Orleans

dotnet.github.io/orleans

David Gristwood
David.Gristwood@microsoft.com
@ScroffTheBad

github.com/dotnet/orleans
- V1.5.1 latest stable build, Aug 28th

Actor model

- Model for concurrent computation
  - Defined in the 1973 paper by Carl Hewitt
  - Popularised by Erlang language (at Ericsson), Akka (Java, Scala and .NET)
  - Service Fabric Reliable Actors framework
  - Project “Orleans” - Microsoft Research, used in Xbox Halo 4 and 5
- Actors universal primitives of concurrent computation
- Improve performance through private middle tier state
- Reduce code complexity through messages and isolation

Orleans: Distributed Virtual Actors for Programmability and Scalability

Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, Jorgen Thelin
Microsoft Research

Abstract

High-scale interactive services demand high throughput with low latency and high availability, difficult goals to meet with the traditional stateless 3-tier architecture. The actor model makes it natural to build a stateful middle tier and achieve the required performance. However, the popular actor model platforms still pass many distributed systems problems to the developers.

The Orleans programming model introduces the novel abstraction of virtual actors that solves a number of the complex distributed systems problems, such as reliability and distributed resource management, liberating the developers from dealing with those concerns. At the same time, the Orleans runtime enables applications to attain high performance, reliability and scalability.

This paper presents the design principles behind Orleans and demonstrates how Orleans achieves a simple programming model that meets these goals. We describe how Orleans simplified the development of several scalable production applications on Windows Azure, and report on the performance of those production systems.

1. Introduction

Building interactive services that are scalable and reliable is hard. Interactivity imposes strict constraints on availability and latency, as that directly impacts end-used experience. To support a large number of concurrent user sessions, high throughput is essential.

The traditional three-tier architecture with stateless front-ends, stateless middle tier and a storage layer has limited scalability due to latency and throughput limits of the storage layer that has to be consulted for every request. A caching layer is often added between the middle tier and storage to improve performance [9][14][19]. However, a cache loses most of the concurrency and semantic guarantees of the underlying storage layer. To prevent inconsistencies caused by concurrent updates to a cached item, the application or cache manager has to implement a concurrency control protocol [11]. With or without cache, a stateless middle tier does not provide data locality since it uses the *data shipping paradigm*: for every request, data is sent from storage or cache to the middle tier server that is processing the request. The advent of social graphs where a single request may touch many entities connected dynamically with multi-hop relationships makes it even more challenging to satisfy required application-level semantics and consistency on a cache with fast response for interactive access.

The actor model offers an appealing solution to these challenges by relying on the *function shipping paradigm*. Actors allow building a stateful middle tier that has the performance benefits of a cache with data locality and the semantic and consistency benefits of encapsulated entities via application-specific operations. In addition, actors make it easy to implement horizontal, “social”, relations between entities in the middle tier.

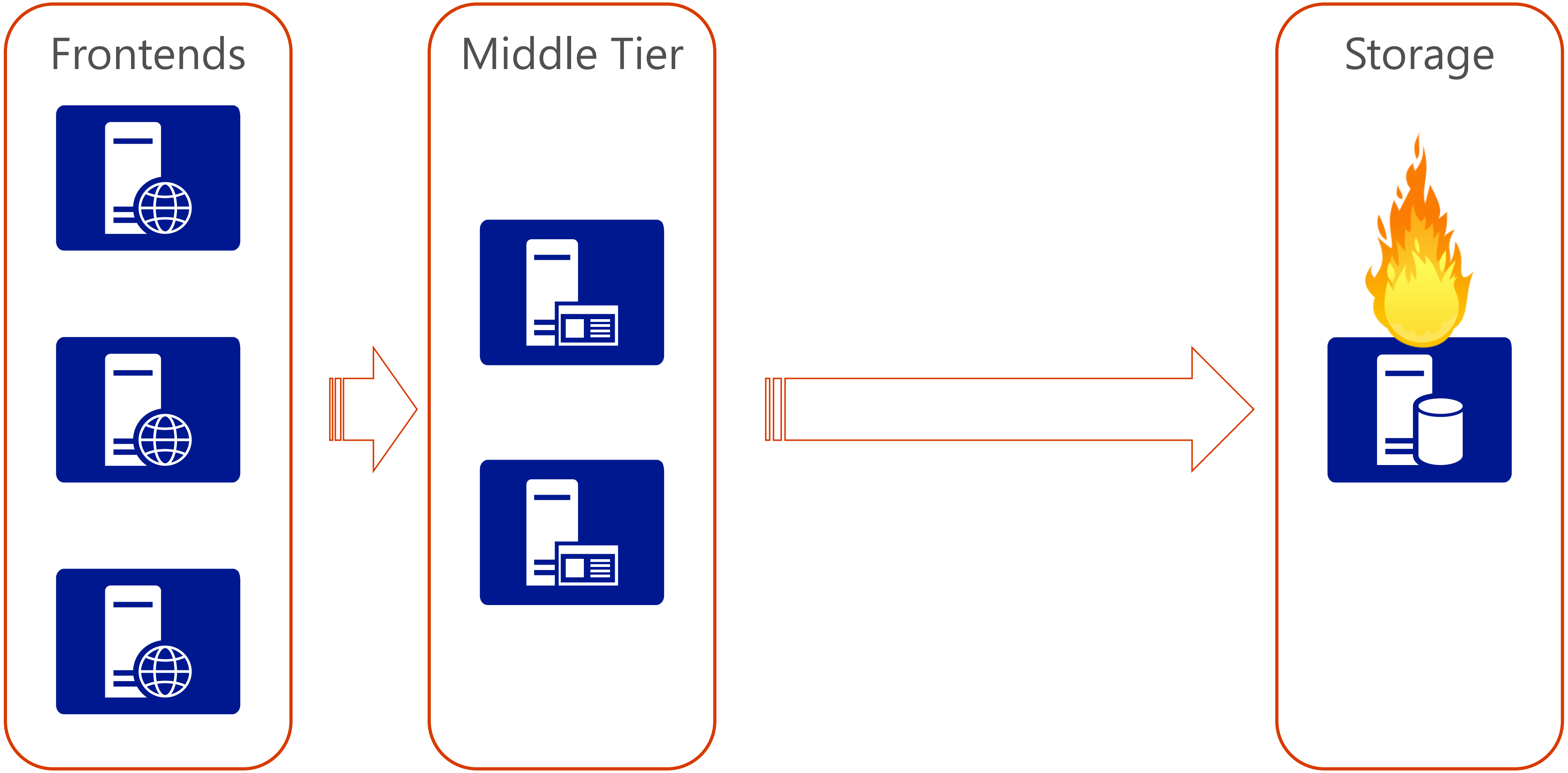
Another view of distributed systems programmability is through the lens of the object-oriented programming (OOP) paradigm. While OOP is an intuitive way to model complex systems, it has been marginalized by the popular service-oriented architecture (SOA). One can still benefit from OOP when implementing service components. However, at the system level, developers have to think in terms of loosely-coupled partitioned services, which often do not match the application’s conceptual objects. This has contributed to the difficulty of building distributed systems by mainstream developers. The actor model brings OOP back to the system level with actors appearing to developers very much like the familiar model of interacting objects.

Actor platforms such as Erlang [3] and Akka [2] are a step forward in simplifying distributed system programming. However, they still burden developers with many distributed system complexities because of the relatively low level of provided abstractions and system services. The key challenges are the need to manage the lifecycle of actors in the application code and deal with inherent distributed races, the responsibility to handle failures and recovery of actors, the placement of actors, and thus distributed resource management. To build a correct solution to such problems in the application, the developer must be a distributed systems expert.

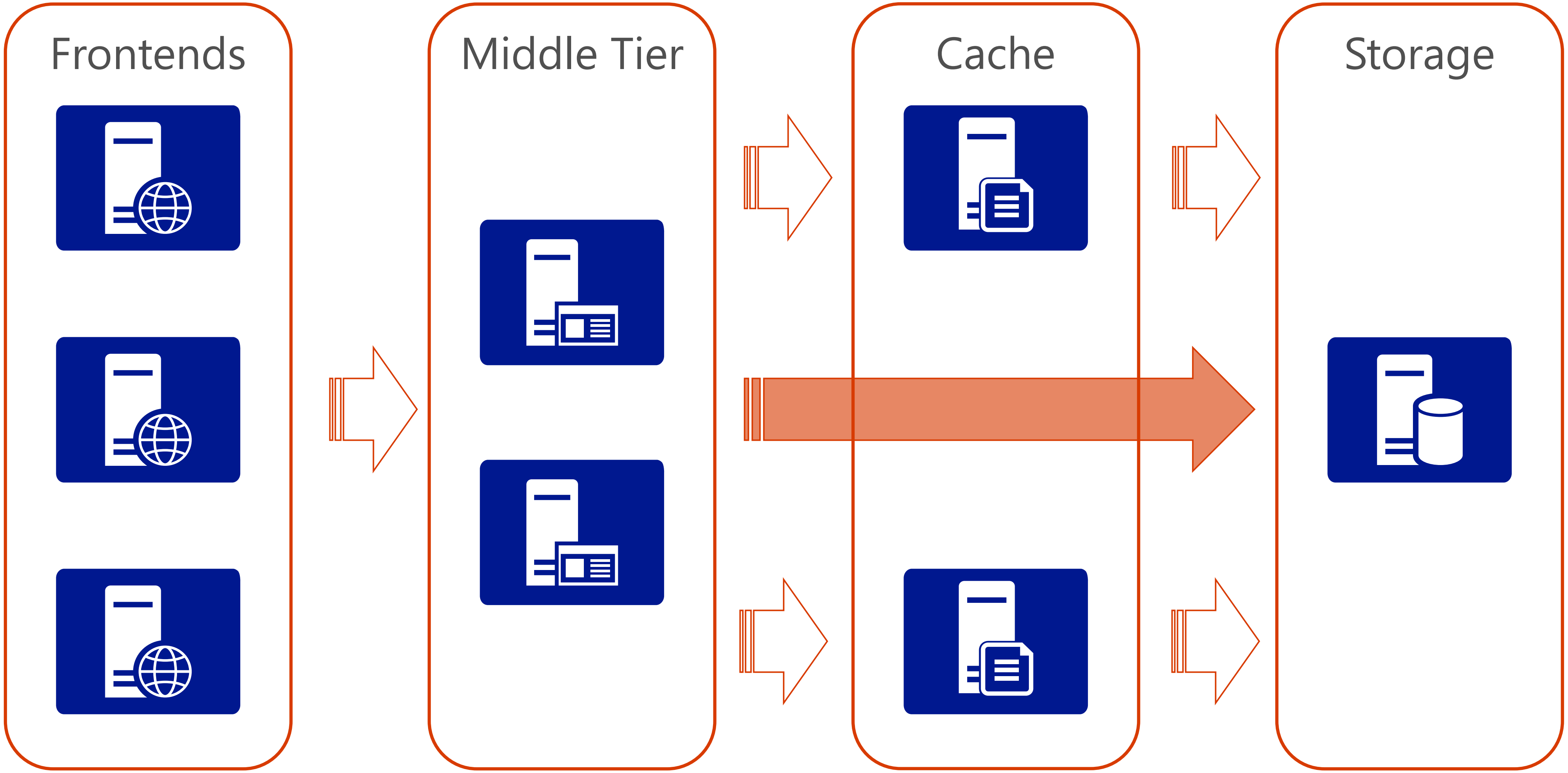
To avoid these complexities, we built the Orleans programming model and runtime, which raises the level of the actor abstraction. Orleans targets developers who are not distributed system experts, although our expert customers have found it attractive too. It is actor-based, but differs from existing actor-based platforms by treating actors as virtual entities, not as physical ones. First, an Orleans actor always exists, virtually. It cannot be explicitly created or destroyed. Its existence transcends the lifetime of any of its in-memory instantiations, and thus transcends the lifetime of any particular server. Second, Orleans actors are automatically instantiated: if there is no in-memory

www.microsoft.com/en-us/research/wp-content/uploads/2016/02/Orleans-MSR-TR-2014-41.pdf

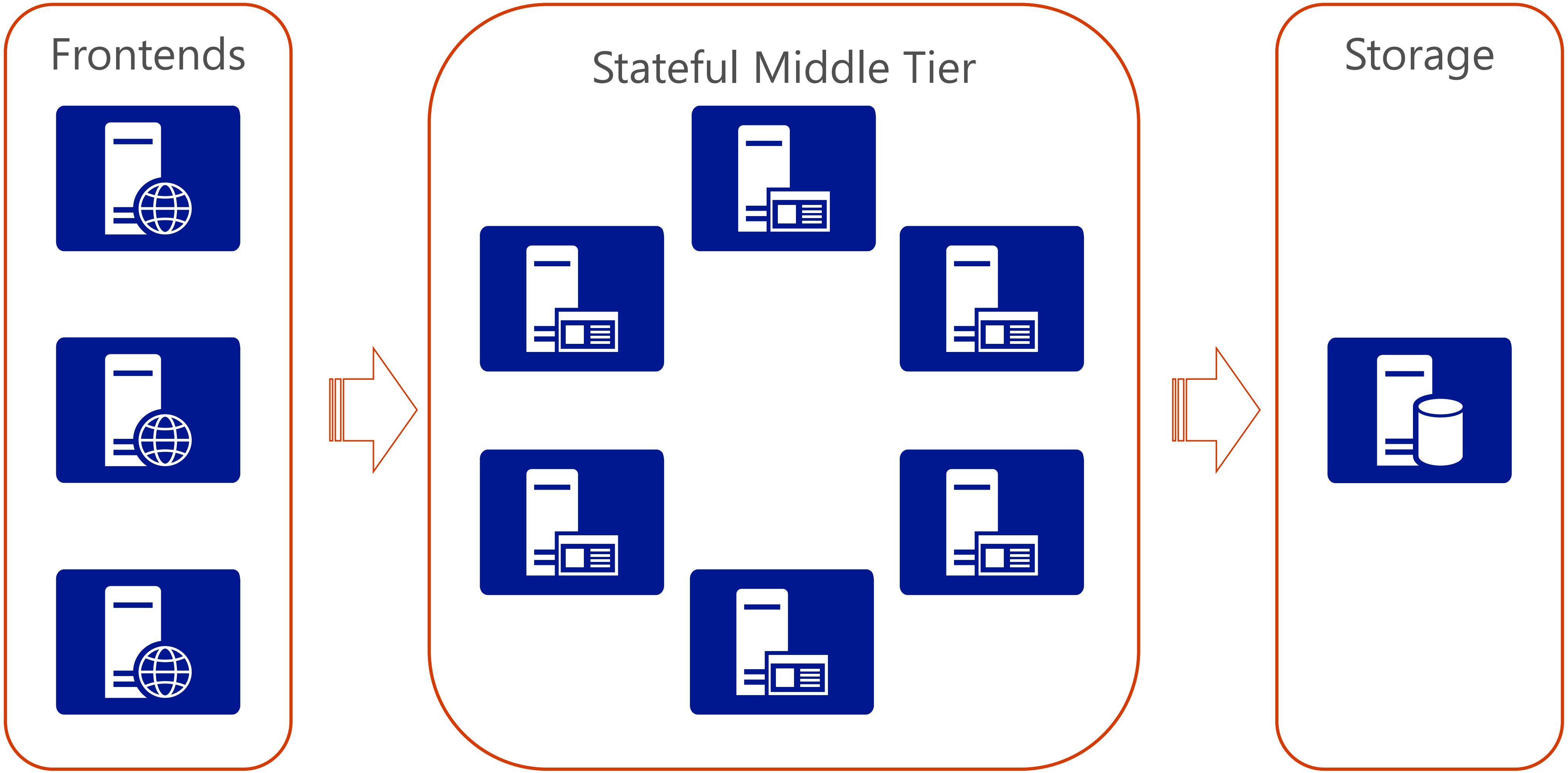
Orleans



Orleans



Orleans



Orleans

Isolated, individually addressable entities that exchange messages

Orleans: Actor Model for the Cloud

Programming model

For all developers

Simple yet powerful

3x –10x less code

Scalable by default

No inherent bottlenecks

No single points of failure

Known best patterns & practices

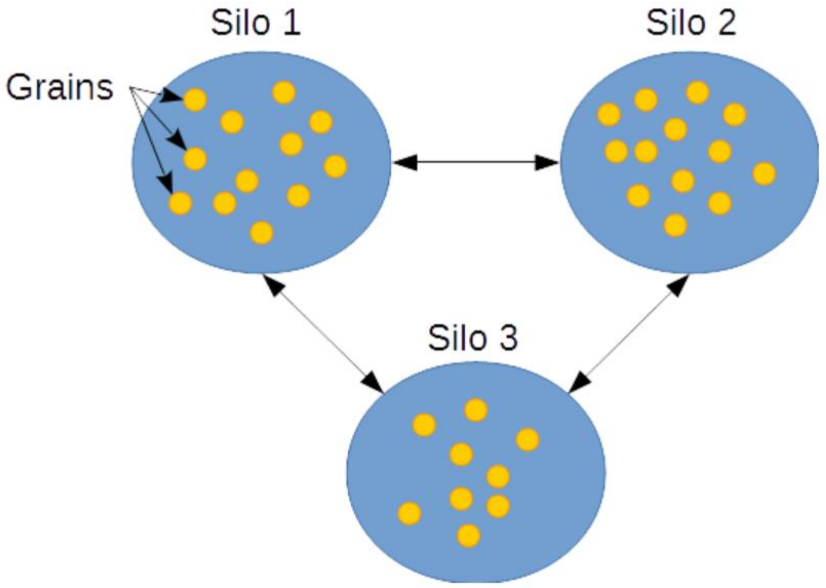
Grains: Virtual Actors

Actors always exist, virtually

- Needn’t be created, looked up or deleted
- Can call any actor at any time

Orleans Runtime performs heavy lifting

- Manages silos and grain location
- Transparently instantiates and GC’s actors
- Routes requests & responses, invokes actors
- Transparently recovers from server failures



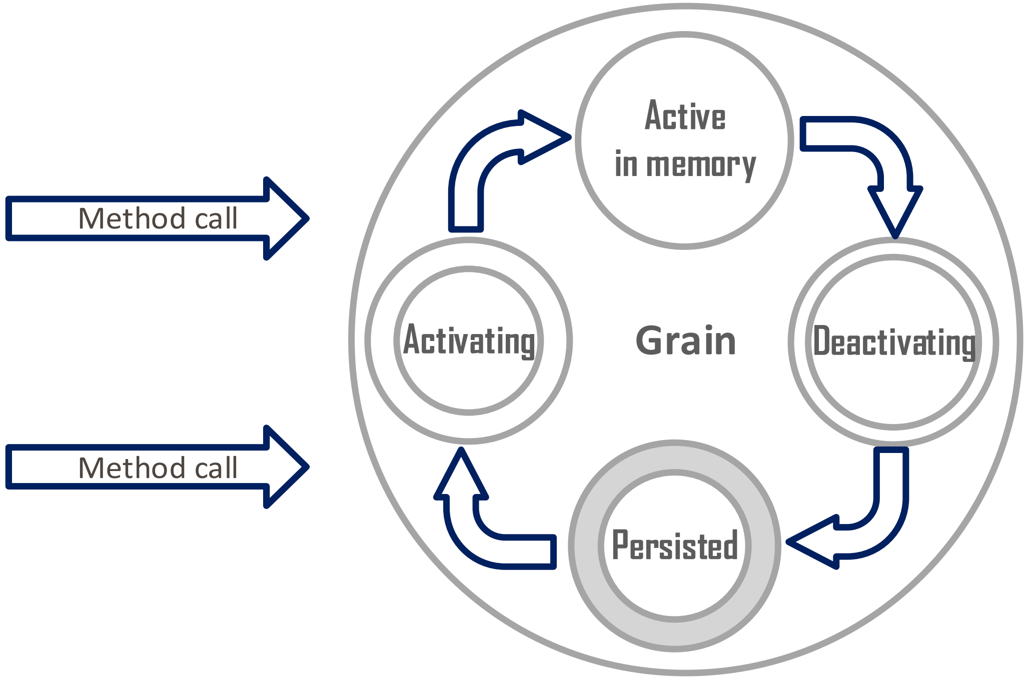
Basic anatomy of a grain

```
// grain interface
public interface IUser : IGrainWithGuidKey
{
    Task<string> SayHello(string name);
}
```

```
// invoking a grain
IUser user = GrainFactory.GetGrain<IUser>(userId);
string reply = await user.SayHello(name);
Console.WriteLine("User {0} said: {1}", userId, reply);
```

```
// grain implementation class
public class UserGrain : Grain, IUser
{
    private int _counter;

    public async Task<string> SayHello(string name)
    {
        return string.Format("Hello, {0}. You are caller #{1}",
            name, ++_counter);
    }
}
```

Methods are always single-threaded



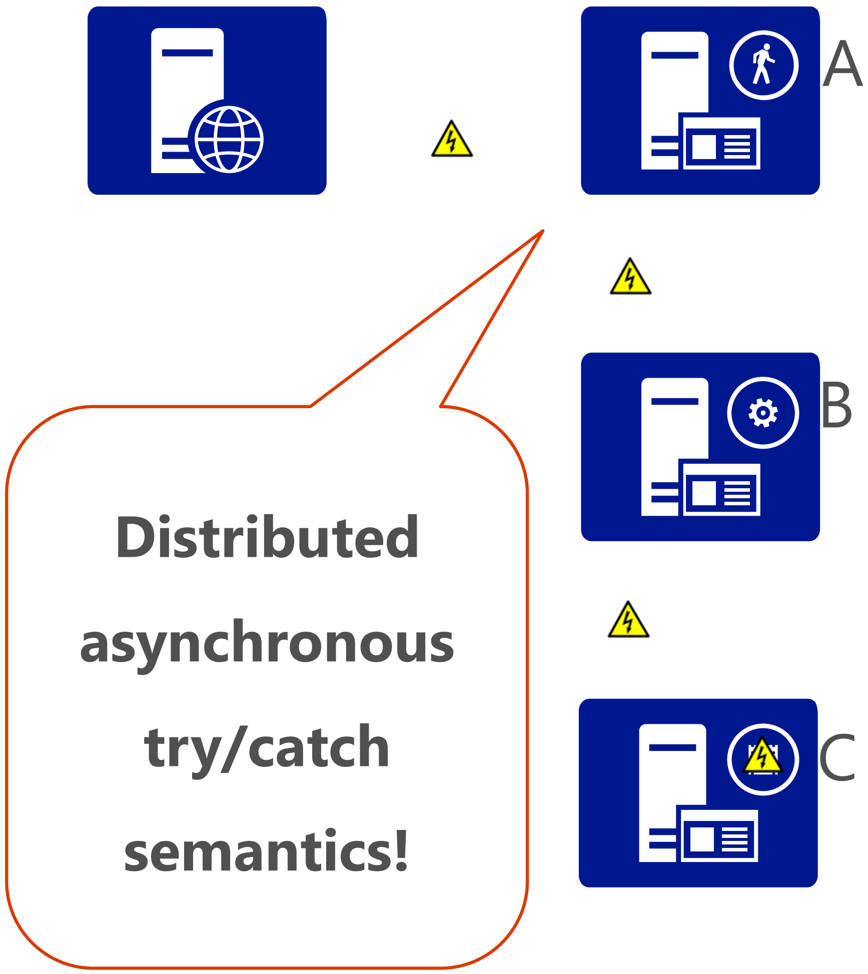
Distributed Try/Catch

```
public interface IUser : IGrainWithGuidKey
{
    Task AddFriend(IUser friend);
}
```

```
IUser me = GrainFactory.GetGrain<IUser>(myId);
IUser friend = GrainFactory.GetGrain<IUser>(friendId);

try
{
    await me.AddFriend(friend);
    Console.WriteLine("Added friend { 0}.", friendId);
}
catch (Exception exc)
{
    Console.WriteLine("Failed to add {0} as friend: {1}",
        friendId, exc);
    throw;
}
```

Grain reference as an argument

Handle exceptions as if local



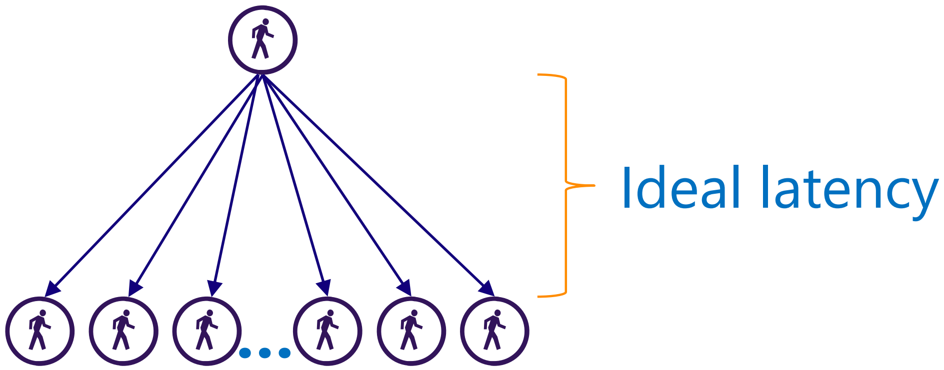
Async pattern

```
public class UserGrain : Grain, IUser
{
    private List<IUser> _friends;

    public async Task<string> GetFriendsStatus()
    {
        List<Task<string>> tasks = new List<Task<string>>();

        foreach (var friend in _friends)
            tasks.Add(friend.GetStatus());

        await Task.WhenAll(tasks);

        StringBuilder sb = new StringBuilder();

        foreach (var t in tasks)
            sb.AppendLine(t.Result);

        return sb.ToString();
    }

    public Task<string> GetStatus()...
}
```

Fan out

Join promises and await them

Process results when ready



No multi-threading

No blocking of threads

Easy parallelism

> Orleans is an important step in furthering a goal of the Actor Model that application programmers need not be so concerned with low-level system details.[i] For example, in moving to the current version, Orleans reinforces the current trend of not exposing customer Actors[72] to application programmers.[73]

Carl Hewitt. Actor Model of Computation for Scalable Robust Information Systems: One computer is no computer in IoT. Inconsistency Robustness, 2015, 978-1-84890-159-9. <hal-01163534v4>

Declarative persistence

```
public class UserState
{
    public string Name { get; set; }
    public DateTime LastLoggedIn { get; set; }
}

[StorageProvider(ProviderName = "UserProfileStorage")]
public class UserGran : Grain<UserState>, IUser
{
    public Task OnLogin()
    {
        State.LastLoggedIn = DateTime.UtcNow;
        return WriteStateAsync();
    }
}
```

Property bag as state class

Link to config

Type argument

One line to save state

Included providers

- Azure Table
- Blob
- SQL
- In-memory (for development)

Community built

- Redis
- Mongo
- Document DB, etc.

Advance features in Orleans

- Stateless workers
- Re-entrant Grains
- Grain call filters (‘Interceptors’)
- Observers (one-way asynchronous calls)
- Timers and reminders
- Streams

Orleans Video

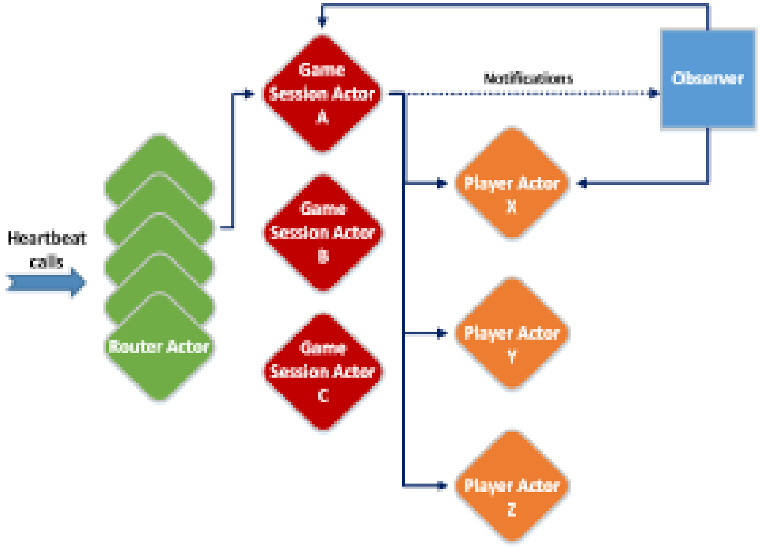
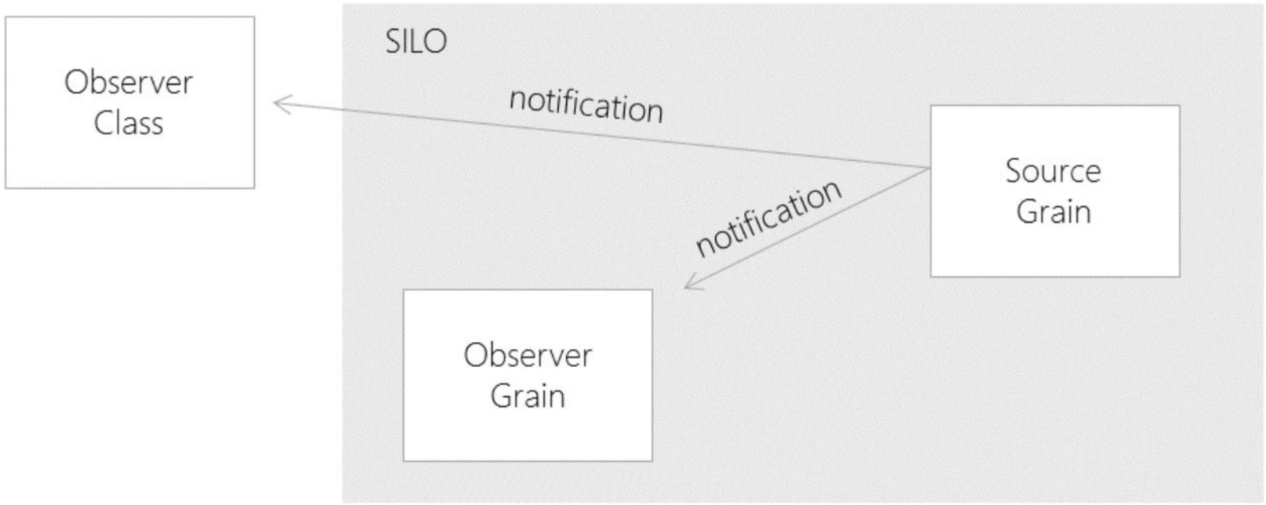


Figure 1: Halo 4 Presence Service

Demo



- Service Fabric Sample
  - Based on Stateless Calculator WCF Application Sample
  - Build on NuGet Microsoft.Orleans.ServiceFabric
- Implemented as Service Fabric stateless service
  - i.e. no replicated state
  - This sample doesn't implement persistence
- Simulates calculator in a grain
  - Set, add, subtract, etc
  - Observer pattern to track changes in calculator value
  - Timer increments value every few seconds

```
18 references
public interface  ICalculatorGrain : IGrainWithGuidKey
{
    6 references
    Task<double> Add(double value);
    4 references
    Task<double> Subtract(double value);
    4 references
    Task<double> Divide(double value);
    4 references
    Task<double> Multiply(double value);
    4 references
    Task<double> Set(double value);
    4 references
    Task<double> Get();

    4 references
    Task Subscribe(ICalculatorObserver observer);
}

14 references
public interface ICalculatorObserver : IGrainObserver
{
    8 references
    void CalculationUpdated(double value);
}
```

Run Orleans….

Orleans runs anywhere

- Can be hosted in any process
- Cloud or on-premises
- Azure, AWS, GCP, others
- Plugins for MySQL, ZooKeeper, Consul
- Open provider model
- Soon cross-platform

Imitation is sincerest form of flattery

- Orleans has a JVM clone “Orbit”
- Attempt to implement Virtual Actors in Go

ORBIT

What is Orbit?

Orbit is a framework to write distributed systems using virtual actors on the JVM. A virtual actor is an object that interacts with the world using asynchronous messages.

At any time an actor may be active or inactive. Usually the state of an inactive actor will reside in the database. When a message is sent to an inactive actor it will be activated somewhere in the pool of backend servers. During the activation process the actor's state is read from the database.

Actors are deactivated based on timeout and on server resource usage.

It is heavily inspired by the Microsoft Orleans project.