Orleans: Distributed Virtual Actors for Programmability and Scalability

BY MICROSOFT RESEARCH
PHILIP A. BERNSTEIN, SERGEY BYKOV, ALAN GELLER, GABRIEL KLIOT, JORGEN THELIN

PRESENTED BY RON YITZHAK
Introduction
Cloud computing platform combining diverse client devices – PCs, smartphones, etc.

Each client has unlimited, on-demand computation and data storage offered by cloud computing services.

Data and computation are no longer tied to a physical location or computing device.

Cloud computing combines many of the most difficult aspects of programming:

- Systems are inherently **parallel and distributed**.
- Running computations across **many servers** in **multiple data centers** and **diverse clients**.

Thus, programming cloud applications become a fundamental challenge.
Key Challenges

- High-scale interactive services demand:
  - High throughput – to support a large number of concurrent user sessions
  - Low latency & high availability – constrains that imposed by the interactivity demand
  - Scalability
  - Reliability
Key Challenges: Scale

- A successful and popular application must handle demand that grows orders of magnitude in a very short time.

- What is growing?
  - No. servers, no. network connections, size of data structures, cost of algorithms, likelihood to failures

- Techniques suitable for small websites, become bottlenecks and must be replaced by more scalable solutions.

- New scalable solutions often requiring architectural changes and rewrites of a system.

- Software should be elastic and capable of expanding to meet demand.
Reliability is a measure of the continuous delivery of correct service.

- Definition: Reliability of a system at time $t$ is the probability that the system operates without failures in the interval $[0, t]$.

- Time is specifying in a varies way depending on the nature of the system.

- High reliability is required when a system is expected to operate without interruptions.

- In many cases, we are interested not only in the probability of failure, but also in the number of failures and the time required to make repairs.
Key Challenges: Availability & Reliability

- Availability is the fraction of time that the system is in the operational state
- Often specified in terms of downtime per year
- Typically used as a measure for systems where short interruptions can be tolerated

<table>
<thead>
<tr>
<th>Availability</th>
<th>Downtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>90%</td>
<td>36.5 days/year</td>
</tr>
<tr>
<td>99%</td>
<td>3.65 days/year</td>
</tr>
<tr>
<td>99.9%</td>
<td>8.76 hours/year</td>
</tr>
<tr>
<td>99.99%</td>
<td>52 minutes/year</td>
</tr>
<tr>
<td>99.999%</td>
<td>5 minutes/year</td>
</tr>
<tr>
<td>99.9999%</td>
<td>31 seconds/year</td>
</tr>
</tbody>
</table>

Table 2.1. Availability and the corresponding downtime per year.
The Traditional Model
The Traditional Model V1: 3-Tier Architecture

- Stateless frontends
- Stateless middle-tier
- Storage is the bottleneck
  - Latency
  - Throughput
  - Scalability
- Data shipping paradigm
A client makes a service request

The service talks to the database and the database replies with some data

The service does some computation

A reply is sent to the client, the data disappears from the service
The Traditional Model V2: 3-Tier Architecture with Cache Tier

- Much better performance
- Lost concurrency - concurrent updates to a cached item require a concurrency control protocol
- Data shipping paradigm
The Actor Model
The Actor Model as **Stateful** Middle Tier

- Stateless frontends
- Performance of cache
- OOP paradigm regained
- Data locality
- Function shipping paradigm
- But there are still problems...
Data Locality

- The idea that requests are shipped to the machine holding the data that it needs to operate on.

- Benefits:
  - **Low latency**
    - Don’t have to hit the database for every single request, just in case that the data falls out of memory.
    - The number of network accesses is reduced.
  - **Data intensive applications**
    - If a client needs to operate a bunch of data it will all be accessible so a response can be returned quickly.
A client makes a request or starts a session

The database is accessed one time to get the data

The data moves into the service and *left there* after the request has been handled

The next time the client makes a request, it is *routed to the same machine* so it can operate on data that’s already in memory
Problems with Actor Model Frameworks

- App manages lifecycle of actors
- App has to deal with inherent distributed races
- App has to deal with actor failures and recovery
- App manages placement of actors – distributed resource management
- Developer must be a distributed systems expert
ORLEANS
Programming Model
and Runtime
Orleans: Programming Model & Runtime

- Orleans is a software framework for building client + cloud applications
- A novel virtual actor abstraction that enables a simplified programming model
  - Encourages use of simple concurrency patterns that are easy to understand and implement correctly
- An efficient and scalable implementation of the distributed actor model
Programming Model
Virtual Actors

- **Actors** are the basic building blocks of Orleans applications
- Virtual entities, not physical ones
- Used as the basic units of isolation and distribution
- Encapsulate behavior and mutable state, like any object
- Identified by a unique identity, composed of its type and primary key
- Interact with other actors only by sending messages

Grain = identity + behavior [+ state ]

Class User : Grain, IUser

User/jach@gmail.com

In-memory or persisted
Virtual Actors

- The virtualization of actors in Orleans remove from the application the need to explicitly:
  - Activate or deactivate an actor
  - Supervise actor’s lifecycle
  - Re-create an actor on failures
  - Scale-out actors on demand
Virtualization of Actors in Orleans (1)

- **Perpetual Existence**
  - Logical entities that always (virtually) exist, thus always addressable
  - Can not be explicitly created or destroyed
  - Its virtual existence is unaffected by the failure of a server that executes it
Virtualization of Actors in Orleans (2)

- **Automatic Instantiation**
  - **Activations** – in-memory instances of an actor that created automatically by Orleans' runtime
  - An actor may have zero or more activations
  - An actor will not be instantiated if there are no requests pending for it
  - When a new request is sent to an actor that has no activations, the runtime automatically creates an activation by picking a server and instantiating the object on that server
  - If a server fails, the runtime will automatically re-instantiate the actor on a new server on its next invocation
  - Runtime resource management is reclaiming unused in-memory actor’s instance
Virtualization of Actors in Orleans (3)

- **Location Transparency**
  - An actor may be instantiated in different locations at different times, and might not have a physical location at all
  - An application interacting with an actor does not know the actor's physical location
  - Orleans' runtime automatically instantiate a non-instantiated actor on-demand
Virtualization of Actors in Orleans (4)

- **Automatic Scale-Out**
  - **Stateless Worker Mode**: many independent activations of an actor are created automatically by Orleans on-demand to increase throughput
    - There is no state reconciliation between different activations of the same actor
    - Appropriate for actors with no-state (logical actors)
    - Can be used as read-only cache
Actors Client Code
Actor Interfaces

- Interaction between actors is made through methods and properties declared as part of **strongly-typed** interfaces.
- All methods and properties of an actor interface are required to be **asynchronous**, i.e. they must return a promise.

```csharp
public interface IGameActor : IActor
{
    Task<string> GameName { get; }
    Task<List<IPlayerActor>> CurrentPlayers { get; }
    Task JoinGame(IPlayerActor game);
    Task LeaveGame(IPlayerActor game);
}
```
An actor reference is a strongly-typed virtual actor proxy that allows actor and non-actor code to invoke methods on it.

Can be passed as an input argument to actor method calls.

How to obtain an actor reference:

- Orleans’ automatically generates factory class at compile time.
- Calling the `GetActor` method of the factory class and specify the actor’s primary key.
Actor References

- Actor references are virtual
- Does not expose to the programmer any location information of the target actor
- In contrast to traditional RPC models, the programmer doesn’t need to explicitly bind the object to the service
- Pros:
  - Simplifies programming
  - Maximizing throughput – no need to wait for binding or resolving service end point
Promises

- Actors interact by sending asynchronous messages implemented as method calls.
- Orleans’ methods return immediately with a promise for a future result rather than blocking until the result is returned.
- Allow for concurrency without explicitly thread management.
- Promise’s three-state lifecycle:
  - **Unresolved** – the initial state, represents the expectation of receiving a result at some future time.
  - **Fulfilled** – the result has been received, and it became the value of the promise.
  - **Broken** – an error occurs in the calculation or in the communication.
Promises in C#

- Usage: schedule a closure (or continuation) to execute when the promise is resolved
- Schedule is done implicitly using the `await` C# keyword on a promise
- The compiler transforms the code after `await` into a closure that executes after the promise is resolved
- Enable the developer to write code that looks sequential but executes asynchronously
Actor activations are **single-threaded** and do work in chunks, called **turns**

- An activation executes **one turn at a time**

- A turn can be a method invocation, or a closure executed on resolution of a promise

- Orleans may execute turns of different activations in parallel

- Two modes:
  - **Default** – waiting for an activation to finish processing one request before dispatching the next one
  - **Re-entrant** – turns belonging to different requests to the same activation of a re-entrant actor may be freely interleaved
Persistence

- The execution of actor requests may modify the actor state which may be persistent.
- An actor class can declare a property as `interface bag` which represents the persistent actor state.
- The runtime then provides each actor of that type with a `state` object.
- The `state` object implements the `interface bag` – methods for persisting and refreshing the state.
- Now it is up to the application when to checkpoint an actor's persistent state:
  - After each request is completed, timer based, number of requests based.
The execution of actor requests may modify the actor state which may be persistent.

If multiple activations of a grain concurrently modify their persistent state, the changes must be reconciled into a single consistent state.

Orleans runtime provides reconcilable data structures that track updates and automatically reconcile conflicting changes.

- Fine-grained reconciliation policies
- Mechanisms to implement a different reconciliation algorithm or other data structures
Runtime Implementation
Overview

- Orleans runs on a cluster of servers in a datacenter
- A server has three key subsystems:
  - **Messaging**
    - Connects each pair of servers with a single TCP connection
    - Uses as set of communication threads to multiplex messages between actors hosted on those servers
  - **Hosting**
    - Maintains a distributed directory to keep track of all actor activations in the cluster
    - Decides where to place activations
    - Manages activations' lifecycle
  - **Execution**
    - runs actors' application code on a set of compute threads with the single-threaded and reentrancy guarantees
Overview

Actor a client

Actor a Execution

Actor a Messaging

Actor a Hosting

Actor b

Actor b Execution

Method's invocation

Calls actor b: b.doSomething0

converts the method call into a message and passes it along with b's identity

passes b's identity to determine b's server

checks its distributed actor activations directory

b's server

serialized message

parameters deserialized into strongly typed objects

schedules for invocation
Distributed Directory

- Orleans keeping the location of each actor in a **distributed directory (DHT)**
- Each server in the cluster holds a partition of the directory
- Each record is a mapping: \( actorId \rightarrow location(s) \) of its activations
- When a new activation is activated or deactivated, a registration / un-registration request is sent to the appropriate directory partition
- Orleans maintains a large local cache on every server of the recently resolved mappings

- Allows **completely flexible placement** – enables the runtime managing system resources by placing and moving actors as the load on the system changes
Cooperative Multitasking

- Preemptive multitasking with a thread for each activation can not handle million activations per server

- **Cooperative Multitasking** –
  - Once started, an application turn runs to completion, without interruption
  - The OS never initiates a context switch from a running process to another
  - Instead, processes voluntarily yield control periodically or when idle or logically blocked
  - Orleans scheduler uses a small number of compute threads that it controls to execute all application actor code
  - Provides a much better overall system performance and CPU utilization
Reliability

- Orleans manages all aspects of reliability automatically
- Built-in membership mechanism
  - Failure detection & membership view – servers automatically detect failures via periodic heartbeats and reach an agreement on the membership view
- The lifespan of an individual actor completely decoupled from the lifespan of the hosting server
  - When a server fails, all activations on that server are lost
  - Since actors are virtual, no actor fails when a server fails
- The only part left for the developer is maintaining the persistent state
Eventual Consistency

- In failure-free times, Orleans guarantees that an actor only has a single activation.
- However, when failures occur, this is only guarantees eventually.
- After a server has failed but before its failure has been communicated to all servers, it may be that two activations of the same actor are registered in different directories.
- Once the membership has settled, one of the activations is dropped.
- **Availability over Consistency** trade-off to ensure that applications can make progress.
- Sufficient for most of the applications.
**Additional Runtime Properties**

- **Storing isolation** – actors on the same machine are completely isolated and communicate only through messaging
  - Using `Immutable<T>` objects to avoid deep copying
  - Serialization subsystem uses highly optimized copying module

- **Asynchrony** – all calls to actor methods are asynchronous
  - Simplicity – using async/await C# keywords
  - Scalability – preventing application code from blocking ensures that the system throughput is minimally impacted by the cost of remote requests
Additional Runtime Properties

- **Single-threading** – at most one thread runs at a time within each activation
  - Synchronization is unnecessary
  - The parallelism across many activations handling different requests is sufficient to efficiently use the available CPU resources
  - Leads to better overall system responsiveness and throughput
Applications

Halo 4 Video Game
Halo 4 Presence Service

Presence Service Responsibilities

- Keeping track of all active game sessions, their participating players, and evolving game status
- Allows joining an active game
- Enables real-time viewing of a game session

Integration with the game console

- Each game console makes regular heartbeats calls to the service to report its status of the game in progress
- In a multiplayer game session, each console sends its status independently

Persistency – no persistent storage, game state is only kept on-memory

- The ground truth is on the consoles
- In case of a failure, it takes one heartbeat to recover
Halo 4 Presence Service

- Incoming heartbeat requests from consoles arrive to the **Router actor**
- **Router** forwards the request to the right **Game Session actor** according the session ID
- The **Game Session actor** updates its internal in-memory state
- Periodically calls **Player actors** using the player IDs extracted from the heartbeat data
The code the developers had to write for making calls to actors was rather simple:

- Create a reference to the target actor based on its type and identity
- Immediately invoke a method on the promptly-returned actor reference object

There was no need to write code to locate or instantiate the target actor.

No need to manage failures of servers.
Halo 4 Statistics Service

- **Which kind of data?**
  - Results of completed and in-progress games with details of important game's related events
  - Players' achievements, rank progression, personal statistics
  - Handles queries about players' details sent by game consoles and the game's web site

- Halo 4 statistics are very important, as players hate to lose their achievements
- Therefore, any statistics report posted to the service is pushed through a Windows Azure Service Bus reliable queue
- All data can be recovered and processed in case of a server failure
Halo 4 Statistics Service

- The front-end receives an HTTP request with a statistics data and saves it in the Service Bus
- Worker processes pull the requests and call the corresponding Game Session actors
- The Game Session actor saves the data as-is to Azure BLOB store
- Then unpacks it and sends relevant pieces to the relevant Player actors
- Each Player actor processes its piece and writes the results to Azure Table store
- That data is later used for serving queries about player’s status, accomplishments, etc.
Halo 4 Statistics Service

- In case of a failure, the process resubmits the request
- No data loss
  - Except the rare case that a process dequeued a request and failed before submitting it
- Reduce number of I/O operations by using local cache in the Players actors, to be used for queries
Questions?
References

- Orleans: Distributed Virtual Actors for Programmability and Scalability by Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, Jorgen Thelin (Microsoft Research)
- Orleans: Cloud Computing for Everyone
- Availability & Reliability
- Shipping paradigms
- Orleans tutorial