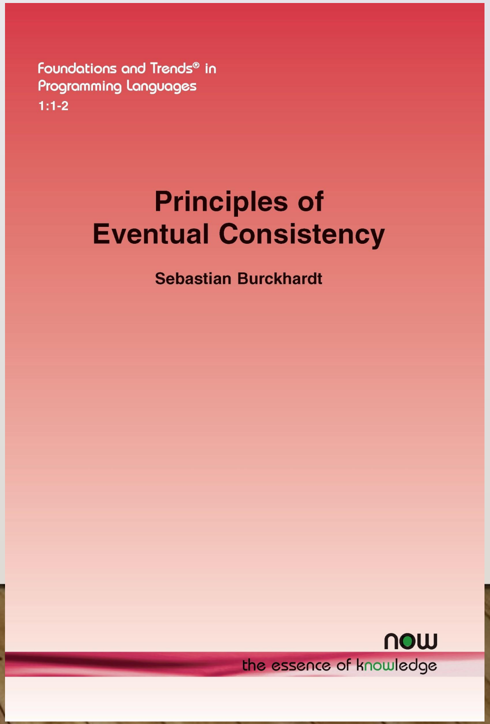
SEBASTIAN BURCKHARDT

From Basel, Switzerland.

At Microsoft Research since 2007

General interest: programming models for distributed/parallel/concurrent systems.

- Particular interests:
  - Eventual Consistency (e.g. specifications, CRDTs)
  - Elastic services, actor systems (e.g. Orleans)
  - Serverless programming models



BACKGROUND:
HOW TO BUILD SCALABLE, AVAILABLE SERVICES TODAY?

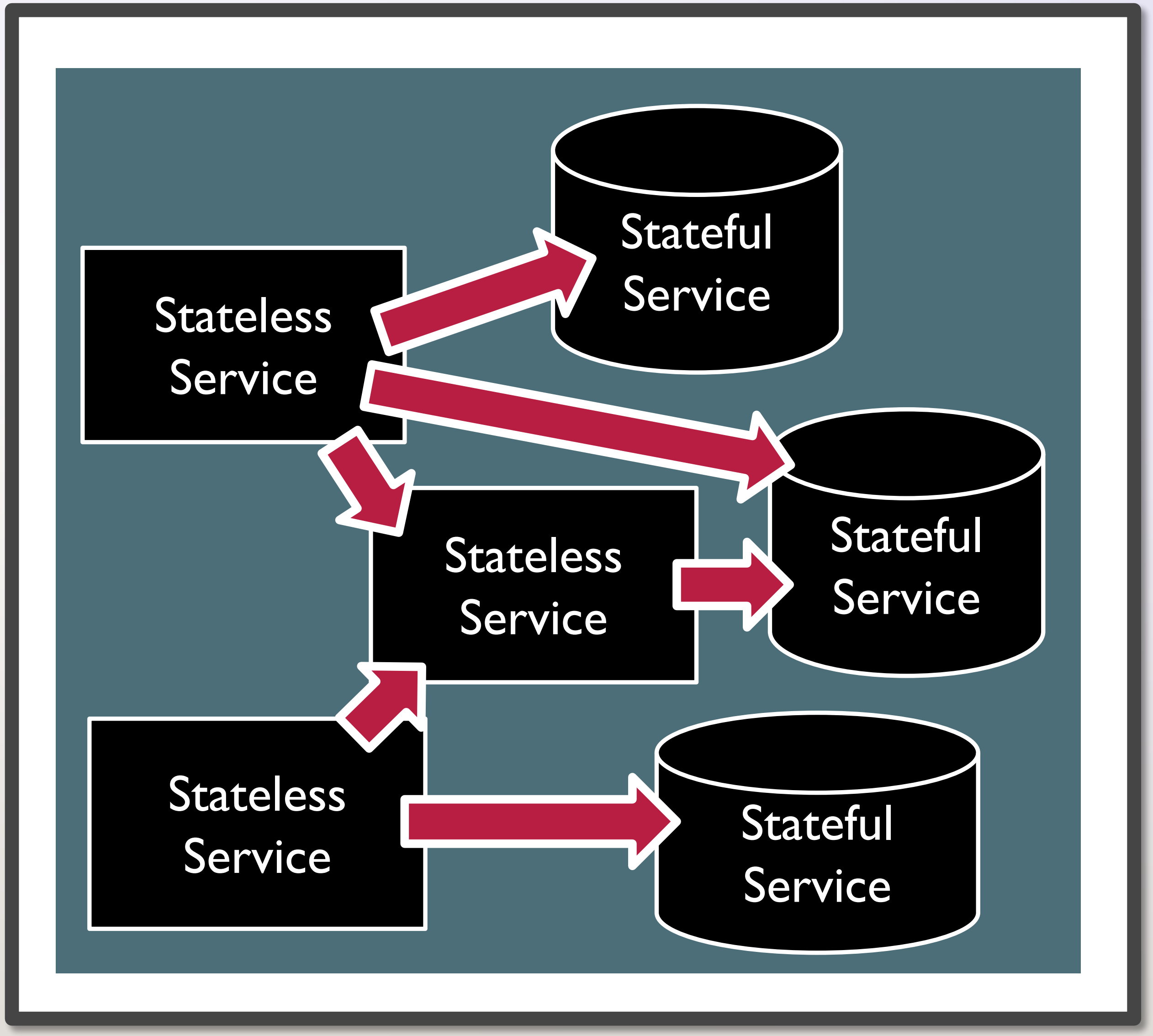
- From scalable, available components.

- Leave all the real hard stuff
  (distributed protocols, failure detection, failure recovery,...)
  for others to solve.

COMPOSED SERVICES

Services represent logical functionality, not physical machines.

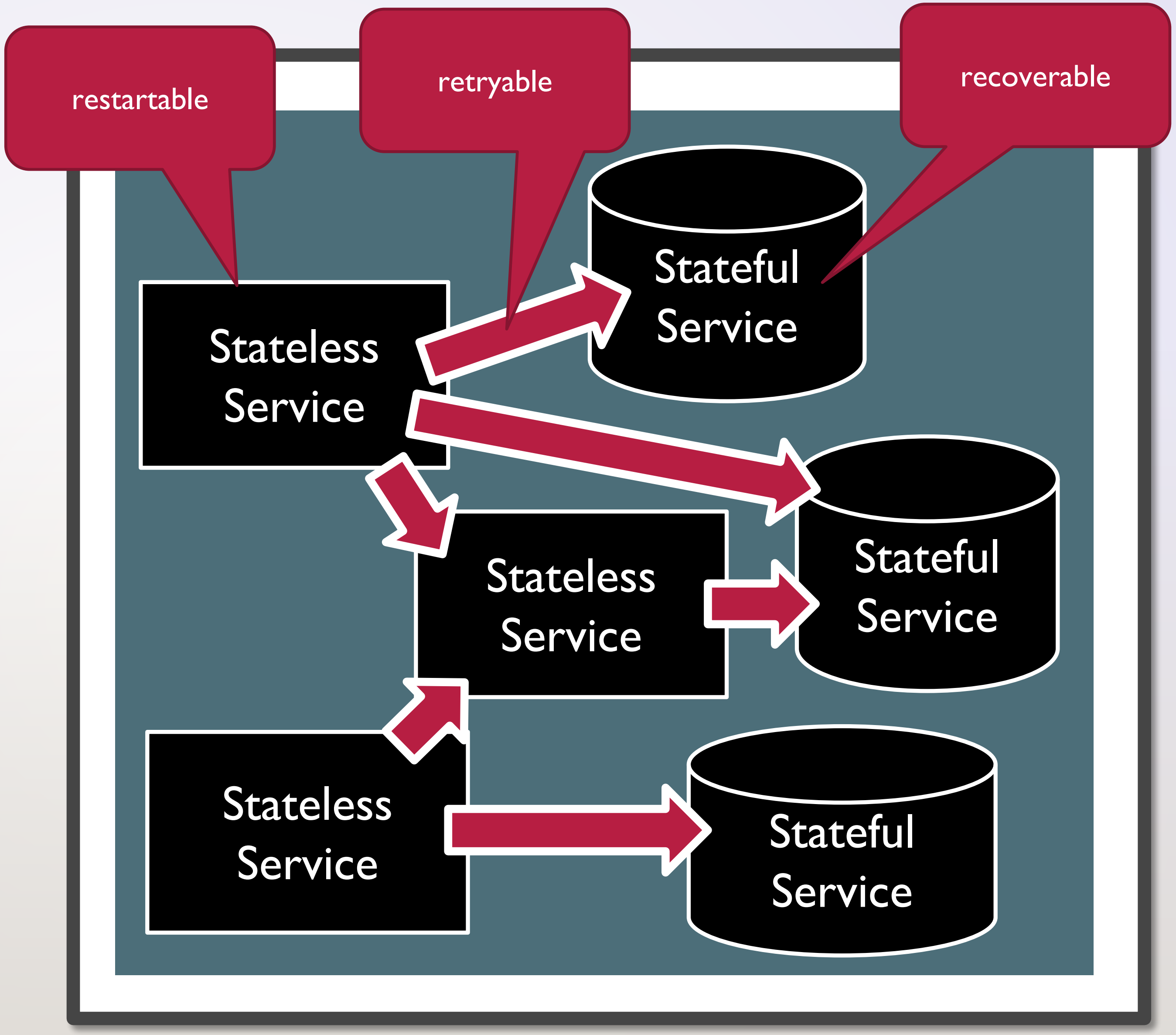
Often: multiple languages, REST APIs



COMPOSED SERVICES

All components are highly available.

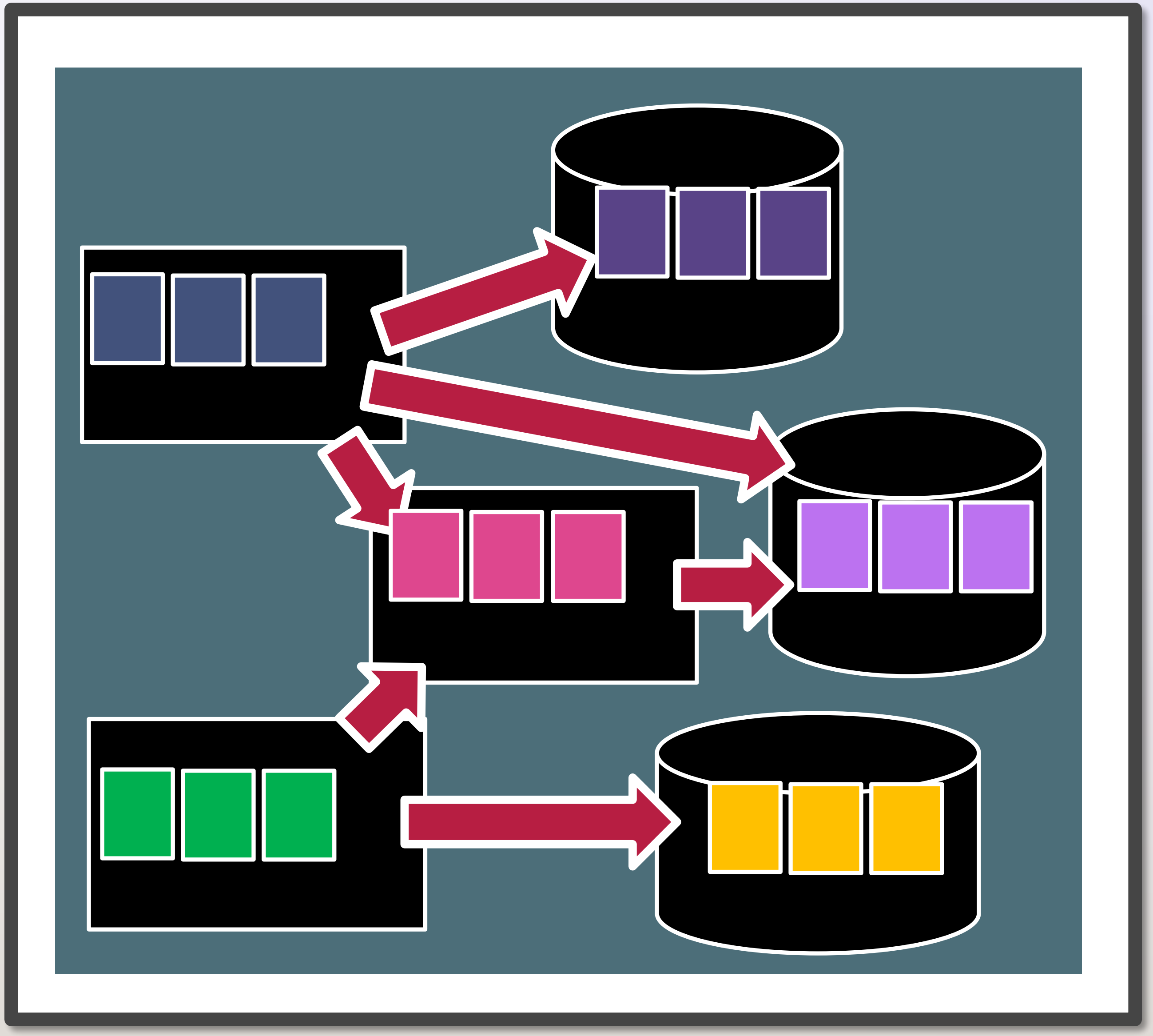
Localized failure recovery via idempotent or testable APIs.



INTERNAL PARTITIONING

services are internally partitioned so they can scale beyond the capacity of a single machine.

Partitions can be very fine-grained (e.g. key-value pair in a key-value store).



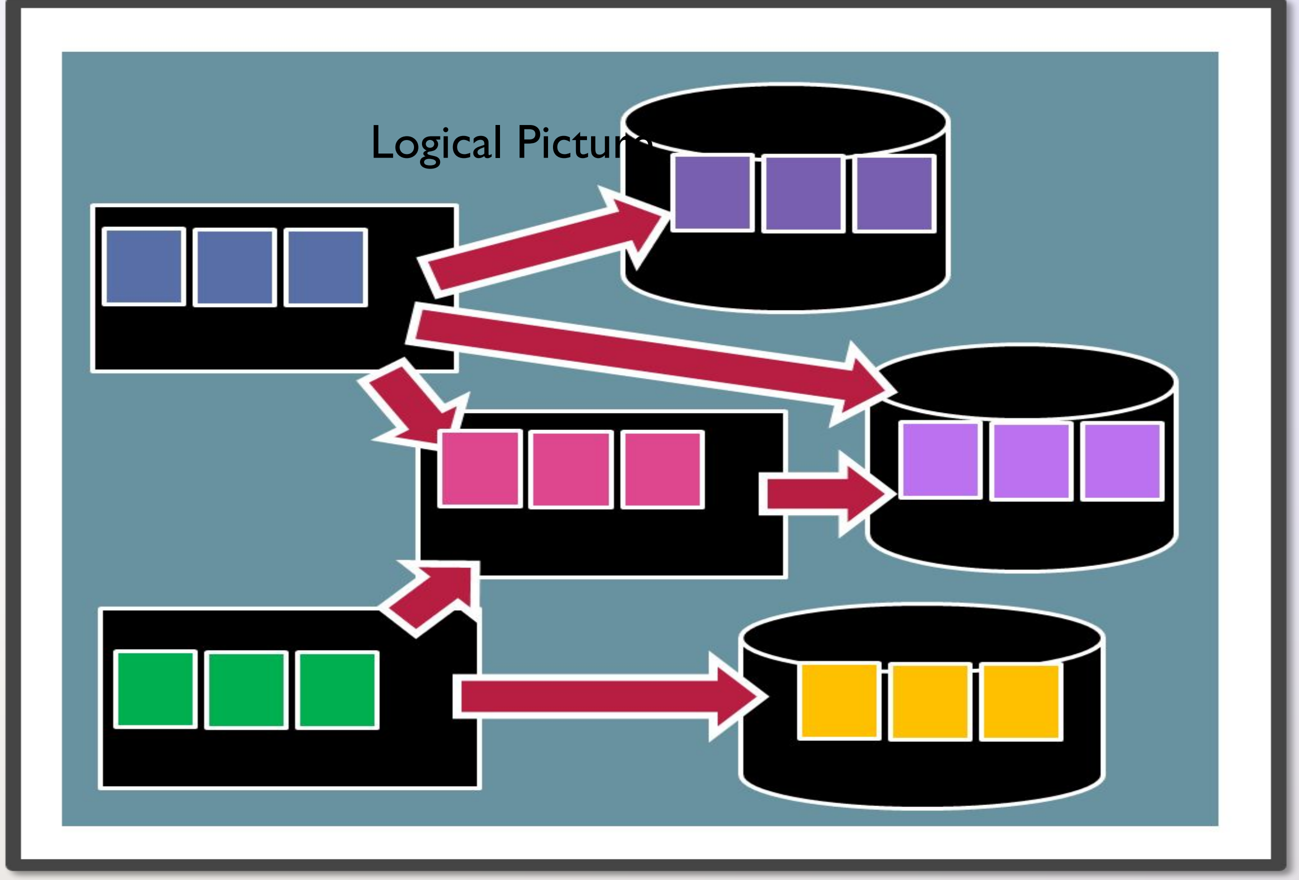
ELASTIC PACKING

Needed for price/performance.

Much recent innovation in this space.

General goals: make this

- a bit easier (K8s, Docker)
- fully automatic (serverless)



DEVELOPERS CHOOSE

Control

Productivity

Assembly

C,
C++

Java,
C#

Python

DEVELOPERS CHOOSE

Control

Productivity

Infrastructure
as a Service

Containers
as a service

Platform
as a service

Functions
as a Service
e.g.
AWS Lambda, Azure Functions

DEVELOPERS CHOOSE

SERVERLESS
(meaning: concept of a “server” is not visible in the application code, but managed within lower layer)

Control

Productivity

Infrastructure as a Service

Containers as a service

Platform as a service

Functions as a Service
e.g.
AWS Lambda, Azure Functions

TOP-GROWING CLOUD SERVICES 2019

Place	Service	Growth	2018 Use	2019 Use
#1 (tie)	Serverless	50%	24%	36%
#1 (tie)	Stream Processing	50%	20%	30%
#3	Machine Learning	44%	18%	26%
#4	Container-as-a-Service	42%	26%	37%
#5	IoT	40%	15%	21%
#6	Data warehouse	38%	29%	40%
#7	Batch processing	38%	26%	36%

Source: Forbes, RightScale 2019 state of the cloud report

SERVERLESS FUNCTIONS

```
string helloworld(string name)
{
    return "Hello, " + name;
}
```

- Easy to deploy
- Elastic scale
- Load-based cost (e.g. pay per invocation)
- Free language choice, easy REST interface

```
> curl http://my-function-app.azure.com/helloworld?name=Shonan
Hello, Shonan
```

SERVERLESS FUNCTIONS ARE NOT “PURE”. THEY CAN CALL OTHER SERVICES.

Functions can call external services:

key-value stores, queues, blob storage, pub-sub, databases, ...

= the “standard library” of cloud programming!

```
async void delete_all()
{
    await cloudstorage.delete_file(“*”);
}
```

```
async void counter_increment()
{
    var current = await cloudstorage.read(“counter”);
    current = current + 1;
    await cloudstorage.write(“counter”);
}
```

Stateless Functions are great

but…

… WE SEE SOME PAIN POINTS AROUND STATE MANAGEMENT AND SYNCHRONIZATION.

- Sychronization

  functions can interleave and race, synchronization via storage is challenging

- Partial execution

  hosts can fail in the middle of a function, leaving behind inconsistent state

- Performance

  Lots of calls to storage, lots of data movement => wastes time and CPU.

AZURE DURABLE FUNCTIONS

State & Synchronization for Serverless

2 NEW TYPES OF STATEFUL FUNCTIONS

Stateless Functions

+ Orchestrations (≈ Workflows)

provide
- durable **execution** state
- critical sections

+ Entities (≈ Actors)

provide
- durable **application** state
- operation sequencing

ORCHESTRATIONS
= DURABLE EXECUTION STATE

- Reliably execute some combination of functions.
  - e.g. a simple sequence of functions, or multiple parallel function calls
- Eliminates the partial execution problem.

EXAMPLE 1

- Upload file, then add to index

```
void upload_image(string name, byte[] data)
{
    await addtoblobstorage(name, data);

    await addtoindex(name);
}
```

```
void addtoblobstorage(string name, byte[] data)
{
    …
}
```

```
void addtoindex(string name)
{
    …
}
```

ORCHESTRATIONS

- Similar to workflows, but straightforward async-await code.
  No XML or state machines.
- Thus, benefit from complete experience of the host language for control flow (sequential composition, parallel composition, all kinds of loops, exception handling, ...).

EXAMPLE 2

- Send many messages in parallel

```
void send_text_to_all_friends(string text)
{
    var friends = await getFriends();

    await Task.WhenAll(
     friends.Select(f => send(f, text)).ToList());
}
```

```
list<userid> getFriends()
{
    …
}
```

```
send(userid destination, string text)
{
    …
}
```

IMPLEMENTATION: HOW DOES IT WORK?

- Runtime uses record & replay
- Under failures, calls may be duplicated (but only the very last one)

```
void upload_image(string name, byte[] data)
{
    await addtoblobstorage(name, data);
    await addtoindex(name);
}
```

Log:

1. start function, input = (name, data)
2. call addtoblobstorage, input =(name, data)
3. response received, output = ()
4. call addtoindex, input = (name)
5. response received, output = ()
6. finish

IMPLEMENTATION: HOW DOES IT WORK?

```
void upload_image(string name, byte[] data)
{
    await addtoblobstorage(name, data);
    await addtoindex(name);
}
```

- Runtime uses record & replay.
- Under failures, calls may be duplicated
  (but only the very last one)
- Requires code to be deterministically replayable.
  Source of bugs (e.g. user must call context.UtcTime, not DateTime.UtcTime)
- PL research could help:
  - language could use type/effect system to track determinism / nondeterminism
  - language could provide serializable execution state, making replay unnecessary

2 NEW TYPES OF STATEFUL FUNCTIONS

Stateless Functions

Orchestrations
(≈ Workflows)

provide
- durable **execution** state
- critical sections

Entities
(≈ Actors)

provide
- durable **application** state
- operation sequencing

ENTITIES
= DURABLE APPLICATION STATE

- Entity = small piece of state identified by (name, key) string pair.
- Runtime delivers “operations” (messages) to entities via ordered async channels
- Runtime executes operations on entities, *one at a time*. Operations can
  - read and update state
  - send messages
  - perform external calls
  - return a value to caller (if the caller is an orchestration)
- Durable: All state (incl. messages) reliably kept in cloud storage

EXAMPLE ENTITY: BANK ACCOUNT

- each entity identified by a (name,key) pair, e.g. (“AccountEntity”, “32974-234093-00”)
- Serverless function defines how the entity handles operations, e.g. untyped interface in C#:

```
[FunctionName("AccountEntity")]
public static void Run([EntityTrigger] IDurableEntityContext context)
{
    switch(context.OperationName)
    {
        case "get":
            context.Return(context.GetState<int>());
            break;
        case "add":
            context.SetState(context.GetState<int>() + context.GetInput<int>());
            break;
    }
}
```

CALL VS. SIGNAL

- An actor can signal another actor
  send message, fire and forget

- An orchestration can call an actor
  and wait for ack/result

- But actors cannot call actors (to prevent deadlock)

FEATURE: EXPLICIT LOCKING

- Orchestrations can use critical sections
- very effective for preventing unwanted data races and interleavings

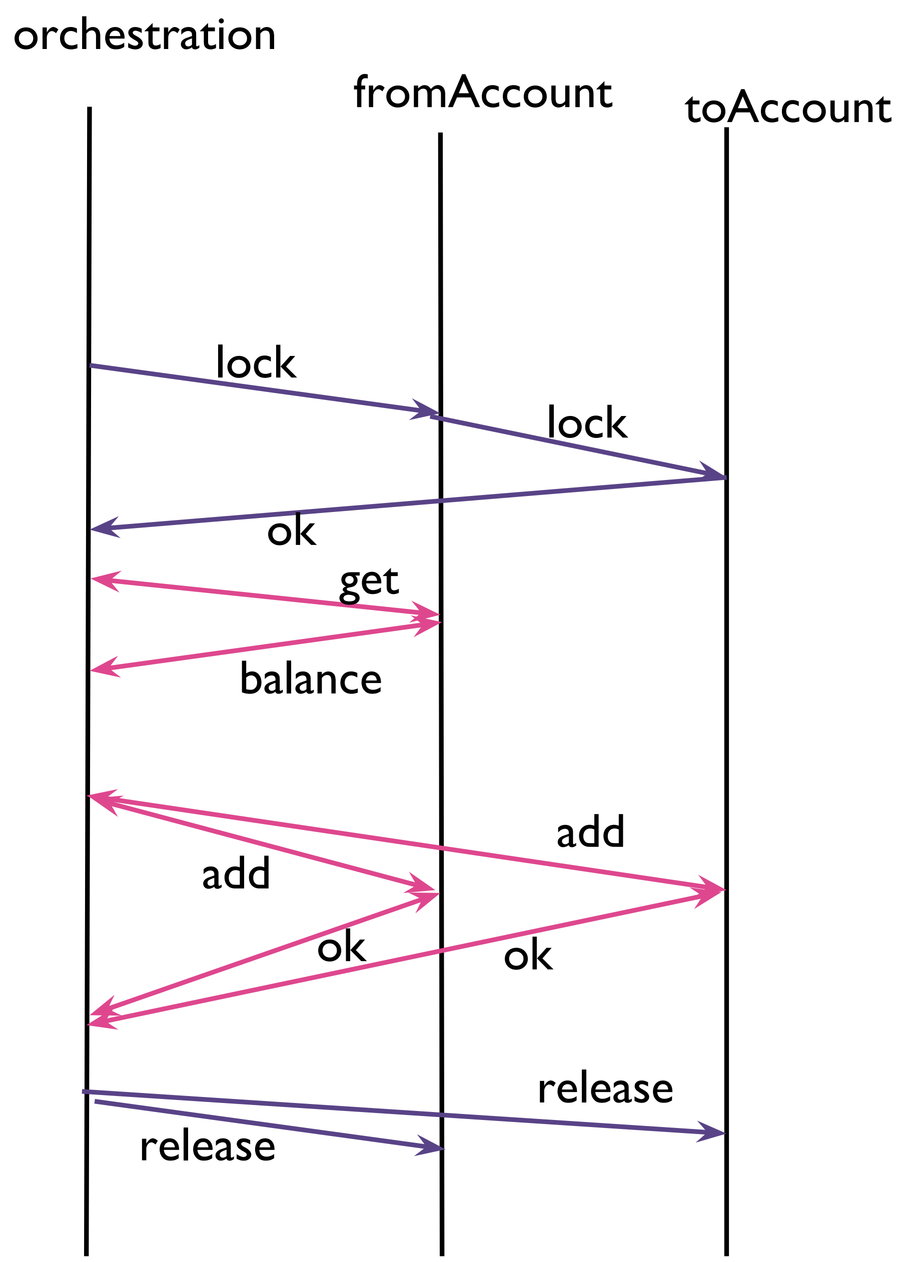
- Critical sections never fail spuriously and require no rollback or compensations

EXAMPLE: TRANSFER FUNDS

```
var fromAccount = new EntityId("AccountEntity", from);
var toAccount = new EntityId("AccountEntity", to);

using (await ctx.LockAsync(fromAccount, toAccount))
{
    var availablebalance = await ctx.CallEntityAsync<int>(fromAccount, "get");

    if (amount <= availablebalance)
    {
        await Task.WhenAll(
            ctx.CallEntityAsync(fromAccount, "add", -amount),
            ctx.CallEntityAsync(toAccount, "add", amount)
        );
    }
}
```

MESSAGE DIAGRAM

```
var fromAccount = new EntityId("AccountEntity", from);
var toAccount = new EntityId("AccountEntity", to);

using (await ctx.LockAsync(fromAccount, toAccount))
{
    var availablebalance = await ctx.CallEntityAsync<int>(

    if (amount <= availablebalance)
    {
        await Task.WhenAll(
            ctx.CallEntityAsync(fromAccount, "add", -amount
            ctx.CallEntityAsync(toAccount, "add", amount)
        );
    }
}
```



DEADLOCK PREVENTION

We enforce some simple rules to prevent deadlocks:

- Runtime acquires locks in order (fixed global total order).
- Critical sections cannot be nested.
- Within a critical section:
  - can call only entities that were locked.
  - can signal only entities that were not locked.

STATUS

- Azure Durable Functions have been out for about a year now. (thanks to Chris Gillum & Durable Functions team)
- Entities (& critical sections) are a new feature I implemented over the last couple months, building on research done last year w/ intern Christopher Meiklejohn, now in public preview since last month.
- No paper yet.