A Survey of Actor-Like Programming Models for Serverless Computing

Jonas Spenger[1,2], Paris Carbone[1,2], Philipp Haller[1]

[1] EECS, KTH Royal Institute of Technology, Stockholm, Sweden
[2] Computer Systems, RISE Research Institutes of Sweden, Stockholm, Sweden

ABS Workshop 2023, Lyon, France, Thursday 5th October, 2023

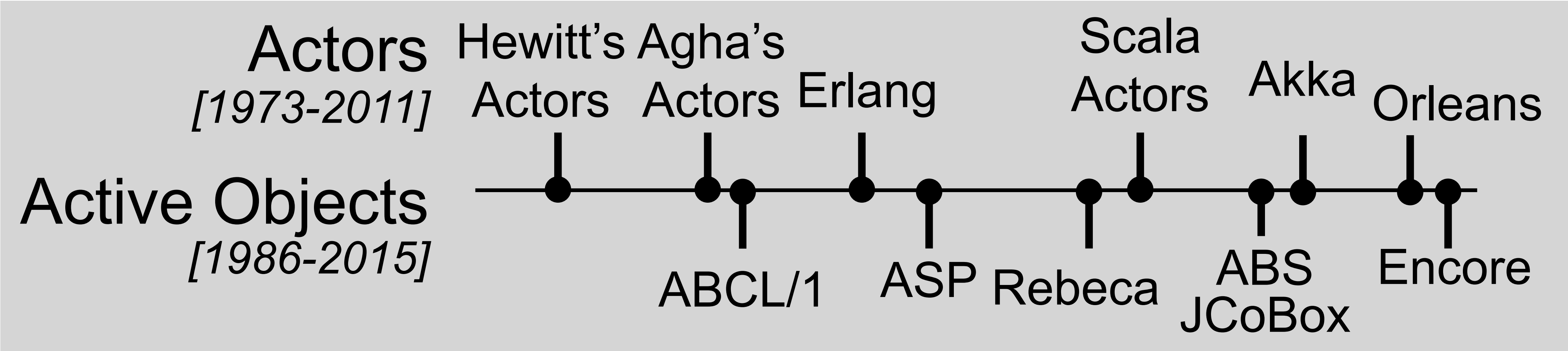
Motivation

Distributed programming is difficult

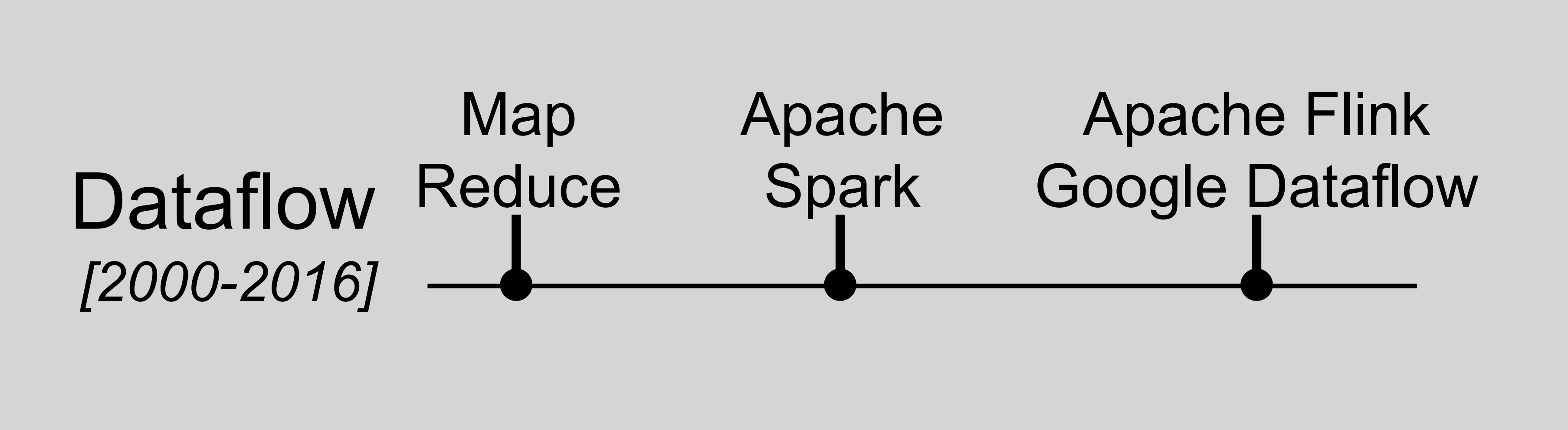
- *Messages get lost, duplicated, reordered*
- *Nodes crash, restart*
- *Delays are unknown*

Distributed programming requires good abstractions

Distributed Programming Models

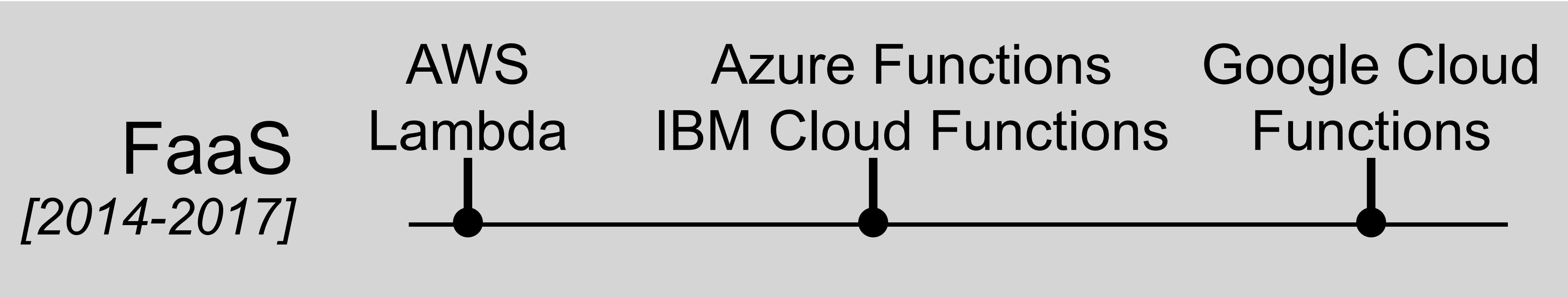


- *Actors, Active Objects*
    - Low-level abstraction
    - Execution over "isolated turns" (Koster et al. 2016)



- *Dataflow Streaming*
    - Acyclic graphs of stateful tasks
    - Data processing, performance, fault-tolerance

FaaS, Serverless Computing

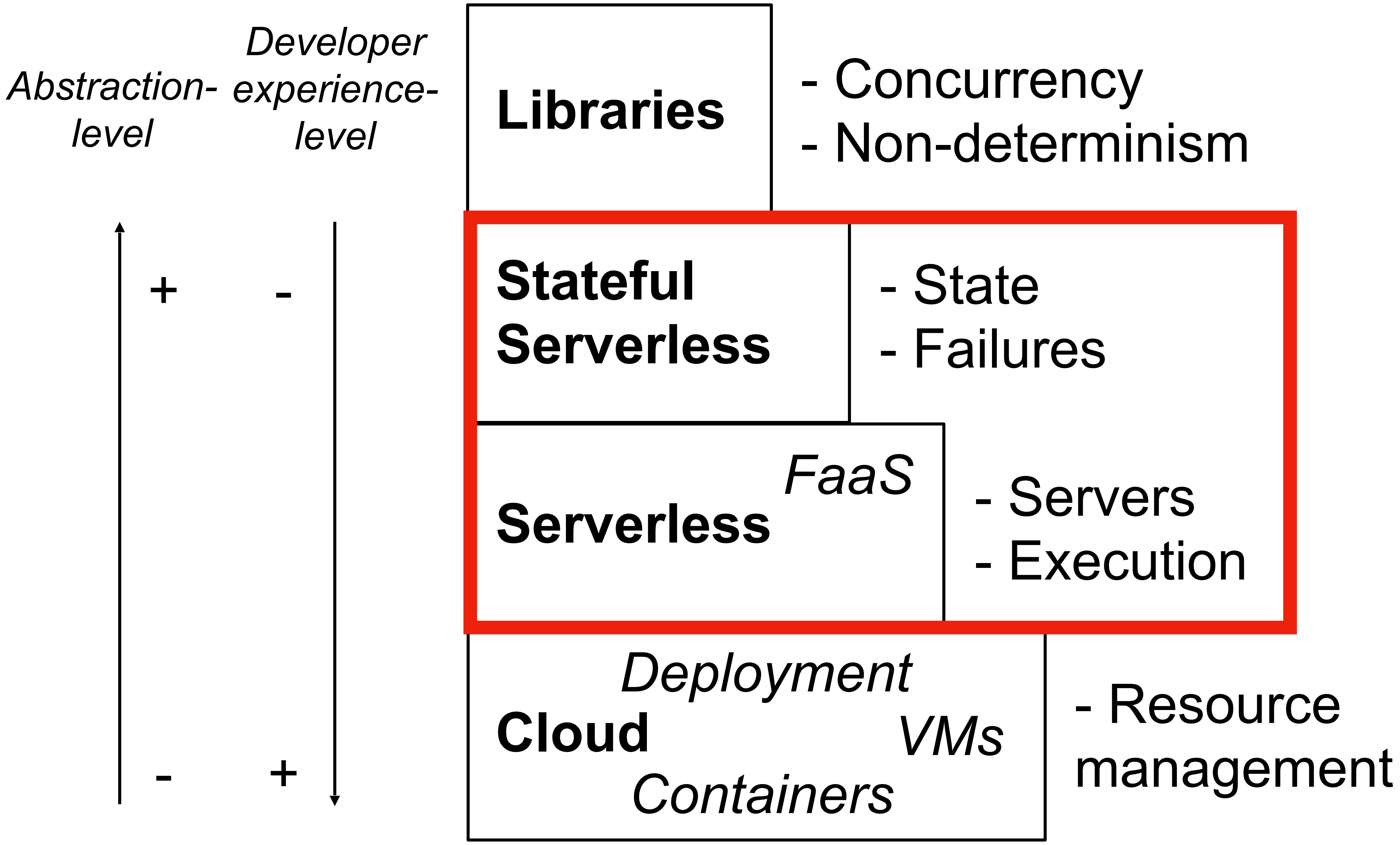


- FaaS (Function-as-a-Service):
  - *Stateless functions*
  - *Triggers*

- **Serverless simplifies building cloud applications**
  - *Elasticity: scaling down to zero, and up to infinity*
  - Serverless framework fully manages the function execution, abstracts away server, execution; billing per use

(Castro et al. 2019)

Levels of Abstraction for Utility Computing



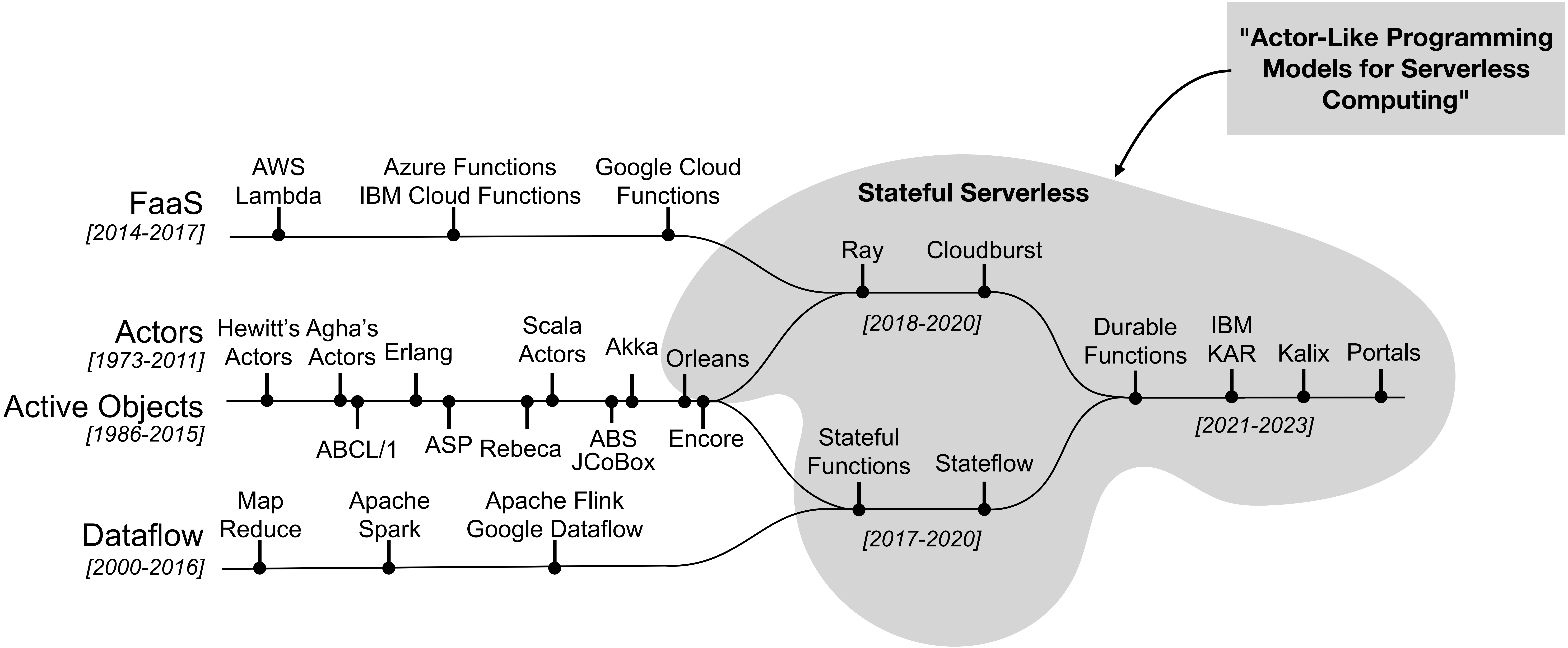
Stateful Serverless

- *Fully manages compute, state, messaging*
- **Consistency is the *framework*'s *responsibility***

FaaS / Serverless

- *Fully manages compute*
- *Application challenges:*
  - *Functions are stateless*
  - *Functions cannot call other functions*
  - **Consistency is the *application*'s *responsibility***

Stateful Serverless



Serverless Actor / Active Object Example

Bank Account Entity

```
class Account(ctx: Context):
  val balance = PerKeyState[Int](ctx).withDefault(0)

  def get(): Int =
    balance.get()

  def deposit(amount: Int): Unit =
    balance.set(balance.get() + amount)

  def withdraw(amount: Int): Unit =
    balance.set(balance.get() - amount)

  def transfer(amount: Int, to: String): Unit =
    val otherAccount = EntityRef[Account](ctx).withKey(to)
    if balance.get() > amount then
      balance.set(balance.get() - amount)
      otherAccount.deposit(amount)
```

- Bank Account Entity
  - ***Balance** of the account*
  - ***Get** the acc's balance*
  - ***Deposit***
  - ***Withdraw***
  - ***Transfer***
- Inspired by stateful serverless systems

Serverless Actor / Active Object Example

Bank Account Entity

```
class Account(ctx: Context):
  val balance = PerKeyState[Int](ctx).withDefault(0)

  def get(): Int =
    balance.get()

  def deposit(amount: Int): Unit =
    balance.set(balance.get() + amount)

  def withdraw(amount: Int): Unit =
    balance.set(balance.get() - amount)

  def transfer(amount: Int, to: String): Unit =
    val otherAccount = EntityRef[Account](ctx).withKey(to)
    if balance.get() > amount then
      balance.set(balance.get() - amount)
      otherAccount.deposit(amount)
```

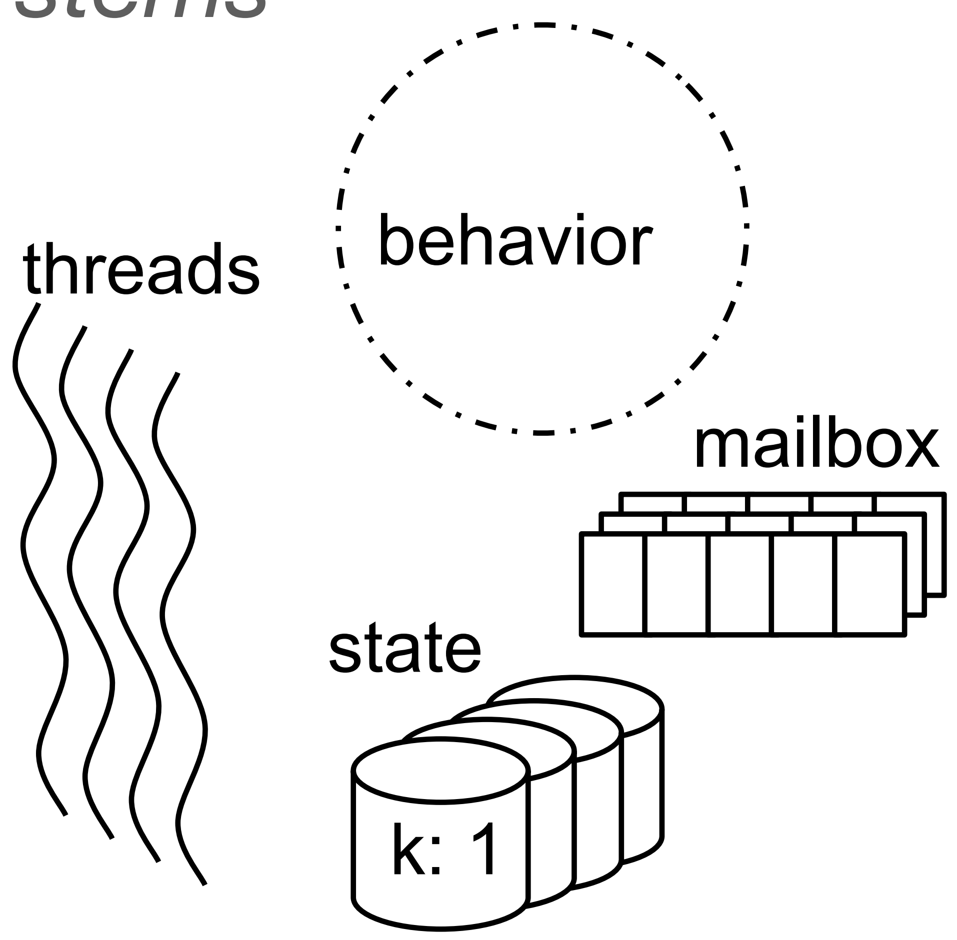
- Virtual Actors
  - *Microsoft Orleans (Bykov et al. 2011, Bernstein et al. 2014)*
  - *Virtual life-cycle*
  - *Virtual Refs*
- Data-parallel
  - *Virtual Ref: < Type + Key >*
  - *State disjoint over keys*

Serverless Actor / Active Object Example

Bank Account Entity

```
class Account(ctx: Context):
  val balance = PerKeyState[Int](ctx).withDefault(0)

  def get(): Int =
    balance.get()

  def deposit(amount: Int): Unit =
    balance.set(balance.get() + amount)

  def withdraw(amount: Int): Unit =
    balance.set(balance.get() - amount)

  def transfer(amount: Int, to: String): Unit =
    val otherAccount = EntityRef[Account](ctx).withKey(to)
    if balance.get() > amount then
      balance.set(balance.get() - amount)
      otherAccount.deposit(amount)
```

- Decoupled, explicit state
- Framework disaggregated:
  - *Mailbox*
  - *State*
  - *Threads*
  - *Behavior*
  - *All managed on different subsystems*



Serverless Actor / Active Object Example

Bank Account Entity

```
class Account(ctx: Context):
  val balance = PerKeyState[Int](ctx).withDefault(0)

  def get(): Int =
    balance.get()

  def deposit(amount: Int): Unit =
    balance.set(balance.get() + amount)

  def withdraw(amount: Int): Unit =
    balance.set(balance.get() - amount)

  def transfer(amount: Int, to: String): Unit =
    val otherAccount = EntityRef[Account](ctx).withKey(to)
    if balance.get() > amount then
      balance.set(balance.get() - amount)
      otherAccount.deposit(amount)
```

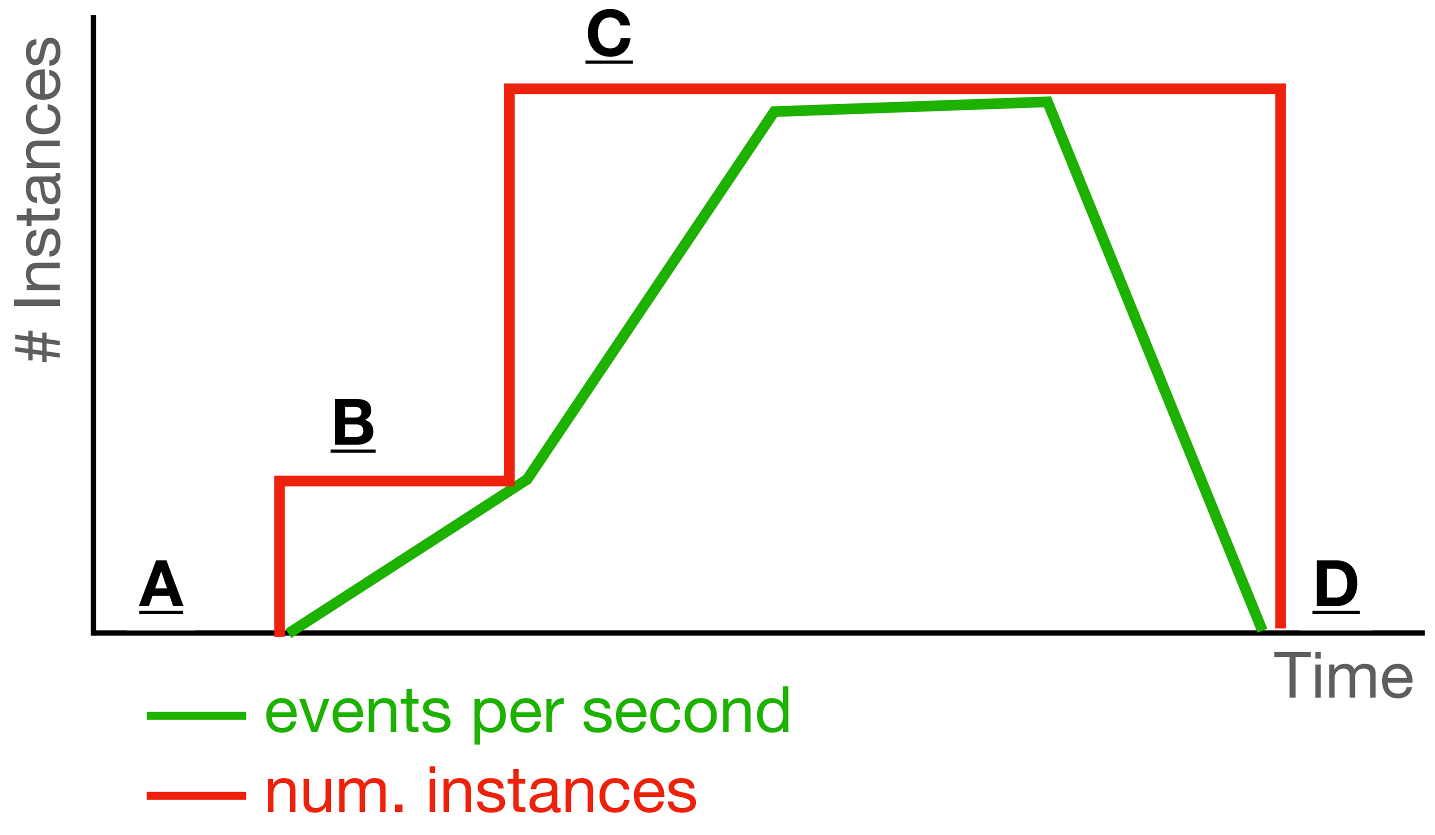
- Discussed on next slides:
  - *Fault Tolerance*
  - *Serverless Execution*

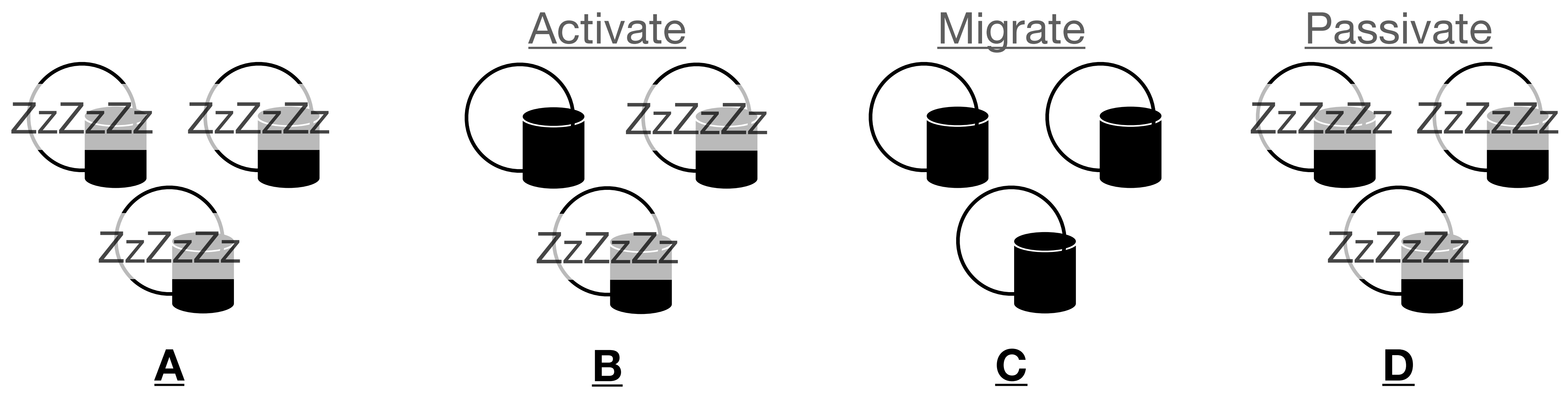
Challenges of Serverless Actors and Active Objects

- Serverless state management
- Fault tolerance

Serverless State Management

- Serverless requires on-demand execution, provisioning





- Framework needs to **activate, migrate, passivate** execution state
  - This is already a challenging problem
  - Especially challenging with actors / active objects

Serverless State Management Challenges with Actors / Active Objects

Activation, migration, passivation of *actors / active objects*

Execution state needs to be *serialized, deserialized*

- ***Code might contain suspended calls***
  - *e.g. await a future / guard*
  - *Coroutines*
  - *Stackful continuations*
  - *Continuation closures*

```
val balance = PerKeyState[Int](ctx).withDefault(0)
def withdraw(amount: Int): Unit =
  await balance.get() >= amount
  balance.set(balance.get() - amount)
```

- **Suspended state needs to be serialized / deserialized**
- **All cases are non-trivial to handle**

Serverless State Management Challenges with Actors / Active Objects

Techniques

- *Ensuring continuation closures are safe to serialize/deserialize*
  - *e.g. Spores (Miller et al., 2014)*
- *Virtual life-cycle*
  - *No creation, deletion*
- *Static actor behavior*
- *Explicit state*

Fault Tolerance

Failures make programs hard to reason about

Processing guarantees:

- Messages are processed ***At-Most-Once***
- Messages are processed ***At-Least-Once***
- Messages are processed ***Exactly-Once***

Processing Guarantees

Bank Account Entity

```
class Account(ctx: Context):
  val balance = PerKeyState[Int](ctx).withDefault(0)
  ...
  def deposit(amount: Int): Unit =
    balance.set(balance.get() + amount)
```

Exactly-once processing makes programs significantly easier to write and reason about!

Challenge: how to provide exactly-once processing

At-Most-Once

- *Issues: deposit **may not occur***
- *App needs to: **re-send dropped events***
- *e.g. Actor systems*

At-Least-Once

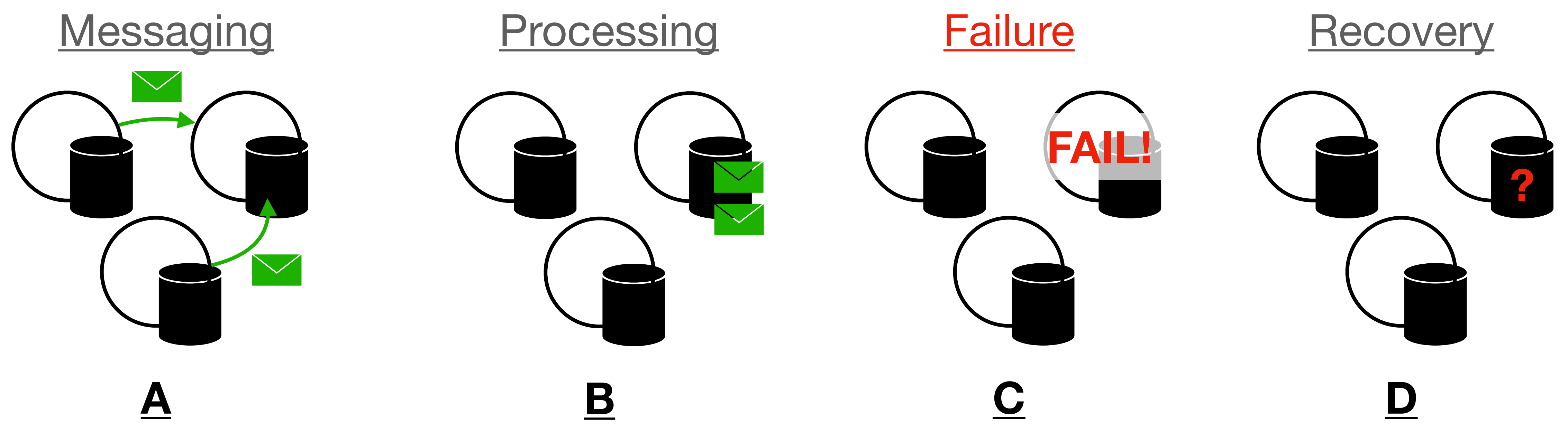
- *Issues: deposit **may occur multiple times***
- *App needs to: **deduplicate, ensure idempotency***
- *e.g. FaaS*

Exactly-Once

- ***No issues***
- *App: **works as expected***
- *e.g. Stateful Serverless*

Challenges with Fault Tolerance

- Failures happen, and need to be dealt with



- Fault tolerance requires recovering to a consistent state after failure
  - *Finding a consistent state* ***?*** *is a challenging problem*

Challenges with Fault Tolerance for Actors / Active Objects

Challenges

- Non-determinism
  - ***Actor behavior non-deterministic***
    - *e.g. timers, random numbers*
  - ***=> Cannot use event logging for replay***
- Dynamic topology
- External systems cannot rollback on events once emitted

Challenges with Fault Tolerance for Actors / Active Objects

Techniques

- Capture all non-determinism together with event log
  - *e.g. Immortals (Goldstein et al., 2020)*
- Checkpoint-recovery
  - *Rollback-recovery from a causally consistent snapshot (Elnozahy et al., 2002), e.g. using Chandy-Lamport (Chandy, Lamport, 1985)*
- Limiting dynamicity of model
  - *e.g. Virtual Actors*
- Transactional streams connecting to external systems
  - *e.g. Kafka, Atomic Streams (Spenger et al., 2022)*

Challenges with Fault Tolerance

Ensuring that state is consistently updated

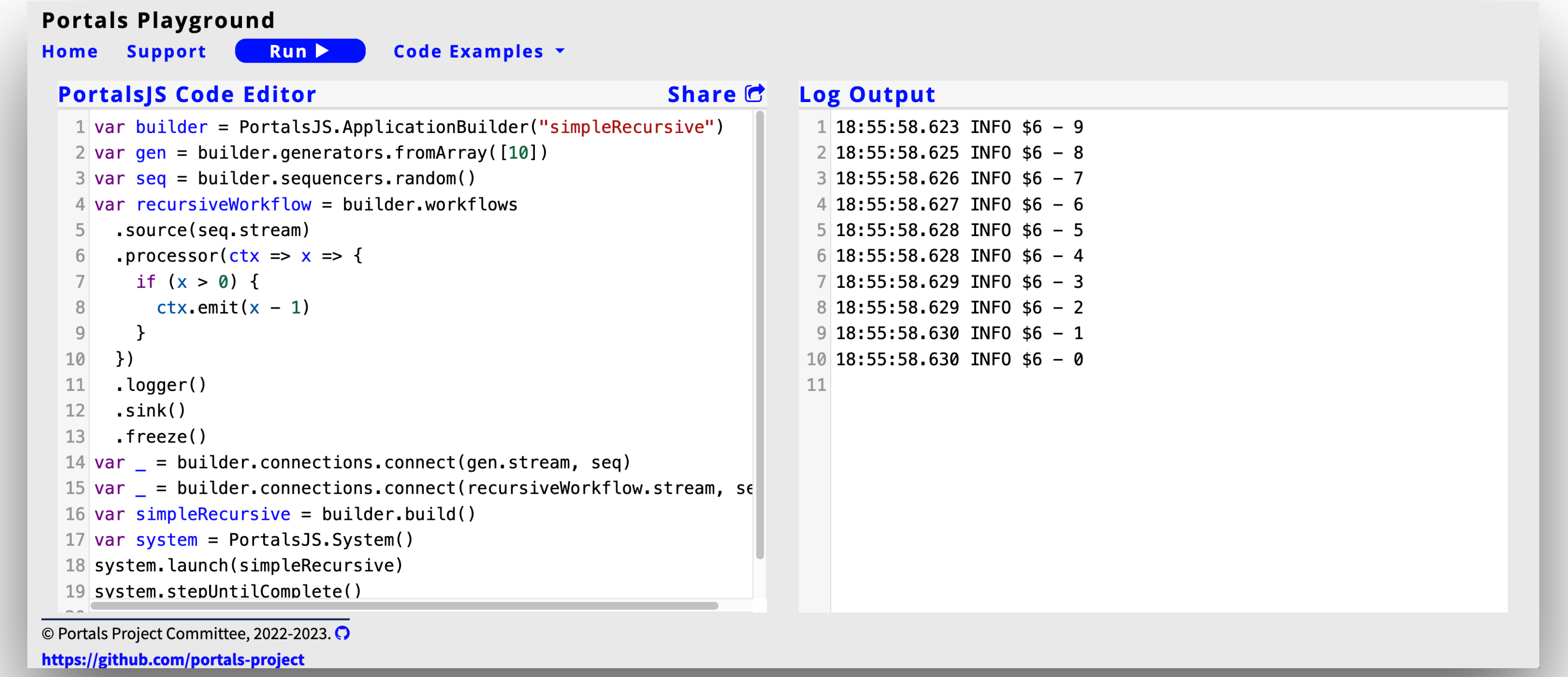
- Transactional updates to state
  - **Processing as atomic steps:**
    - *Consume event*
    - *Process event*
    - *Produce all side-effects*

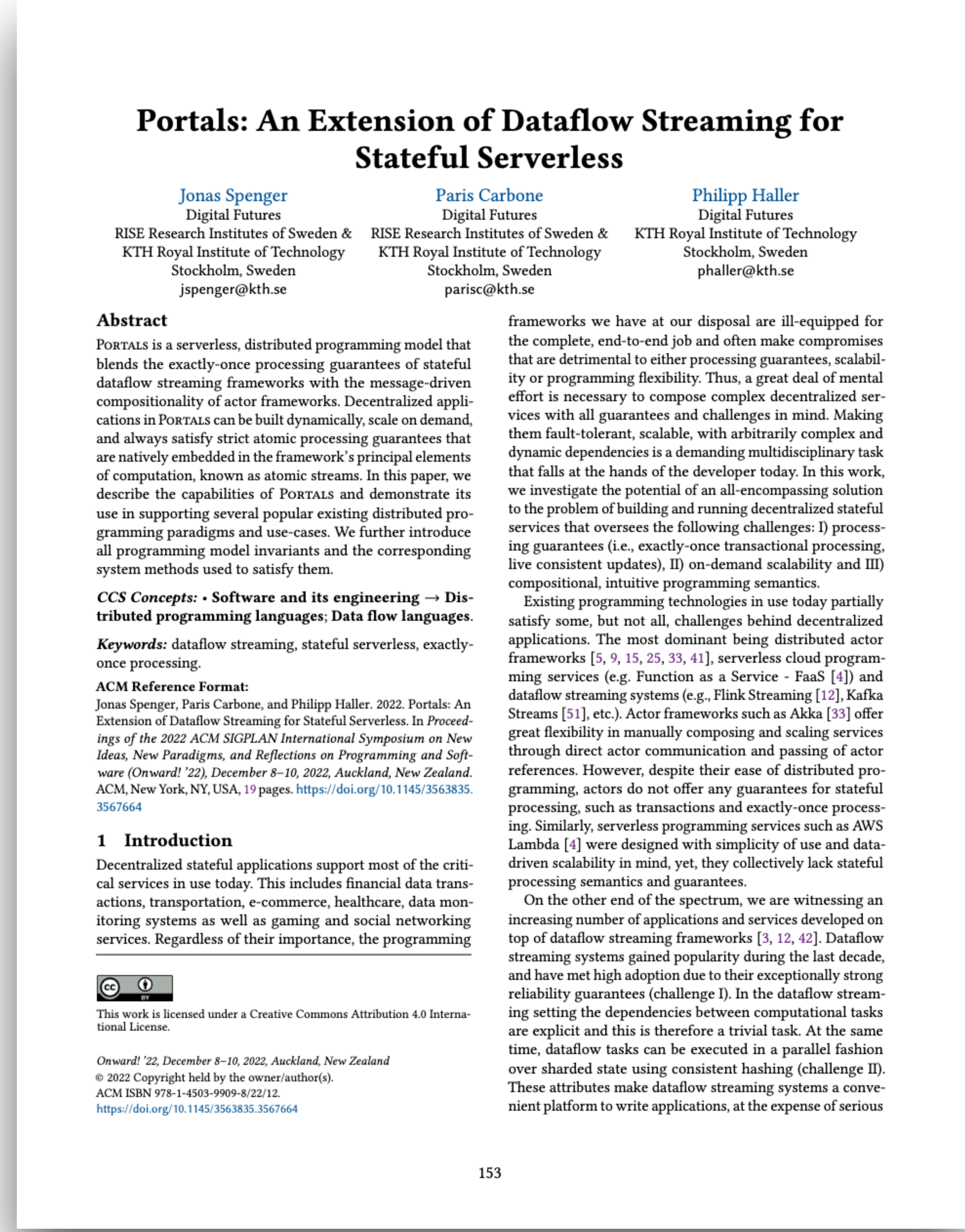
Techniques

- Distributed Two-Phase Commit
- Batched, pipelined for efficiency, e.g. Flink (Carbone et al., 2017)

Portals: Unifying Stateful Dataflow Streaming with Actors

- Disclaimer: under development by the authors
- Restriction: *actors cannot form new connections dynamically*
- **Leverages atomic streams for exactly-once processing**
- https://www.portals-project.org/
- https://www.portals-project.org/playground/





Portals: An Extension of Dataflow Streaming for Stateful Serverless

Jonas Spenger
Digital Futures
RISE Research Institutes of Sweden &
KTH Royal Institute of Technology
Stockholm, Sweden
jspenger@kth.se

Paris Carbone
Digital Futures
RISE Research Institutes of Sweden &
KTH Royal Institute of Technology
Stockholm, Sweden
parisc@kth.se

Philipp Haller
Digital Futures
KTH Royal Institute of Technology
Stockholm, Sweden
phaller@kth.se

Abstract

PORTALS is a serverless, distributed programming model that blends the exactly-once processing guarantees of stateful dataflow streaming frameworks with the message-driven compositionality of actor frameworks. Decentralized applications in PORTALS can be built dynamically, scale on demand, and always satisfy strict atomic processing guarantees that are natively embedded in the framework's principal elements of computation, known as atomic streams. In this paper, we describe the capabilities of PORTALS and demonstrate its use in supporting several popular existing distributed programming paradigms and use-cases. We further introduce all programming model invariants and the corresponding system methods used to satisfy them.

CCS Concepts: • **Software and its engineering → Distributed programming languages**; **Data flow languages**.

Keywords: dataflow streaming, stateful serverless, exactly-once processing.

ACM Reference Format:
Jonas Spenger, Paris Carbone, and Philipp Haller. 2022. Portals: An Extension of Dataflow Streaming for Stateful Serverless. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '22), December 8–10, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3563835.3567664

1 Introduction

Decentralized stateful applications support most of the critical services in use today. This includes financial data transactions, transportation, e-commerce, healthcare, data monitoring systems as well as gaming and social networking services. Regardless of their importance, the programming frameworks we have at our disposal are ill-equipped for the complete, end-to-end job and often make compromises that are detrimental to either processing guarantees, scalability or programming flexibility. Thus, a great deal of mental effort is necessary to compose complex decentralized services with all guarantees and challenges in mind. Making them fault-tolerant, scalable, with arbitrarily complex and dynamic dependencies is a demanding multidisciplinary task that falls at the hands of the developer today. In this work, we investigate the potential of an all-encompassing solution to the problem of building and running decentralized stateful services that oversees the following challenges: I) processing guarantees (i.e., exactly-once transactional processing, live consistent updates), II) on-demand scalability and III) compositional, intuitive programming semantics.

Existing programming technologies in use today partially satisfy some, but not all, challenges behind decentralized applications. The most dominant being distributed actor frameworks [5, 9, 15, 25, 33, 41], serverless cloud programming services (e.g. Function as a Service - FaaS [4]) and dataflow streaming systems (e.g., Flink Streaming [12], Kafka Streams [51], etc.). Actor frameworks such as Akka [33] offer great flexibility in manually composing and scaling services through direct actor communication and passing of actor references. However, despite their ease of distributed programming, actors do not offer any guarantees for stateful processing, such as transactions and exactly-once processing. Similarly, serverless programming services such as AWS Lambda [4] were designed with simplicity of use and data-driven scalability in mind, yet, they collectively lack stateful processing semantics and guarantees.

On the other end of the spectrum, we are witnessing an increasing number of applications and services developed on top of dataflow streaming frameworks [3, 12, 42]. Dataflow streaming systems gained popularity during the last decade, and have met high adoption due to their exceptionally strong reliability guarantees (challenge I). In the dataflow streaming setting the dependencies between computational tasks are explicit and this is therefore a trivial task. At the same time, dataflow tasks can be executed in a parallel fashion over sharded state using consistent hashing (challenge II). These attributes make dataflow streaming systems a convenient platform to write applications, at the expense of serious

Onward! '22, December 8–10, 2022, Auckland, New Zealand
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9909-8/22/12.
https://doi.org/10.1145/3563835.3567664

153

Comparison of Actor-Like Serverless Systems

System	Actor Style	Processing Guarantees
Orleans		
Durable Functions		
Flink Stateful Functions		
IBM KAR		
Kalix		
Portals		
Ray		
Cloudburst		

Comparison of Actor-Like Serverless Systems

System	Actor Style	Processing Guarantees
Orleans	Virtual	At-Most-Once/At-Least-Once
Durable Functions	Virtual	***Exactly-Once***
Flink Stateful Functions	Virtual	***Exactly-Once***
IBM KAR	Virtual	At-Least-Once
Kalix	Virtual	At-Least-Once
Portals	Virtual	***Exactly-Once***
Ray	Non-virtual	At-Most-Once/At-Least-Once
Cloudburst	Non-virtual	At-Least-Once

Comparison of Actor-Like Serverless Systems

System	Actor Style	Processing Guarantees	Msg Ops	Msg Futures
Orleans	Virtual	At-Most-Once/At-Least-Once	Send, Call, Reply	✓
Durable Functions	Virtual	***Exactly-Once***	Send, Reply-	✗
Flink Stateful Functions	Virtual	***Exactly-Once***	Send, Reply	✗
IBM KAR	Virtual	At-Least-Once	Send, Call, TailCall, Reply	✓
Kalix	Virtual	At-Least-Once	Send, Call, Reply, Forward	-
Portals	Virtual	***Exactly-Once***	Send, Call, Reply	✓
Ray	Non-virtual	At-Most-Once/At-Least-Once	Call, Reply	✓
Cloudburst	Non-virtual	At-Least-Once	Call, Reply	✓

More in the Paper

A Survey of Actor-Like Programming Models for Serverless Computing

Jonas Spenger[1,2][0000−0002−7119−5234], Paris Carbone[1,2][0000−0002−9351−8508], and Philipp Haller[1][0000−0002−2659−5271]

[1] Digital Futures and EECS, KTH Royal Institute of Technology, Stockholm, Sweden
{jspenger,parisc,phaller}@kth.se
[2] Computer Systems, RISE Research Institutes of Sweden, Stockholm, Sweden
{jonas.spenger,paris.carbone}@ri.se

Abstract. Serverless computing promises to significantly simplify cloud computing by providing Functions-as-a-Service where invocations of functions, triggered by events, are automatically scheduled for execution on compute nodes. Notably, the serverless computing model does not require the manual provisioning of virtual machines; instead, FaaS enables load-based billing and auto-scaling according to the workload, reducing costs and making scheduling more efficient. While early serverless programming models only supported stateless functions and severely restricted program composition, recently proposed systems offer greater flexibility by adopting ideas from actor and dataflow programming. This paper presents a survey of actor-like programming abstractions for stateful serverless computing, and provides a characterization of their properties and highlights their origin.

Keywords: Actor Model · Active Objects · Serverless Computing · Dataflow · Stateful Serverless · Distributed Programming · Cloud Computing

1 Introduction

Serverless computing has greatly simplified building cloud applications by providing Functions-as-a-Service (FaaS), a programming model consisting of *functions* and *event triggers*. These functions are automatically scheduled for execution on compute nodes, elastically scaling with the load [22]. In effect, the serverless model fully abstracts away the underlying computing infrastructure, billing and running user code on-demand. As a consequence, serverless computing can reduce costs and make scheduling more efficient.

While early serverless models were restricted, recent developments have introduced more flexible abstractions. The first major serverless frameworks, such as AWS Lambda [6] and similar [31,40,51], were restricted to: 1) *stateless functions*; and 2) limited compositional primitives such as no direct function-to-function messaging, often-cited challenges with serverless computing [12,36,42]. Recent developments, however, have seen programming models supporting stateful serverless that overcome these challenges through abstractions closely related

- Introduction
- **Background**
- **Challenges**
- **Analysis of Systems**
  - *Programming model and properties*
  - *Communication properties*
  - *Serverless execution properties*
- **Research Directions**
- Conclusions

Conclusions

Stateful serverless simplifies writing distributed applications

- Actor-like programming models for stateful serverless
  - *Virtual, decoupled, disaggregated, data-parallel, fault tolerant, serverless*
- Challenges with state management, fault tolerance
  - *Difficult with actors / active objects*
- Analysis of systems shows diversity
  - *Processing guarantees, dynamicity, message operations, futures*
- Research directions
  - *Static guarantees, end-to-end fault tolerance, new programming abstractions*

Thank You for Listening!

A Survey of Actor-Like Programming Models for Serverless Computing

Jonas Spenger[1,2][0000−0002−7119−5234], Paris Carbone[1,2][0000−0002−9351−8508], and Philipp Haller[1][0000−0002−2659−5271]

[1] Digital Futures and EECS, KTH Royal Institute of Technology, Stockholm, Sweden
{jspenger,parisc,phaller}@kth.se
[2] Computer Systems, RISE Research Institutes of Sweden, Stockholm, Sweden
{jonas.spenger,paris.carbone}@ri.se

Abstract. Serverless computing promises to significantly simplify cloud computing by providing Functions-as-a-Service where invocations of functions, triggered by events, are automatically scheduled for execution on compute nodes. Notably, the serverless computing model does not require the manual provisioning of virtual machines; instead, FaaS enables load-based billing and auto-scaling according to the workload, reducing costs and making scheduling more efficient. While early serverless programming models only supported stateless functions and severely restricted program composition, recently proposed systems offer greater flexibility by adopting ideas from actor and dataflow programming. This paper presents a survey of actor-like programming abstractions for stateful serverless computing, and provides a characterization of their properties and highlights their origin.

Keywords: Actor Model · Active Objects · Serverless Computing · Dataflow · Stateful Serverless · Distributed Programming · Cloud Computing

1 Introduction

Serverless computing has greatly simplified building cloud applications by providing Functions-as-a-Service (FaaS), a programming model consisting of *functions* and *event triggers*. These functions are automatically scheduled for execution on compute nodes, elastically scaling with the load [22]. In effect, the serverless model fully abstracts away the underlying computing infrastructure, billing and running user code on-demand. As a consequence, serverless computing can reduce costs and make scheduling more efficient.

While early serverless models were restricted, recent developments have introduced more flexible abstractions. The first major serverless frameworks, such as AWS Lambda [6] and similar [31,40,51], were restricted to: 1) *stateless functions*; and 2) limited compositional primitives such as no direct function-to-function messaging, often-cited challenges with serverless computing [12,36,42]. Recent developments, however, have seen programming models supporting stateful serverless that overcome these challenges through abstractions closely related

Stateful serverless simplifies writing distributed applications

- Actor-like programming models for stateful serverless
  - *Virtual, decoupled, disaggregated, data-parallel, fault tolerant, serverless*
- Challenges with state management, fault tolerance
  - *Difficult with actors / active objects*
- Analysis of systems shows diversity
  - *Processing guarantees, dynamicity, message operations, futures*
- Future directions
  - *Static guarantees, end-to-end fault tolerance, new programming abstractions*

Thanks to the co-authors: Paris Carbone, Philipp Haller.
Many thanks to the reviewers for their helpful comments.
Check out our work at https://www.portals-project.org/.

This work was funded by Digital Futures, SSF, Horizon Europe, RISE AI.

KTH VETENSKAP OCH KONST

RI. SE

digital futures

References

De Koster, Joeri, Tom Van Cutsem, and Wolfgang De Meuter. "43 years of actors: a taxonomy of actor models and their key properties." Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control. 2016.

Castro, Paul, et al. "The rise of serverless computing." Communications of the ACM 62.12 (2019): 44-54.

Bykov, Sergey, et al. "Orleans: cloud computing for everyone." Proceedings of the 2nd ACM Symposium on Cloud Computing. 2011.

Bernstein, Phil, et al. "Orleans: Distributed virtual actors for programmability and scalability." MSR-TR-2014–41. 2014.

Miller, Heather, Philipp Haller, and Martin Odersky. "Spores: A type-based foundation for closures in the age of concurrency and distribution." ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28. Springer Berlin Heidelberg, 2014.

Goldstein, Jonathan, et al. "Ambrosia: Providing performant virtual resiliency for distributed applications." Proceedings of the VLDB Endowment 13.5 (2020): 588-601.

Elnozahy, Elmootazbellah Nabil, et al. "A survey of rollback-recovery protocols in message-passing systems." ACM Computing Surveys (CSUR) 34.3 (2002): 375-408.

Chandy, K. Mani, and Leslie Lamport. "Distributed snapshots: Determining global states of distributed systems." ACM Transactions on Computer Systems (TOCS) 3.1 (1985): 63-75.

Spenger, Jonas, Paris Carbone, and Philipp Haller. "Portals: An extension of dataflow streaming for stateful serverless." Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. 2022.

Burckhardt, Sebastian, et al. "Durable functions: Semantics for stateful serverless." Proceedings of the ACM on Programming Languages 5.OOPSLA (2021): 1-27.

Foundation, T.A.S.: Apache flink stateful functions. https://nightlies.apache.org/flink/flink-statefun-docs-stable/ (2023), accessed on 2023-05-18

Tardieu, Olivier, et al. "Reliable Actors with Retry Orchestration." Proceedings of the ACM on Programming Languages 7.PLDI (2023): 1293-1316.

Lightbend, Inc.: Kalix. https://www.kalix.io/ (2023), accessed: 2023-05-18

Moritz, Philipp, et al. "Ray: A distributed framework for emerging {AI} applications." 13th USENIX symposium on operating systems design and implementation (OSDI 18). 2018.

Sreekanti, Vikram, et al. "Cloudburst: Stateful functions-as-a-service." arXiv preprint arXiv:2001.04592 (2020).

Carbone, Paris, et al. "State management in Apache Flink®: consistent stateful distributed stream processing." Proceedings of the VLDB Endowment 10.12 (2017): 1718-1729.