Project Orleans
Distributed Virtual Actors for Programmability and Scalability

Philip A. Bernstein
Microsoft Research
Joint work with Sergey Bykov, Alan Geller, Gabriel Kliot, Michael Roberts, Jorgen Thelin

October 12, 2014
Presented at DISC 2014, Austin, TX

Project “Orleans” is a programming model and runtime for building *cloud native* services

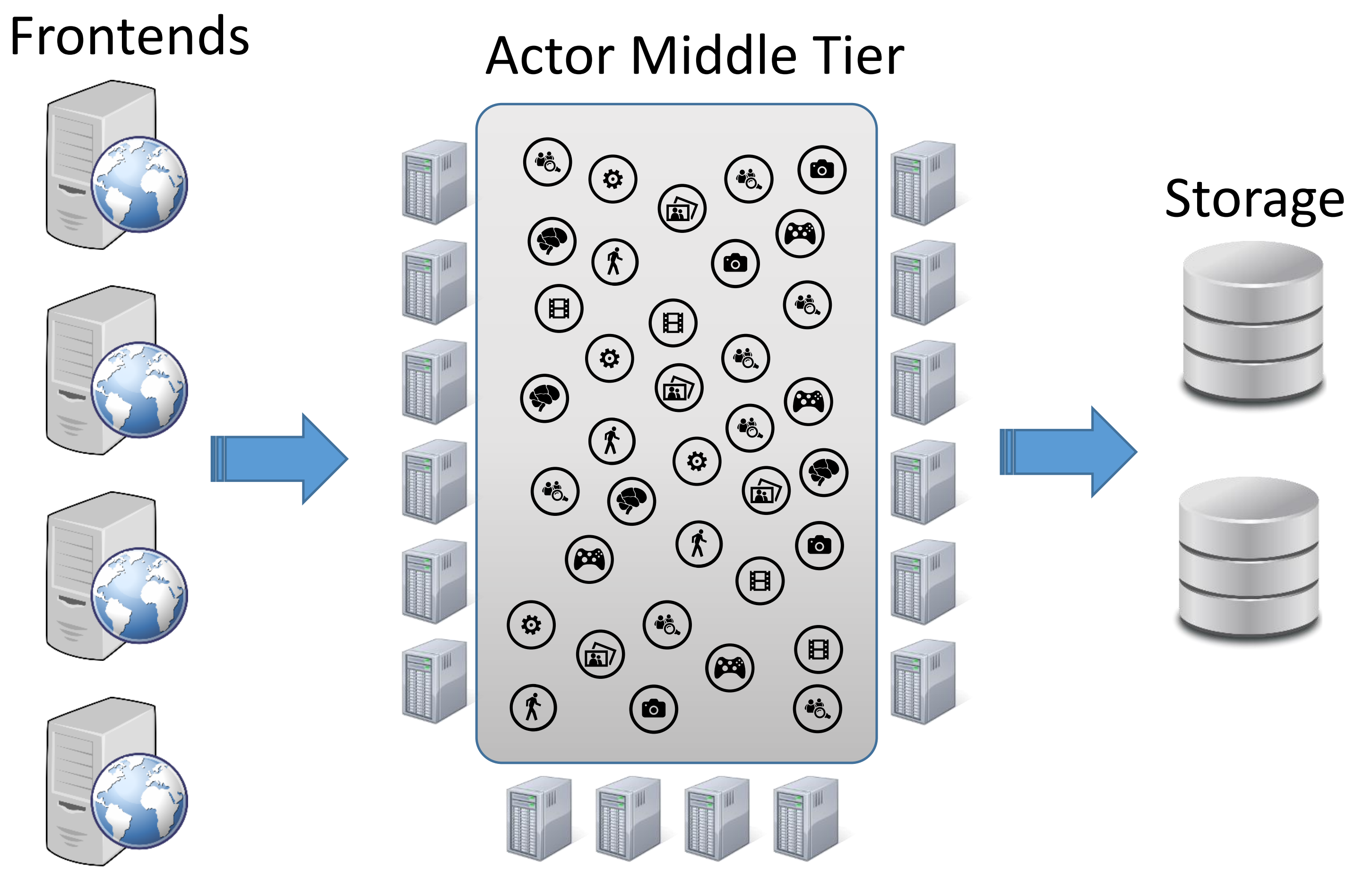
What is Project “Orleans”?

- Oversimplifying it: “Distributed C#”
  - Orleans runs your .NET objects on a cluster as if within a single process
  - Define .NET interfaces and classes, deploy to Azure, send requests to them
- Practically: “Toolset for building cloud-native services”
  - Encapsulates best practices for building scalable, reliable, elastic services
  - Framework for stateful near-real-time backends
  - 3-5x less and simpler code to write, scalability by default
- Academically: “Distributed virtual actor model”
  - Adaptation of the Actor Model for challenges of the Cloud
  - Actors that exist eternally and never fail

Motivation

- Developer Productivity
  - Concurrency, distribution, fault tolerance, resource management…
  - Modern workloads are even ‘worse’
  - Domain of distributed systems experts
  - Help desktop developers [and experts] succeed
  - *Write less code*
- Scalability by default
  - Designs and architectures break at scale
  - Failure to scale may be fatal for business
  - Code must be scale-proof – must scale out without rewriting

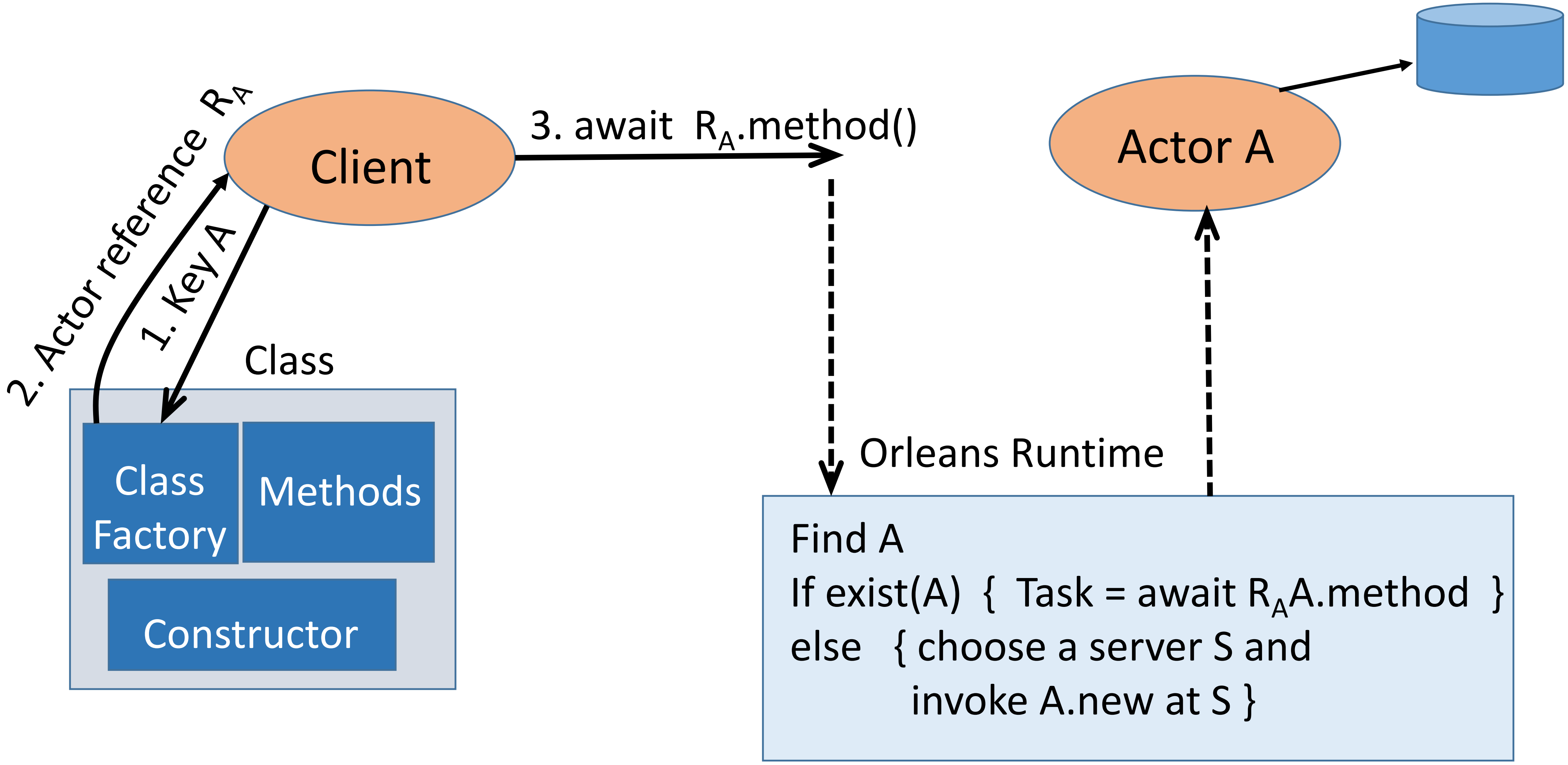
Actor Model as Stateful Middle Tier



Orleans Programming Model

- Each class has a key, whose values identify instances
  - Game, player, phone, device, scoreboard, location, etc.
- To invoke an actor A, the caller passes the key to its local class factory and gets back an actor reference R_A
- The actor invokes a method on R_A
- Method invocations are asynchronous
  - Return a “task” (i.e., a promise)
  - An attempt to reference a task’s result blocks the caller until the task completes
  - .NET has language support for this (Task-Await)

Invoking a method on actor A



Key Innovation: *Virtual* actors

1. Actor instances always exist, virtually
    - Application neither creates nor deletes them. They never fail.
    - Code can always call methods on an actor
2. Activations are created on-demand
    - If there is no existing activation, a message sent to it triggers instantiation
    - Transparent recovery from server failures
    - Lifecycle is managed by the runtime
    - Runtime can create multiple activations of stateless actors (for performance)
3. Location transparency
    - Actors can pass around references to one another or persist them
    - These are logical (virtual) references, always valid, not tied to a specific activation

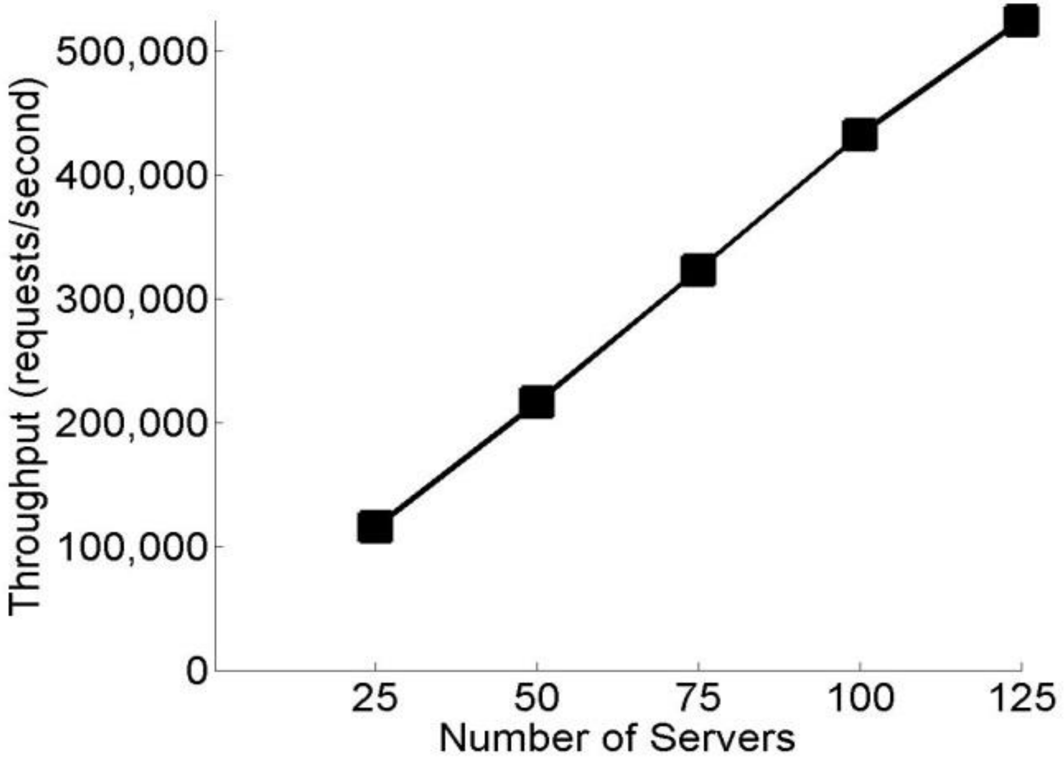
Actor State Management

- The runtime instantiates an actor by invoking the actor’s constructor
  - The constructor typically reads the actor’s state based on its key
  - Usually from storage, but possibly from a device (e.g. phone, game console, sensor)
- The actor saves its state to storage whenever it wants
  - Typically before returning from a method call that mutates its state
  - Or could be after *n* seconds, or after *n* calls, etc.
- Orleans does not support transactions (yet)
- Declarative persistence
  - Attach all state variables to an interface that inherits from IState
  - Declare a persistence provider for the class (Azure Table, Azure SQL DB, Redis…)
  - Invoke “WriteState” to save the state to the persistent store

Scalability

- Near linear scaling to hundreds of thousands of requests per second
- Also scalable in number of actors
- Multiplexed resources for efficiency
- Location transparency simplifies scaling up or down
- Elastic – transparently adjusts to adding or removing servers

Test Lab Numbers



Request: Client → Actor 1 → Actor 2

Orleans was built for…

Scenarios

- Social graphs
- Mobile backend
- Internet of things
- Real-time analytics
- ‘Intelligent’ cache
- Interactive entertainment

Common characteristics

- Large numbers of independent actors
- Free-form relations between actors
- High throughput/low latency
- Fine-grained partitioning is natural
- Cloud-based scale-out & elasticity
- Broad range of developer experience

- Not good for a service where different requests span different combinations of records over a large database

Other features

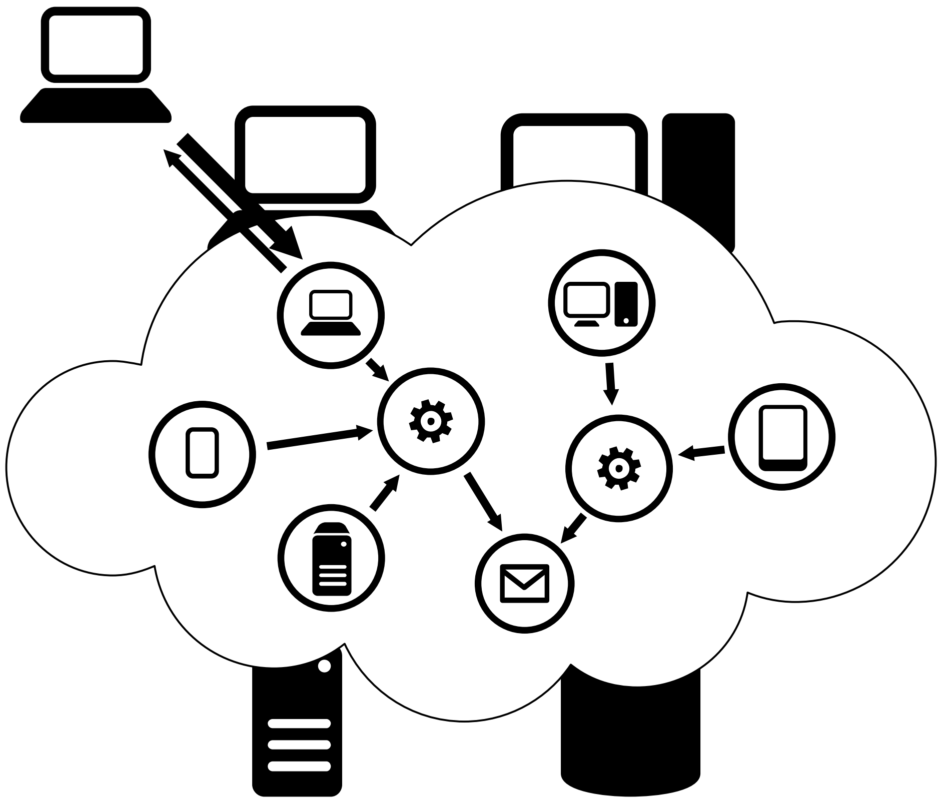
- Exceptions are automatically propagated
- Timers that live as long as the hosting activation
- Fault-tolerant timers, for infrequent events

Production usage

- Halo 4 - all back end services
  - Players, games, weapon caches, regions, scoreboards, ….
  - Dozens of services, 10s to 100s of machines each
  - 100Ks of requests per second
  - Bursty load (evenings, weekends) and peak load at product launch
- Back end services of many other game studios
- About ten other Microsoft services run on Orleans
  - Examples: intelligent cache, telemetry.
- Public preview since April 2014

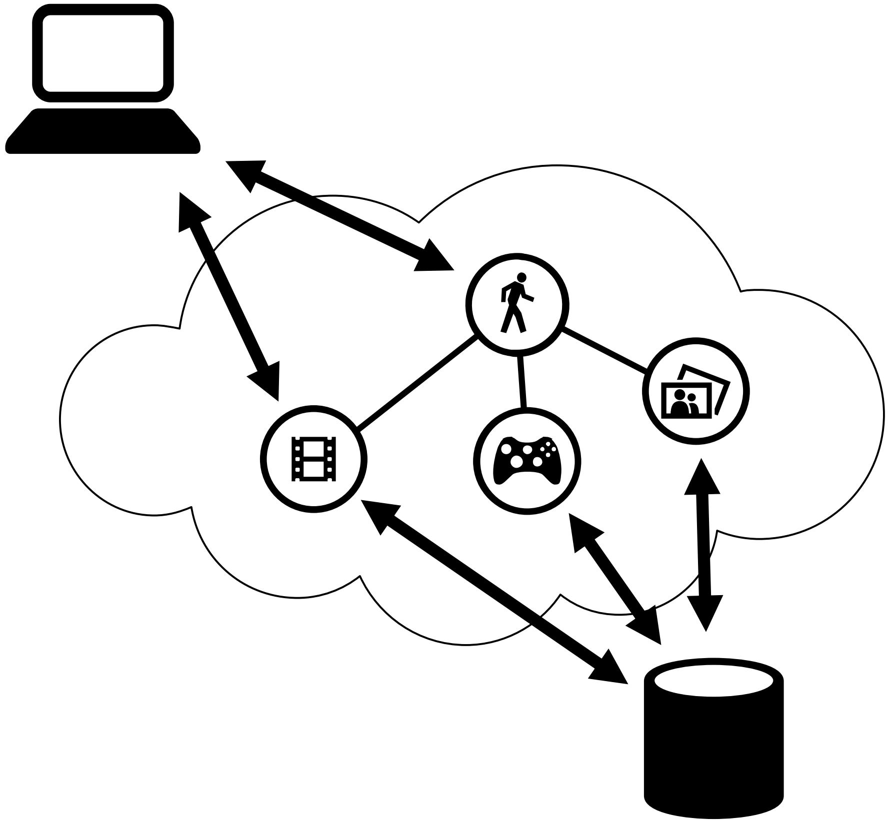
Near real-time analytics

- Devices send telemetry to the Cloud
- Per-device actors process and pre-aggregate incoming data
- Grouping by location, category, etc.
- Statistics, predictive analytics, fraud detection, etc.
- Control channel back to devices
- Elastically scales with number of devices and groupings



Intelligent cache

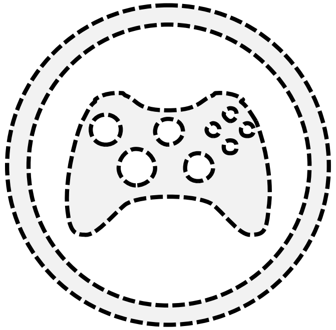
- Actors hold cache values
- Semantic operations on values
- Function shipping (method calls)
- Coordination across multiple values
- Automatic LRU eviction
- Transparent on-demand reactivation
- Write-through cache with optional batching



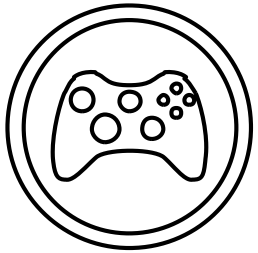
Outline

- ✓Orleans Overview
- Runtime Library
- Cluster Membership
- Actor Directory

Actors in Orleans

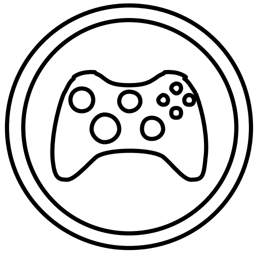


Game Actor Type



Game Actor (Instance)
#2,548,308

Game Actor #2,548,308
Activation #1 @ 192.168.1.1



Game Actor (Instance)
#2,031,769

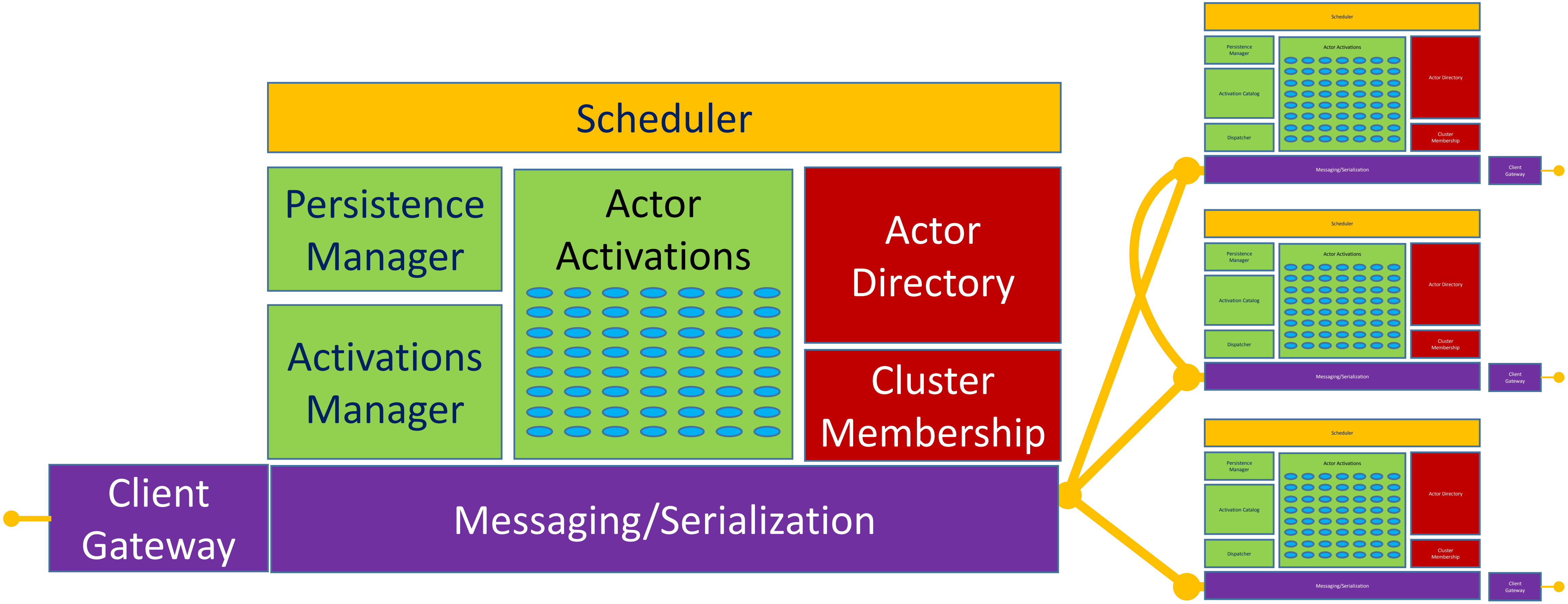
Game Actor #2,031,769
Activation #1 @ 192.168.1.5

Actor Type | **Actor (Instance)** | **Actor Activation**

Actor execution model

- Activations are single-threaded
  - Optionally re-entrant
  - Runtime schedules execution of methods
  - Multiplexed across threads
- No shared state
  - Avoid races
  - No need for locks
- Cooperative multitasking
  - Everything must be asynchronous

Distributed Runtime



Achieving Efficiency and Scalability

- Cooperative multitasking
- Multiplexed Communication
- Balanced placement
- Custom Serialization
- Support for Immutability

Outline

- ✓Orleans Overview
- ✓Runtime Library
- Cluster Membership
- Actor Directory

Cluster Membership

- A <u>cluster</u> is a set of servers
- Each server must know the identity of every other available server in its cluster
- Orleans uses reliable storage to store the consensus view
  - A table, with one row per server describing the server's state
- We use Microsoft Azure Table service
  - Supports optimistic concurrency control via http ETags.
    Read returns a row's ETag.
    Write only if the row's ETag is unchanged.
  - Supports transactions over rows with the same partition key

Cluster membership protocol

- The servers form a ring using consistent hashing
- Each server pings the next 3 in the ring, every few seconds.
- If a server S gets N successive failures to ping server T,
  S writes its timestamped suspicion into T’s row
- If T has more than M suspicions within K seconds, then
  - S writes that T is dead into T’s row, using an Etag to avoid lost updates
  - and broadcasts a request for all servers to re-read the membership table (which they’ll do anyway periodically)
- T kills itself upon learning it is dead.
  System infrastructure will restart it with a new name

It’s useful to totally order membership states

- Avoids two servers killing two other servers, and neither of them knowing right away about the other one’s actions.
- Serializes the joining of new servers to the cluster
  - Allows a new joining server to validate two-way connectivity to every other server that has already started.
  - Ensures that at least when a server starts, there is full connectivity between all servers in the cluster.

Totally ordering membership states

- So we add a membership-version row that tracks state changes
- Within a transaction, S writes that T is dead and,
  if S's membership-version is still fresh, S increments the version number in the membership-version row, else S aborts.
  - If S's membership-version was stale, it re-reads the membership state and re-runs the transaction
- That way the membership configurations are totally ordered with increasing version number

Algorithm properties

- Our algorithm can handle any number of failures.
  - I.e., does not require quorum.
  - We have seen production situations when over half of the servers were down.

- Our algorithm can handle thousands and probably even tens of thousands of servers
  - Paxos-based solutions generally do not scale beyond tens of servers

Outline

- ✓Orleans Overview
- ✓Runtime Library
- ✓Cluster Membership
- Actor Directory

Actor Directory: ActorID → ServerID

- Stored in a DHT, spread across all active servers
- Each server owns a partition of the key space
- Each actor is assigned to a partition by consistent hashing
- Directory enforces the single-activation constraint
- Each server caches recently used actor-to-server mappings
- A mapping entry can be wrong (stale cache, failed unregister)
  - Recipient of a misdirected message reroutes it or returns an exception
  - Sender and receiver correct the error by invalidating a cache entry or updating the directory entry

Server failures

- When a server F fails, its directory partition is lost
- When server S_1 learns of F's failure
  - It purges its directory partition of actor entries that map to F
  - It kills local actor activations that were mapped by F
- While resolving a failure, the directory might be inconsistent
  - We favor availability over consistency
  - This "eventual single-activation consistency" semantics has been the right tradeoff for most applications

Eventual single-activation consistency

- Single instancing may be compromised during recovery
  - Suppose actor α was mapped by F to server S_2
  - If S_2 is slow at learning of F's failure, a server with a cached entry for α may invoke α at S_2
  - Meanwhile, if S_1 invokes α, it will create a new directory entry for it at (say) server S_3 and activate it at (say) S_4

  ☹ So now there are two activations of α, at S_2 and at S_4
- Eventually, S_2 will learn of F's failure and kill α
  - α at S_2 might save its state during its rundown, which might conflict with state saved by α at S_4
  - If so, α at S_2 will have to merge its state with the one saved by α at S_4

Geo-distributed Actor Directory (prototype)

- Suppose an application is distributed in many clusters
- To ensure single-instancing, a request to activate actor α in cluster C_1 triggers a consensus protocol with other clusters
  - C_1 asks other clusters if they have a copy.
  - If all clusters reply “no”, then C_1 can safely instantiate α
  - If C_2 says “yes” at server SC_2, then C_1 maps α to SC_2
  - If a cluster C_3 doesn’t reply, then C_1 instantiates α anyway, favoring availability over consistency
- Optimistic consensus: instantiate α and then run consensus, if it’s very unlikely another cluster will instantiate α.
- When C_3 reunites with C_1, they run a reconciliation protocol for actors they created when they were disconnected.

Geo-distributed directory reconciliation

- When server membership changes, servers exchange lists of in-doubt actors.
- If ≥ 2 servers have instantiated an actor with at least one in-doubt
  - Then use a fixed precedence relation over server ID's to choose a winner.
  - Precedence could be global, per class, or per class+key

Conclusion

- Orleans Benefits
  - Significantly improved developer productivity
  - Makes cloud-scale programming attainable to desktop developers
  - Scalability by default. Excellent performance
  - Proven in multiple production services
- Main innovation: Virtual actor programming model
- Runtime algorithms
  - Cluster membership
  - Actor Directory
- Future work: transactions, streams, dynamic optimization

Website
http://research.microsoft.com/en-us/projects/orleans/

Get the public Community Preview:
http://aka.ms/orleans

Get the Samples:
https://orleans.codeplex.com/

Microsoft