# Distributed Order Dependency

Sebastian Schmidl, Juliane Waack[1]

**Abstract:** Abstract goes here.

**Keywords:** Actor Model; Akka; Distributed Computing; Parallelization

## 1 Introduction

Most web-based applications process data at ever growing rates. Whether user logins, the processing and storage of sensor data, or dynamic information provisioning, data is at the center of online applications. Cluster or cloud deployments and multi-core hardware architectures allow scaling application logic in terms of computational power. Data management systems, however, are at risk of becoming the bottleneck in data-centric software systems. Such applications create an increasing demand for scalable data storage solutions, that are capable of fulfilling real-time requirements for OLTP workloads at low latency.

Traditionally, system architectures for these types of software are comprised in two tiers: An application tier containing the bulk of the business logic, as well as a data tier, storing the application's data. The separation of concerns within this design in terms of concrete functionality depends on the choice of data storage solution. A relational database management system (RDBMS) enables data manipulation and the execution of most data-related computation in the data tier by using e.g. stored procedures and a complex, declarative query language. However, the monolithic design of conventional RDBMSs proves a hurdle for scalability and reduces modularity, which has a negative impact on code maintainability. More recently key-value storage solutions have gained popularity especially for highly scalable online applications. This is due to the fact that their less rigid schema guarantees allow for improved scalability. The trade-off of this solution is having to offload much of the data-handling logic into the application tier, thereby increasing load on the application as well as not keeping true to the separation of concerns.

In either case, the data and application tier do not necessarily share a format for data objects. Application logic and database exhibit differences in available data types and modelling

[1] Hasso-Plattner-Institut, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, {sebastian.schmidl, juliane.waack}@student.hpi.de

capabilities. Object-oriented Programming (OOP) languages provide concepts, such as inheritance, polymorphism, and nested objects. They enforce strict object encapsulation. RDBMSs are not able to express all of the same concepts due to the representation of data as relations in tables. While data storage solutions with a less rigid schema, such as document-oriented databases, are able to represent e.g. nested objects, they still face the same problems as RDBMSs with regard to data formats, inheritance, and encapsulation. The described set of difficulties when persisting application state and data objects in a database is commonly referred to as *object-relational impedance mismatch*. The use of object-relational mapping (ORM) tools, such as Hibernate for Java or Active Record for Ruby on Rails, is one approach to provide a middle tier for mapping between the data formats of the application and data tier.

Shah; Salles [SS17a] call for a new paradigm for designing a scalable data storage solution using the Actor model. The core primitive in this model are actors, which are objects comprised of state and behavior that execute computational tasks concurrently. Individual actors communicate with each other exclusively via asynchronous message passing. Incoming messages are stored in a mailbox allowing for the separate and independent processing of each message. When receiving a message an actor can modify its state, send a finite amount of messages to other actors, and create new actors. An actor's internal state is only available to said actor, which encourages a shared-nothing system architecture. The self-contained nature of actors and the fact that, through asynchronous message passing, actors provide a lock-free concurrency model, allows for naturally scaling out applications and systems [Ve15]. Shah; Salles propose an implementation of the data tier using the actor model as a logically distributed runtime. They predict that this programming abstraction will allow for a modular, scalable application design.

We build on this idea and present an application development framework for actor-based data-centric applications. In this framework actors employ relational structures internally to manage application state and data and provide functionality based on this data in the form of actor behavior. Since these actors are not part of a dedicated database runtime, but are defined and instantiated using the application framework, this approach dissipates the strict line between data and application tier. It allows for utilizing and interacting with data objects in the same representation throughout business logic and data storage.

With this framework we make the following contributions: We introduce domain actors (Dactors) to model application data in an Akka-based actor database system, similarly to the work of Shah; Salles on reactors [SS17b]. Dactors encapsulate application data and logic, therefore bridging the aforementioned impedance mismatch between data and the application logic tier. We implement a SQL-like data interface for defining internal Dactor logic, and provide Functors to allow users to define application logic relying on data contained within multiple Dactors. We also present experimental results on the memory overhead introduced by using Akka actors for data management in this manner to demonstrate the feasibility of this approach.

To our knowledge, we present the first implementation of this concept using the Akka framework. Comparable, existing work has been presented using the virtual actor concept provided by the Orleans framework for Microsoft .NET. Therefore, we discuss differences between the classical notion of actors and the virtual actor concept in **??**. In this section, we also give an outline of the history of actor model implementations as well as the characteristics of *Actor Database Systems* as proposed by Shah; Salles. In **??**, we outline our approach conceptually in more detail, before explicitly describing the implementation of our proof-of-concept framework, as well as some examples for usage patterns in **??**. We present setup and evaluation of experiments on the memory overhead introduced by the use of actors for data storage using our framework in **??**, and offer a concluding statement about this and future work in **??**.

## References

[SS17a] Shah, V.; Salles, M. V.: Actor Database Systems: A Manifesto. CoRR abs/1707.06507/, 2017.

[SS17b] Shah, V.; Salles, M. V.: Reactors: A Case for Predictable, Virtualized OLTP Actor Database Systems. CoRR abs/1701.05397/, 2017.

[Ve15] Vernon, V.: Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Pearson Education, 2015.