EUROPEAN CLOUD
SUMMIT 2024

Serverless Actor Model with Durable Functions!

Massimo Bonanni

Paranormal Trainer, with the head in the Cloud and all the REST in Serverless!

EUROPEAN CLOUD SUMMIT

Use ECS Coins for Swag!

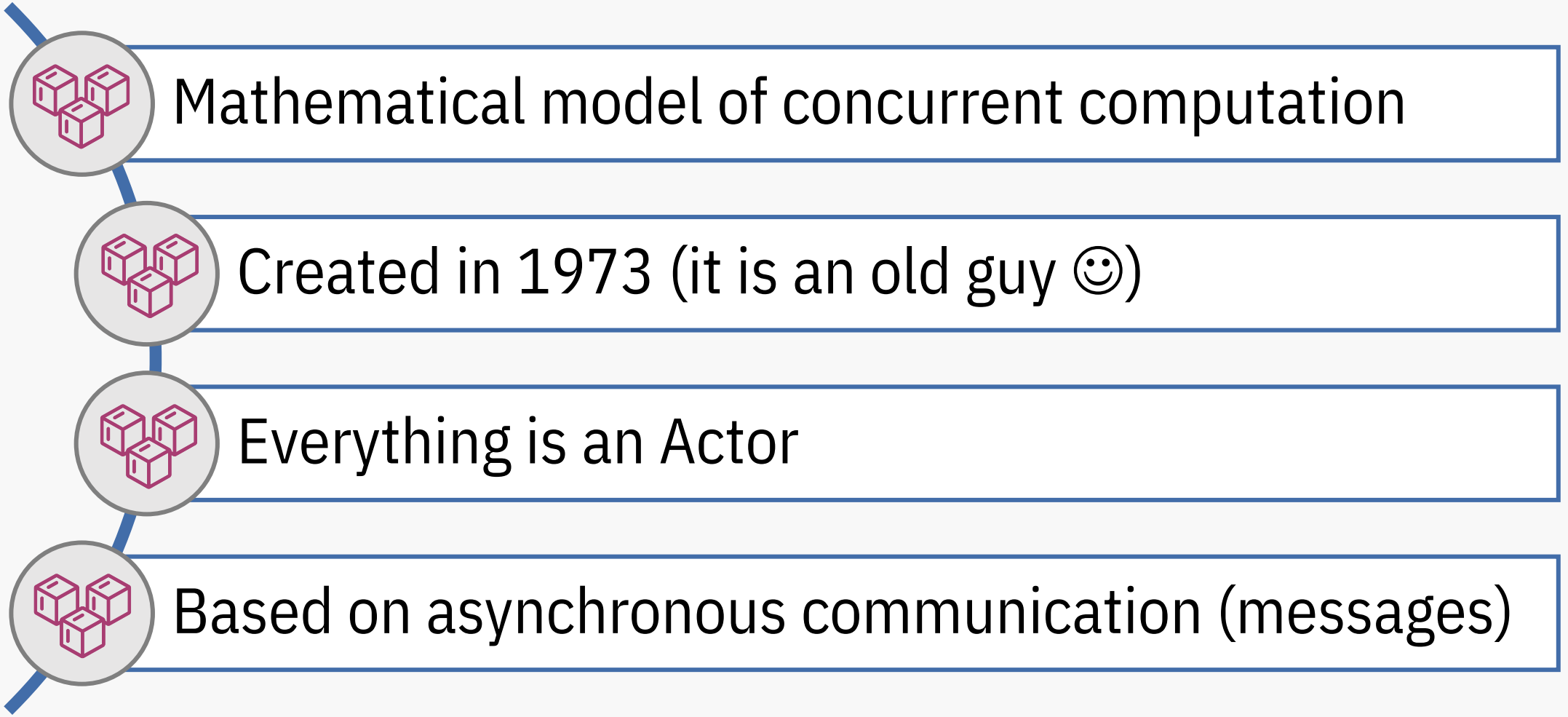
Top 3 win an Atari 2600+

1. Get the app
2. Visit sessions and sponsors, rate sessions
3. Earn ECS Coins
4. Spend ECS Coins

csmmt.eu/app

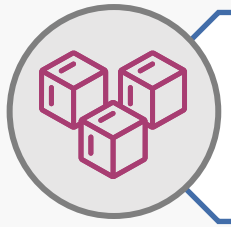
run events

What Actor Model is?

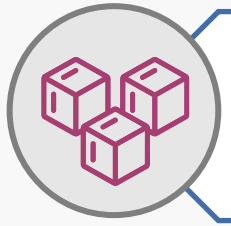


Fundamental concepts

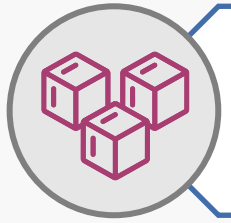
An actor is a computational entity that, in response to a message it receives, can:



Change its status based on the message received (one message at time)



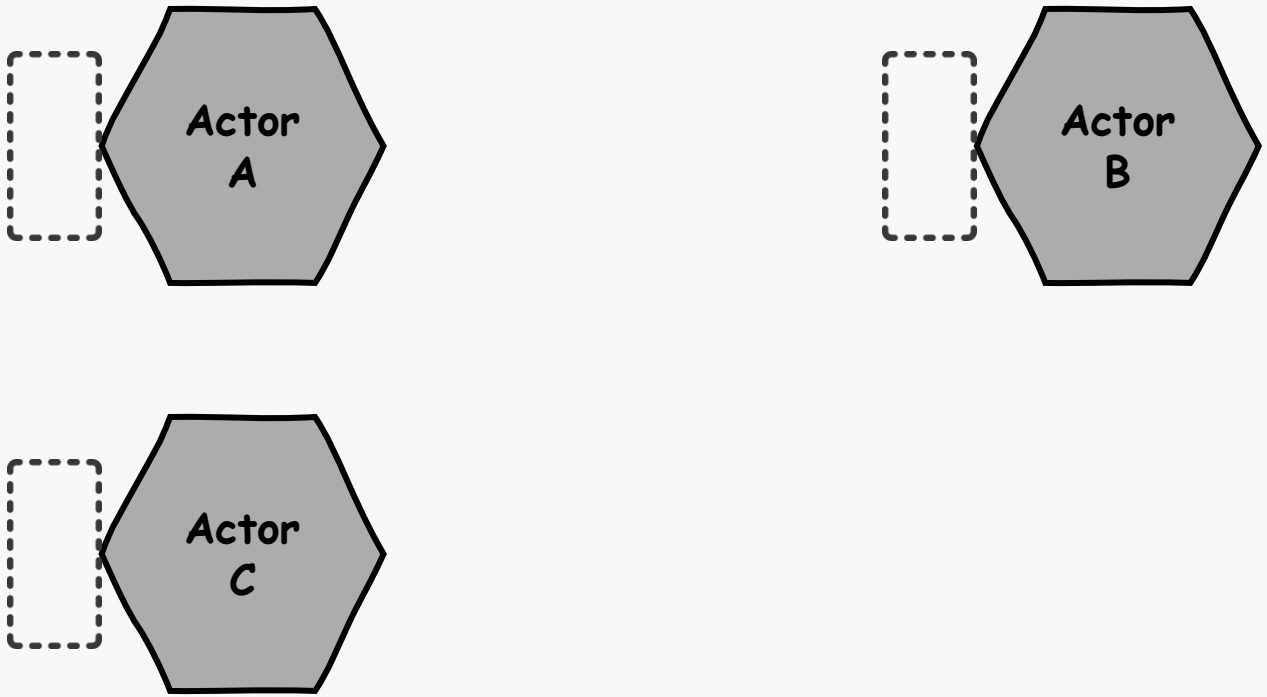
Send a finite number of messages to other actors (asynchronously)



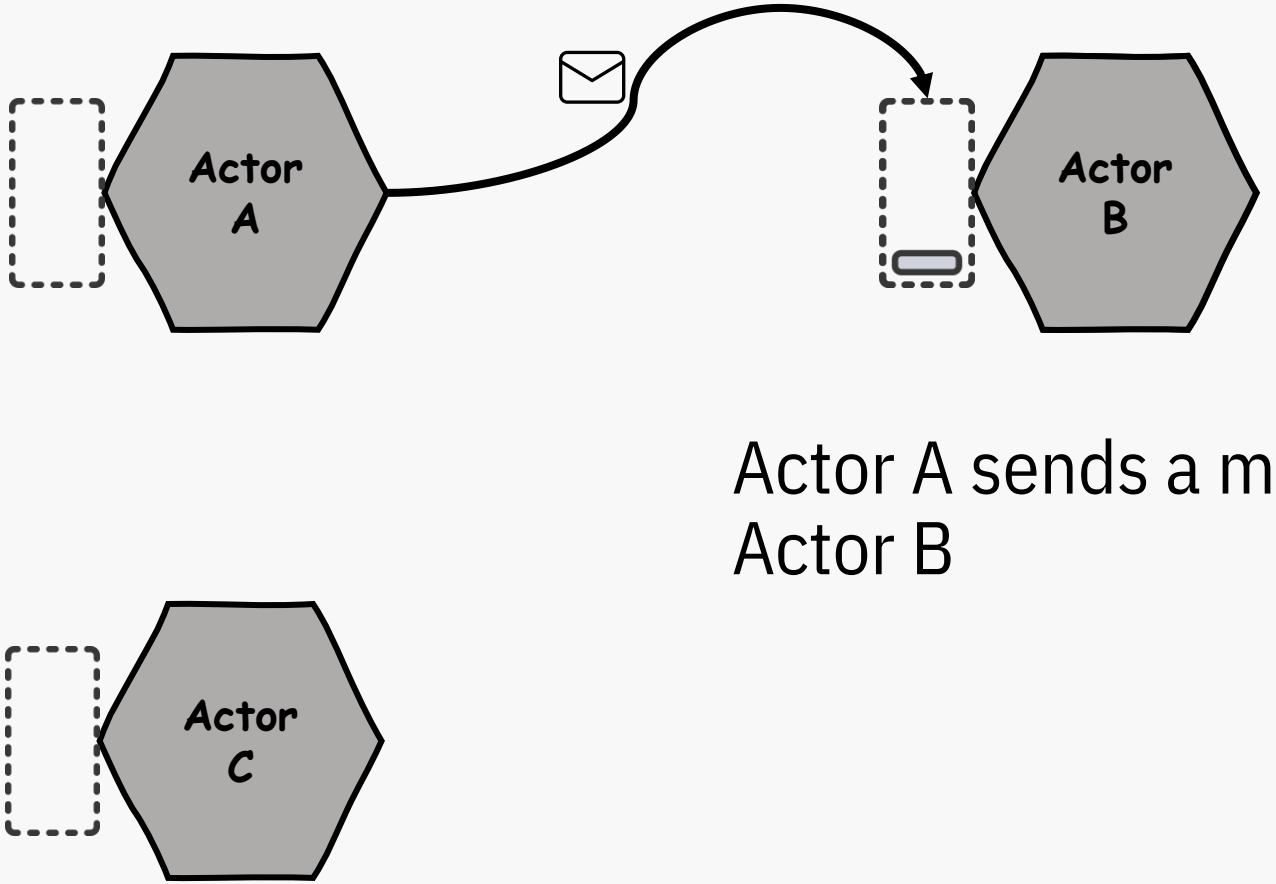
Create a finite number of new actors

Asynchronous Messaging Communication (1/5)

Each actor has its own message queue in which messages from other actors are stored.

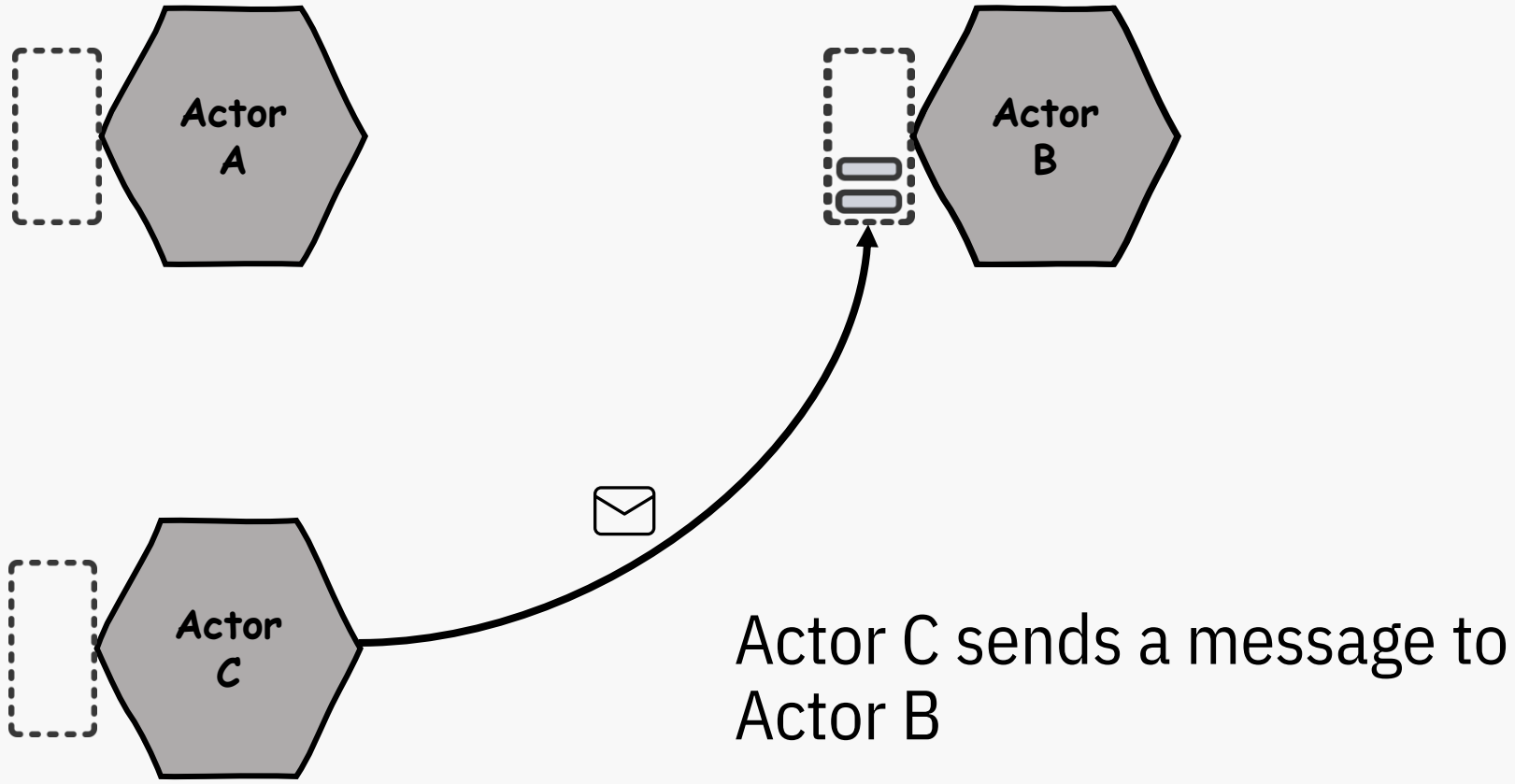


Asynchronous Messaging Communication (2/5)



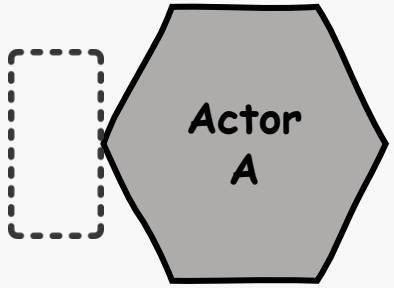
Actor A sends a message to Actor B

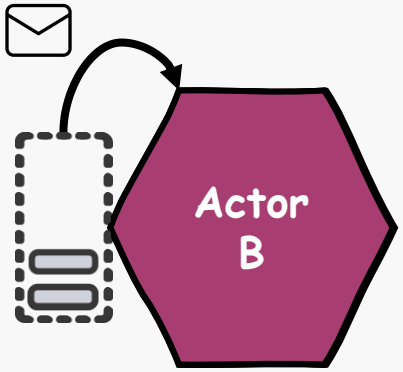
Asynchronous Messaging Communication (3/5)

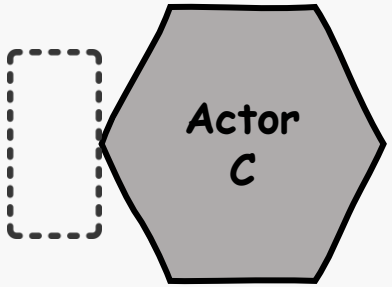


Actor C sends a message to Actor B

Asynchronous Messaging Communication (4/5)

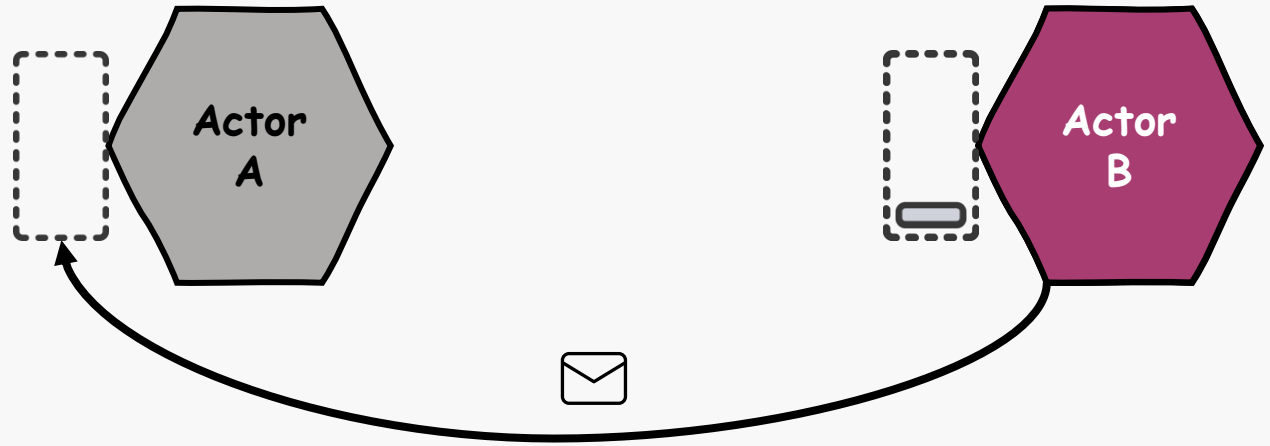


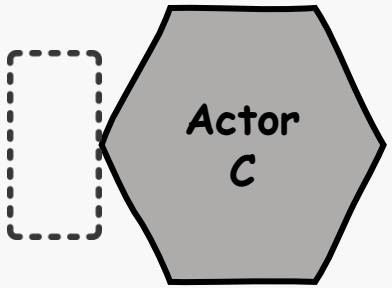




Actor B manages message received from Actor A and change its status

Asynchronous Messaging Communication (5/5)





Actor B send confirmation to Actor A using a message

Actor Model vs Azure Functions

Actor is stateful	Azure Function is stateless
Actor calls other Actors (using messages)	Azure Function cannot call other Azure Functions (directly)
Actor can live for long time	Azure Function has a finite timeout

Actor Model vs Azure Functions

Actor is stateful

Azure Function is stateless

Actor calls other Actors (using messages)

Function cannot all other Azure nctions (directly)

Actor can live time

Azure Function has a ite timeout

Durable Functions

Durable Functions are Azure Functions!!!

Azure Functions Extension

- Based on Azure Functions
- Adds new Triggers and Bindings
- **Manages state, checkpoints, and restarts**

Durable Task Framework

- **Long running persistent workflows in C#**
- Used within various teams at Microsoft to reliably orchestrate long running operations

Languages

- C#
- JavaScript
- Java
- Python
- Powershell

Function Types

- Client
- Orchestrator
- Activity
- **Entity** (no Powershell or Java)

Durable Entities

One of the Durable Functions function types

Expose operations for reading and updating internal state or interact with other entities

Accessible via Entity Instance ID composed by:

- Entity Name
- Entity Key

Every operation can be accessed using:

- Entity Instance ID
- Operation Name
- Operation Input
- Scheduled time (optional)

The Entity Instance ID

An **Entity Instance ID** is simply a pair of strings that uniquely identifies an entity instance.

Entity Name

- It is a name that identifies the type of the entity.
- This name must match the name of the entity function that implements the entity.
- It isn't sensitive to case

Entity Key

- It is a string that uniquely identifies the entity among all other entities of the same name

Implementing an Entity

Function-based

Entities are represented as functions and operations are explicitly executed in the function body

For entities with simple state, few operations, or a dynamic set of operations

This syntax doesn't catch type errors at compile time

Class-based

Entities are represented by classes

This syntax produces more easily readable code and allows operations to be invoked in a type-safe way

Can implement an interface.
The framework gives you a base class to manage state.

Class-based constraints

- The class must be constructible
- The class must be JSON-serializable
- Operations must have at most one argument
- Not overloads permitted for operations
- Arguments and return values must be serializable values or objects (not generics)
- You can define an interface for the entity

```
public class CarEntity : TaskEntity<CarData>, ICarEn
{
    private readonly ILogger _logger;

    public CarEntity(ILogger<CarEntity> logger)
    {
        this._logger = logger;
    }

    #region [ Public methods ]
    public void Initialize(InitializeCarDto carInfo)

    public void Rent(RentCarDto rentInfo)...

    public Task<ReturnCarResponseDto> Return(ReturnC

    public void Update(UpdateCarDto info)...

    public void Delete()...
    #endregion [ Public methods ]

    [ Private methods ]

    [Function(nameof(CarEntity))]
    public static Task Run([EntityTrigger] TaskEntit
        => ctx.DispatchAsync<CarEntity>();

}
```

Define an interface for an entity

- Must only define methods
- Must not contain generic parameters
- Methods must not have more than one parameter
- Methods must return void, Task, or Task<T>

```
public interface ICarEntity
{
    void Initialize(InitializeCarDto carInfo);

    void Rent(RentCarDto rentInfo);

    Task<ReturnCarResponseDto> Return(ReturnCarDto returnInfo);

    void Update(UpdateCarDto info);
}
```

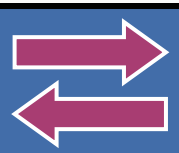
Communicate with the entities

Signaling

One-way (fire and forget) communication

You send an **operation message** but don't wait for a response.

Communication **between entities**, orchestrator and clients



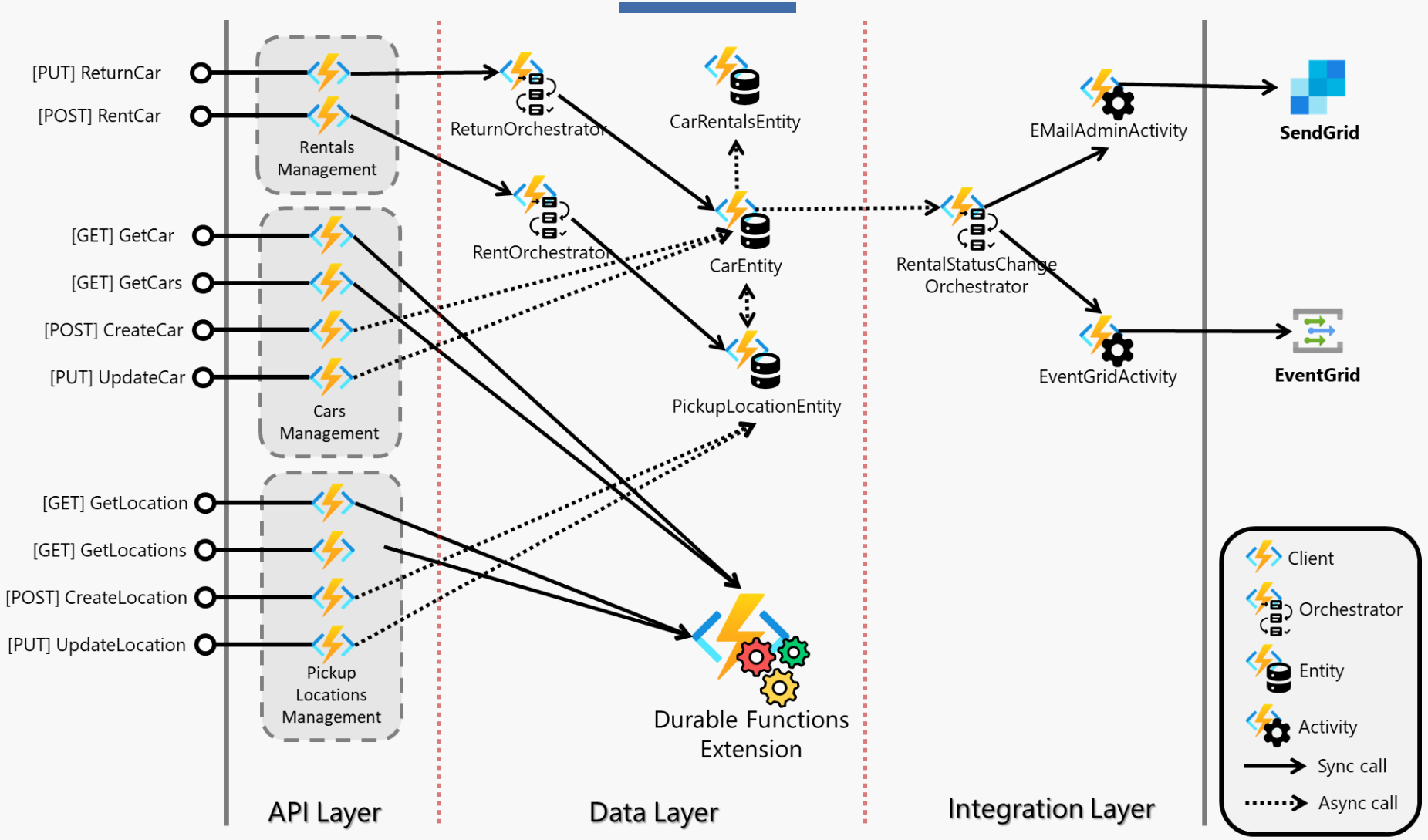
Calling

Two-way (round-trip) communication.

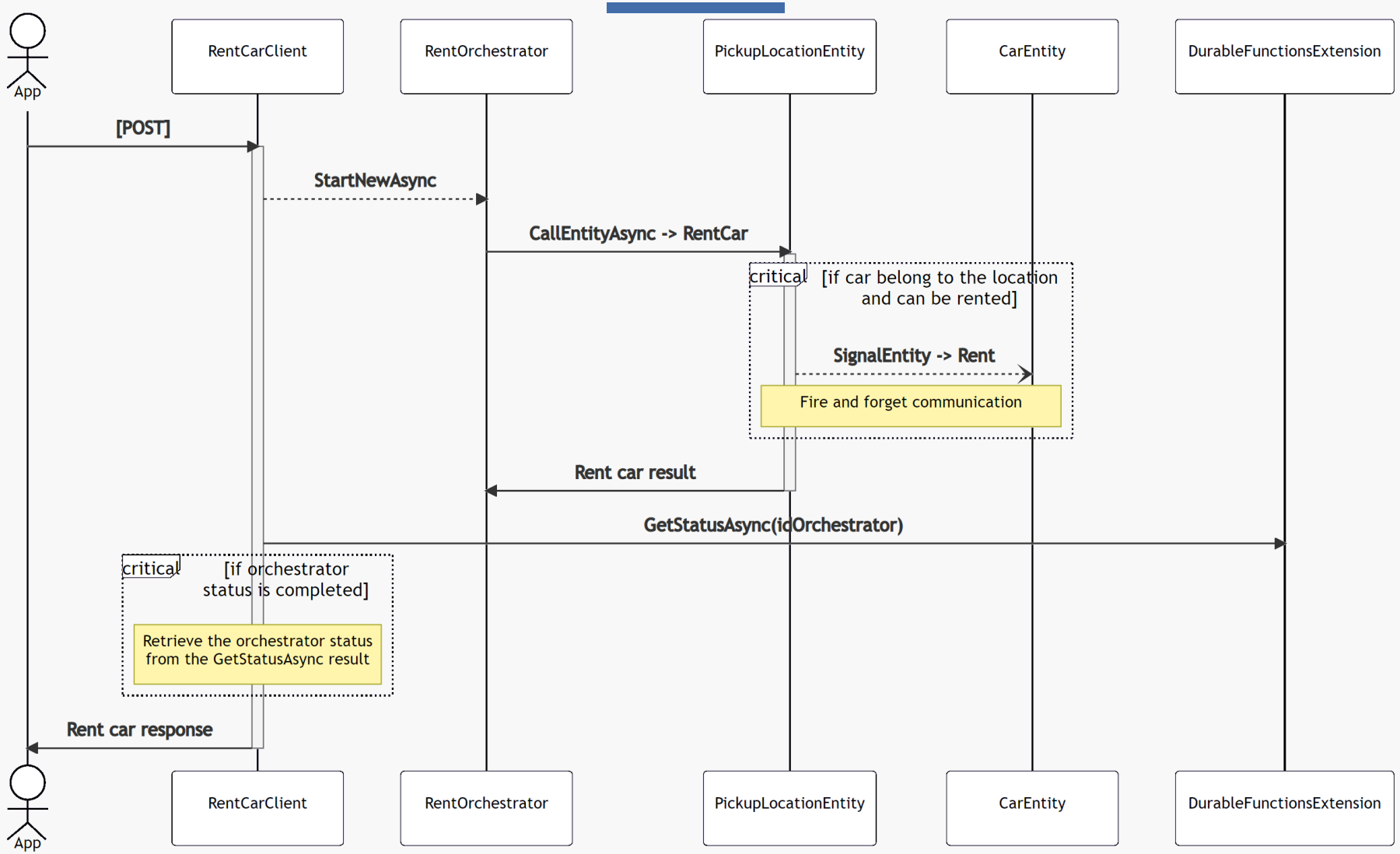
You send an **operation message** to the entity, and then wait for the response message before you continue

Communication only from orchestrators and entities

Serverless Car Rent - Architecture



Serverless Car Rent – Rent a Car scenario



DEMO

Serverless Car Rent

Durable Entities vs Virtual Actors

Like a Virtual Actor

- Durable Entities are addressable via an entity ID
- Durable Entity operations execute serially, one at a time, to prevent race conditions
- Durable Entities are created implicitly when they're called or signaled
- When not executing operations, durable entities are silently unloaded from memory
- Durable Entities don't deadlock

Unlike a Virtual Actor

- Durable Entities prioritize durability over latency
- Durable Entities don't have built-in timeouts for messages
- Request-response patterns in entities are limited to orchestrations
- Durable Entities can be used in conjunction with durable orchestrations to support distributed locking mechanisms

THANK YOU, YOU ARE AWESOME ❤

PLEASE RATE THIS SESSION IN THE MOBILE APP.

Massimo Bonanni

Microsoft Technical Trainer

massimo.bonanni@microsoft.com
@massimobonanni

aka.ms/maxlinkedin