Stream Processing in an Actor-Oriented Database System

Philip Bernstein
Microsoft Research

BIRTE 2019
August 26, 2019
Most new interactive services are stateful, object-oriented middle-tier applications

They need database technology, but are currently poorly served

One such technology is stream processing
Interactive Application Services

Clients

Internet of Things (IoT)

Multi-Player Games

Social Networking

Mobile Apps

Telemetry
What’s a Middle Tier?

Clients → Frontends → Middle Tier → Storage
Stateful Object-Oriented Applications

- These applications manage state, usually represented as objects
  - 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: thermometers, motion detectors, cameras, GPS receivers, and virtual sensors built on top (room presence, traffic jams)
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

Game object reports activities as a stream of events

Game object writes to Scoreboard object
  At the end, it might update the Leaderboard
Stateful Micro-Services

- Many micro-services execute stateful middle-tier OO apps
  - Data ingestion – event streams for real-time analytics
  - Workflow – manage long-running multi-step jobs
  - Smart contracts – workflows on blockchains

- Example – merge event streams from 100K servers
  - Index them, store them in batches, run continuous queries, publish query results to dashboards

- These services aren’t naturally object-oriented
  - But for scalability, OO is a good design approach
Application Properties

- 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, image rendering, continuous queries, computations over graphs, ...
- App does **heavy communication**: high-bandwidth message streams
Service is highly available

Compute, storage, and communications must scale out independently

That’s why the three-tier architecture is popular
Many of these apps are implemented using an **actor system**
- Greatly 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 Actor Programming Framework

- 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

- Object 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

Player_A

Storage

Placement Strategy

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 }

Orleans magic: A fault-tolerant DHT maps object-ID to server-ID
Fault Tolerance

- Object can save state at any time, e.g., to storage

- Runtime automates fault-tolerance

```java
public class Account {
    int balance;

    Task async 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 treat the app as a database of objects
  - Offer standard database abstractions
Actor-Oriented Database System (AODB)

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

- **Transaction** – Player X buys a kryptonite shield
- **Index** – Get all players in Los Angeles
- **Query** – Get all players in L.A. who are playing Halo with $\geq 8$ other players
- **View** – the number of active instances of each game
- **Trigger** – notify a chess player when it’s his/her move
- **Stream** – Watch player actions, looking cheaters
Actor-Oriented Database (AODB) Unique Requirement

- Storage independent, using cloud storage
- In particular, stream-transport independent. It should work with
  - Azure Event Hubs
  - Azure ServiceBus
  - Azure Queues
  - Apache Kafka
  - TCP/IP messages
  - ...
AODB Streams Requirements

- Allow fine-grained free-form compute over stream data
- Allow stream topology and processing logic to change dynamically
- Example – A stream per online user
  - Users come and go
  - Their interests change – weather location, sports, flight status, stock
  - ... based on external context not on events in the stream
- Example – detect new ways of cheating in an online game
  - Re-route certain events to a cheat detector object
  - Change the logic of the cheat detector
And of course the system must be …

- Scalable
- High throughput
- Low latency
- Highly available
Conceptual System View

- **Sensors**
- **Devices**
- **Data Sources**
- **Streams**
- **Stateful Processing Agents**
- **Output**
Actor Model Clusters Storage Writes

- Events relevant to an object are sent to that object
  - E.g., a player in a game, or a room in an IoT system
- The object decides when to write to storage
- Alternative model: cluster writes based on event type
  - Many event types are relevant to the same object
  - Too many writes
  - Writes to the object conflict
Orleans Streams

- A highly customizable pub-sub system
  - Defines the programming model and its implementation
  - Any Orleans object can be a stream producer or consumer
  - The queue manager is a plug-in (wrapped by a Queue Adaptor)

- A consumer can:
  - run any .NET code: C#, Trill, .NET Reactive Extensions, state machine, ...
  - call other objects, e.g., for notification

- Flexible, dynamic stream topology
Programming model

1. Object calls stream provider to get a stream based on GUID+Namespace (a local call)

2. To consume from a stream, an Object subscribes to it, which returns a subscription handle

3. Producer calls `Stream.OnNext` to send an event to all subscribers
Stream Provider

- Can be a lightweight driver
- Can contain substantial logic
  - Split a firehose into fine-grained streams
  - Aggregate fine-grained streams into a firehose
  - Replicate events into many streams
Virtual Streams

- Like a virtual actor, a stream always exists
  - It is activated on demand by sending events to it or subscribing to it
- Each subscription is durable
  - An object (subscriber) must explicitly **Unsubscribe**
- If an object deactivates and later is reactivated, it must invoke **SubscriptionHandle.Resume()** to reattach event-processing logic
  - It typically does this in its **OnActivate** method
  - If it didn’t persist its subscription handles, then it can get them by calling **GetAllSubscriptionHandles**
Event Ordering

- Stream provider determines the event order between producer and consumer

- A producer can pass a `StreamSequenceToken` to the `OnNext` call
  - The `StreamSequenceToken` is delivered with the event so the consumer can reconstruct event order

- An object can checkpoint its state with its `StreamSequenceToken`
  - At recovery, the object loads its state and passes the `StreamSequenceToken` to `Subscribe` to identify the first event it should receive
  - Only some stream providers support this “rewinding”
How - Components

- Orleans server process instantiates **Pulling Agents** that get messages from the queues
- Each **Pulling Agent** loads a **Stream Provider** for the specified queuing service
  - Generic provider code is abstracted into **Queue Adaptors**
- **Queue Balancer** balances work across pulling agents and servers to prevent bottlenecks and support elasticity. It’s customizable.
- **Pub-Sub** tracks all stream subscriptions, persists them, and matches stream consumers with stream producers.
Flow Control

- Agent delivers events to consumer via async RPC
  - Sends a small batch and wait for completion before sending the next batch
- A per-agent cache buffers the event stream
  - Decouples dequeuing events from delivering them to consumers
  - As the cache fills, the agent slows the dequeuing rate, thereby applying backpressure
Open source since Orleans V1, January 2015
  http://dotnet.github.io/orleans

Used by Halo and other Microsoft games

Many 3rd-party users
  Over 500 issues in Orleans GitHub mention streams

Developed by Sergey Bykov, Jason Bragg, Alan Geller, Gabriel Kliot, Jim Larus, Ravi Pandya, Jorgen Thelin
Other Database Features

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

- **Indexing**
  - P.A. Bernstein, M. Dashti, T. Kiefer, D. Maier: Indexing in an Actor-Oriented Database. CIDR 2017

- **Geo-distribution**
Orleans

http://dotnet.github.io/orleans