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

Philipp Haller creates Scala Actors (the original standard library actors). His work becomes a major influence to Akka and the main reason for Jonas Boner to choose Scala as the platform for Akka.

First commit: https://github.com/scala/scala/commit/0d8b14c6055e76c0bff3b65d0f428d711abe1f5a

Philip Haller publishes his influential paper on Scala’s Actors, ‘Actors that Unify Threads and Events’. http://infoscience

Jonas gets seriously interested in Erlang.

Jonas starts tinkering with Scala Actors.
Run next: '19/hotswap-cod

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.
Google’s apps crash in a worldwide outage.

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

<table>
<thead>
<tr>
<th>Distributed Programming Patterns</th>
<th>Guarantees</th>
<th>Distributed Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclic Dependencie s</td>
<td>Dynamic Communication Topology</td>
<td>Dataflow Composition</td>
</tr>
<tr>
<td>Dataflows</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Actors</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Stateful Serverless</td>
<td>X*</td>
<td>X</td>
</tr>
</tbody>
</table>

* Supported with restrictions

---

# Limitations of Distributed Programming Models

## Distributed Programming Patterns

<table>
<thead>
<tr>
<th></th>
<th>Cyclic Dependencie s</th>
<th>Dynamic Communication Topology</th>
<th>Dataflow Composition</th>
<th>Typed Communication</th>
<th>Request/Reply with Futures</th>
<th>Guarantees</th>
<th>Distributed Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataflows</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>Actors</td>
<td>X</td>
<td>X</td>
<td>X*</td>
<td>X*</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Stateful Serverless</td>
<td>X*</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X*</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

* Supported with restrictions

---

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

Philipp Haller
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
TheCanonicalStatefulServerlessExample:ShoppingCart

Step 1: Define the application logic

Shopping Cart App

- Cart
- Orders
- Inventory

Step 2: Launch the app

Step 3: Stateful Serverless System Manages Execution

Stateful Serverless System

Launched Apps:
- Shopping Cart
- Recommendations
- Analytics
- ...

Compute Nodes

Scheduler

Storage Nodes

- ...

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

Scalable

Model composition, expressiveness

Shopping Cart App

Expected semantics: reliable, fault-tolerant

Request/reply, cycles

Cart
Orders
Inventory

Stateful Serverless System

Launched Apps:
• **Shopping Cart**
• Recommendations
• Analytics
• ...

• Compute Nodes
• Storage Nodes
• Scheduler

Dynamic evolution

Multiservices, decentralized

Privacy, sensitive data

Philipp Haller
The Canonical Stateful Serverless Example: Shopping Cart

Requirements and challenges

Stateful Serverless System

- Launched Apps:
  - Shopping Cart
  - Recommendations
  - Analytics
  - ...
- Compute Nodes
- Storage Nodes
- Scheduler
- Multiservices, decentralized
- Dynamic evolution
- Privacy, sensitive data

Scalable

- AddToCart
- RemoveFromCart
- Checkout

Model composition, expressiveness

Expected semantics: reliable, fault-tolerant

Request/reply, cycles

Philipp Haller
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
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](http://www.portals-project.org)
Summary

www.portals-project.org

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

People

Jonas Spenger  
(KTH, RISE)

Paris Carbone  
(KTH, RISE)

Philipp Haller  
(KTH)