Relational Actors

Rodrigo Laigner

Presentation given for course Distributed Systems – 2019.1 at PUC-Rio

Outline

- Introduction
- Actor Programming Model
- ReactDB
- IoT and Actor-Oriented Databases
- Case Study Implementation
- Evaluation
- Conclusion

Introduction

- Latency-sensitive OLTP applications has emerged in diverse areas
  - Computer games, high-performance trading, and web
- Databases are moving towards in-memory storage
  - Hardware systems are integrating increasingly more cores in a single machine
- New requirements for database architecture
  - Efficiency in multi-core machines and careful design of concurrency control strategies
- There is a lack of abstractions to reason about the parallelizable application logic

Introduction

- Challenges on adapting database architecture without affecting application code
- Reactors model logical computational entities
  - Encapsulating state abstracted as relations
- Objective is to address architectural flexibility and high resource efficiency in multi-core machines
- Then, Shah & Salles (2018) designed an in-memory database system that exposes reactors as a programming model

Programming Model

- Actor programming model
  - Desirable primitives for concurrent and distributed programming
  - Each communication is described as a message
- Reactors are special types of actors
  - Model logical computational entities encapsulating state abstracted as relations
  - They can represent application–level scaling units such as accounts in a banking application
  - Classic database programming features such as declarative querying over the encapsulated relations
  - Transaction across multiple reactors provides serializability guarantees as in traditional databases

Programming Model

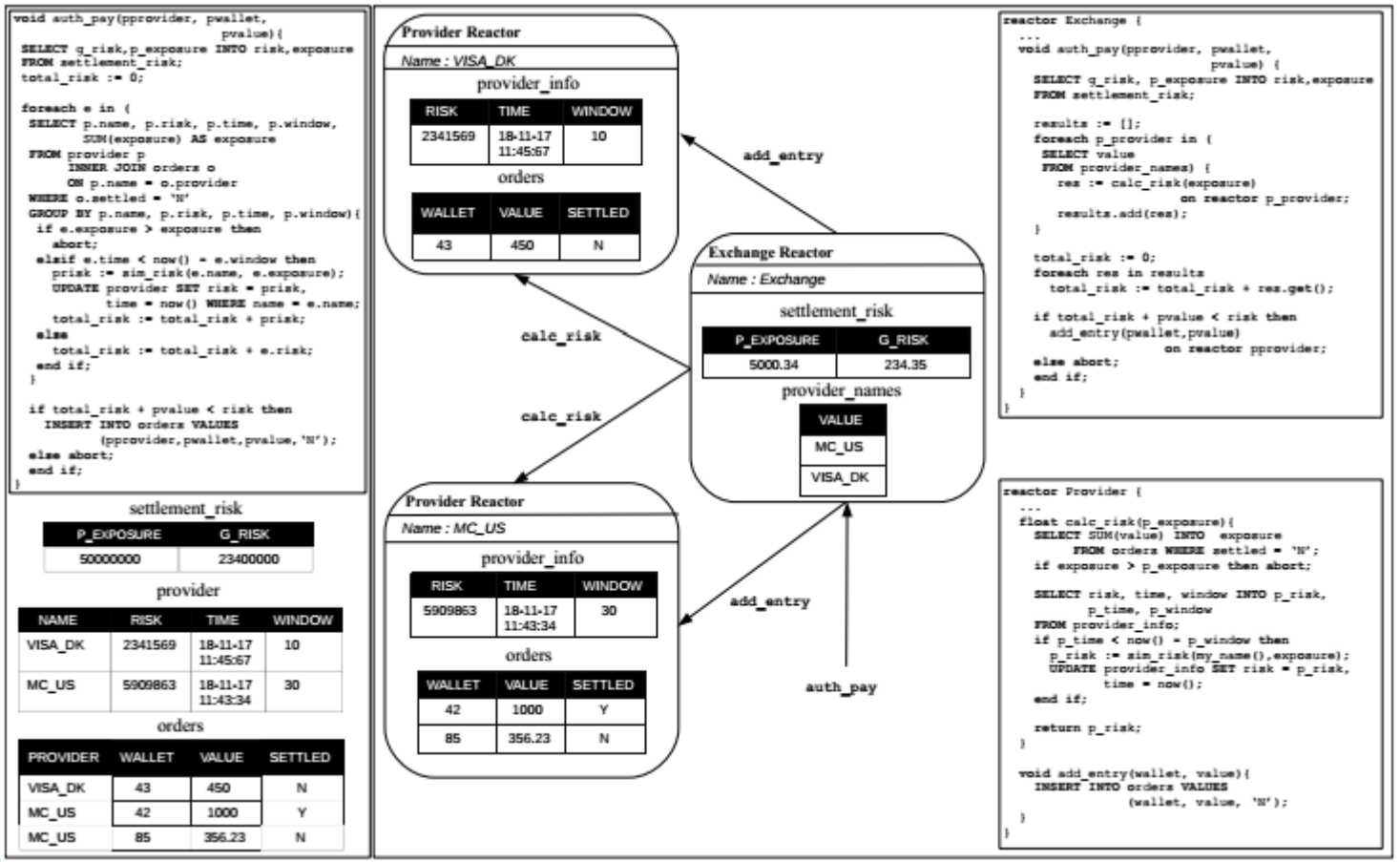


Figure 1: A simplified currency exchange application in: (a) the classic transactional model, and (b) the reactor model.

ReactDB

- In memory database system that exposes the reactor programming model
- Design aims at providing control over:
  - Mapping of reactors to physical computational resources
  - Memory regions under concurrency control
- Architecture is organized as a collection of *containers*
  - Abstracts a (portion of a) machine with its own storage (main memory)
  - Associated with computational resources (cores) disjoint from other containers, abstracted by transaction executors
  - A reactor is mapped to one and only one container

ReactDB

- Transaction Executor
  - Consists of a thread pool and a request queue
  - Responsible for executing requests, namely asynchronous procedure calls
  - Each one is pinned to a core
  - Each one maintains a thread pool to process (sub-transactions)

- Concurrency
  - Sub-transaction invoked in different container yields different management of transaction

System Architecture

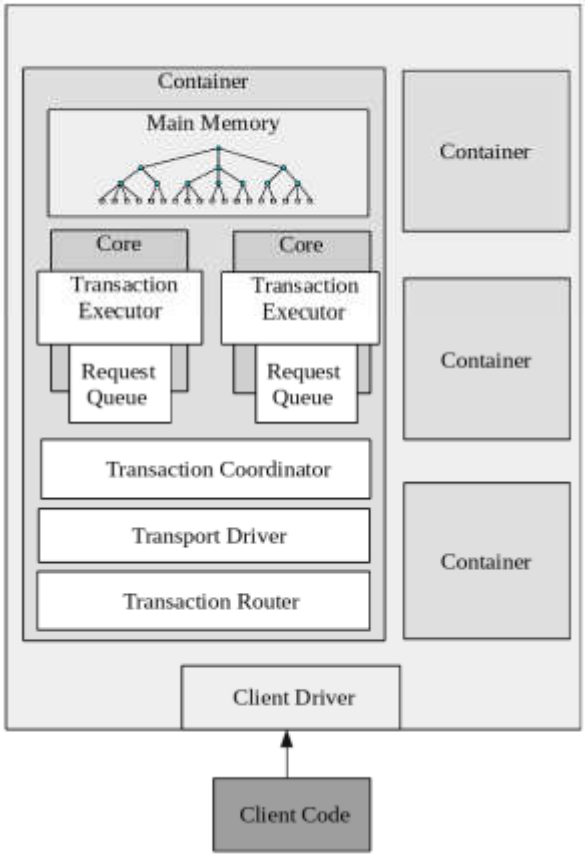


Figure 4: ReactDB's architecture.

IoT and Actor–Oriented Databases

- Huge amount of data is being generated with IoT
  - Challenges on volume, variety, dynamicity and ubiquity of IoT data
- Approach to model and build IoT data platforms
  - Based on the characteristics of an Actor–Oriented Database (AODB)
- Work aimed to illustrate the challenges and benefits provided by AODB to meet these requirements
  - Case study on Beef Cattle Tracking and Tracing

Non-Functional Requirements for IoT Data Platforms

- Data ingestion from endpoints
  - Capability to receive and store data from IoT devices
- Multi-tenancy
  - Must provide varied information services to different users
- Support for heterogeneous data
  - Allow for communication employing different data formats
- Cloud-based deployment
  - For ease of operation, management, and maintenance
- Scalable data platform
  - Must not degrade in functionality or performance while expanding

Non–Functional Requirements for IoT Data Platforms

- High efficiency
  - Must process massive amounts of concurrently generated data effectively
- Access control and data protection
  - Should support data protection, enforcing authentication and access control

Why Actor–Oriented Databases?

- Facilitate the management of distribution and the encapsulation of data
- Actor modularity supports representation and sharing of heterogeneous data
- Multiple actor types and concurrent execution among actors to achieve scalability
- Parallelism across actors allows for processing of massive amounts of concurrently generated data
- Encapsulation and modularity in AODBs support data protection and access control

Modeling AODBs

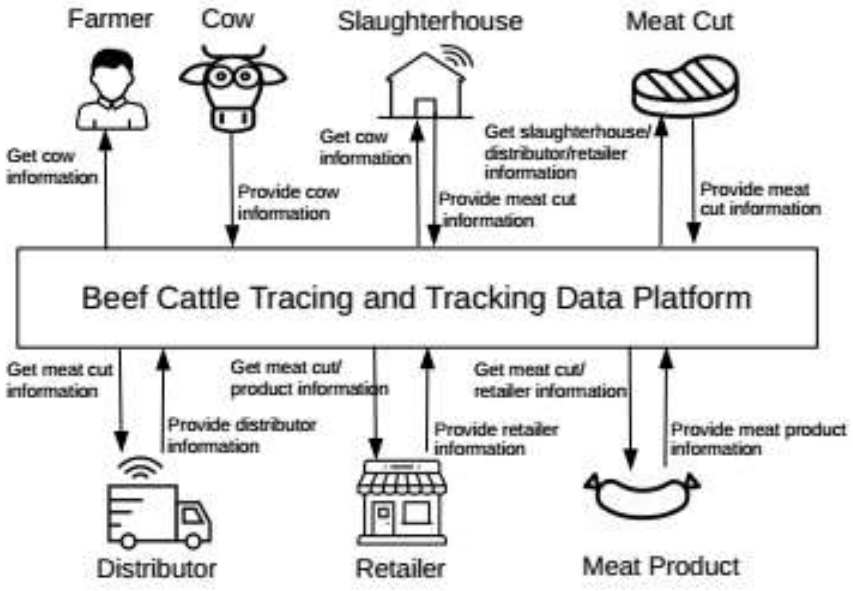
- How can actors be identified?
  - One actor is designed to carry out one specific real-world task with associated logic
- What should the Granularity of Actor State be?
  - An actor should represent the functionality of one active entity for which detailed tracking is required
- What about the absence of distributed transactions?
  - Communication is asynchronous, it is a challenge to keep consistency across actors in the presence of updates
  - Keep data related to a constraint in a single actor or design a multi-actor workflow for updates

Beef Cattle Tracking and Tracing

- Agricultural supply chains involve a complex network of producers, retailers, distributors, transporters, storage facilities, and consumers in the sale, delivery, and production of a particular product
- Trackability and traceability are essential requirements in food marketing
  - Tracking refers to following the path of an entity from the source to destination
  - Tracing refers to identifying original information regarding an entity and tracing it back in the system

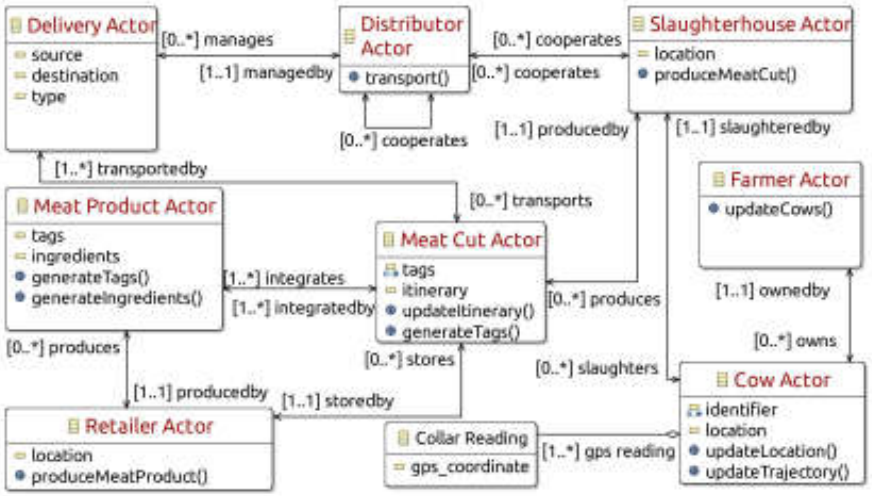
Beef Cattle Tracking and Tracing

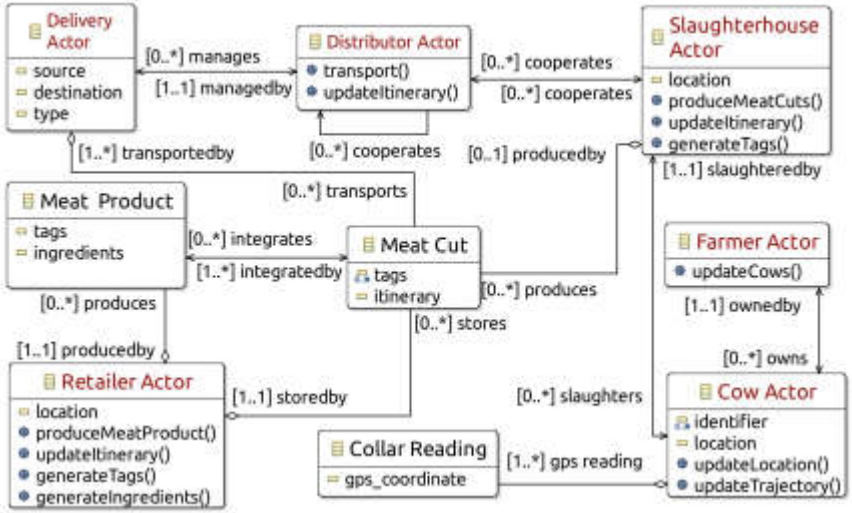
- Case study refers to a part of the beef cattle supply chain, concentrating on cow tracking and meat product tracing, providing tracking information and helping consumers trace meat products



Beef Cattle Tracking and Tracing

- Every actor encapsulates its state and communicates with other actors via asynchronous messages
- Accesses to data in the state of an actor are rendered as asynchronous communication events across actors





Case Study Implementation

- Choice of AODB
  - The vision for AODBs was proposed in the context of the Orleans project
  - Orleans provides an explicit storage model for actor state
  - Thus, a cloud storage system is employed by Orleans
  - Features such as indexing and ACID are being implemented
- Support for Non–Functional Requirements
  - Modularity, data encapsulation, and asynchronous communication were provided by virtual actors in Orleans
  - Support a number of data types and structures

Case Study Evaluation

- Development of a tool that simulates data requests from sensors
  - To generate variable load for the data platform
- Amazon DynamoDB [9] was used for Orleans grain state storage
  - 200 writes and 200 reads per second
  - Populated with synthetic data
- Amazon AWS to setup environment
  - m5.xlarge instances were employed for the Orleans silos, RDS db.t2.small for Orleans system storage

Case Study Evaluation

- Configuration are set to not benchmark DynamoDB storage, but rather the execution of in-memory actors
- Ingestion to the grain state storage has been configured to only happen when the Orleans silo service is shut down

Case Study Evaluation

- How many sensor readings can the SHMDP ingest using a single cloud server?
- Does the SHMDP scale simultaneously on the number of sensors and servers?

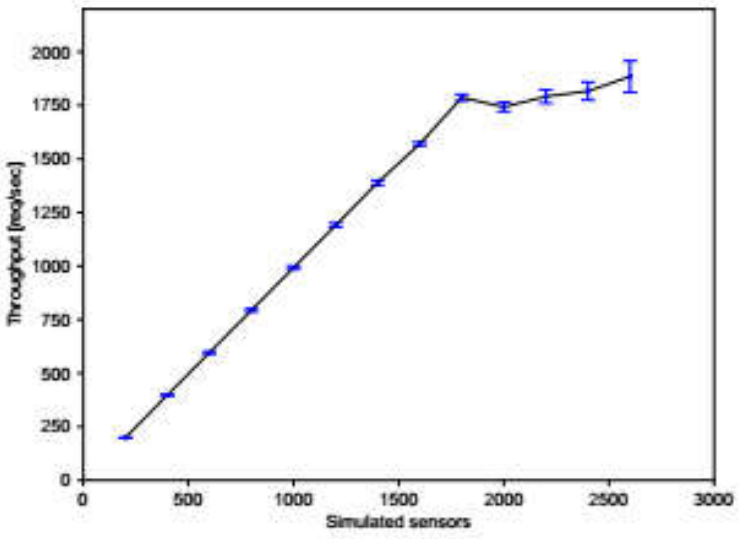


Figure 6: Single-server throughput experiment.

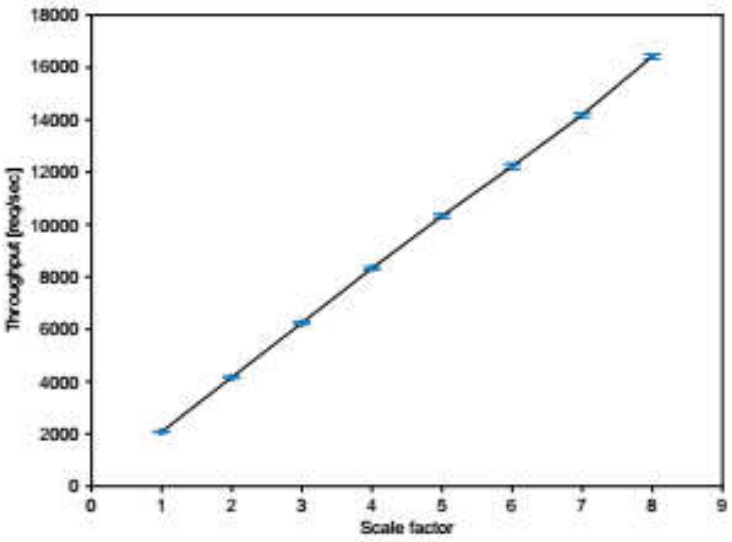


Figure 7: Scale out experiment over multiple servers.

Case Study Evaluation

- **Does the SHMDP deliver low latency on online query functions concurrently with data ingestion?**

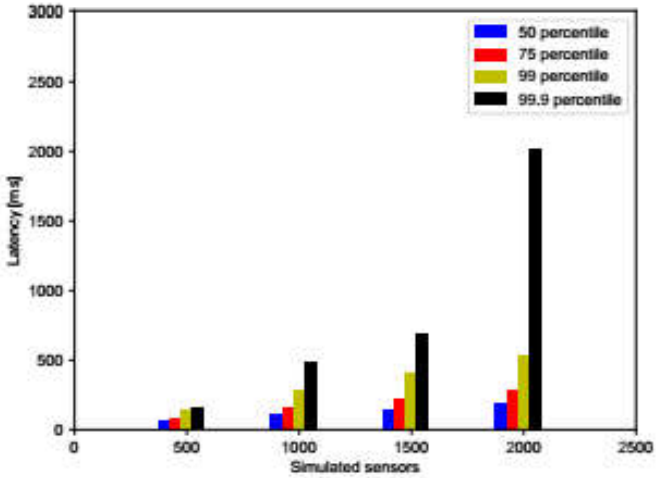


Figure 8: Latency percentiles for raw sensor channel data point time range requests.

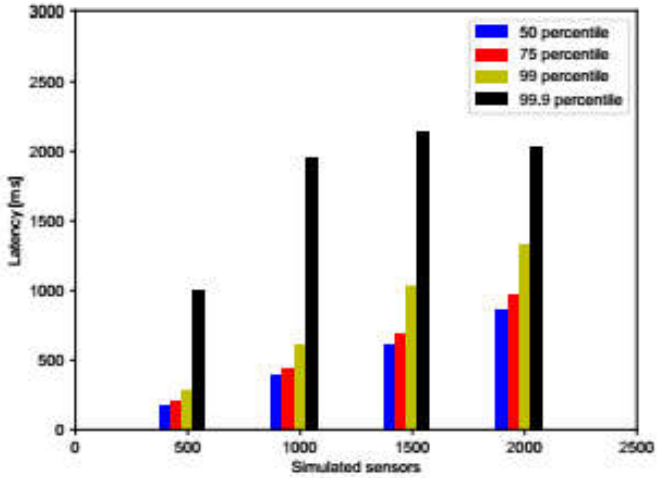


Figure 9: Latency percentile for organization live data requests.

References

- Modeling and Building IoT Data Platforms with Actor-Oriented Databases . Wang et al. EDBT. 2019.
- Reactors: A Case for Predictable, Virtualized Actor Database Systems. Shah, V. & Salles, M. SIGMOD. 2018.