Extending Dataflow Streaming for Stateful Serverless

Philipp Haller
Associate Professor

School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden

CASTOR Software Days 2022

September 1, 2022
KTH, Stockholm, Sweden

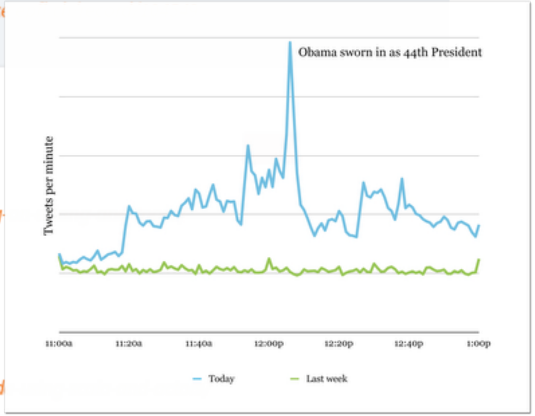
Philipp Haller: Background

- Associate professor at KTH (2014–2018 Assistant professor)
  - PhD 2010 EPFL, Switzerland
- 2005–2014 ***Scala language team***
  - 2012–2014 Typesafe, Inc. (now Lightbend, Inc.)
  - Co-author Scala language specification
- Focus on ***concurrent and distributed programming***
  - Creator of Scala Actors, co-author of Scala's futures and Scala Async

2019: **ACM SIGPLAN Programming Languages Software Award** for Scala
Core contributors:
Martin Odersky, Adriaan Moors, Aleksandar Prokopec, Heather Miller, Iulian Dragos, Nada Amin, Philipp Haller, Sebastien Doeraene, Tiark Rompf

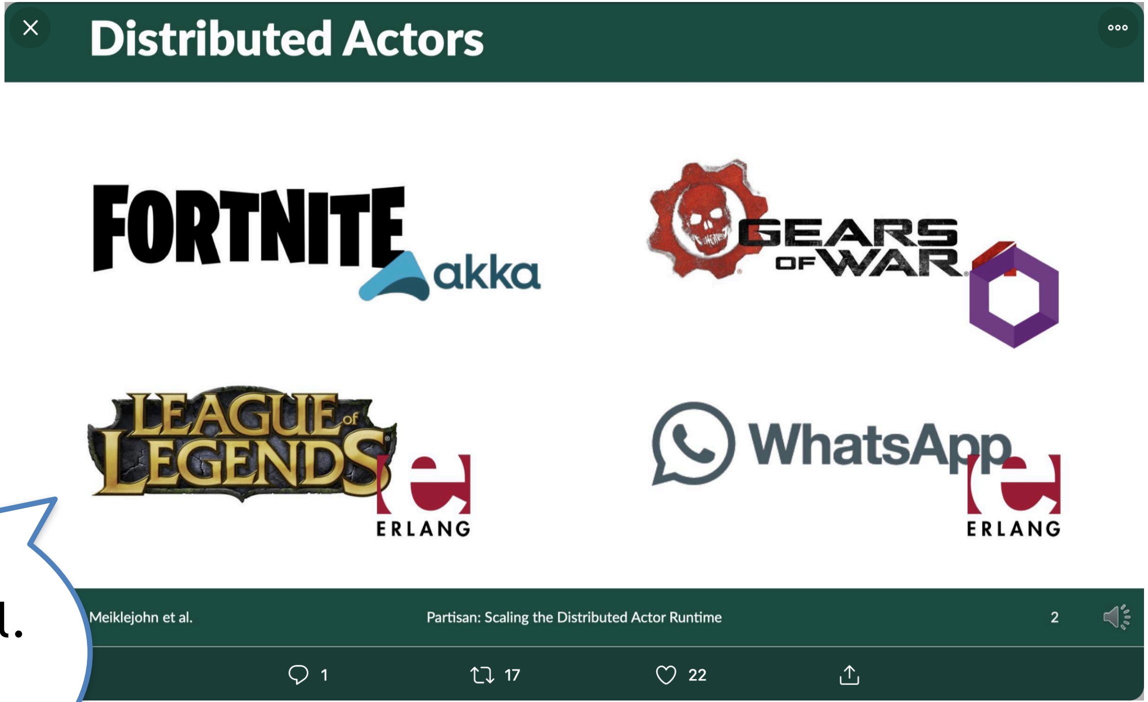
Scala Actors and Akka

Scala Actors used, e.g., in core message queue system of Twitter:



https://www.lightbend.com/akka-five-year-anniversary

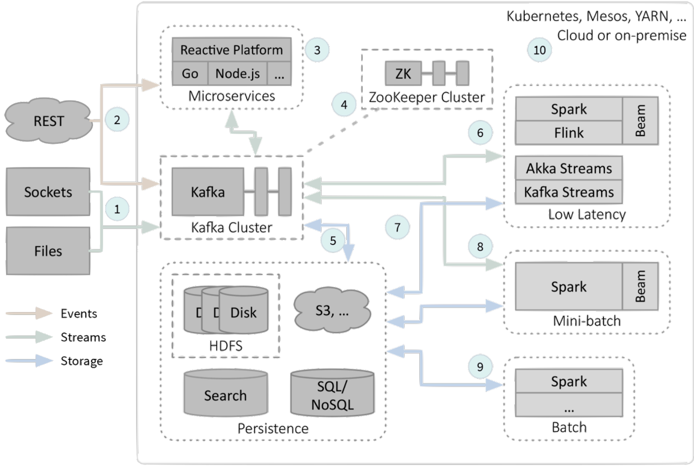
The use of actors is common in industry



Slide from: Meiklejohn et al. “Partisan” at USENIX ATC ‘19

All Modern Services are Distributed

- Each of these systems is a distributed system itself
- User data and services scattered across multiple systems
- This is not suited for classic monolith architectures: microservices architecture to the rescue*



Source: Dean Wampler: Fast Data Architectures For Streaming Applications (2nd edition), O'Reilly

*Till Rohrmann: Keynote: Rethinking how distributed applications are built. DEBS 2022.

The New York Times

PLAY THE CROSSWORD

F.T.C. Scrutinizes Tech Companies' Data Use: Live Updates

Google's apps crash in a worldwide outage.

Dec. 14, 2020

Failures in cloud-based distributed systems can be catastrophic.

Internet users worldwide received a jarring reminder on Monday about just how reliant they were on Google, when the Silicon Valley giant suffered a major outage for about an hour, sending many of its most popular services offline.

At a time when more people than ever are working from home because of the pandemic, Google services including Calendar, Gmail, Hangouts, Maps, Meet and YouTube all crashed, halting productivity and sending angry users to Twitter to vent about the loss of services. Students struggled to sign into virtual classrooms.

Reality of Distributed Systems

- **Reliability:** computers crash, messages get lost
- **Scalability:** workloads increase or decrease
- **Cloud and edge:** execution in heterogeneous environments
- **Response time:** services require low latency
- **Privacy:** systems manage sensitive, regulated data (GDPR, CCPA)

We are asking too
much of distributed software
programmers!

Limitations of Distributed Programming Models

	Distributed Programming Patterns					Guarantees		Distributed Execution		
	Cyclic Dependencie s	Dynamic Communication Topology	Dataflow Composition	Typed Communication	Request/ Reply with Futures	Exactly-once Processing	Serializable Updates	Decentralize d Deployments	Data Parallelism	Task Parallelism
Dataflows	-	-	X	X	-	X	-	-	X	X*
Actors	X	X	X*	X*	X	-	-	X	-	X
Stateful Serverless	X*	X	-	X	X*	X	-	X	X	X

* Supported with restrictions

J Spenger, P Carbone, P Haller. *Portals: an Extension of Dataflow Streaming for Stateful Serverless.* 2022, preprint

Limitations of Distributed Programming Models

	Distributed Programming Patterns					Guarantees		Distributed Execution		
	Cyclic Dependencie s	Dynamic Communication Topology	Dataflow Composition	Typed Communication	Request/ Reply with Futures	Exactly-once Processing	Serializable Updates	Decentralize d Deployments	Data Parallelism	Task Parallelism
Dataflows	-	-	X	X	-	X	-	-	X	X*
Actors	X	X	X*	X*	X	-	-	X	-	X
Stateful Serverless	X*	X	-	X	X*	X	-	X	X	X

* Supported with restrictions

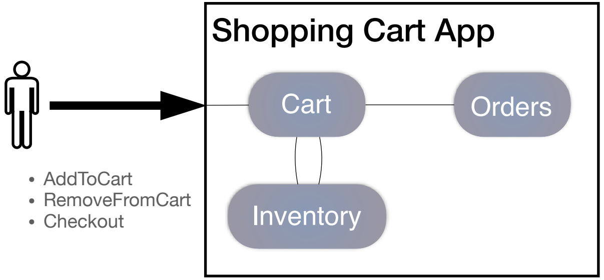
No current programming system is well-equipped for the complete job!

The Stateful Serverless Dream

- The programmer should only need to write business logic
- The stateful serverless system should automate everything else:
  - ⚙ **Reliability:** exactly-once-processing guarantees
  - ⚙ **Scalability:** scale up and down with demand
  - ⚙ **Execution:** cloud, edge, performance, latency
  - ⚙ **Privacy:** primitives for handling sensitive data

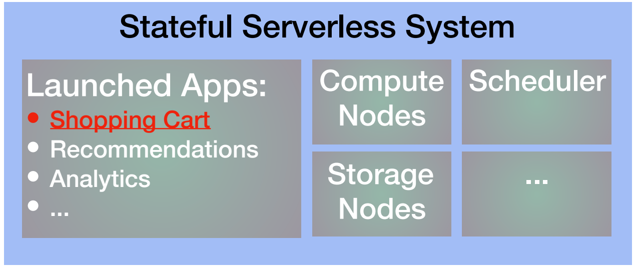
The Canonical Stateful Serverless Example: Shopping Cart

Step 1: Define the application logic



Step 2: Launch the app

Step 3: Stateful Serverless System Manages Execution

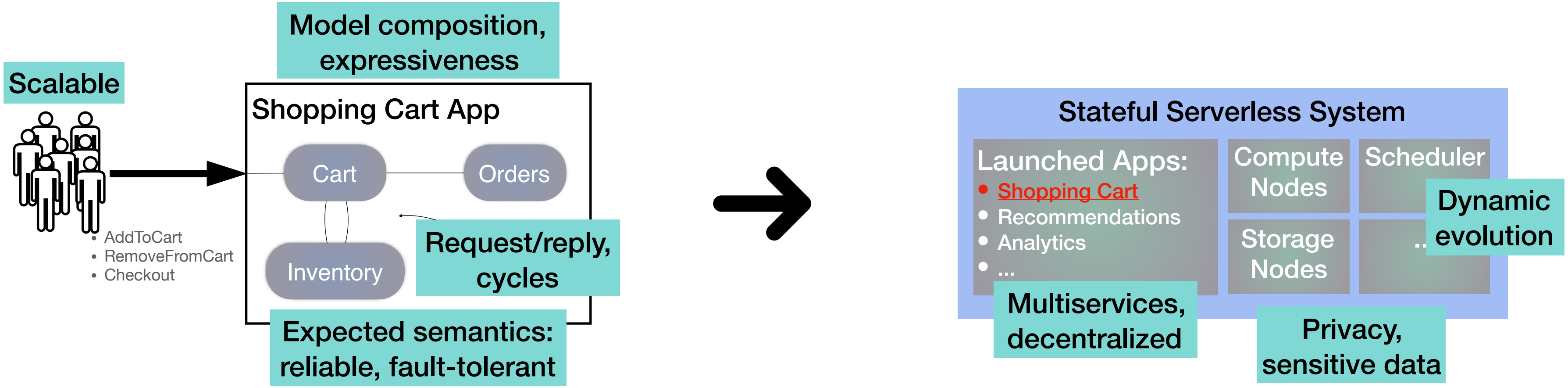


Automatically:

- End-to-end processing guarantees: checkpointing, recovery
- Manage running applications
- Manage multiple, decentralized deployments
- Scale up/down, dynamic reconfiguration
- Handle requests for live application updates, privacy

The Canonical Stateful Serverless Example: Shopping Cart

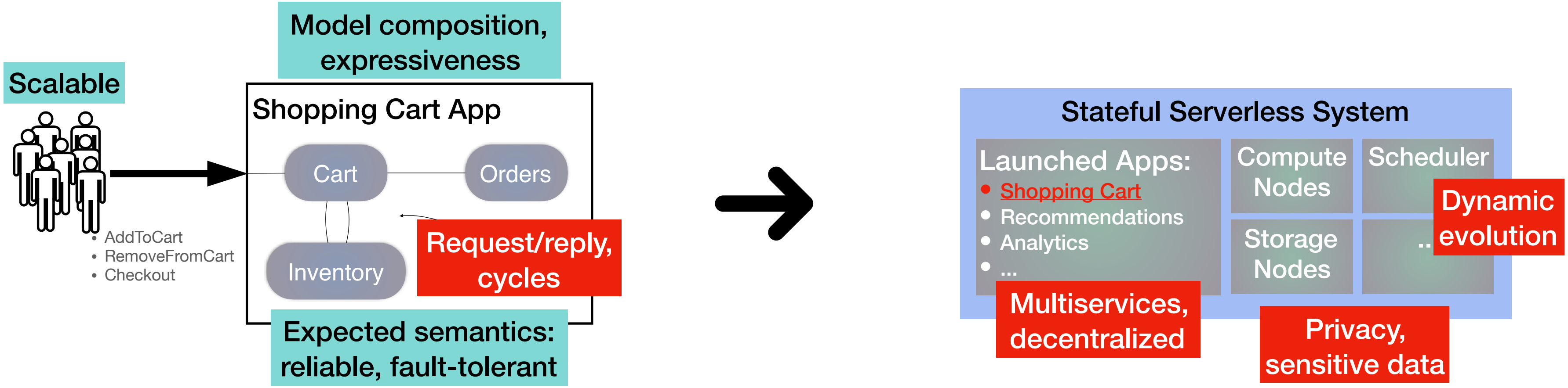
Requirements and challenges



The Canonical Stateful Serverless Example: Shopping Cart

Requirements and challenges

Requirements not supported by Dataflow Streaming



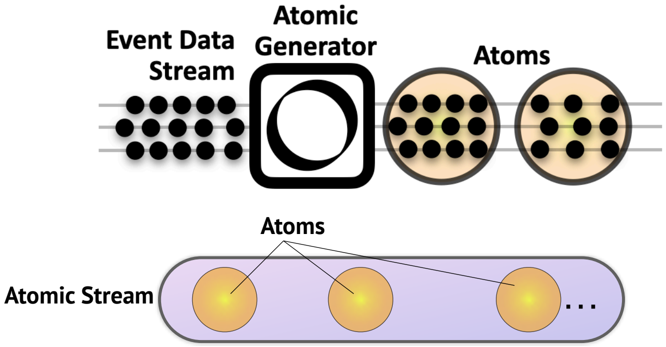
Extending Dataflow Streaming for Stateful Serverless

The **Portals programming model** introduces new abstractions:

- **Atomic Streams**
- **“Portals”**
- Workflows
- Live consistent updates (serializable)

Atomic Streams

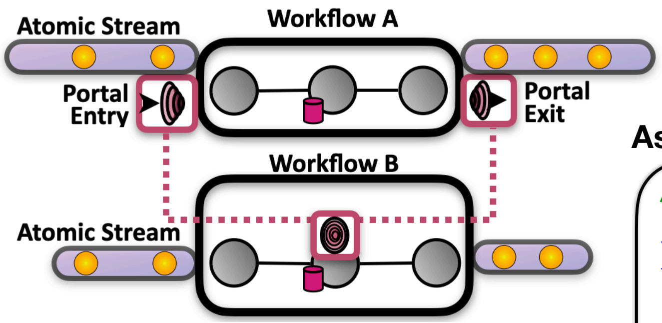
- Totally ordered, distributed stream of atoms.
    - **Atom:** Sequence of events, transactional unit of computation.
- Atomic Streams enforce end-to-end exactly-once-processing guarantees.
    - **The Atomic Processing Contract:** "*The consumer/producer must always consume and process the whole atom, before consuming and processing the next atom.*"



“Portals”

Request-reply style programming with workflows, includes futures API

- **Portals** enable **request/reply**, **futures**



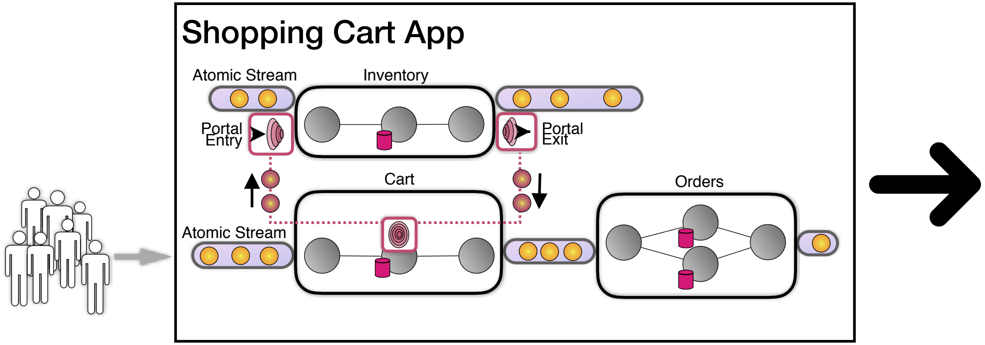
Replier: Workflow A

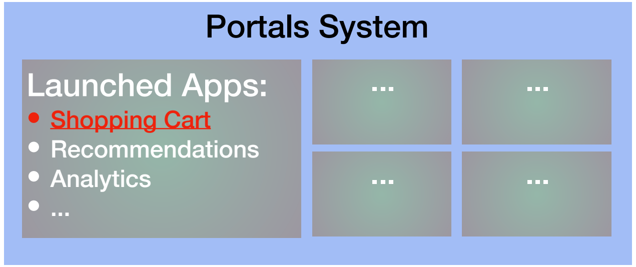
```
// Workflow A
...
val portal = builder.portals("portalName")
val workflow = builder.workflows("Workflow A")
  .source(...)
  .replier(portal)
    {  event => .... /* handle regular events */ }

    {  request => // handle requests
      ...
      val response = ...
      reply(response) // reply to request
    }
  .sink()
  .freeze()
...
```

Asker: Workflow B

```
// Workflow B
...
val portal = builder.registry.portals.get("portalName")
val requester = builder.workflows(“Workflow B”)
         .source(...)
         .asker(portal) { event=>
                 val request: T = ... // build request
                 val future:Future[R]=
                 portal.ask(request)
                 await(future) { ... /* continue */ }
         }
         .sink("sink")
```

The Canonical Stateful Serverless Example: Shopping Cart





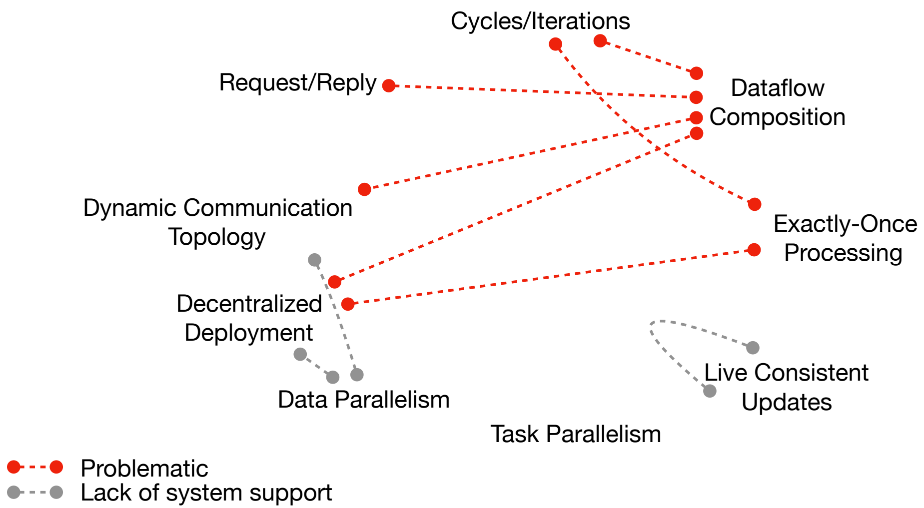
Semantically sound application logic

Fully automated deployment

When to use Portals

Applications that have/need certain combinations that are problematic.

Common solution: resort to plumbing together different systems.



Use Cases

- Complex event processing applications
- ML model training and serving
- Dynamic workflow reconfiguration
- Sagas, distributed transactions
- Serializable updates (e.g., for consistent execution of GDPR requests)
- Secure workflows / privacy-preserving computing (future work)
- …

Outlook

- **Portals programming model**
  - Express/simulate other distributed programming models in Portals
  - Operational semantics and soundness of Portals
  - Integration of Secure Multi-Party Computation (future)
- **Portals system**
  - Exploit use-cases
  - Performance optimization & evaluation
  - Release Portals 1.0: distributed, decentralized runtime
  - Sign up for launch at www.portals-project.org

Sign up for the launch at

www.portals-project.org

Summary

People

Jonas Spenger
(KTH, RISE)

Paris Carbone
(KTH, RISE)

Philipp Haller
(KTH)

Key takeaways

- Dataflow streaming a great candidate for composing stateful serverless services
  - Not so great for cycles, request/reply-style communication, decentralized dynamic deployments
- The **Portals programming model** extends dataflow streaming:
  - **Atomic streams** ensure processing guarantees over decentralized dynamic deployments
  - **Portals** enable request/reply-style communication with futures

This work was partially funded by the Swedish Foundation for Strategic Research (SSF grant no. BD15-0006) and by Digital Futures.