Multiple Papers ....

(W4) Paper Reading

Nan Lin

Shanghai Jiao Tong University

2024-07-21

Outline

Outline

Decentralized and Stateful Serverless Computing on the Internet Computer Blockchain

Maksym Arutyunyan, Andriy Berestovskyy, Adam Bratschi-Kaye, Ulan Degenbaev, Manu Drijvers, Islam El-Ashi, Stefan Kaestle, Roman Kashitsyn, Maciej Kot, Yvonne-Anne Pignolet, Rostislav Rumenov, Dimitris Sarlis, Alin Sinpalean, Alexandru Uta, Bogdan Warinschi, and Alexandra Zapuc, *DFINITY, Zurich*

https://www.usenix.org/conference/atc23/presentation/arutyunyan

Basic concepts: State, Stateless and Stateful

- What is **state** ? (e.g. cookies)
- **Stateless** vs **Stateful**: Between requests, the server retains data or state?
  - Scalability & Cost Efficiency
  - Simplified management
  - vs
  - Persistent application data
  - Checkpointing communication channels
  - Thus provide failure, availability and scalability *isolation* (!)
  - **Persistent queues**: *Kafka*
- Serverless functions: primarily stateless applications
- Connecting event-driven functions with stateful services?

Basic concepts: Internet Computer (IC), comparaison

Internet Computer (IC): Decentralized blockchain-based platform for the execution of general-purpose applications in the form of smart contracts.

- Distributed worldwide
- State maintained automatically
- **Orthogonal persistence**

Drawbacks of Stateless Serverless Computing

- *Ephemeral and Stateless Functions*: Short-lived
- *Scalability*
- *Developer Burdern*: Externel support

Outline

Netherite: Efficient Execution of Serverless Workflows

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

Badrish Chandramouli
Microsoft Research
badrishc@microsoft.com

Chris Gillum
Microsoft Azure
cgillum@microsoft.com

David Justo
Microsoft Azure
dajusto@microsoft.com

Konstantinos Kallas
University of Pennsylvania
kallas@seas.upenn.edu

Connor McMahon
Microsoft Azure
comcmaho@microsoft.com

Christopher S. Meiklejohn
Carnegie Mellon University
cmeiklej@cs.cmu.edu

Xiangfeng Zhu
University of Washington
xfzhu@cs.washington.edu

Basic concepts: Durable Function (DF): 3 parts

- **Durable Function**: Part of *Azure Functions*
- Operation
  - Compose tasks into **Orchestration**
  - Store application state in **Entites**
  - Concurrency control by **Critical Sections**

Basic concepts: Durable Function (DF): 3 parts

- **Orchestration**: Task-parallel workflows (1. Sequential Composition)

```
[FunctionName("SimpleSequence")]
public static async Task<int> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext c)
{
    try
    {
        var x = c.GetInput<int>();
        var y = await c.CallActivityAsync<int>("F1", x);
        var z = await c.CallActivityAsync<int>("F2", y);
        return z;
    }
    catch (Exception) {
        // Error handling or compensation can go here.
    }
}
```

Figure 1: Sequencing two functions F1 and F2 using a durable functions orchestration in C#.

Basic concepts: Durable Function (DF): 3 parts

- **Orchestration**: Task-parallel workflows (2. Parallel Composition)

```
const df = require("durable-functions");
module.exports = df.orchestrator(function*(context) {
  // Get the directory input argument
  const directory = context.df.getInput();
  // Call an activity and wait for the result
  const files = yield context.df.callActivity(
    "GetImageList", directory);
  // For each image, call activity without waiting
  // and store the task in a list
  const tasks = [];
  for (const file of files) {
    tasks.push(context.df.callActivity(
      "CreateThumbnail", file));
  }
  // wait for all the tasks to complete
  const results = yield context.df.Task.all(tasks);
  // return sum of all sizes
  return results.reduce((prev, curr) => prev + curr, 0);
});
```

Figure 2: Example orchestration using the Durable Functions JavaScript API. It calls an activity `GetImageList`, and then, in parallel, `CreateThumbnail` for each image. It then waits for all to complete and returns the aggregated size.

Basic concepts: Durable Function (DF): 3 parts

- **Entities**: shared objects storing application state, working *serially*, supports synchonization and concurrency control

```
public class Account
{
    public int Balance { get; set; }
    public int Get() => Balance;
    public void Modify(int Amount) { Balance += Amount; }

    // boilerplate for Azure Functions (feel free to ignore)
    [FunctionName(nameof(Account))]
    public static Task Run([EntityTrigger]
      IDurableEntityContext ctx)
        => ctx.DispatchAsync<Account>();
}
```

Figure 3: Example entity using the Durable Functions C# API. Its state is an integer `Balance`, and it has operations `Get` and `Modify` to read or update it.

Basic concepts: Durable Function (DF): 3 parts

- **Critical sections**: address synchronization challenges involving durable state stored in more than one place

Basic concepts: Durable Function (DF): 3 parts

```
[FunctionName("Transfer")]
public static async Task<bool> Transfer(
    [OrchestrationTrigger] IDurableOrchestrationContext ctx)
{
  (string source, string dest, int amount) =
    ctx.GetInput<string,string,int>();
  EntityId sourceId = new EntityId("Account", source);
  EntityId destId = new EntityId("Account", dest);

  using (await ctx.LockAsync(sourceId, destId))
  {
    int bal = await ctx.CallEntityAsync<int>(sourceId, "Get");
    if (bal < amount)
    {
      return false;
    }
    else
    {
      await Task.WhenAll(
        ctx.CallEntityAsync(sourceId, "Modify", -amount),
        ctx.CallEntityAsync(destId, "Modify", +amount));
      return true;
} } }
```

Create Entity

Exclusive Access, locked

Figure 4: Example of an orchestration with a critical section that reliably transfers money between account entities.

Basic concepts: Stateless functions with stateful instances

Core problem: How the serverless message passing model combines stateless functions with stateful instances that communicate via messages?

Message-passing layer: From serverless models to lowever level

Basic concepts: Stateless functions with stateful instances

Serverless message-passing model: queue + "key-queue-value" state

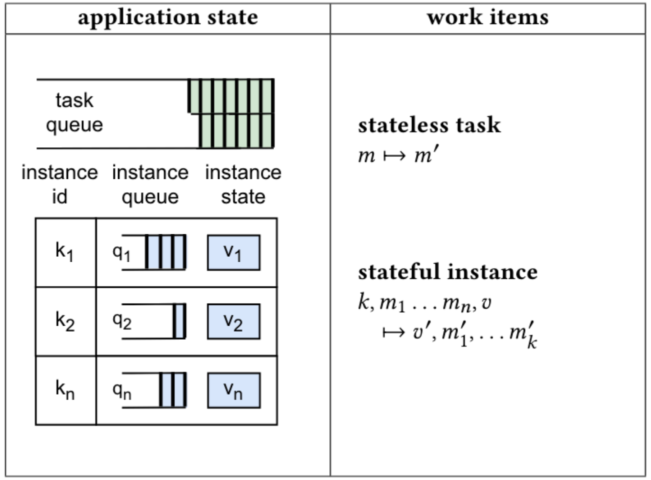

Figure 5: The serverless message-passing model. Messages in the task queue represent stateless functions scheduled for execution. The key-queue-value stores the current state and message queue of each instance. The application progresses by fetching, executing, and committing work items. Work items can consume and produce messages and update instance states.

Basic concepts: Stateless functions with stateful instances

Transition: task $m \mapsto m'$; $(m_1, \ldots, m_n) \mapsto (m'_1, m'_2, \ldots, m'_k)$

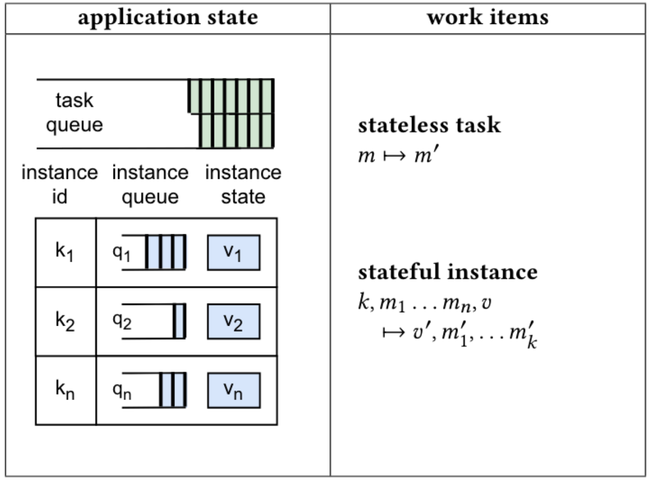

Figure 5: The serverless message-passing model. Messages in the task queue represent stateless functions scheduled for execution. The key-queue-value stores the current state and message queue of each instance. The application progresses by fetching, executing, and committing work items. Work items can consume and produce messages and update instance states.

Basic concepts: Stateless functions with stateful instances

DF applications → instances (state of orchestrations and entities), tasks (actitvites), messages (calls and response)

- **Orchestration**: Checkpointing → **Partial history** of eventes, which could be replayed
- **Entites** → Instances
- **Critical Sections**: Mutual exclusion achieved by *two-phase locking protocol*

Motivation

Problem: **DF** ...

- Use individiual storage operations to update instance states and to en(de)queue messages
- Thus creates a throughput bottleneck due to limited **I/O per second (IOPS)**

Motivation

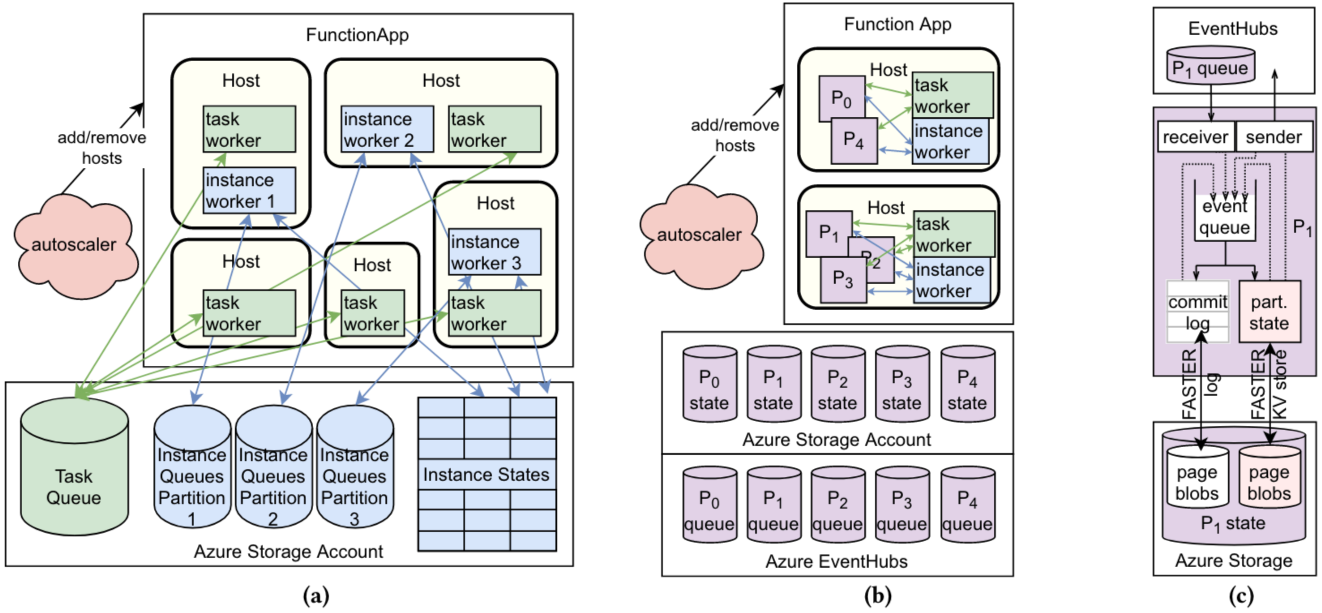

Figure 7: (a) Illustration of the original AzureStorage engine architecture, showing 6 hosts. A single task queue delivers task work items to all hosts. Instance work items are partitioned over 3 control queues, each connected to one affinitized worker. The instance states are stored in tables. (b) Illustration of the Netherite engine architecture, showing 5 partitions $P_0, \ldots, P_4$ distributed over 2 hosts. Workers do not connect to storage directly, but to locally hosted partitions. Each partition has its own state and uses an optimized persistence mechanism. Partitions communicate with each other via ordered persistent queues (EventHubs). (c) Partition-internal event processing, state management, and persistence with FASTER.

Their benefits

Illustration of a **message-passing model** for stateful serverless.

A replacement of **DF** with benefits of ...

- Improve throughput
- Reduce storage traffic
- Pipelining reduce latency

Outline

Triggerflow: Trigger-based orchestration of serverless workflows

Aitor Arjona [a,*], Pedro García López [a], Josep Sampé [a], Aleksander Slominski [b], Lionel Villard [b]

[a] *Universitat Rovira i Virgili, Tarragona, Spain*
[b] *IBM Watson Research, NY, USA*

Motivation

Serverless orchestration systems are not designed for *long-running* data analytics tasks.

Goals:

- Heterogeneous workflows
- Extensibility and Computational Reflection
- Serverless desig: reactive, pay per use, flexible scaling, ...
- High-volume workloads