Actor-Oriented Database Systems

Philip Bernstein
Microsoft

ICDE 2018
April 19, 2018
Most new services are written as stateful middle-tier applications
These apps do a lot of data management
But they are poorly served by data management technology
There are technical reasons for this
This is a research opportunity!
What's a Middle Tier?

Clients → Frontends → Middle Tier → Storage
Interactive services are built as a stateful, object-oriented middle tier
- Multi-player games, IoT, social networking, mobile, telemetry
- They comprise a large fraction of new app development
- Naturally object-oriented, modeling real-world objects

Examples of objects
- Gaming: players, games, grid positions, lobbies, player profiles, leaderboards, in-game money, and weapon caches
- Social: chat rooms, messages, photos, and news items
- IoT: sensors, virtual sensors (flood, break-in), buildings, vehicles, locations
Scenario

- Player logs into game console
- Console connects to cloud service, creating Player object
- Player object connects to a Game-Lobby object
- Game-Lobby runs an algorithm to group players into a Game
  - Returns a reference to the Game object to all players
Many micro-services are stateful middle-tier apps
- Data ingestion – event streams, real-time analytics
- Workflow – manage long-running jobs, e.g., ETL, resource allocation
- Smart contracts – workflows on blockchains

Example – merge event streams from 100K servers
- Index them, store them in batches, run standing queries

To scale out, they’re partitioned by keys or key-range
- Stream ID, workflow ID, contract ID

A partition is identified by a key = object
Objects are **active for minutes to days**, sometimes forever.

App manages **millions of objects**, streams, images, and videos, and huge knowledge graphs.

App does **heavy computation**: complex actions, render images, standing queries, compute over graphs, ... 

App does **heavy communication**: high-bandwidth message streams.
Service is highly available
Scale out to large number of servers
Compute servers must scale out independently of storage servers
... and independently of communication servers
Geo-distributed for worldwide low-latency access
Middle-tier Objects Comprise a Distributed DB

- Many (but not all) objects are persistent
  - Player is persistent, Lobby is not
- Active objects are in-memory for fast response
- Latest state is in main memory. Storage might be stale
  - Sensor object persists state periodically
Many of these apps are implemented using **actor systems**

- Simplifies distributed programming

Actors are objects that ...

Communicate only via asynchronous message-passing

- Messages are queued in the recipient's mailbox
- No shared-memory state between actors

Process one message at a time

- No multi-threaded execution inside an actor
Orleans is an open-source actor framework in C#  
https://dotnet.github.io/orleans/

- Invented the Virtual Actor model
  - Like virtual memory, actors are loaded and activated on demand
  - Deactivated after an idle period
- Supports scalability by load-balancing objects across servers
- Supports fault-tolerance by automatically reactivating failed objects
Orleans Programming Model

- Actor is fully-encapsulated and single-threaded

- Each class has a key, whose values identify instances
  - Game, player, phone, device, scoreboard, input stream, workflow, etc.

- Asynchronous RPC
  - Key.Method(params) returns a “task” (i.e., a promise)
  - **Await** Task - blocks the caller until the task completes
  - .NET has language support for this (Async-Await)
Calling an Actor’s Method

Client

PlayerKey_A.Move()

Orleans Runtime

Lookup Player_A’s location
If (Player_A is active)
{ invoke Player_A.Move }
else { activate Player_A on some server S;
    invoke Player_A’s constructor;
    invoke Player_A.Move at S }

Player_A

Storage

Placement Strategy
Fault Tolerance

- Actor can save state at any time, e.g., to storage.
- Runtime automates fault-tolerance.
- Orleans magic: A fault-tolerant DHT that maps actor-ID to server-ID.

```java
public class Account {
    int balance;

    Task Withdraw(int x)
    {
        if (balance >= x)
        {
            balance = balance - x;
            Save State;
            return (1);
        }
        else return (0);
    }
}
```
Good news / Bad news

Good news

- The virtual actor model automates scalability and fault tolerance

Bad news

- App is responsible for managing its state

Let’s help them out!
Actor-Oriented Database System (AODB)

- Indexes
- Transactions
- Queries
- Streams
- Views
- Triggers
- Replication
- Geo-distribution
Examples

- Index – Get all players in Paris
- Transaction – Player X buys a kryptonite shield
- Query – Get all players in Paris who are playing Halo with \( \geq 8 \) other players
- Stream – Watch player actions, looking for players who might be cheating
- View – the number of active instances of each game
- Trigger – notify a chess player when the other player made a move
AODB’s Distinguishing Features

- Developer friendly - Compatible with actor framework’s programming model
- Elastically scales out to hundreds of servers
- Data is in-memory and on cloud storage
- Works with any cloud storage system
  - Files, BLOBs, KV store, document (JSON) store, SQL DBMS
Been There, Done that

- Object-oriented database
- Persistent programming language
- Object-to-relational mapper
- Application server
- Main memory database
- Graph database
Object-Oriented Database

- C++ objects are mapped to persistent storage
  - Gemstone, Vbase, ObjectStore, O₂, Objectivity, ONTOS, Versant, ...
  - ODMG standard

- Target markets: CAD, telecom, scientific apps

- Like AODB, it’s compatible with the OO programming language

- Unlike AODB, it’s targeted at workstation apps, all shared state is in a custom storage system
Persistent Programming Language

- Annotate some program variables as persistent
- Variation: Persistence by reachability
- Very similar to OODB’s, but driven from a PL viewpoint
- Typically, the app runs in one OS process
- Negligible commercial market
- Examples – PS Algol, Galileo, Argus
Object-to-Relational Mapper

- Map OO classes to relational tables
- Translate queries and updates on classes into SQL on tables
- They’re popular, but only target SQL databases, no distributed transactions, ...
- Examples – Hibernate, .NET Entity Framework
**Application Server**

- Middle-tier objects communicate with DB’s
  - OLTP monitors (1970s & 80s) -> .NET transactions, J2EE (1990s)

- Each class executes as an OS process (not actor-oriented)
  - multi-threaded
  - synchronous RPC

- Static mapping of classes to servers

- Offers distributed transactions over DBMS’s that support XA interface

- Offers dynamic SQL or an object-to-relational mapper
Main Memory Database

- Like AODB, state is in main memory
- Unlike AODB . . .
- Manages records, not objects
- Not integrated with OO programming language
- Doesn’t scale to large number of servers
Graph Database

- Nodes are passive data, not active objects
- Could be a storage target for actors
Why do it again?

- Different combination of requirements ...
- Scalable to large number of servers
- Highly available
- Uses cloud storage
- Storage independence
- Geo-distributed for worldwide low-latency access
Scalability Implies …

- Limited ability to co-locate functionality
- Functionality must be parallelizable
- Scale-out is more important than a fast path
High Availability Implies …

- Tolerates server failures
- Fast recovery from failure
- Add or remove servers without shutting down
- Best effort to tolerate storage unavailability
Storage Independence Implies …

- Works with any cloud storage system
- Works for persisted and non-persisted objects
- Doesn’t require DB-feature-support by the storage system
- Should benefit from DB-feature-support by the storage system
- Copes with latency of cloud storage
Elastically scale out to hundreds of servers
Data is in-memory and on cloud storage
Works with any cloud storage system
Works for persisted & non-persisted objects
Limited ability to co-locate functionality
Tolerates server failures
Fast recovery from failure

Functionality is parallelizable
Scale-out is more important than a fast path
Add/remove servers without shutting down
Tolerates storage unavailability
Doesn’t need built-in storage system support
Benefits from a storage system’s built-in support
Copes with latency of cloud storage
Let’s Explore Features

- Transactions
- Geo-distribution
- Indexing
- Queries
Transactions

- Programming model
  - App server model is fine

- Performance challenges
  - No shared log
  - Cloud storage latency
  - Object migrate between servers
  - Many/most transactions are distributed

```csharp
public interface IAccountActor
{
    [TransactionOption.Required]
    Task Withdraw(uint amount);

    [TransactionOption.Required]
    Task Deposit(uint amount);

    [Transaction(TransactionOption.Required)]
    Task&lt;uint&gt; GetBalance();
}
```
Transaction Implementation

- TM coordinates 2PC
- Objects are participants
Early Lock Release

- Problem: object remains locked until it receives Commit
- When object o receives Prepare, it releases T₁’s lock
- If T₂ reads/writes o, it takes a “commit dependency” on T₁
  - TM commits transactions in dependency order
- When T₂ terminates, it releases locks, allowing T₃ to read/write o. Etc.
- Cascading abort is possible only due to server failure
- When T₁ commits, [T₂, T₃, …] prepare in a batch (= group-commit).
Early Lock Release (cont’d)

Benefits

- Conflicting transactions execute in parallel with 2PC
- Enables group commit without a shared log
- Up to 20x throughput improvement

_NOTE:_ Single-object transaction must ask TM to validate its dependency
Solution: One TM per Object

- Single-object transactions resolve dependencies locally
- Other benefits
  - No central TM bottleneck or single point-of-failure
  - Less configuration complexity
  - TM’s are naturally geo-distributed, with the objects
Geo-Distribution

- Extend single-instance invariant world-wide
  - Requires a global mutual-exclusion protocol on actor activation
- Multi-master replication
  - Programming model – eventually linearizable
Updates are specified as functions and queued locally.

App sees a **local state** and **global state** of each actor.

Can read confirmed state

 Optionally with local updates applied.

Can read global state with local updates applied (slow).
Versioned Actor

Updates are applied asynchronously to the global state.

Application Code

Confirmed State

Update
Update
Update

Local

Remote

Global State

Actual State

Updates are applied asynchronously to the global state.
All changes to global state are pushed to confirmed state

Updates are removed from queue when confirmed
Each Orleans class has a unique key

Support indexing of other members

```csharp
public class PlayerProperties {
    public int Rank { get; set; }

    [Index]
    public string Location { get; set; }
}

public interface IPlayer : IndexableActor<PlayerProperties> {
    Task Move(Direction d);
    Task<string> GetLocation();
}
```
Indexing Examples

- Ensure every player has a unique email address
- Offer an ad hoc tournament to all Halo players who are on-line
- Identify all players with weapons stashes in a given location
- Survey all players who logged in after 3PM today
Index Requirements

- Can index persistent and non-persistent actors
- Leverage actor storage that supports indexing
- Works if storage does not support indexing
- Can index active actors only
- Both hashed and B-tree indexes must scale out
- Plus unique indexes, transactional consistency, fault tolerance, ...
Queries over Actors

- Extent – all actors of a class, all active actors, explicit collection, and index
- Split execution between active and inactive actors
- Joins and aggregates
  - Reward the player with the best score in the last 15 minutes of a Microsoft game
- Materialized views – can use mid-tier caching technology
- Streams – Dynamically reconfigure distributed operators
- Triggers – for reactive applications
Developers of mid-tier stateful applications need our help

Whatever database topic interests you, there’s an opportunity to help
P.A. Bernstein, M. Dashti, T. Kiefer, D. Maier: Indexing in an Actor-Oriented Database. CIDR 2017


T. Eldeeb, P. Bernstein, “Transactions for Distributed Actors in the Cloud”, MSR-TR

P.A. Bernstein, S Bykov, A. Geller, G. Kliot, J. Thelin: Orleans: Distributed Virtual Actors for Programmability and Scalability, MSR-TR

Sebastian Burckhardt
Sergey Bykov
Natacha Crooks
Mohammad Dashti
Tamer Eldeeb
Jose Faleiro
Alan Geller
Tim Kiefer
Alok Kumbhare
Gabriel Kliot
David Maier
Christopher Meiklejohn
Muntasir Rahman
Vivek Shah
Adrienne Szekeres
Jorgen Thelin
Alejandro Tomsic