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

ACM ISBN 978-1-4503-9909-8/22/12.
https://doi.org/10.1145/3563835.3567664

programming limitations. None of the dataflow streaming systems currently in use provides support for cyclic data dependencies or the ability to use nesting or delegation across different application entities (challenge III). This hinders the use of dataflow streaming systems beyond simple, non-cyclic application deployments.

In this paper, we present a principled way to compose applications, with strong guarantees and uncompromising flexibility. Portals takes some of the most used communication patterns known in actor systems (e.g., request-reply, futures) and applies them in a generalized dataflow model that is adapted to the modern needs of serverless programming (challenge III). Applications in Portals can be built dynamically by using and manipulating atomic streams of events, a transactional variant of distributed streams. Atomic streams are pre-discretized into distinct units of computation that enforce atomic processing by design, which we call *atoms* (challenge I). Portal's workflows are computational graphs of tasks that implement transactional microservices on atomic streams using dataflow semantics. An additional benefit of atomic streams is the ability to establish a serializable order across selected events on demand, a new capability that has not been explored in the past among dataflow systems.

Portals introduces several novel abstractions that greatly simplify the programming of decentralized services with complex communication patterns. To enable arbitrary nested composition and rich actor-like message-based communication schemes we propose *portals* and *reply streams*. The portal abstraction emulates direct, continuation-based event handling logic of requests across workflows through implicit atomic stream dependencies. *Atomic streams* enable the transactional processing. All together, this enables seamless compositionality, with dynamic scalability, and exactly-once processing guarantees. In this paper, we further provide an analysis of system needs and use cases that led to the creation of Portals , and present a set of microbenchmarks that showcase the benefits in scalability as well as the costs of atomic processing guarantees.

Contributions

- We present Portals, a programming model for composing decentralized data-driven services. Our approach combines key ideas from the stateful dataflow streaming model with the following capabilities (section 3, section 4):
  - Transactional processing guarantees across dataflow programs and serializable event order via the use of *atomic streams* and the *atomic processing contract* (subsection 4.1), and *serializable updates* (subsection 4.5).
  - Cyclic and intuitive data-parallel service composition through the use of *portals* and *reply streams* (subsection 4.4), and multi-dataflow deployments (subsection 4.2).
- We show use cases of Portals applications (section 7): Sagas, decentralized datalakes, complex event processing, and GDPR-support. We also demonstrate the generality of the Portals model by showing how we can model other distributed programming models in Portals (i.e., MapReduce, BSP, Pregel, Virtual Actors).
- We have implemented a prototype of the model and discuss its design (section 5), and evaluated the performance of our implementation (section 6).

2 Motivation and Challenges

Dataflow streaming systems [3, 12, 42], actor systems [5, 9, 15, 25, 33, 41], and stateful serverless systems [7, 20] have proven to be of great utility as witnessed by their widespread use. The programming technologies in use today, however, do not address all of the following required challenges (Table 1).

2.1 Exactly-Once Processing Guarantees Across Workflows

Exactly-once processing greatly simplifies the programming model for the end user, and is typically expected of stateful serverless systems (*e.g.,* [8, 20]). The challenging aspect of providing exactly-once processing, is that large ecosystems of microservices typically span different deployments and different systems (decentralized, Table 1), over which we need to provide end-to-end guarantees.

Dataflow systems [3, 12, 42] can provide exactly-once-processing guarantees, but not for the composition of multiple decentralized dataflow pipelines. End-to-end guarantees over multiple pipelines are non-trivial and usually tackled via externally connected persistent queues, such as Kafka [51] or Pravega [53]. Actor systems, on the other hand, offer no built-in processing guarantees, making the problem of exactly-once-processing a developer responsibility. Common approaches to providing end-to-end exactly-once processing guarantees on an actor system such as Erlang [5] or Akka [33] typically fall back to logging and filtering. For example, most actor applications rely on event-sourcing [21] the actors, which also entails deduplicating channels between the actors. In contrast, stateful serverless provides decentralized exactly-once processing [7]. The problem becomes difficult to manage due to cyclic dependencies that may be formed outside of the system. That is, certain fault-tolerance protocols that only work on acyclic dependency graphs may no longer work [11], and other protocols may result in unbounded rollbacks [18].

We propose *Atomic Streams* as a principled solution for decentralized exactly-once processing (subsection 4.1). The core idea is that the *atomic processing contract* enforces end-to-end fault tolerance over participating applications by imposed transactional processing.

Table 1. Feature comparison across existing distributed programming models and Portals.

	Distributed Programming Patterns					Guarantees		Distributed Execution		
	Cyclic Dependencies	Dynamic Communication Topology	Dataflow Composition	Typed Communication	Request/Reply with Futures	Exactly-once Processing	Serializable Updates	Decentralized 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
Portals	X	X	X	X	X	X	X	X	X	X*

* Supported with restrictions

2.2 Dynamic Scalability

The ability to scale up or down according to demand is critical for stateful serverless. For this, we consider two forms of parallelism: task parallelism and data parallelism. Task parallelism executes different tasks in parallel, whereas data parallelism executes the same task on partitions of data in parallel. Dataflow [3, 12, 42] and stateful serverless [7, 16, 20] systems have good support for data parallelism and task parallelism (with restrictions as they cannot spawn new tasks dynamically), as the data is typically partitioned over some key, and the pipeline is typically composed of tasks. Actor systems [5, 9, 15, 25, 33, 41], on the other hand, do not offer any form of data parallelism, as the actor instance is usually not defined over some range of keys.

We propose extensions to the dataflow streaming model (section 4), so that we can harness its inherent data- and task-parallelism.

2.3 Composition and Intuitive Programming Semantics

Microservice composition is a common development pattern in stateful serverless [46]. Common microservice patterns require support for cyclic dependencies, for example, request-response communication, transactions, iterations, etc. Microservice deployments also requires support for dynamic communication topology, as the services evolve over time.

Contrary to actor systems, dataflow systems [3, 12, 42] cannot handle cycles, dynamically evolving or multi-pipeline topologies. In the middle ground between these, stateful serverless systems aim to serve both purposes, yet, with known serious restrictions for microservice composition [7]. We believe that dataflow-style composition is a particularly good fit for large-scale microservices. This style of composition is not well supported in the actor model [32], and stateful serverless models [7, 20].[1] Further, we see a need for request-reply communication with futures as an intuitive programming construct; typically supported by actor systems and stateful serverless systems, but not dataflow systems.

We propose *Portals* and *Reply Streams* as a streaming abstraction for request-reply interaction with futures (subsection 4.4); and cyclic dependencies and dynamic multi-workflow composition as part of the Portals programming model (subsection 4.2).

2.4 Serializable Updates

If we consider every stream update in the system as a transaction, it is often necessary to provide serial ordering guarantees across selected updates. For example, special system events (e.g., consistent snapshots [11], GDPR requests [49] for data deletion) need to impose such guarantees. Currently, serializable updates are not supported by distributed programming systems today. A naïve method to support this would be to enforce processing order-sensitive events in strict serial and non-concurrent fashion. However, this can impact the performance of dataflows.

We propose *serializable updates* as a user-specified feature for live consistent updates (subsection 4.5).

2.5 Complexities and Inconsistencies: An Example

The example in Figure 1 summarizes some of the major challenges presented so far in a typical decentralized dataflow service. Consider a service that pipelines operations of two different workflows A and B. This can be materialized with strong guarantees by combining two stream workflow applications (*e.g.,* using Flink) and a distributed log (*e.g.,* Kafka) for logged, replayable channels. However, complexities arise when we need to add a third service C to issue requests and process responses back from A and B. Since a new level of

[1] We have added it as supported with restrictions for actors because nesting and chaining futures (*dataflow futures*) allow for the construction of asynchronous process pipelines, adding support for dataflow style composition in actor languages [13, 14, 19].

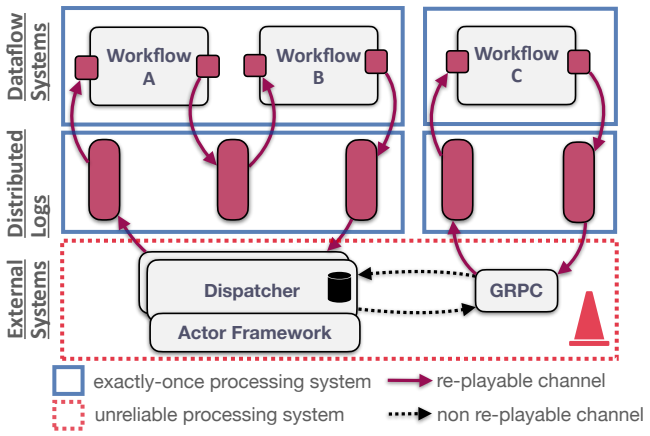


Figure 1. Decentralized composition with existing systems.

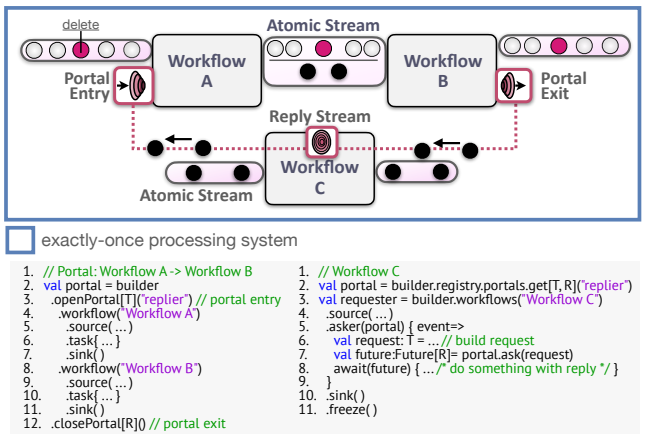


Figure 2. Decentralized composition with Portals.

indirection is necessary, the developer is forced to implement the handling of requests and filtering of responses outside the dataflow system, *e.g.,* in a custom RPC-based system using an actor framework. While the respective dataflow computations maintain exactly-once processing guarantees, the custom logic executed externally can break end-to-end transactional guarantees. Such problems hinder compositionality and raise trust issues while making development highly complex and unreliable.

3 Highlights: Portals

We present our highlights of the Portals programming model in Figure 2. The programming model consists of atomic streams, workflows, tasks, and portals. Figure 2 shows how Portals can seamlessly support the previous example from subsection 2.5 with end-to-end processing guarantees. Workflows A and B process some atomic stream, and also a portal stream, while workflow C is using *portals* to send a request and receive back a reply from A and B.

Atomic Streams. Atomic streams (subsection 4.1) are totally-ordered streams of atoms (atoms are sequences of events). Workflows consume and produce atomic streams. The atomic streams enable the *decentralized exactly-once processing* across workflows, as they impose the *atomic processing contract* on all workflows: the workflow must process one atom at a time (in the order of the atomic stream), and do this transactionally. This means that Workflow A must process one atom at a time, and then transactionally commit to the produced atom, before Workflow B can process the atom. Atomic streams are used to connect workflows together (even to form cycles), and come with many more features such as multi-lane composites (between workflows A and B), and atomic generators, sequencers and splitters. The atomic streams are a principled approach to end-to-end exactly-once-processing.

Workflows. Workflows (subsection 4.2) allow for dataflow style composition of typed task graphs (see the code examples). In addition, they allow for advanced composition of tasks (subsection 4.3), see *e.g.,* the code snippet for Workflow C. The Portals model can dynamically add new workflows. For example, Workflow C could be launched at a later point in time and connect to the existing portal. The workflows provide seamless compositionality and scalability.

Portals. The portal stream (subsection 4.4) spans from a portal entry to a portal exit, the entry is where requests come in and the exit is where the replies are consumed. Workflow C is connected to the portal stream which spans over A and B. In doing this, Workflow C can ask the portal, and await the reply from the portal. The portal receives the requests at the portal entry, which is forwarded to A and B for processing, and the reply is finally consumed by the portal exit and sent back to the requester. We provide a convenient future API as shown in the code snippet for Workflow C. The portals abstraction enables the intuitive composition of workflows with futures.

Serializable Updates. The serializable updates allow performing live consistent updates on systems. We can define serializable updates (subsection 4.5) as operations executed last of their respective dedicated atoms. In the figure, the red atom has a delete event. This event is then executed at the end of processing the red atom. We can ensure that no events are reordered over this execution, as the atoms cannot be reordered on the atomic streams.

Key Applications. We find that the pain point in existing scalable systems is the combination of exactly-once processing, dataflow composition, cycles/request-reply communication, and dynamic communication topology; as argued in section 2, no system exists that solves this. The Portals model provides a unifying solution to this, and enables applications that require a combination of the discussed features to be implemented. One such concrete use case is presented in subsection 7.1. If implemented with existing

systems, this requires much manual labour (subsection 2.5). Instead, we suggest, using the Portals system would lead to less development effort and more reliable applications as these features are automatically provided.

4 Programming Model

This section gives a more detailed overview of the highlights in the Portals programming model.

Portals applications comprise atomic streams, workflows, and portals. The general flow and structure of an application is described in the following listing. To build a portals application we first create a builder, and pass it the name of the application as a string literal. From the builder we can add workflows, portals, sequencers, etc. We can also look up external portals, streams, workflows, via the registry. Finally, we can assemble the PortalsApplication by calling build() on the builder. Next, the application is launched using a system.

```
val builder = Builders.application("appName")
val workflowBuilder = builder
    .workflows[String, Int]("exampleWorkflow")
...
val externalWorkflow = builder
    .registry.workflows.get[T, R]("externalWorkflowName")
...
val app = builder.build()
val system = Systems.local()
system.launch(app)
```

Building and launching the application triggers a *well-formedness* check on the application. The checks include naming collisions, and verifying the existence of external dependencies.

4.1 Atomic Streams

Atomic Streams are the core data type of the Portals model (Figure 3). An atomic stream is a totally ordered, distributed, immutable stream of atoms. Processors produce atomic streams, and consumers consume them. The atomic processing contract says that the consumer must always consume the whole atom and process the whole atom before consuming and processing the next atom. This allows us to safely compose together producers and consumers, *i.e.,* microservices, whilst preserving end-to-end fault-tolerance across the microservices.

Atoms. Each atom contains a finite distributed stream of events, with each event belonging to exactly one atom. The events, in turn, are associated to some contextual implicit key. That is, from each event we can extract some key. Events and atoms are distinct with different purposes. Whereas events represent the input in form of data that is consumed and processed; atoms represent a group of events that are to be processed together atomically (which is used for the exactly-once aspect). Events within an atom, in contrast, may be processed concurrently. This distinction is useful as

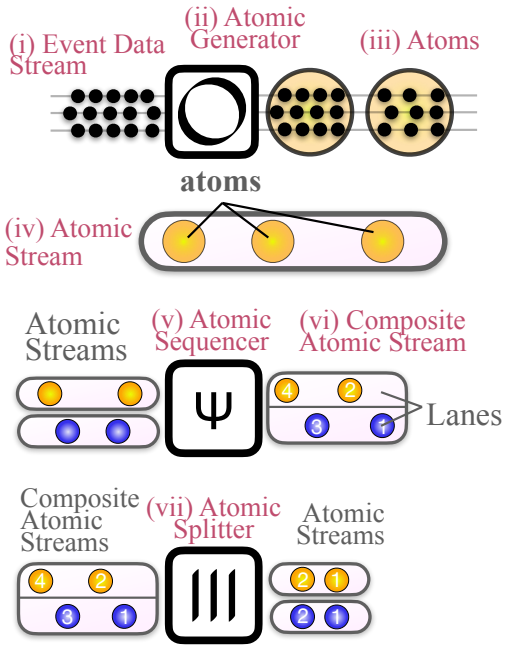


Figure 3. Basic atomic processing elements and operators.

there is a cost associated with atoms. Smaller atoms have a higher relative cost, and this can be adapted depending on the execution environment.

Atomic Streams. An atomic stream is a totally ordered, distributed, immutable stream of atoms. The atomic stream AtomicStream[T] transports atoms of type Atom[T] with events of type T. The atomic streams are distributed over the implicit key of the events. Additionally, within the atom, the sequence of events within an atom are partially ordered over this key. Atomic streams are created from the outputs of generators, sequencers, workflows, etc. They stop when the stream is completed.

Multi-Lane Composite Atomic Streams. A composite atomic stream consists of multiple lanes, one lane for each atomic stream. However, the atoms are still in a totally-ordered sequence across the lanes. The type of the composite atomic stream is a composite of the individual types, *e.g.,* AtomicStream2[T, U] is the composite of AtomicStream[T] and AtomicStream[U].

Atomic Sequencers. The *Atomic Sequencer* combines (or, sequences) inputs from one or more atomic streams. This operation creates a new atomic stream. The sequencer can combine atoms by some strategy, *e.g.,* random, round-robin, zip, etc. The composite strategy takes two atomic streams and produces the composite thereof, from *e.g.,* AtomicStream[T] and AtomicStream[U] we get AtomicStream2[T, U].

It is allowed to mutate the inputs of an atomic sequencer. This capability allows to statically define atomic streams and use them dynamically in a publish-subscribe fashion. Note that the output still remains immutable and totally ordered.

Atomic Splitters. Atomic splitters are used for splitting a composite atomic stream into separate atomic streams. The simplest form of a split is to take an AtomicStream2[T, U] and split it into two separate atomic streams: AtomicStream[T] and AtomicStream[U]. The two new atomic streams are totally-ordered, but there is no total order across these two streams.

Atomic Generators. Atomic generators are used for generating new atomic streams from sources other than existing atomic streams. In practice, generators can be used to connect from an external system; or for running batch jobs, by starting a new generator.

Connecting Atomic Streams. Atomic streams can be connected by connecting an atomic stream to an atomic sequencer. In doing so, the consuming atomic sequencer starts consuming atoms from the atomic stream. Note that it does not consume the full history of atoms, rather, starts consuming atoms as they are produced.

```
...
val generator = builder.generators.fromRange(0, 1024, 128)
val sequencer = builder.sequencers.random[Int]("seqName")
builder.connections.connect(generator.stream, sequencer)
...
```

Guarantees and Contracts. Atomic streams provide various guarantees and contracts.

Guarantee 4.1 (Atomic Total Order). *The atoms $a_1, a_2, \ldots$ in an atomic stream A are totally ordered. That is, the atomic stream is a totally ordered sequence $A = a_1, a_2, \ldots$, and there is some atomic ordering relation over this atomic stream $<_A$ that orders any two atoms $a_i \in A$ and $a_j \in A$ so that either $a_i <_A a_j$ or $a_j <_A a_i$.*

This, together with the partial order of events within an atom, have the following implication.

Guarantee 4.2 (Atomic Event Order). *Events are partially ordered within an atom, and totally-ordered across atoms of the same stream. That is, for some atomic stream A, if two events are in two different atoms of the same atomic stream,* i.e., *$e_i \in a_i$ and $e_j \in a_j$ with $a_i \neq a_j$ and $a_i, a_j \in A$, then either $e_i <_A e_j$ or $e_j <_A e_i$. If two events are in the same atom a, then they are ordered according to the partial order $<_a$ within that atom.*

Atomic streams are fault-tolerant. We capture the fault-tolerance through the following definition.

Guarantee 4.3 (Atomic Fault-Tolerance). *Atomic streams are fault-tolerant (exactly-once processing). That is, any observed atom order $A = a_1, a_2, \ldots, a_n$ is a prefix of any subsequently observed atom order $A' = a_1, a_2, \ldots, a_n, \ldots, a_m$.*

To process an atomic stream, the consumer must adhere to the atomic processing contract. Atomic sequencers and workflows are processors, they both consume and produce atomic streams.

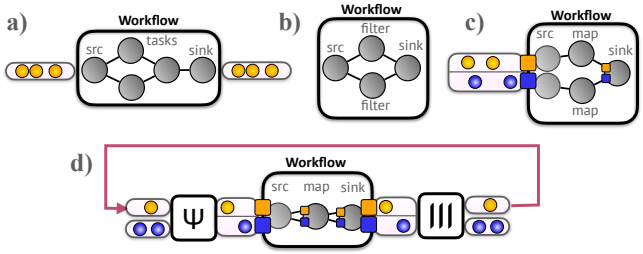


Figure 4. Workflow composition examples in Portals.

Contract 4.4 (Atomic Processing Contract). *The consumer must always consume and process the whole atom before consuming and processing the next atom. The consumption must follow the order as given by the* atomic order. *The producer must produce an atom after consuming and processing an atom, and before consuming and processing the subsequent atom. This produced atom production order will establish the atomic order.*

4.2 Workflows

A *workflow* (Figure 4) consumes an atomic stream and produces an atomic stream. The processing happens over an acyclic graph of sources, tasks, and sinks. For this the workflow needs to conform to the *atomic processing contract* (Contract 4.4)). That is, the workflow must always consume and process the atoms one atom at a time, *i.e.,* no two atoms are to be processed concurrently. Nesting workflows is not supported.

Workflows. A workflow is a directed acyclic graph (DAG) of *sources*, *tasks*, and *sinks* (Figure 4.a).[2] The workflow has one source which is the ingestion point of events to the workflow. Tasks are stateful operators that consume events and produce events. The workflow has one sink which is the egress point of the workflow.

A workflow has an input type and an output type Workflow[T, U], corresponding to the source and sink types. To consume an atomic stream, the types of the workflow need to match the contravariance and covariance of the input and output type, respectively.

Creating, Starting, and Stopping Workflows. To build a workflow we use the builder pattern for the following illustrations. The string literal passed to the workflow builder factory is the workflow name, which can later be used to find a workflow in the workflow registry. The following example shows a workflow that consists of a source with type String which receives as an argument a stream of the corresponding type, followed by a map invocation that transforms strings to integers, and a sink. Freezing a workflow blocks it from further modifications. To execute a workflow it needs to be

[2] By convention, we graphically depict workflows with mutiple sources and multiple sinks (see, *e.g.,* Figure 4.c), this corresponds to the single source and sink representation with multiple ports.

launched on some system, which in turn returns a reference to the atomic stream that the workflow produces (and optionally the atomic sequencer it consumes). Workflows continue processing until the atomic stream it consumes has completed.

```
val stream: AtomicStream[String] = ...
val workflow = Workflows.builder[String, Int]("workflowName")
  .source[String]( stream )
  .map { s => s.length() }
  .sink()
  .freeze()
```

Ports. Workflows have a single input source and a single output sink. The sink and source may have multiple ports (Figure 4.c). To create a workflow with multiple input and output ports, we specify the port numbers, *e.g.,* source_1, source_2. The multi-port workflow consumes from a composite atomic stream with compatible lanes, and similarly, produces an atomic stream with compatible lanes (*i.e.,* the individual ports match the individual lanes).

Cycles Across Workflows. Cycles are not allowed within a workflow. However, cycles can be created across workflows, including cycles that go directly from a workflows sink back to the workflows source (Figure 4.d). Trans-workflow cycles are safe, as they are broken up by atoms: within the processing of one atom there are no cycles.

Combinators. The workflow builder supports combinators (prefixed by *with* that are later defined for the tasks). For example, the withWrapped wraps around the defined task to extend its behavior. In addition, the allWith combinator is applied to all sources, sinks, and tasks of a workflow. For example, the allWithWrapper, wraps around all sources, tasks, and sinks. This convenient notation can be used to inject some behavior for the entire workflow. For example, we could simply log every event on the workflow by the following.

```
val _ = workflowBuilder.allWithWrapper { wrapped => event =>
  log.info(event) // log all events
  wrapped(event) // execute the wrapped behavior
}
```

Guarantees. Workflows provide the following guarantees.

Guarantee 4.5 (Workflow Atomic Processing)**.** *The workflow adheres to the atomic processing contract. That is, the workflow processes one atom at a time, and no two atoms are processed concurrently.*

Guarantee 4.6 (Workflow Fault-Tolerance)**.** *A workflow processes an atom exactly once, and events may be replayed due to failures.*

Further, the workflow guarantees to respect the atomic event order, and process events in a per-key FIFO order internally on the tasks.

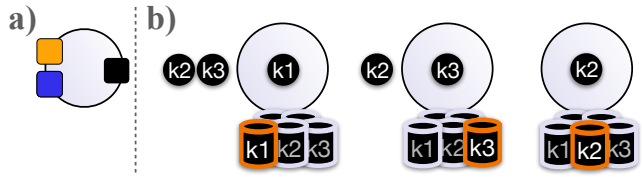


Figure 5. Tasks: a) task; b) task with disjoint state access.

Guarantee 4.7 (Workflow Event Processing)**.** *The workflow processes events of an atom in a per-key FIFO order on the source/sink/task graph.*

Lastly, the workflow processing needs to trigger the atom completion handlers on all tasks in the correct way.

Guarantee 4.8 (Workflow Atomic Handler Trigger)**.** *The workflow triggers on all sources/sinks/tasks the onAtomPreComplete handler as the last event before the atom has been processed to completion, and the onAtomPostComplete handler as the first event of the new atom.*

4.3 Tasks

Tasks are the core computational unit in our programming model. They allow us to describe the processing logic. For this we provide a suite of helpful abstractions, that can help build advanced and complex task logic.

Tasks. A task has one or more input ports, and one or more output ports (Figure 5). It processes one event at a time, may modify its state during processing, and produces zero or more output events. A Task[T, U] implements five event handlers. The onNext handler gets triggered with the next event to be processed. The onError and onComplete handlers get triggered when there is an error from an upstream task, or the upstream tasks have completed. The onAtomPreComplete handler is invoked directly before an atom has been fully processed to completion, the onAtomPostComplete directly after.

Task context. The task handlers are invoked with access to the TaskContext[T, U]. This gives access to side effects such as logging, emitting events, and accessing the external state of the task. The task invocation is done under some implicit key context. The state accessed by the task is scoped by this implicit key (see Figure 5.b), *i.e.,* the state access of one key is disjoint to the state access of a different key.

Task Factories. The Tasks factory methods are a convenient way to define tasks. They can be used to define a map, flatMap, filter operators, and more.

As an example, the following creates a processor behavior for a Task[String, Int]. The processor handles an event, and may modify state and emit zero or more events. It takes a function as a parameter that is used for handling the incoming events. The example shows how we can log the event, followed by emitting the length of the event.

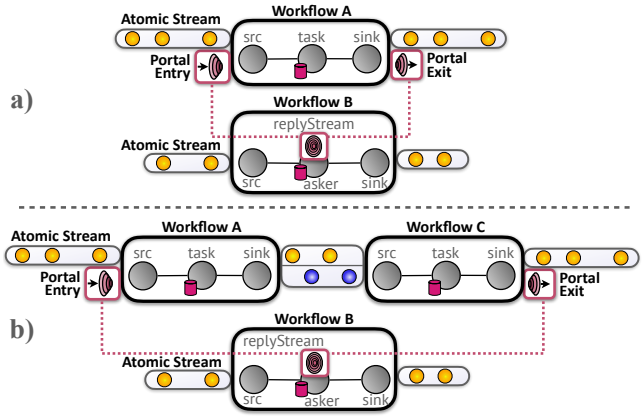


Figure 6. PORTALS reply streams: a) example with a requester and a replier; b) example with a workflow forwarding/delegating a reply stream.

```
Tasks.processor[String, Int] { event =>
  log.info(event)
  emit(event.length())
}
```

State. There are two types of state available, PerKeyState and PerTaskState. The PerKeyState is scoped by the invocation key, whereas the PerTaskState is scoped per task instance that is executing the task. It is convenient to create the typed state during the initialization phase of a task, as shown in the following example.

```
Tasks.init[...] {
  // create state with default values 0, and 1 resp.
  val perKeyState = TaskStates.perKey[Int](0)
  val perTaskState = TaskStates.perTask[Int](1)
  Tasks.map { ... /* do something useful with the state */ }
}
```

4.4 Portals and Reply Streams

Portals enable request-reply interactions. The requesting workflow can create requests. The replying workflow can reply to these requests. This interaction is distinct from the regular atomic streams and workflows, as the replies are only sent back to the requester (there may be multiple requesters). In contrast, events of the atomic stream are sent to all of the subscribers of the atomic stream. Furthermore, the portals request-reply style interaction is more similar to direct-style messaging, whereas atomic streams and workflows are more similar to a publish-subscribe type of interaction. It enables many patterns in microservices, for example, request-reply style communication, query streams, sagas, and more. Nesting portals is not supported.

Portals. A *portal* has two endpoints, the portal entry and the portal exit (Figure 6). The portal type encapsulates the type of the requests and the type of the response, Portal[T, R]. The endpoints span a portal stream, the requester's requests come in at the entry, the replies exit at the exit. The portal stream transports the atoms together with some contextual (not first-class) information about the sender. One, or many workflows, may operate, transform, and reply on the requests.

Replier, Creating Portals. The replier consumes the requests from the portal's entry (workflow A Figure 6). It may then operate and transform the requests to produce the final reply. The replier replies by emitting events that are consumed by the portal's exit.

The following creates a replier portal, by opening and closing the portal gates, and performing some transformation on the requests to produce the replies.

```
val replier: Portal[T, R] = builder
  .openPortal[T, R] ("portalName") // portal entry
  .map { ... } // create replies from requests
  .closePortal()  // portal exit
```

We also support the direct syntax of accessing a portal from a task by creating a *replier* task, which takes two handlers, one for regular events from the consumed atomic stream, and one for the requests from the portal.

```
val portal = builder.portals.portal[T, R]("portalName")
val workflow = builder.workflows[I, O]("replyingWorkflow")
  .source( ... )
  .replier(portal)
    { ... /* handle regular data events */}
    { ... /* handle portal requests */}
  .sink()
```

Requester, Ask and Await. A requester can connect to the portal from within an asker task via the ask and await API. The asker task takes two sets of parameters, the first are the list of portals that it wants to connect to, the second is the event handler logic for the asker. The requester can obtain a portal reference (either through the registry, or some other way). With this reference, the requester can ask the portal (requesting some value), and await the reply (workflow B Figure 6).

Asking the portal (portal.ask(request)) returns a future and sends a request to the portal. The replier, then, consumes the request from the portal entry, and produces a reply that is consumed by the portal exit. This, in turn, completes the future at the requester. The requester can await for the future completion, and will upon completion continue operating by executing the continuation closure. The following example shows how we can create a requester that binds to a portal.

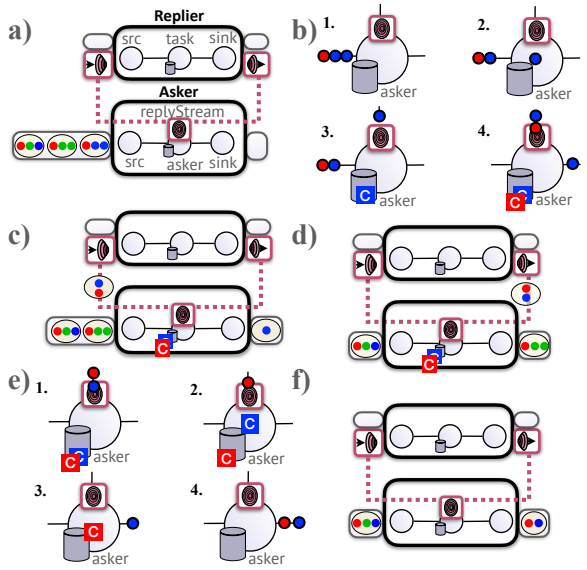


Figure 7. Await semantics in Portals. The example shows the interaction between an asking and replying workflow, with highlighted await semantics on the asking task. The events are coloured by their keys.

```
val portal = builder.registry.portals.get[T, R]("replier")
val requester = builder.workflows[I, O]("workflowName")
  .source( ... )
  .asker(portal) { event =>
    val request: T = ... // build request
    val future: Future[R] = portal.ask(request)
    await(future) { ... /* continuation with completed future */ }
  }
  .sink()
```

Await Semantics. The await semantics of the Portals model are illustrated in Figure 7. (a) The events in the figure are coloured by the respective event's key. (b) The *asker* task processes the events of an atom. The action of processing an event that triggers an ask creates the following side-effects (b.3): i) a request event for the replier, and ii) a non-completed future; thus, invocations of ask are non-blocking. By using await, the asking task provides the continuation as a closure, which is saved to the implicit task context for the given key. Subsequent events are executed as normal. (c) The request is sent to the replier, who processes the request and (d) responds with a reply, during this time the asker workflow may process other atoms. (e) The asker receives the reply, completes the future, and executes the continuation. While the continuation executes, the task does not process any events for the same key to avoid data races on the per-key state. (f) The workflow eventually produces an atom with two events and has no pending continuations.

The await semantics introduce an alternative form of operation to the regular task execution. An asking task may send requests, and receive replies that are executed at a later point in time. The sent requests and received replies are two different events that are also in two different atoms. Each takes one processing loop cycle in a task, and this form corresponds to a regular atomic processing program that consumes and produces atoms.

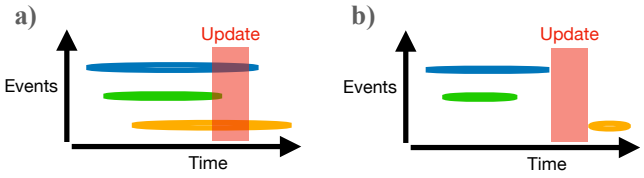


Figure 8. Serializable updates: a) non-update-serial schedule; b) update-serial schedule.

Forwarding Portal Streams. It is possible to build a portal stream that spans over multiple workflows. For example (Figure 6), workflow A can forward all requests to Workflow C, which then connects to the portal exit and replies.

4.5 Serializable Updates

Serializable updates are consistent updates on the system [49]. The observable effect of a serializable update is equivalent to that of an operation applied to the system atomically and in isolation, with no other commands or operations concurrently to this operation (Figure 8).

Writing Serializable Updates. To define a serializable update we need: 1) some update function that is executed on a task's state; 2) and some way of triggering this update function. The user can define what the update function should do on the task's state. To trigger the update we utilize the onAtomComplete handler. The idea behind this is that the invocation of the onAtomComplete handler is a serializable/atomic event on the workflow.

```
// wrap all tasks (wrapped) to register the update
.allWithWrapper { event => wrapped => event match
    case CLEAR_STATE =>
        state.set(CLEAR_STATE, 1) // set flag
        emit(event) // broadcast CLEAR_STATE downstream
    case _ => wrapped(event) // wrapped handles other events
}
// execute the update on invocation of `onAtomPreComplete`
.allWithOnAtomPreComplete {
    if state.get(CLEAR_STATE).isdefined then
        state.clear()
}
```

The example shows how we can define a serializable update for all tasks within a workflow with the allWithWrapper. When the CLEAR_STATE operation is submitted and processed, all tasks will set a flag to then clear the state during the next invocation of the onAtomPreComplete handler.

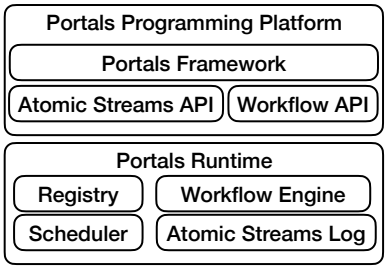


Figure 9. PORTALS architecture.

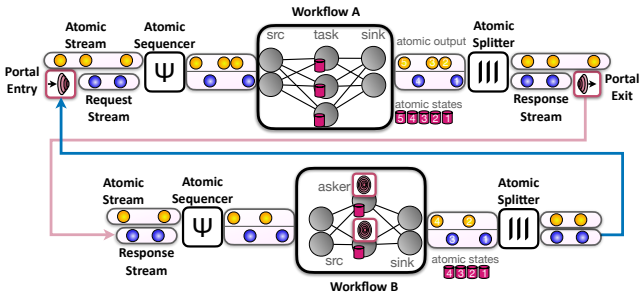


Figure 10. Physical plan of workflows in Figure 6.

Guarantee 4.9 (Serializable Update)**.** *A serializable update is an update with observable effect equivalent to that of an update applied atomically and in isolation to the system. That is, either strictly before or after any other processed event in the system.*

4.6 Distributed Semantics

The *atomic execution model* defines how workflows and atomic streams are executed in PORTALS. The model can be defined at the granularity of atoms, and proceeds by taking steps over atoms. For each step, a workflow is chosen, and executes the atom that is next in the queue on the atomic stream, this in turn produces a new atom, which is added at the end of the queue of the produced atomic stream. The onAtomPreComplete is invoked at the end of atom completion, and the onAtomPostComplete handler is invoked right at the beginnning of the next atom being processed. Internally, the workflow processes each atom whilst respecting the per-key FIFO order of the events. The distributed execution model guarantees *exactly-once processing*.

5 Design and Implementation

We have designed and implemented a local prototype of PORTALS with a subset of the functionality (subsection 5.5).[3] We will discuss the system architecture on a high-level, and deep dive into the implementation of atomic streams, portals, and ensuring end-to-end exactly-once-processing guarantees.

5.1 PORTALS System Architecture

On a high level, the system architecture consists of the portals programming platform and the portals runtime (Figure 9). The main components of the portals runtime are the scheduler; the registry; the atomic streams; the workflows; and the portals. The atomic streams are used to glue the processing units together. The implementation leverages the capabilities of distributed transactional logs such as Apache Kafka [51], and Pravega [53]. The workflow engine implements the runtime capabilities of executing the processing units and the workflows. The workflows are to be implement with similar techniques as other dataflow streaming systems [3, 11, 42], enabling the scalable data-parallel execution. The registry is used for service discovery and name management. We plan to implement it using replicated state machines to ensure high availability, using *e.g.,* Zookeeper [26], or similar systems. The scheduler needs to implement scheduling logic for executing the streams and workflows, and monitoring logic for monitoring the system and making decisions on live reconfiguration of the deployment.

5.2 Atomic Streams

The atomic streams expose a transactional interface in the implementation. A producer can begin building a new atom, pre-commit updates to the new atom (or aborting to the new atom), and finally commit the atom. The commit operation is idempotent. The consumer interface allows the consumer to receive an atom, and to replay from a certain atom-id. The interface is similar to both Kafka and Pravega, the implementation is a matter of an abstraction layer. For Kafka this abstraction entails: building the atom by running beginTransaction() followed by send()ing the events of the atom; pre-commiting by flush()ing all events; and committing by running commitTransaction(). With Kafka, however, some care must be taken as it does not fully provide a two-phase commit interface out-of-the-box [52]. We intend to use the underlying distributed log system (Kafka, Pravega) solely for committing and transmitting the atoms, the processing of workflows is to be executed separately.

5.3 Portals

Implementing portals will require the addition of portal streams, and its dual, the reply streams. The portal streams transport the contextual information of the request, so that the reply can be sent back to the correct requester. Similarly, the reply streams require transporting contextual information about the request, so that the reply can be matched with the request. The portal streams and reply streams are implemented on top of atomic streams. To exemplify this, the logical representation of a portal corresponds to the physical representation Figure 10.

[3] https://github.com/portals-project/portals

We can guarantee the completion of the reply future (either to some value, or to no value), using the atomic processing contract. Because the portal processes the atom exactly once, and this creates one new atom with replies, the requester processing the reply atom can complete all futures with values (if they were in the reply), or otherwise complete the futures with no value (if they were not in the reply).

5.4 Exactly-Once Processing

We introduced the notion of exactly-once processing guarantees in section 2. In this section, we will overview how Portals achieves these guarantees per workflow, and discuss how this naturally generalizes to end-to-end exactly-once (atomic) processing guarantees across workflows and external systems.

Exactly-once processing generally refers to the observable behavior of a system mimicking a system that is failure-free (events are delivered and processed exactly-once, no computing nodes crash). We can describe the exactly-once processing behavior of Portals through the observable effects of atomic streams: 1) an observed prefix of an atomic stream is immutable (atomic streams cannot be changed once committed), and 2) atoms are produced by an observably failure-free execution of the producer. [4] In other words, (1) ensures that once an atom is observable, it will always remain observable, whereas (2) ensures that only failure-free executions are observable.

We should note that the definition does not state that the events of an atom may be processed at most once. Indeed, if the processing of an atom fails, the internal events of the atom might be processed again. This may be an issue if user-defined-functions cause external side-effects, such as sending requests to some external database.[5] The end-to-end guarantees do not apply to such code. Instead, we recommend connecting external services to the transactional APIs of Atomic Streams and Generators for end-to-end guarantees (subsubsection 5.4.3). These are the only restrictions we put on user-defined functions; notably, they are even allowed to be non-deterministic.

5.4.1 Atomic Processing in Workflows. Workflows are responsible for processing each incoming atom in a transactional fashion. The full computation of an atom generates

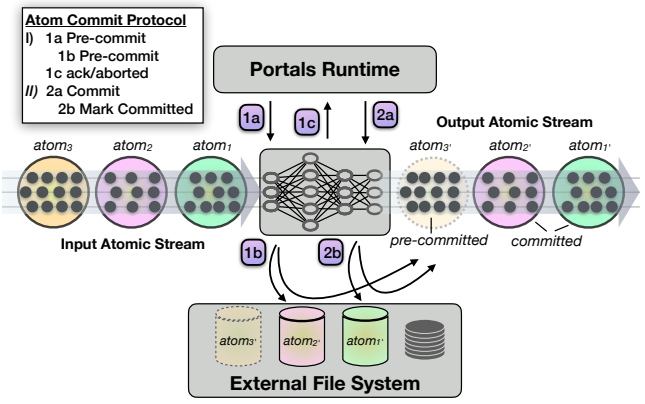


Figure 11. Asynchronous Atomic Commit Protocol, adapted from [10].

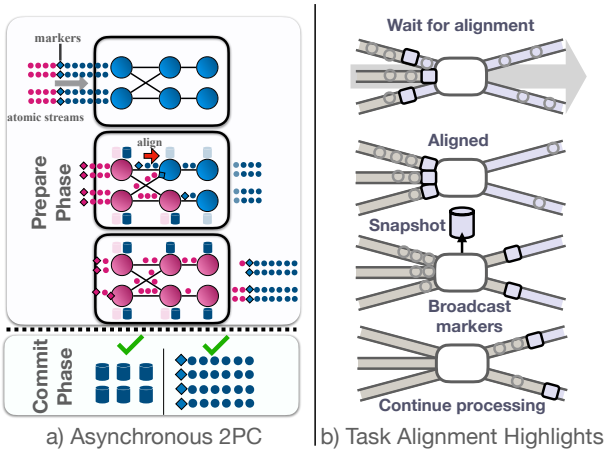


Figure 12. Atomic Commit and Alignment Protocol.

two side-effects: i) new workflow task states and ii) a new output atom. Both of these side-effects need to be committed before the processing of the next atom. A naïve implementation would be to ingest and commit each input atom through the workflow in a micro-batch fashion, similarly to Apache Spark's Discretized Streams model [58]. Despite its simplicity, this solution can suffer from high latency penalties and leave most task instances of the workflow idle until all state and output changes have been reliably replicated. Instead, in Portals we adopt a variant of the asynchronous two-phase commit protocol of Apache Flink [11]. When an atomic stream is ingested by a workflow, its computation is instrumented using a series of transactions, one per input atom (Figure 11).

I) Phase 1 (Pre-commit): All tasks of the workflow process input records of an atom and capture its effects. There are two types of captured effects: task state updates, and a new output atom. The effects are aligned to represent the view of the effects caused by an atom through an alignment protocol [11], summarised in Figure 12. During the prepare

[4] We can define an *observably failure-free producer* by the produced atoms of this producer being *equivalent* to *some* failure-free producer. Two atoms are considered equivalent if they contain the same events with the same order.

[5] If an event is replayed, it may issue the same operation twice to the external database. Clearly, this is not an end-to-end exactly-once semantics, as the operation was issued twice. We do not guarantee end-to-end exactly-once to cover external resources that are accessed from within the user-defined functions. Although, we would like to note here that the Portals system will preserve the exactly-once guarantees internally within the system regardless, the non-exactly-once external side-effects can be seen as a form of non-deterministic behavior.

phase of the atomic commit protocol the records of an atom are streamed through the workflow and the output is logged. The end boundaries of each atom in an atomic stream are signified with special marker events which trigger a local snapshotting action on each task before being sent further (Figure 12). Tasks with multiple inputs such as composite stream sources synchronize their input so that all markers are processed in a task before proceeding with the records of the next atom. This is also known as the "alignment" phase. Once snapshotting is complete, all state copies and logged outputs are replicated asynchronously (in a pre-commit) on respective replicated file systems and output atomic streams.

Phase 1 either completes or fails. Completion is signaled when all tasks have (asynchronously) notified the Portals runtime that their effects are captured, whereas failure is detected once any task fails. In case of a failure the system would roll back and redeploy tasks to re-execute from the latest atom and the latest state. In case of no failures it moves to Phase 2. Completing the first phase ensures property (2) of exactly-once processing: only failure-free executions are observable.

II) Phase 2 (Commit): The portals runtime has recorded all side-effects of an atom, this includes the effects on state, and a new output atom. What remains is to make these effects visible to the system outside of the workflow. This is done by the Portals runtime through signalling the output atomic stream to finalize the commit of the pre-committed output, and similarly to commit to the pre-committed state output. Notice that during this phase, possible failures may occur, but such failures will not result in a roll-back. A failure might at-worst yield more than one signal of completion, an operation that is idempotent by nature. Completing the second phase ensures property (1) of exactly-once processing: once an atom is observable, it will always remain observable.

5.4.2 Atomic Processing across Workflows.

Each workflow using the above protocol is guaranteed to adhere to the exactly-once processing guarantees, by processing atomic streams, and producing atomic streams as its output (Figure 13). Other workflows may in turn ingest these atomic streams. As we know, each workflow preserves the exactly-once processing guarantees under the assumption that the input is an atomic stream. Thus the composition preserves the exactly-once processing guarantees across all workflows.

5.4.3 End-to-End Atomic Processing across Workflows and External Systems:

Portals exposes a transactional producer and consumer atomic streams interface to external systems, as a means to support end-to-end exactly-once processing to also cover these external systems (Figure 13). The exposed producer and consumer interfaces adhere to the transactional "pre-commit" and "commit" phases over atoms in an atomic stream. For example, the consumer interface allows for *replaying* atoms (if needed for a roll-back recovery). The producer interface further allows the external system to *pre-commit* to an atom, and to *commit* to a pre-committed atom, in addition to a *rollback* method to communicate that a non-committed atom can be discarded. If the external system uses this interface in its intended way, then we can ensure the end-to-end guarantees to include the external system. This enables external systems to transactionally consume and produce atomic streams, adhering to the atomic processing contract. Through transitivity over atomic streams, the end-to-end guarantees will cover end-to-end systems connected via atomic streams.

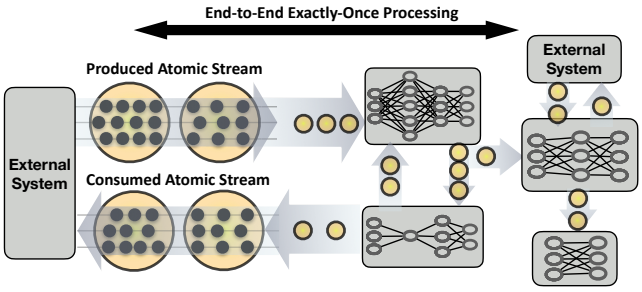


Figure 13. End-to-end exactly-once processing via Atomic Streams.

5.5 Prototype Implementation

We have implemented a local runtime prototype with a subset of the functionalities of Portals in Scala 3 [60]. The implementation consists of an asynchronous multi-threaded runtime that implements the alignment protocol in subsubsection 5.4.1, implemented on top of the Akka Actor framework [33]. The implementation produces an AST of the Portals Application which it checks for well-formedness, and upon launch is submitted to the runtime. The runtime translates the Portals concepts into actors, which entails implementing the alignment protocol and executing the encapsulating behavior on sources, tasks, etc. In total, the entire local runtime system is less than 600 lines of code. We have made it so the runtime eliminates empty atoms at the sequencer, as otherwise cycles cause the empty atoms to be circulated ad infinitum and accumulated over time.

6 Evaluation

We have evaluated the local Portals prototype runtimes structured around three questions. The code for the prototype implementation, and the executed benchmarks are available online.[6] The experiments were run on a MacBook Pro (14-inch 2021, macOS 12.5.1) with an Apple M1 Pro chip and 16 GB of memory, 8 cores (6 performance, 2 efficiency, no hyper-threading). The experiments were run using Scala version 3.2.0 and Java version 17 for which 1 GB java heap size was allocated. Each individual benchmark followed the same procedure. All configuration of parameters would be

[6] https://github.com/portals-project/portals-benchmarks

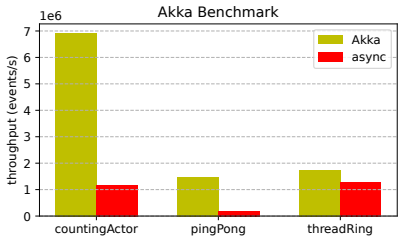


Figure 14. SAVINA Microbenchmarks. Higher is better.

run in a sequence. For each configuration of parameters we would first run five warmup runs, followed by five timed runs. The timed runs were then averaged, and here we report the average as the number of events processed per second, higher is better.

Q1: How does the Portals implementation compare to other distributed programming frameworks such as Akka? To answer this question we have chosen three microbenchmarks from the Savina Actor Benchmark Suite [27].

- *PingPong:* Two actors that repeatedly send and reply a Ping message back and forth.
- *ThreadRing:* A ring of actors (in our case, a cyclic workflow with a chain of tasks), each actor decrements a token and forwards it until it has been decremented to zero.
- *CountingActor:* A generator actor sends messages to a receiving actor who increments a counter upon receiving a message.

For this we compare two systems:

- *async:* The async implementation implements the alignment protocol as described in section 5. This system is implemented on top of Akka Actors [33].
- *Akka:* The Akka implementation corresponds to the microbenchmarks implemented directly with actors in the Akka Actor framework [33].

The results in Figure 14 show that there is a high relative overhead of using the Portals implementation in comparison to the Akka Actor implementation. For the CountingActor, PingPong, and ThreadRing microbenchmarks, we measured throughput 6.9 mil, 1.5 mil, 1.7 mil events/s for Akka, and 1.2 mil, 180k, 1.3 mil events/s for the async runtime. The throughput is a factor 6.0, 8.3, 1.4 times higher for the Akka implementation compared to our async runtime. We can explain part of this on the additional structure required for the portals programming model. The PingPong example, for example, is compiled down to 8 actors (2 sequencers, 2 sources, 2 sinks, 2 tasks); compared to the Akka implementation which consists of 2 actors (2 pingerpongers). One PingPong roundtrip, then, goes over 8 actors in the async runtime, whereas it only goes over 2 actors in the Akka runtime. This would explain why we have a factor 4 in reduced performance. This relative difference in structure-size is smaller in the ThreadRing example that spans 128 actors/tasks. For this benchmark we still observe a factor 1.4 of performance difference. We would think that this performance difference is mostly due to the additional work associated with the alignment protocol on each task/actor within the workflow, and the virtualization of our sequencers, sources, tasks etc. on actors.

Q2: How large is the atom alignment overhead? In order to study the atom alignment overhead, we have implemented two additional runtimes as baselines:

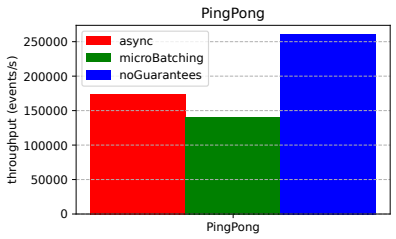
- *micro-batching:* The micro-batching runtime processes atoms sequentially. That is, it blocks subsequent atoms from being processed until the whole atom has been processed by the workflow. This buffering and blocking happens at the sources, internally there is no buffering. Still, the protocol uses a method similar to alignment for detecting when an atom has been processed to completion.
- *noguarantees:* The noguarantees runtime does not implement the atomic processing contract. All events are processed eagerly, there is no buffering or alignment in this runtime. The physical actor-topology of programs in the noguarantees runtime correspond to the async runtime, this lets us assess the relative cost of the alignment protocol.

We have also added three more benchmarks.

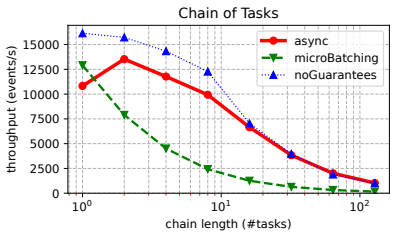
- *ChainOfTask:* A workflow with a chain of tasks, for which each task performs some non-trivial work.
- *AtomAlignment:* A workflow with two columns of three tasks. The first column is fully connected to the second column. The reason for this choice is that the alignment protocol is exercised on the inner nodes of the second column. Additionally, only the top row of tasks forward events, so that only one event is produced per consumed event. The WithWork variant performs some non-trivial work on each task when processing an event.
- *NEXMark Benchmark:* The NEXMark benchmark [55] is a set of queries on synthetic data streams for auctions. We have implemented the first four queries.

The results are presented in Figure 15. The relative difference of throughput between the async and noguarantees runtime in the PingPong benchmark is 1.5 (Figure 15a). That is, we could account 50% of the overhead to the additional work related to the alignment protocol.

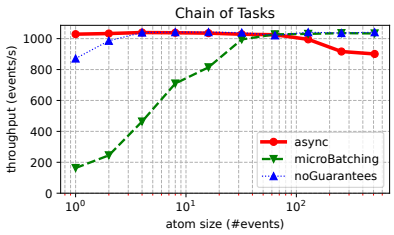
There are some general trends which we would like to highlight. The atom alignment overhead is larger for smaller atom sizes, as the relative work per event is larger, and this relative cost reduces when the atom size increases. We can see this for small atoms in the CountingActor benchmark (Figure 15d), Chain of Tasks benchmark (Figure 15c), and in the AtomAlignment (Figure 15e) benchmark, for which



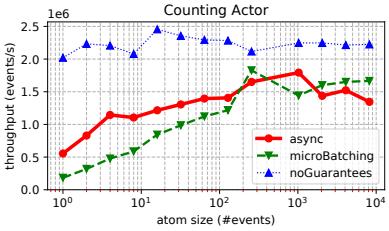
(a) PingPong Microbenchmark.



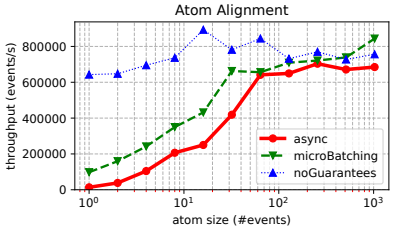
(b) Chain of Tasks over the length of the chain.



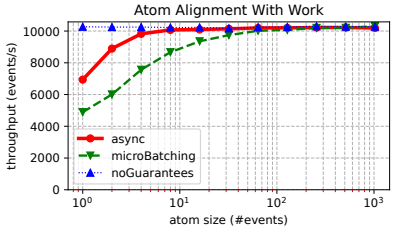
(c) Chain of Tasks over the atom size.



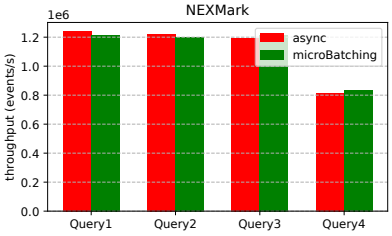
(d) CountingActor over the atom size.



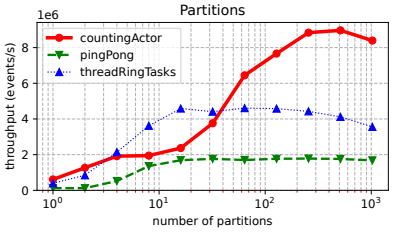
(e) Atom Alignment microbenchmark.



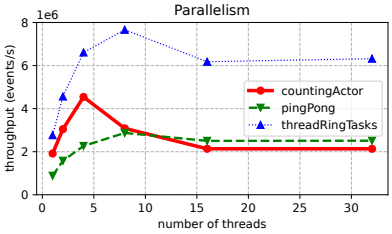
(f) Atom Alignment microbenchmark with Work.



(g) First four queries from the NEXMark benchmark [55].



(h) Data parallelism over the number of partitions.



(i) Thread-level parallelism over the number of threads.

Figure 15. Portals benchmarks. Higher is better.

the relative cost is larger for smaller sizes, and reduced for larger sizes. Another trend is that for longer chain lengths, some of the relative costs associated with the sources and sinks are reduced, as in the Chain of Tasks over chain length benchmark (Figure 15b) when comparing the async and the noGuarantees runtime.

Comparing the micro-batching runtime and the async runtime, we can also note the following trends. Both the micro-batching runtime and async runtime perform better for larger atom sizes. For longer workflows with larger end-to-end latency, however, the async runtime will perform better than the micro-batching runtime. In the ChainOfTasks (Figure 15c) and AtomAlignmentWithWork (Figure 15f) benchmarks, the end-to-end latency causes the micro-batching runtime to have a large amount of idle processing time on its tasks. This accounts for the difference in throughput to the async runtime (for which the async runtime has higher throughput).

The NEXMark benchmark (Figure 15g) consists of three streams: persons, auctions, and bids [55]. We have chosen to implement the first four queries. The first query converts the currency of bids from U.S. Dollar to Euro. The second query filters auctions for specific auction identifiers, representing a selection. The third query performs a join and a filter to find out which persons are selling articles of the provided category in the provided states. The fourth query performs a join with an aggregation to compute the average winning bid prices of all auctions for each category. We implemented these queries with atom-size set to 1024. The results (Figure 15g) show that async and micro-batching have similar throughput performance across these queries. Even though these queries correspond to more complex computations, the performance around one million events/s still compares to the other microbenchmarks.

Q3: How does Portals scale with data parallelism? We have evaluated the microbenchmarks on a data-parallelism-enabled version of the *async* runtime, executed

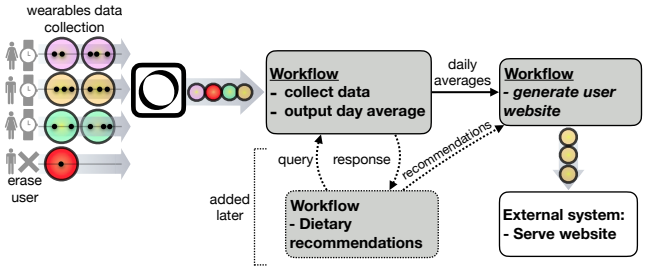


Figure 16. Wearables end-to-end application use case.

on a machine with eight physical hardware threads on eight cores. The results (Figure 15h, Figure 15i) indicate that there is a large gain possible from the data-parallel execution. For the CountingActor, PingPong, and ThreadRing workloads, the largest relative throughput gains were 14-, 14-, and 11-fold, respectively, with the number of threads set to 16. Similarly, we see a gain in throughput when increasing the threads (2-, 3-, 3-fold), with the number of partitions set to 16. This gain is to be expected, due to the inherent data-parallel and pipeline-parallel execution of dataflow-streaming. Although, we would have hoped for higher gains with the increasing thread number.

7 Use Cases

We divide use cases in Portals into concrete applications, existing service patterns and distributed programming paradigms. In each case, we detail how the basic programming elements of Portals can cover the required capabilities.

7.1 Application Example

As a prime use case of the Portals model consider a wearables application as depicted in Figure 16. The application needs to collect user data such as heart rate records, blood sugar levels, from a wearable smartwatch. The data can be transactionally ingested into the Portals system. A workflow collects and stores the latest data from each user, and outputs the latest day's averages continually. The daily averages are forwarded via an atomic stream to another workflow that generates websites to present this data to the user. The websites, in turn, are forwarded to an external system for serving. Thus far, the application requires *scalability*, as there are potentially many users; *reliability*, as we do not want to drop or duplicate any datapoints; and *edge cloud* processing. Internally, within the workflows, data collection and processing makes use of *scalable* dataflow streaming composition.

Over time, as the application evolves, a third-party recommendation service is added *dynamically* to the ecosystem. The newly added workflow computes dietary recommendations based on the collected user data. The recommendations workflow connects to the portal of the data-collection workflow to *query it on-demand*. This way, the workflow can query for some user's data, and then process the reply to generate the recommendations for that specific user. The generated recommendations are then forwarded to the website workflow.

Lastly, the application should support erasing a user, in an *atomic state update*. The atomic execution ensures that the erasure is applied correctly. If the erasure had been applied concurrently to a read-write operation of the user's data, then the data could possibly be read before erasure, and written after erasure. This would in turn lead to an inconsistent state in which the data was not erased. This type of operation is critical with respect to privacy regulations [31, 49], for example.

We show Portals code of the wearables application in Listing 1 (internal details omitted for space reasons). Here, we first create a portal; a generator for ingesting the user-data; and a workflow for processing this data and replying to any data queries. We also create the website workflow together with a sequencer, such that we can later add new connections to the website workflow. Finally, we connect the data workflow to the website workflow via the sequencer, and launch the application. At a later point in time, we launch a new recommendations application. The new application connects to the portal and sequencer of the data workflow and website workflow respectively. The recommendations workflow queries the portal, and computes some statistics that it emits to the website workflow. In order to support deletions, we additionally add the capability to erase a user from the application, by running the `allWithWrapper` combinator on the `dataWorkflow` (this should be done before launching the application, but we have it separate here just for clarification). The erasure is executed atomically if the erasure event is the only event in the atom. We can also support batching by collecting the operations to be executed, and execute them on the `onAtomComplete` event (subsection 4.5).

7.2 Subsuming Existing Service Patterns

Sagas. Distributed transactions can be implemented with Sagas in the Portals framework. The ask-await syntax can be used to implement Sagas via orchestration, for which the orchestrator can be a workflow, and the services all implement a portal.

Decentralized Datalakes. The workflow composition together with portals enable implementing decentralized datalakes in Portals. Multiple workflows can co-exist, for example, an OLTP workflow, an OLAP analytics workflow, and these can in turn query the datalake workflow. The scalability of the Portals model enables the scale needed for such applications.

Complex Event-Processing Applications, Stateful Serverless. Another area we think Portals would be well

Listing 1. Wearables application.

```
/** Wearables Application **/
val builder = Applications.builder("wearables")
val portal = builder.portals[Query]("data-query")
val generator = builder.generators
    .fromExternal[Data]("resource-identifier")
val dataWorkflow = builder
    .workflows[Data, Statistics]("dataWorkflow")
        .source(generator.stream)
        ...
        .replier(portal)
          { ... /* handle regular data events */ }
          { ... /* handle requests */ }
        ...
        .sink()
        .freeze()
val sequencer = builder.sequencers.random("website-sequencer")
val websiteWorkflow = builder
    .workflows[Statistics, Website]("websiteWorkflow")
        .source(sequencer.stream)
        ... // generate a website
        .sink()
        .freeze()
builder.connections.connect(dataWorkflow.stream, sequencer)
val app = builder.build()
system.launch(app)

/** Recommendations Application **/
val builder = Applications.builder("recommendations")
val extPortal = builder.registry.portals
    .get[Query]("wearables/portals/data-query")
val extSequencer = builder.registry.sequencer
    .get[Statistics]("wearables/sequencers/website-sequencer")
val generator = builder.generators
    .fromExternal[Trigger]("another-resource-identifier")
val recommendations = builder
    .workflows[Trigger, Statistics]("recommendations")
        .source(generator.stream)
        .asker(portal) {
            val future = portal.ask( ... )
            await(future){ ... }
            ... // emit statistics
        }
        .sink()
        .freeze()
builder.connections.connect(recommendations.stream, sequencer)
val app = builder.build()
system.launch(app)

/** Serializable Update */
def erase[T, U](): TaskContext[T, U] ?=> Unit =
    state.clearPerKey() // clear per-key state
dataWorkflow
    .allWithWrapper{ ctx ?=> wrapped => event => event match
        case Erase() => erase()
        case _ => wrapped(event)
  }
```

suited for, is for writing complex event processing logic, because of the expressivity of the tasks and workflows. Many of these services can be connected, and still retain the end-to-end exactly-once processing guarantees.

Dynamic Workflow Reconfiguration. Another area to explore is dynamic workflow reconfiguration. We can envision the possibility of updating a workflow to a newer version. We can do this by starting the new workflow by consuming the same atomic stream as the old workflow, then, in a serializable update, we can stop the old workflow, and start the new workflow. This transition would be seamless due to the atomic processing guarantees.

7.3 Subsuming Distributed Programming Models

Portals shares many aspects with other distributed programming models. Here we will look at how we could implement other models with Portals, to showcase some more features of the model. The Streaming Dataflow and MapReduce [17] model are trivial to implement in Portals, thus we do not describe them here.

Bulk Synchronous Parallel (BSP). To support BSP [24, 38, 56] requires some form of synchronization barriers. The model proceeds in supersteps, and for each superstep the processes receive messages from the last superstep, and produce the messages for the next superstep. The idea here, is that we can use an atom as a superstep/synchronization barrier, and iterate by cycling the events for the next superstep back into the same workflow. This way we can also perform step-wise communications.

Pregel. The pregel model [37] is similar to the BSP model, in that the computation happens over supersteps. In addition to this, we would need to support some form of aggregations and program termination to model pregel. Aggregations can happen via some reduce task. Termination could be managed by a second workflow, that controls the first workflow to see if all vertices have voted to terminate, and no new messages have been spawned.

Virtual Actors. The virtual actor model [9] consists of virtual actors (sometimes also called grains, entities, or data-parallel actors). Virtual actors differ from regular *actors* [1] in two ways: they cannot dynamically spawn new actors; and the actor reference is a composite of the virtual actor reference and the identity. Virtual actors have been supported within dataflow systems but limited to a single dataflow worflow as it has been showcased in Flink [2, 20]. This can be materialized as a single task over some contextual key which corresponds to the virtual actor identity, together with a shuffle and cycle.

8 Related Work

Atomic streams reflect existing fault-tolerant distributed log systems such as Kafka [51], Pravega [53] and Azure queue storage [40]. In practice, these systems are used to connect

stateful streaming systems together, such as the combination of Kafka [51] and Flink [12]. Much like the atomic streams in Portals it enables reasoning about end-to-end exactly-once guarantees across applications. The Portals workflows closely resemble pipelines in other stateful dataflow streaming systems [3, 12, 22, 42]; these all provide scalability and native data parallelism. Our proposed implementation of atom alignment is based on the asynchronous snapshotting barriers from Flink [12]. What distinguishes Portals from other streaming models is that it models multiple decentralized pipelines. With respect to iterations and cycles, there are streaming systems that have support for this [23, 42].

Other stateful serverless programming models [7, 16, 20, 34, 50] share many of the features of Portals and are great alternatives in their own right. This includes scalability, exactly-once processing guarantees, dynamic deployments, cyclic dependencies, and more. What differs is the dataflow-style composition and serializable updates present in Portals. Noteworthy considerations are data-parallel actor systems [7, 9, 20, 30, 33] (sometimes also called virtual actors or entities), as an alternative to our workflows. They provide data parallelism with actor-like dynamic messaging. In some sense, Portals Workflows resemble process actors [5, 29, 33] but with static messaging and dataflow-style compositionality. There has been work on the compositionality of actors; for example, Reactors [43, 44] are an actor-dataflow hybrid which can have multiple incoming streams and compose together sub-protocols within the actor (sub-protocol composition is otherwise problematic in actors [32]). Portals Workflows lack the first-class references of sources as it exists in the Reactors model, but support data parallelism unlike Reactors. The spirit of the atomic processing contract can be found in other related work, such as work by Yoo et al. [57] which processes events over transactional turns to guarantee composable reliability and enable exactly-once processing even under arbitrary failures. A "portals" construct has been proposed previously in the context of dataflow streaming systems, but with a completely different semantics, such as enabling broadcast messaging on streams [54].

The definition and use of futures [6, 28, 35] has come to change over time [45]. The future semantics presented in this work can be classified as explicit and typed, with asynchronous synchronization as await invocations do not block but register a continuation closure. Other future implementations might be implicit (implicit synchronisation visibility), untyped, or synchronous (*e.g.*, with a blocking get operation), with control-flow or data-flow synchronization [45]. To our knowledge, we present the first dataflow model that has an abstraction for request-reply communication together with futures.

The presented serializable updates resemble dynamic system updates and transactions. Whereas dynamic system updates are primarily concerned with safely swapping out some running program's part [47], ensuring the semantic correctness of the update [48], serializable updates in Portals specifically target modifying the application state and leave semantic consistency concerns to the user. Similarly, this is true for dynamic streaming systems updates [36] which is more concerned with system configuration and less with application state. Transactions on streaming systems share the same idea as serializable updates, i.e., to provide transactional guarantees for certain operations [16, 39, 59]. The use of transactional application state for GDPR compliance is also suggested in other work [31].

9 Conclusions

We have presented the Portals programming model, an extension of the dataflow streaming model for stateful serverless systems. Portals combines: exactly-once processing guarantees of *atomic streams* and the *atomic processing contract*, serializable event orderings with *serialziable updates*, seamless dynamic compositionality through *portals*' ask-await style direct messaging, and multi-*workflow* service compositions, and dynamic scalability. The Portals model has successfully combined these features into a single programming model, and we believe that it is a great alternative for writing scalable stateful serverless applications due to its guarantees and intuitive expressiveness.

For future work, we plan to complete the distributed and decentralized implementation of Portals, work out the operational semantics and provide a sound type system, and explore extensions such as dynamically splitting and fusing atoms as well as actor-like references.

Acknowledgments

The authors would like to thank Chengyang Huang and Siyao Liu for help with the implementation of Portals, as well as the anonymous referees for their valuable comments and helpful suggestions.

This work was partially funded by Digital Futures ("Resilient Decentralized Computing" research pairs project), the Swedish Foundation for Strategic Research under Grant No.: BD15-0006, as well as RISE AI.

References

[1] Gul Agha. 1986. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA.

[2] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful Functions as a Service in Action. *Proc. VLDB Endow.* 12, 12 (2019), 1890–1893. https://doi.org/10.14778/3352063.3352092

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[4] Amazon Web Services, Inc. 2022. Serverless Computing - AWS Lambda - Amazon Web Services. https://aws.amazon.com/lambda/. Accessed: 2022-07-07.

[5] Joe Armstrong, Robert Virding, and Mike Williams. 1993. *Concurrent programming in ERLANG*. Prentice Hall.

[6] Henry G. Baker and Carl Hewitt. 1977. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, USA, August 15-17, 1977*, James Low (Ed.). ACM, 55–59. https://doi.org/10.1145/800228.806932

[7] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: Efficient Execution of Serverless Workflows. *Proc. VLDB Endow.* 15, 8 (2022), 1591–1604. https://www.vldb.org/pvldb/vol15/p1591-burckhardt.pdf

[8] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485510

[9] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: cloud computing for everyone. In *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*. 16. https://doi.org/10.1145/2038916.2038932

[10] Paris Carbone. 2018. *Scalable and Reliable Data Stream Processing*. Ph. D. Dissertation. Royal Institute of Technology, Stockholm, Sweden. https://nbn-resolving.org/urn:nbn:se:kth:diva-233527

[11] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[13] Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Tobias Wrigstad, and Albert Mingkun Yang. 2018. Attached and detached closures in actors. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2018, Boston, MA, USA, November 5, 2018*, Joeri De Koster, Federico Bergenti, and Juliana Franco (Eds.). ACM, 54–61. https://doi.org/10.1145/3281366.3281371

[14] Nicolas Chappe, Ludovic Henrio, Amaury Maillé, Matthieu Moy, and Hadrien Renaud. 2022. An Optimised Flow for Futures: From Theory to Practice. *Art Sci. Eng. Program.* 6, 1 (2022), 3. https://doi.org/10.22152/programming-journal.org/2022/6/3

[15] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. 2014. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Comput. Lang. Syst. Struct.* 40, 3-4 (2014), 112–136. https://doi.org/10.1016/j.cl.2014.05.002

[16] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Distributed transactions on serverless stateful functions. In *15th ACM International Conference on Distributed and Event-based Systems, DEBS 2021, Virtual Event, Italy, June 28 - July 2, 2021*, Alessandro Margara, Emanuele Della Valle, Alexander Artikis, Nesime Tatbul, and Helge Parzyjegla (Eds.). ACM, 31–42. https://doi.org/10.1145/3465480.3466920

[17] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, 137–150. http://www.usenix.org/events/osdi04/tech/dean.html

[18] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 34, 3 (2002), 375–408. https://doi.org/10.1145/568522.568525

[19] Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. 2019. Godot: All the Benefits of Implicit and Explicit Futures. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:28. https://doi.org/10.4230/LIPIcs.ECOOP.2019.2

[20] The Apache Software Foundation. 2022. Apache Flink Stateful Functions. https://nightlies.apache.org/flink/flink-statefun-docs-release-3.2/. Accessed on 2022-06-26.

[21] Martin Fowler. 2005. Event Sourcing. https://martinfowler.com/eaaDev/EventSourcing.html. Accessed: 2022-07-10.

[22] Can Gencer, Marko Topolnik, Viliam Durina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz, József Bartók, Grzegorz Gierlach, Frantisek Hartman, Ufuk Yilmaz, Ondrej Lukás, Mehmet Dogan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Hazelcast Jet: Low-latency Stream Processing at the 99.99th Percentile. *Proc. VLDB Endow.* 14, 12 (2021), 3110–3121. https://doi.org/10.14778/3476311.3476387

[23] Gábor E. Gévay, Juan Soto, and Volker Markl. 2022. Handling Iterations in Distributed Dataflow Systems. *ACM Comput. Surv.* 54, 9 (2022), 199:1–199:38. https://doi.org/10.1145/3477602

[24] Philipp Haller and Heather Miller. 2011. Parallelizing Machine Learning- Functionally: A Framework and Abstractions for Parallel Graph Processing. In *2nd Annual Scala Workshop, SCALA'11, Stanford, CA, USA, June 2, 2011*. https://infoscience.epfl.ch/record/165111

[25] Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.* 410, 2-3 (2009), 202–220. https://doi.org/10.1016/j.tcs.2008.09.019

[26] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, Paul Barham and Timothy Roscoe (Eds.). USENIX Association. https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems

[27] Shams Mahmood Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos A. Varela (Eds.). ACM, 67–80. https://doi.org/10.1145/2687357.2687368

[28] Robert H. Halstead Jr. 1985. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (1985), 501–538. https://doi.org/10.1145/4472.4478

[29] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 years of actors: a taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016*, Sylvan Clebsch, Travis Desell, Philipp Haller, and Alessandro Ricci (Eds.). ACM, 31–40. https://doi.org/10.1145/3001886.3001890

[30] Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. 2022. Data-Parallel Actors: A Programming Model for Scalable Query Serving Systems. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, Amar Phanishayee and Vyas Sekar (Eds.). USENIX Association, 1059–1074. https://www.usenix.org/conference/nsdi22/presentation/kraft

[31] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan W. Weckwerth, Brian S. Xia, Peter Bailis, Michael J. Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. 2022. Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework. *CoRR* abs/2208.13068 (2022). https://doi.org/10.48550/arXiv.2208.13068 arXiv:2208.13068

[32] Roland Kuhn. 2016. My journey towards understanding distribution. https://github.com/rkuhn/blog/blob/master/01_my_journey_towards_understanding_distribution.md. Accessed: 2022-06-26.
[33] Lightbend, Inc. 2022. Akka. https://akka.io/. Accessed: 2022-07-07.
[34] Lightbend, Inc. 2022. Kalix. https://www.kalix.io/. Accessed: 2022-07-07.
[35] Barbara Liskov and Liuba Shrira. 1988. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*. 260–267. https://doi.org/10.1145/53990.54016
[36] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanam Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. 2018. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *Proc. VLDB Endow.* 11, 10 (2018), 1303–1316. https://doi.org/10.14778/3231751.3231765
[37] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 135–146. https://doi.org/10.1145/1807167.1807184
[38] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM Comput. Surv.* 48, 2 (2015), 25:1–25:39. https://doi.org/10.1145/2818185
[39] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Çetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. 2015. S-Store: Streaming Meets Transaction Processing. *Proc. VLDB Endow.* 8, 13 (2015), 2134–2145. https://doi.org/10.14778/2831360.2831367
[40] Microsoft. 2021. Azure Queue Storage. https://docs.microsoft.com/en-us/azure/storage/queues/storage-queues-introduction. Accessed: 2022-07-07.
[41] Mark S. Miller, Eric Dean Tribble, and Jonathan S. Shapiro. 2005. Concurrency Among Strangers. In *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3705)*, Rocco De Nicola and Davide Sangiorgi (Eds.). Springer, 195–229. https://doi.org/10.1007/11580850_12
[42] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 439–455. https://doi.org/10.1145/2517349.2522738
[43] Aleksandar Prokopec. 2017. Encoding the building blocks of communication. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, Emina Torlak, Tijs van der Storm, and Robert Biddle (Eds.). ACM, 104–118. https://doi.org/10.1145/3133850.3133865
[44] Aleksandar Prokopec and Martin Odersky. 2015. Isolates, channels, and event streams for composable distributed programming. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Gail C. Murphy and Guy L. Steele Jr. (Eds.). ACM, 171–182. https://doi.org/10.1145/2814228.2814245
[45] Francisco Ramón Fernández Reyes. 2021. *Abstractions to Control the Future*. Ph. D. Dissertation. Uppsala University, Sweden. https://nbn-resolving.org/urn:nbn:se:uu:diva-425128
[46] Till Rohrmann. 2022. Keynote: Rethinking how distributed applications are built. (2022). Proceedings of the 16th ACM International Conference on Distributed and Event-based Systems.
[47] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. 2013. A survey of dynamic software updating. *J. Softw. Evol. Process.* 25, 5 (2013), 535–568. https://doi.org/10.1002/smr.1556
[48] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2022. Change Is the Only Constant: Dynamic Updates for Workflows. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. IEEE, 350–362. https://ieeexplore.ieee.org/document/9793884
[49] Jonas Spenger, Paris Carbone, and Philipp Haller. 2021. WIP: Pods: Privacy Compliant Scalable Decentralized Data Services. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB Workshops, Poly 2021 and DMAH 2021, Virtual Event, August 20, 2021, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12921)*, El Kindi Rezig, Vijay Gadepally, Timothy G. Mattson, Michael Stonebraker, Tim Kraska, Fusheng Wang, Gang Luo, Jun Kong, and Alevtina Dubovitskaya (Eds.). Springer, 70–82. https://doi.org/10.1007/978-3-030-93663-1_7
[50] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 11 (2020), 2438–2452. http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf
[51] The Apache Software Foundation. 2017. Apache Kafka. https://kafka.apache.org/. Accessed: 2022-07-07.
[52] The Apache Software Foundation. 2021. FlinkKafkaProducer. https://nightlies.apache.org/flink/flink-docs-release-1.11/api/java/org/apache/flink/streaming/connectors/kafka/internal/FlinkKafkaProducer.html. Accessed: 2022-07-07.
[53] The Linux Foundation. 2022. Pravega. https://cncf.pravega.io/. Accessed: 2022-07-07.
[54] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer, 179–196. https://doi.org/10.1007/3-540-45937-5_14
[55] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2002. NEXMark: A benchmark for queries over data streams. https://datalab.cs.pdx.edu/niagara/NEXMark/.
[56] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111. https://doi.org/10.1145/79173.79181
[57] Sunghwan Yoo, Charles Edwin Killian, Terence Kelly, Hyoun Kyu Cho, and Steven Plite. 2012. Composable Reliability for Asynchronous Systems. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 27–40. https://www.usenix.org/conference/atc12/technical-sessions/presentation/yoo
[58] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 423–438. https://doi.org/10.1145/2517349.2522737
[59] Shuhao Zhang, Juan Soto, and Volker Markl. 2022. A Survey on Transactional Stream Processing. *CoRR* abs/2208.09827 (2022). https://doi.org/10.48550/arXiv.2208.09827 arXiv:2208.09827
[60] École Polytechnique Fédérale Lausanne (EPFL). 2022. The Scala Programming Language. https://www.scala-lang.org/. Accessed: 2022-07-07.