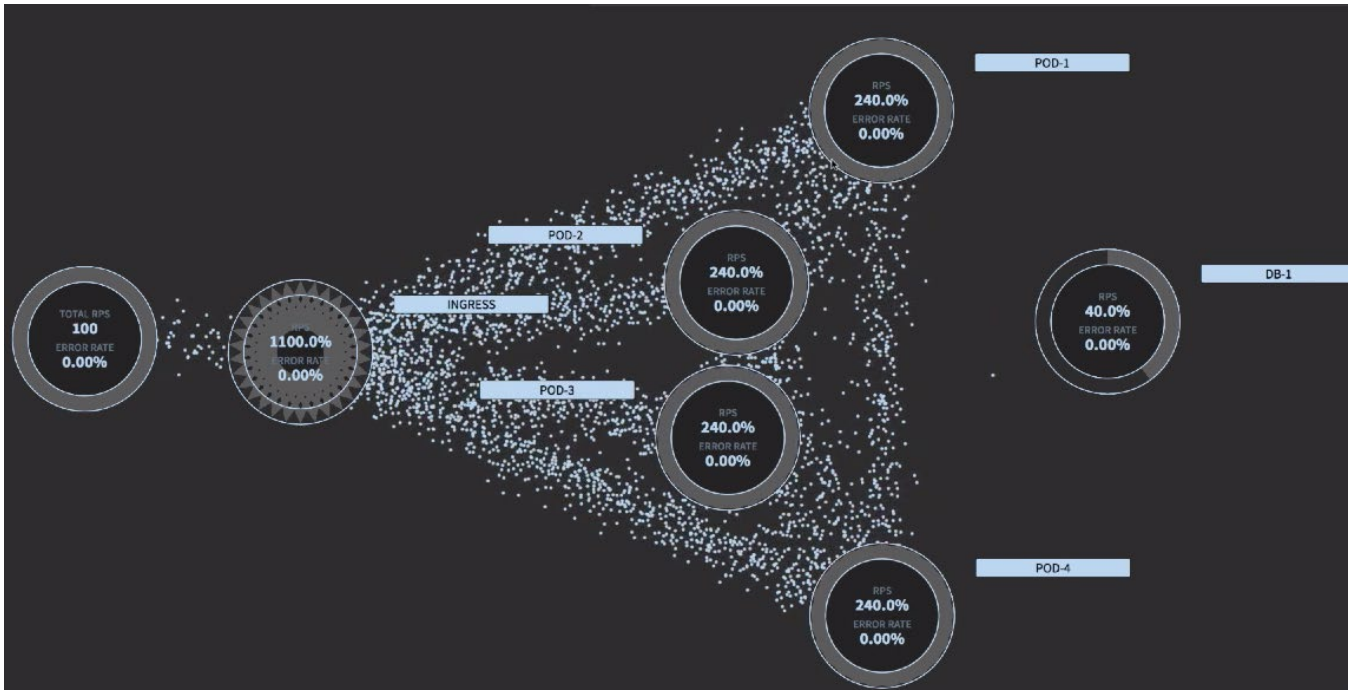
Typical latency for communications between actors in the cluster is in microseconds (µs). In addition to this awesome availability, we also have unparalleled accuracy, since the entities are the keepers of their own state there is only one single place where decisions are being made and events being issued for any particular piece of the domain, such as an order.

As a byproduct of all this, we end up with a cloud-native architecture with significantly less noise; and by that I really mean data flying around between all tiers, all the time, and usually in real time. The following images show what I mean by noise, the first one is a typical "stateless" architecture, the second is a Reactive architecture, utilizing distributed state.



3 Tier Architecture

In a typical 3-tier architecture (above), we are forced to traverse all tiers at least once for every request. In most cases, there are also lateral callouts to other services as well to further add “noise”.



Reactive Architecture

In a Reactive architecture (above), state is the only thing you need to adequately do your job. Here it is properly distributed across your application tier, meaning that the load and behavior are also naturally distributed across your application cluster.

The Value of An Event-Driven, Distributed Domain with Internal State

State can mean very different things to different observers. Events are the things we should be sharing. We have shown how it is not possible to design your systems using actors to model the world without regard to limited resources, state contention or any sacrifices.

Now that we’ve seen how noisy a traditional 3-tier architecture is compared to a Reactive architecture with distributed state, we will now take a deeper look at how messaging, CQRS, and Event Sourcing work in stateful applications.

CHAPTER 4

Working With Messages, CQRS, And Event Sourcing In Stateful Applications

By Duncan DeVore
Principal Solution Architect at Lightbend

How to Do Messaging for Stateful Cloud-Native Apps

In Chapter 3, we looked at the nature of distributed state, and how a traditional 3-tier architecture stands up to a event-driven, Reactive architecture. We now turn to how messaging types, Event Sourcing, and CQRS apply to cloud-native design.

The definition of "cloud-native" we're using here means a friendly and optimized distributed computing environment. Key features of this type of environment are elasticity, resilience, and synchronous (point-to-point), asynchronous, and parallel communication. A Reactive messaging infrastructure provides loosely coupled design, which is paramount to distributed computing or cloud-native design. So let's look at why messages are so important.

What Are Messages?

First, let's define what a message is and the super types of the messages we will discuss. **Wikipedia has a great definition**, so we will use that - "A message is a discrete unit of communication intended by the

source for consumption by some recipient or group of recipients." I will take the liberty to add an additional piece of information to this definition: **messages in a distributed system should (or must) be immutable**. Now that we have a working definition, let's look at the types of messages we will consider.

- Command (imperative)
- Event (declarative / historical)
- Query (question)

The type of message or message pattern you choose will have a direct effect on how your application behaves.

Looking At Commands (Imperative)

Commands are imperative in nature and represent a request from "somewhere" to do "something". That "something" often results in a request to change state on "somewhere". As a result a command can be denied. In addition, a command, if valid, is often translated into one or more events. For example:

Commands In Scala

```
sealed Trait Cmd
sealed Traid Evt

final case class PickupGroceries(...) extends Cmd
final case class CarDrivenToMarket(...) extends Evt
final case class GroceriesPickedup(...) extends Evt
final case class CarDrivenHome(...) extends Evt
final case class GroceriesPutAway(...) extends Evt
...
val receiveCommand: Receive = {

    // validate and PersistAll
    case cmd: PickupGroceries => validate(cmd) fold (
      f => sender ! ErrorMessage(s"error $f occurred on $cmd"),
      s => persistAll(Set(CarDrivenToMarket, GroceriesPickedUp, ...)) { evts =>

          // side-effect after PersistAll updating internal state
          updateState(evts)
          ...
    })
}
```

Commands In Java

```
class Cmd implements Serializable;
Class Evt implements Serializable;
```

```
class PickupGroceries {...} extends Cmd
class CarDrivenToMarket {...} extends Evt
class GroceriesPickedup {...} extends Evt
class CarDrivenHome {...} extends Evt
class GroceriesPutAway {...} extends Evt
...
@Override
public Receive createReceive() {
    return receiveBuilder()
        .match(Cmd.class,c -> {

                // Create set of events based on command
                final HashSet<Evt> events = new HashSet<Evt>(
                    Arrays.asList(CarDrivenToMarket, Groc...));

                // Persist events using PersistAll
                persistAll(events, (HashSet<Evt> evts) -> {

                    // side-effect after PersistAll updating internal state
                    updateState(evts)
                    ...
                })
        })
}
```

Typically a command follows `VerbNoun` format such as `PickupGroceries` as we see above.

Looking At Events (Declarative / Historical)

Events on the other hand are declarative. They represent that "something" has occurred. They are historical in nature. In our example above we see that processing a valid `PickupGroceries` command, four events are generated, `CarDrivenToMarket`, `GroceriesPickedup`, `CarDrivenHome` and `GroceriesPutAway`. You will notice that the structure of an event is the inverse of a command; `NounVerb`. Another thing to note about an event is because of their declarative nature, events cannot be denied.

Events should be atomic in nature, and represent a form of state that has transpired against a domain aggregate.

Account Register Example

One of the best ways to understand event sourcing is to look at the canonical example, a bank account register. In a mature business model, the notion of tracking behavior is quite common. Consider,

for example, a bank accounting system. A customer can make deposits, write checks, make ATM withdrawals, transfer money to another account, etc.

Date	Comment	Change	Balance
7/1/2014	Deposit from 3300	+10,000.00	10,000.00
7/3/2014	Check 001	-4,000.00	6,000.00
7/4/2014	ATM Withdrawal	-10.00	5,990.00
7/11/2014	Check 002	-20.00	5,970.00
7/12/2014	Deposit from 3301	+2,000.00	7,970.00

In the image above, we see a typical bank account register. The account holder starts out by depositing $10,000.00 into the account. Next they write a check for $4,000.00, make an ATM withdrawal, write another check and finally make a deposit. We persist each transaction as an independent event. To calculate the balance, the delta of the current transaction is applied to the last known value. As a result, we have a verifiable audit log that can be reconciled to ensure validity. The current balance at any point can be derived by replaying all the transactions up to that point. Additionally, we have captured the real intent of how the account holder manages their finances.

Looking At Queries (Questions)

Queries are similar to commands in that they are a request for information. Queries do not change state so they usually do not result in an event(s). That said however, they can if auditing of query projections is required, or perhaps we want to log all queries that are executed. Queries will often rely on synchronous or point-to-point communication (not a requirement) where Command/Events typically use async fire-and-forget.

What Are Some Message Based Abstractions?

Event Sourcing (ES)

As we discuss elsewhere, event sourcing provides a means by which we can capture the real intent of our users. In an event sourced system, all data operations are viewed as a sequence of events that are recorded to an append-only store. This pattern can simplify tasks in complex domains by avoiding the requirement to synchronize the data model and the business domain; improve performance, scalability, and responsiveness; provide consistency for transactional data, and maintain full audit trails and history that may enable compensating actions.

A good article about using event sourcing to overcome complexities of distributed systems and **CAP Theorem**.

Command Query Responsibility Segregation (CQRS)

CQRS stands for Command Query Responsibility Segregation. It's a pattern by which we can segregate operations that read data from operations that write data by using separate interfaces. This pattern can maximize performance, scalability, and security; support evolution of the system over time through higher flexibility; and prevent writes (update commands) from causing merge conflicts at the domain level. The pattern implements two distinct paths, a Command-side(Write) and Query-side(Read). In most cases, these will be separate micro-services running on their own JVM's.

Why Consistency Is Explicit In Message Based Systems

Consistency is often taken for granted when designing traditional monolithic systems as you have tightly coupled services connected to a centralized database. These types of systems default to **Strong Consistency** as there is only one path to the data store for a given service and that path is synchronous in nature. In distributed computing, however, this is not the case. By design, distributed systems are asynchronous and loosely coupled and rely on patterns such as atomic shared memory systems and distributed data stores achieve **Availability** and **Partition Tolerance**. Therefore, strongly consistent systems are not distributable as a whole contiguous system as identified by the CAP theorem.

Systems with strong consistency can be distributed. They just need to use something to coordinate their effort - e.g. distributed atomic commit or distributed consensus. Obviously, this emphasizes the C from CAP and AP suffer a lot.

Consistency Models

In distributed computing, a system supports a given consistency model if operations follow specific rules as identified by the model. The model specifies a contractual agreement between the programmer and the system, wherein the system guarantees that if the rules are followed, memory will be consistent and the results will be predictable.

Eventual Consistency

Eventual consistency is a consistency model used in distributed computing that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. Eventual consistency is a pillar in distributed systems, often under the moniker of optimistic replication, and has origins in early mobile computing projects. A system that has achieved eventual consistency is often said to have converged, or achieved replica convergence. While stronger models, like linearizability (Strong Consistency) are trivially eventually consistent, the converse does not hold. Eventually Consistent services are often classified as as **B**asically **A**vailable **S**oft state **E**ventual consistency semantics as opposed to a more traditional **ACID** (Atomicity, Consistency, Isolation, Durability) guarantees.

Causal Consistency

Causal consistency is a stronger consistency model that ensures that the operations processes in the order expected. More precisely, partial order over operations is enforced through metadata. For example. If operation A occurs before operation B, then any data center that sees operation B must see operation A first. There are three rules that define potential causality:

1. **Thread of Execution**: If A and B are two operations in a single thread of execution, then A -> B if operation A happens before B.
2. **Reads-From**: If A is a write operation and B is a read operation that returns the value written by A, then A -> B.
3. **Transitivity**: For operations A, B, and C, if A -> B and B -> C, then A -> C. Thus the causal relationship between operations is the transitive closure of the first two rules.

Causal consistency is stronger than eventual consistency as it ensures that these operations appear in order. Currently, Akka Persistence does not have an out-of-the-box implementation of causal consistency, so the burden of the programmer to implement. The most common way to implement causal consistency in an Akka based actor model is through **Become/Unbecome & Stash**.

Events-First and Message-Driven Is Ideal for Stateful Cloud-Native Apps

Message-driven architectures are well suited for stateful, cloud-native requirements, and should utilize different types of messages or message patterns (Commands, Events, Queries), as well as message based abstractions such as Event Sourcing and CQRS. Because messaging semantics is of such importance, in our next chapter we dig into how message delivery and reliability for stateful applications can occur—from *at-least-once* to *at-most-once* and *exactly-once* delivery.

CHAPTER 5

Message Delivery & Reliability For Stateful Applications

By Hugh McKee
Developer Advocate at Lightbend

‘At Least Once’, ‘At Most Once’ and ‘Exactly Once’ in Cloud-Native Systems

In Chapter 4, we reviewed the concept of event-driven messages as being well-suited to stateful, cloud-native application requirements. Here, the type and intent of messages really matters to overall system functionality–so let’s now take a look at message delivery, including some basic review of RPC, REST, and the Actor model for reliable, distributed messaging at scale.

Message Delivery Basics

As software developers, we use our selected programming languages to run inline snippets of code that perform a series of computational steps that are interspersed with invocations of other modules. In object-oriented languages, objects invoke other methods in other objects. In functional languages, functions invoke other functions. In a way, this is a form of messaging: invoke a function, pass it a message via parameters, the function processes the request, and it returns a response message via the return values.

When we develop software for distributed systems, it would be desirable to continue to use method or function calls that reach across the network divide as simply as we make local calls. However, after decades of experience and numerous attempts to implement **remote procedure calls** we have learned that **the dynamics of RPCs are considerably different from in-process method calls**.

There is a massive difference between in-process procedure calls and remote procedure calls. With in-process method calls, everything happens within a single process on a single machine. An individual process is running, or it is not running. The procedure caller and the callee live together in an atomic processing universe. On the other hand, everything changes when the caller and the callee live in different processes connected across a distributed network.

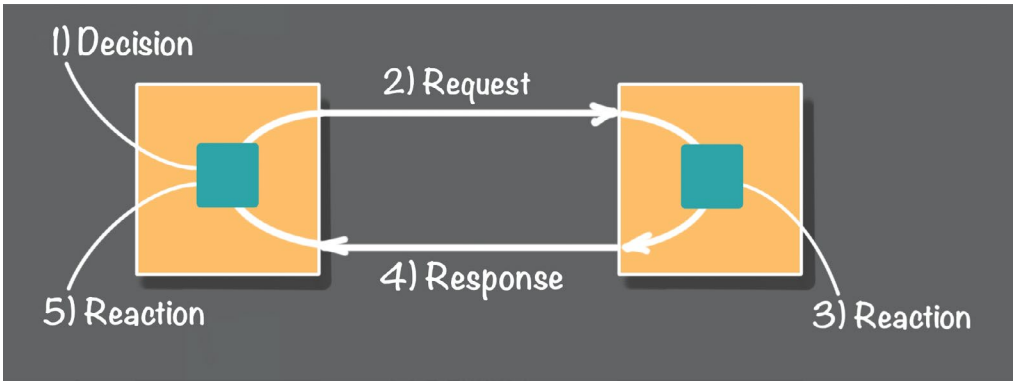


Figure 1. Request/response flow

Looking at Figure 1 above, let's walk through an example to illustrate the distributed process. Starting with the process on the left, in Step 1 we need to ask the process on the right to perform an action. Next, in step 2, the left process sends a network request to the process in the right. In step 3, the process on the right receives the request, and it performs the requested action. Upon completing the requested action in step 4, the right side process returns a response. Finally, in step 5, the left side process receives the response.

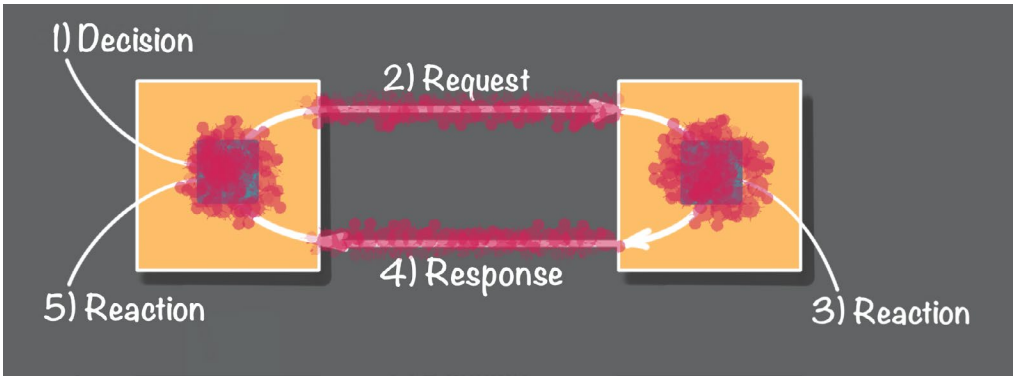


Figure 2. Request/response failure scenarios

While this may look like a method call, the difference is there are a lot of more things that will break (see Figure 2 above) when the caller and the callee are split across a network connection that can fail. The first problem is in step 2. Network issues may prevent the caller from sending requests to the callee. Next, the callee may be offline. Another failure scenario is that the callee received a request, processed it, but the callee is unable to return a response due to a network issue. Or in step 5 the request was sent, processed, but the caller is offline, which prevents the callee from returning a response.

Message Delivery Reliability

So how do we get around these reliability challenges that are ever-present in distributed networks? When you implement various forms of messaging, it helps to categorize your implementation into one of three categories. The following category definitions are copies from the Akka **Message Delivery Reliability** documentation.

- **at-most-once *(maybe once)*** delivery means that for each message handed to the mechanism, that message is delivered once or not at all; in more casual terms it means that *messages may be lost*.
- **at-least-once *(once or duplicates)*** delivery means that for each message handed to the mechanism potentially multiple attempts are made at delivering it, such that at least one succeeds; again, in more casual terms this means that *messages may be duplicated, but not lost*.
- **exactly-once *(essentially once)*** delivery means that for each message handed to the mechanism exactly one delivery is made to the recipient; *messages can neither be lost nor duplicated*. *Note: This approach is complicated, but under certain circumstances it is possible to implement.

With these 3 categories you can identify if the messaging approach you are considering meets your requirements. Let's look at an example scenario to see how this works.

Say you are building a system of microservices. You have two services that interact with each other. In this case when service A processes a request, that triggers the need to notify service B that this has happened. For example, let's say that service A is an order service and service B is a shipping service.

What about using REST? Service A makes RESTful requests to service B. Which of the three categories is this? The RESTful approach is at-most-once, which means sometimes some messages will not be delivered. In this scenario, an at-most-once messaging implementation is not acceptable.

What about using a service bus? With a service bus, such as **Apache Kafka** or **Apache Pulsar**, now service A is a message publisher, and service B is a consumer. What category does this fall into? It looks like it provides at-least-once delivery, as once a message is published it will be delivered to the consumers; however, you need to look for any cracks in the implementation that message may fall through. For example, say service A receives a request, processes it and commits the order to a database,

and then it makes a call to publish a message to the bus. Do you see the crack in this process? The crack is in between the completion of the DB transaction and the call to publish. Messages will be lost when failures occur in between these two steps.

These are solvable problems, but you need to be aware of the consequences of your implementation choices. In the next section, we look at methods for eliminating cracks in message flows.

Circuit Breakers, Retry Loops, and Event Sourcing & CQRS

Let's look at some of the techniques for eliminating failures and some approaches for gracefully handling broken message flows.

Circuit Breakers

When messages cannot be delivered due to failures, the message sender may hang for a relatively long time until the request times out, or in some cases the caller may hang indefinitely. T**his type of problem often causes a log jam on hung requests**. Worse, these log jams may overwhelm the calling process; for example, the calling process fails due to an out-of-memory error. The Circuit Breaker pattern is the most common solution used for handling this type of problem.

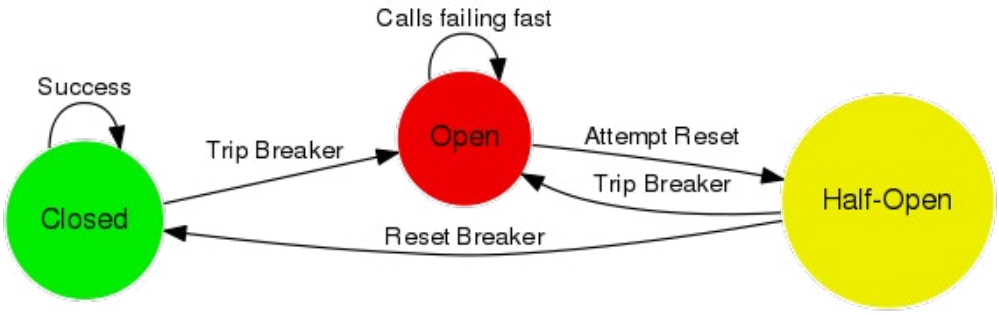


Figure 3. Circuit Breaker States

The general idea is a circuit breaker sits i**n between the message sender and the network interface**. When a specified number of failures occur, the circuit breaker will move to an open state. In keeping with the taxonomy of actual circuit breakers, an open circuit breaker means that it's closed to incoming messages, ironically. So when a circuit breaker is open, all attempts to send more messages are immediately rejected. Also while in the open state, the circuit breaker attempts to occasionally to send a message; this is known as "half-open" state. If the message fails, the circuit breaker goes back into an open state. If the message is successfully delivered the circuit breaker goes back into a closed state, allowing regular message traffic once again.

For more details on this, see the **Akka documentation** or **Martin Fowler**, or take a look at the many other sources available on this topic.

Retry Loops

When the requirement is that for a given message sender all messages must be delivered to a message receiver, it is necessary that the message delivery process uses either an *at-least-once* or *exactly-once* approach. The challenge is that for the message sender the process of sending a message is at least a two-step operation. The first step is to complete a requested action, such as a database transaction. The second step is to attempt to send a message. The fun part is when the second step, delivering the message, cannot be completed.

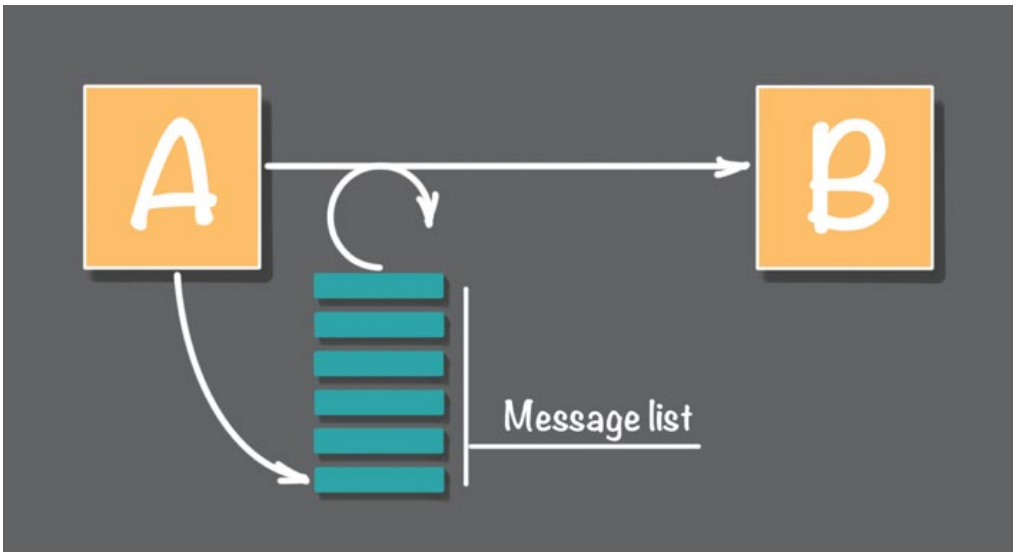


Figure 4. Retry Loop

One approach that is used to ensure that all messages are delivered at-least-once is to use a retry loop (Figure 4 above). **With the retry loop, the message sender will no give up until each message is delivered**. When a message delivery attempt fails, the sender will try again. The sender will keep trying for a limited time and then give up, or it will keep trying indefinitely.

The fun part is that the retry process needs to be durable. That is, the retry process needs to be able to recover from a failure of the sending process. When the sending process fails and is restarted, to satisfy the at-least-once requirement the sending process needs to be able to recover the pending messages when the process is restarted after a failure.

Event Sourcing & CQRS

As always, our friends **Event Sourcing** (ES) and **Command Query Responsibility Segregation** (CQRS) make an appearance in this section as well. Here we are only going to focus on how ES & CQRS is used to facilitate at-least-once message delivery.

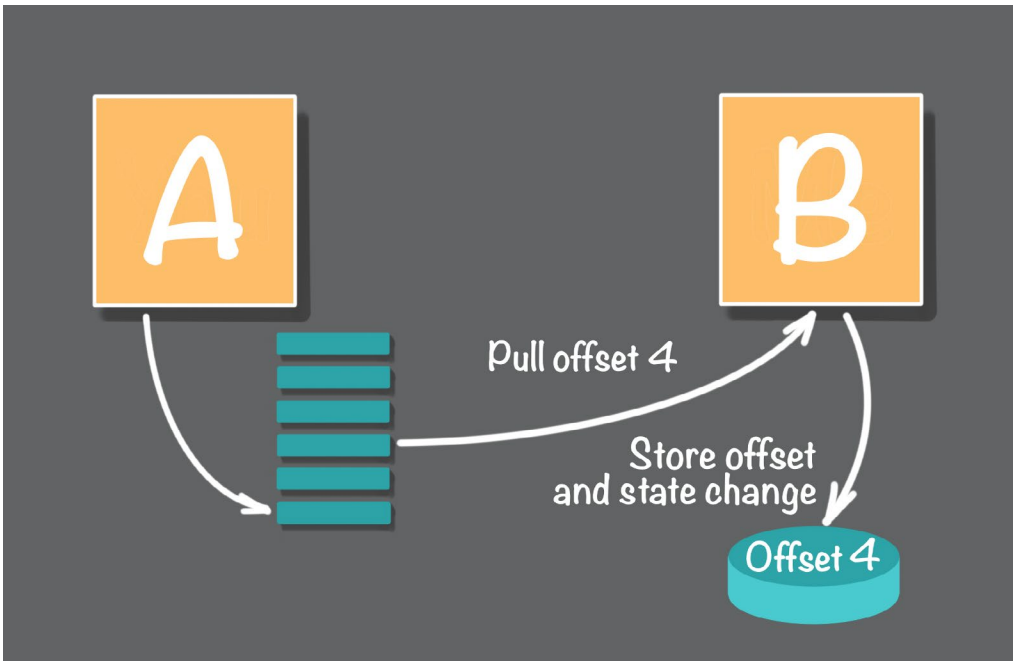


Figure 5. ES & CQRS

Event Sourcing uses an event store, which is an ideal persistent store for pending messages. One approach used is the message sending process is watching the event store (Figure 5 above). As events are written to the event store, the sending process reads these events and attempts to send the associated message. As messages are successfully sent, the sending process advances to the next event. The sending process needs to persist the current offset to the next event to be sent. **With the persisted offset, the sending process is failure resistant.**

Message Delivery with Akka, Play, and Lagom

Message-driven systems is one of the fundamental characteristics of Reactive Systems as defined in the **Reactive Manifesto**. The evolution of Reactive concepts were influenced by the evolution and the development of the **Akka toolkit** and **Play Framework**. **Lagom Framework**, which is a microservice framework built on top of Akka and Play, came along later. Each of these three Lightbend technologies share a message-driven heritage.

Akka

One of the many features of Akka is that it provides an implementation of the actor model. Actors are powerful building blocks based on a simple asynchronous messaging protocol. The only way to interact with an actor is to send it an asynchronous message. See the **actor model Wikipedia page** for a definition. And. of course, see the **Akka documentation** as well.

In Akka, actor messaging falls into the *at-most-once* category, which is similar to the previously discussed RESTful messaging. In some scenarios, at-most-once messaging is perfectly acceptable. However, there is a solution provided that supports *at-least-once* message delivery, see **Akka persistence** for more details on how this is done. The Akka toolkit aslo provides an implementation of **circuit breakers**.

Play Framework

Play Framework has extensive support for handing **RESTful messaging**. Other forms of messaging are typically handled using Akka with Play or using other libraries. For example, see the section of for **handling requests**, including suggestions for **circuit breakers**.

Lagom Framework

Lagom Framework supports both at-least-once messaging and *at-most-once* messaging. See the **Lagom design philosophy** section for a description of how Lagom fulfills the basic requirements of a Reactive Microservice. Also, see the documentation on **Internal and external communication** for more details on how Lagom provides support for a variety of message delivery implementations.

Summary

Implementing the right message delivery is fun. It is easy to build a leaky and brittle message delivery process, but fortunately, it is not that difficult to harden the process. The key concept is to think first about what your requirements are and identify how they fit into the three message delivery categories: at-most-once, at- least-once, or exactly-once. Another way to think of this is this, **is it ok to occasionally drop messages or not**?

It is relatively easy to implement a leaky message delivery process that sometimes drops messages, and this is perfectly acceptable in many applications. However, when your requirements demand that all messages must be delivered, you need to scrutinize closely every stage of the message journey to make sure that no messages can fall through a crack.

Lightbend

Lightbend (@Lightbend) is leading the enterprise transformation toward real-time, cloud-native applications. Lightbend Platform provides scalable, high-performance microservices frameworks and streaming engines for building data-centric systems that are optimized to run on cloud-native infrastructure. The most admired brands around the globe are transforming their businesses with Lightbend, engaging billions of users every day through software that is changing the world.

For more information, visit www.lightbend.com.

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