

# An Actor Database System for Akka

Frederic Schneider,<sup>1</sup> Sebastian Schmidl<sup>2</sup>

**Abstract:** Abstract goes here

**Keywords:** Actor Database System; Akka; Scala; Domain Driven Design

Paper Goals (for us):

- Concept / Approach: Domain design with actors (How does a Actor Database System look like in our view?)
- Memory overhead of actor-design? feasible
- Queries between actors and optimization through concurrency

## 1 Introduction

## 2 Related Work

### 2.1 Actor Model

The Actor model denotes a mathematical concept for concurrent computation which dates back to the early 1970's. The core primitive in this model are actors, which are objects comprised of state and behaviour 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. On 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].

---

<sup>1</sup> Hasso-Plattner-Institut, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, frederic.schneider@student.hpi.de

<sup>2</sup> Hasso-Plattner-Institut, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, sebastian.schmidl@student.hpi.de

Various programming languages and libraries implement the actor model and provide the corresponding functionality: The Erlang programming language supports concurrency using the actor programming model and is designed for distributed and highly available systems in the telecommunication sector [Ar07]. The Scala programming language's standard library included an actor model implementation for concurrent computation starting in 2006, which has now been deprecated for the Akka library [Ha12]. Akka implements the actor model for the Java Virtual Machine (JVM) and can be used with the languages Java and Scala [Li18a]. Akka.Net is a community-driven port of the Akka toolkit for the .NET platform in the languages C# and F# [Ak17].

## 2.2 Virtual Actor Model

While the actor programming model allows for low-level interaction with the actor lifecycle management, handling race conditions, physically distributing actors, and failure handling, the virtual actor programming model raises the abstraction level of actor objects. It still adheres to the plain actor model, but treats actors as virtual entities. In this model actor lifecycle management and distribution is managed by the framework or runtime and not by the application developer. The physical location of the actor is transparent to the application. This model is comparable to virtual memory, which allows processes to access a memory address whether it is currently physically available or not.

The virtual actor model was introduced with the Orleans framework for .NET by Microsoft Research [Be14]. In Orleans, actors (virtually) exist at any time. There is no possibility to create an actor explicitly. Instead, this task is handled by the language runtime, which instantiates actors on demand. The runtime's resource management deals with unused actors and reclaims their space automatically. The Orleans runtime guarantees that actors are always available. This means, that in the view of an application developer an actor can never fail. The runtime will deal with crashing actors and servers and recreate missing actor incarnations accordingly.

The corresponding framework for the JVM is Orbit [El18]. In Orbit, a virtual actor can be active or inactive. But those two states are transparent to the application developer. Similar to Orleans, messages to an inactive actor in Orbit will activate it, so the actor can receive the sent message. Actor state in Orbit is usually stored in a database, which allows for activation and deactivation of actors at runtime. During activation of an actor, the state is recovered from persistent database storage.. Before the actor is deactivated, its state is written back to the database.

Akka provides similar functionality with Akka Cluster Sharding [Li18b]. It introduces the separation of virtual and physical Akka actors. Actors are first grouped into shards, which can then be located at different physical locations. The distribution of the shards and their actors is managed by the framework and it also deals with the routing of the messages and re-balancing the shards to different nodes based on different strategies. A developer must

not know, where a specific actor is physically located to send messages to it. Akka Cluster Sharding also supports passivation of actors. This means that unused persistent actors can be unloaded to free up memory (passivation). A message to passive actors will instruct the framework to load it back into memory.

In this paper we denote virtual actors as *vactors* to clearly distinguish actors of the actor model (actor) from those of the virtual actor model (*vactor*).

## 2.3 Actor Database Systems

The actor and virtual actor programming model is increasingly used to build soft caching layers and cloud applications for various purposes [Er18; Li18c; NE18]. This shows the appeal of the programming model that allows developers to easily develop modular and scalable software, which can be deployed in the heterogenous parallel cloud computing infrastructure.

Despite its popularity for application development, the actor programming model and actor programming frameworks lack state management capabilities, specifically for data-centric applications. The developer has to decide how to handle state persistence and how to satisfy the failure and consistency requirements of the application, because actor model implementations do not provide atomicity or consistency guarantees for state across actors. Shah; Salles [SS17a] find a need for state management guarantees in actor systems and propose to integrate actor-based programming models in database management systems. The authors postulate that *Actor Database Systems* should be designed as a logical distributed runtime with state management guarantees. They should allow for and encourage the design of modular, scalable and cloud-ready applications. Shah; Salles outline four tenets that identify an *Actor Database System* and specify needed features in their manifesto [SS17a]:

**Tenet 1** Modularity and encapsulation by a logical actor construct

**Tenet 2** Asynchronous, nested function shipping

**Tenet 3** Transaction and declarative querying functionality

**Tenet 4** Security, monitoring, administration and auditability

Practical work on actor database systems has been done using the Orleans framework. Shah; Salles tightly follow the propositions of the manifesto and present `REACTDB`, an in-memory OLTP database system that is programmed via application-defined actors with relational semantics, called *reactors* [SS17b]. A reactor is a logical entity encapsulating state as relations, with asynchronous function processing capability, which typically represents an application-level scaling unit. Within individual reactors classic declarative querying can be used, but across actors explicit asynchronous function calls must be defined. Shah;

Salles show that transactions across reactors can have serializability guarantees. Eldeeb; Bernstein [EB16] introduce a new transaction protocol for distributed actors improving the throughput compared to the two-phase commit protocol. Bernstein et al. [Be17] explore indexing mechanisms in an actor database system. They propose a new indexing architecture based on the Orleans framework that is fault-tolerant and eventually-consistent.

Biokoda takes a different approach to combine actor programming models and relational database systems. In their development actorDB, an actor encapsulates a full relational SQL database using the SQLite database engine<sup>3</sup>. Actors can be deployed in a distributed cloud environment and communicate asynchronously. The database is ACID-conform and enforces consistency across actors with the Raft consensus protocol [OO14]. The database can be used via the MySQL-protocol

Most practical research in the field of actor database systems has been presented in conjunction with the Orleans framework and the Erlang programming language. The open-source project *Actorbase* by Cardin constitutes a key-value store implementation utilizing Akka actors for horizontal scaling and partitioning [Ca17]. In this research database system, partitions are managed by storekeeper actors while inter-partition lookup and indexing is managed by so-called storefinder actors. This approach is limited to a key-value store function set and does not provide functionality corresponding to the relational features discussed earlier.

We base our work on the concept of actor database systems as introduced by Shah; Salles [SS17a]. Similar as in [SS17b], actors in our approach consist of state abstracted as relations and asynchronous function processing logic. The application developer defines actors and the asynchronous functions. Different to previous approaches, we use Scala as programming language and Akka as the reference actor model to implement a framework for actor database systems, which is explained in section 4.

---

<sup>3</sup> <https://www.sqlite.org/about.html>

### **3 Domain Actor Database Concept**

#### **3.1 Domain Actors**

#### **3.2 Actor Database Systems**

### **4 Domain Actor Database Framework**

#### **4.1 Actor Design**

#### **4.2 Memory Storage**

#### **4.3 Multi-Actor Queries**

#### **4.4 Discussion**

### **5 Experiments**

#### **5.1 Actor Memory Overhead**

#### **5.2 Query Performance**

#### **5.3 Query Optimization**

#### **5.4 Discussion**

### **6 Conclusion**

### **7 Example**

This example is from the original LNI documentation and therefore in German:

Referenzen mit dem richtigen Namen (*Table*, *Figure*, ...): Sect. 1 zeigt Demonstrationen der Verbesserung von GitHub-LNI gegenüber der originalen Vorlage. Sect. 7 zeigt die Einhaltung der Richtlinien durch einfachen Text.

Referenzen sollten nicht direkt als Subjekt eingebunden werden, sondern immer nur durch Autorenangaben: Beispiel: Abel; Bibel [AB00] geben ein Beispiel, aber auch Azubi et al. [Az]. Hinweis: Großes C bei Citet, wenn es am Satzanfang steht. Analog zu Cref.

Hier sollte die Graphik mittels `includegraphics` eingebunden werden.

Fig. 1: Demographik

Überschriftsebenen	Beispiel	Schriftgröße und -art
Titel (linksbündig)	Der Titel ...	14 pt, Fett
Überschrift 1	1 Einleitung	12 pt, Fett
Überschrift 2	2.1 Titel	10 pt, Fett

Tab. 1: Die Überschriftsarten

Fig. 1 zeigt eine Abbildung.

Tab. 1 zeigt eine Tabelle.

Die LNI-Formatvorlage verlangt die Einrückung von Listings vom linken Rand. List. 1 zeigt uns ein Beispiel, das mit Hilfe der `lstlisting`-Umgebung realisiert ist. Referenz auf `print("Hello World!")` in Line 6.

```

/**
 * Hello World application!
 */
object Hello {
  def main(args: Seq[String]): Unit = {
    println("Hello World!")
  }
  def math: Unit =  $\frac{\sqrt{1-x^2}}{x_t+\lambda} - \infty$ 
}

```

List. 1: Beschreibung

Die korrekte Einrückung und Nummerierung für Formeln ist bei den Umgebungen `equation` und `align` gewährleistet.

$$1 = 4 - 3 + x \tag{1}$$

und

$$2 = 7 - 5 \tag{2}$$

$$3 = 2 - 1 \tag{3}$$

## References

- [AB00] Abel, K.; Bibel, U.: Formatierungsrichtlinien für Tagungsbände. Format-Verlag, Bonn, 2000.

- [Ak17] Akka.NET project: Akka.NET, 2017, URL: <https://getakka.net/>, visited on: 08/15/2018.
- [Ar07] Armstrong, J.: A History of Erlang. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. HOPL III, ACM, San Diego, California, pp. 6-1–6-26, 2007, ISBN: 978-1-59593-766-7, URL: <http://doi.acm.org/10.1145/1238844.1238850>.
- [Az] Azubi, L. et al.: Die Fußnote in LNI-Bänden. In. Pp. 135–162.
- [Be14] Bernstein, P. A.; Bykov, S.; Geller, A.; Kliot, G.; Thelin, J.: Orleans: Distributed virtual actors for programmability and scalability. MSR-TR-2014-41/, 2014.
- [Be17] Bernstein, P. A.; Dashti, M.; Kiefer, T.; Maier, D.: Indexing in an Actor-Oriented Database. In: CIDR. 2017.
- [Bi18] Biokoda: ActorDB – 1. About, 2018, URL: <http://www.actordb.com/docs-about.html>, visited on: 08/16/2018.
- [Ca17] Cardin, R.: Actorbase, or "the Persistence Chaos", 2017, URL: <https://dzone.com/articles/actorbase-or-quotthe-persistence-chaosquot>, visited on: 08/18/2018.
- [EB16] Eldeeb, T.; Bernstein, P. A.: Transactions for Distributed Actors in the Cloud. In. 2016.
- [El18] Electronic Arts Inc.: What is Orbit?, 2018, URL: <https://github.com/orbit/orbit/wiki>, visited on: 08/15/2018.
- [Er18] Erlang Solutions Ltd: Case Studies & Insights, 2018, URL: <https://www.erlang-solutions.com/resources/case-studies.html>, visited on: 08/17/2018.
- [Ha12] Haller, P.: On the Integration of the Actor Model in Mainstream Technologies: The Scala Perspective. In: Proceedings of the 2Nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions. AGERE! 2012, ACM, Tucson, Arizona, USA, pp. 1–6, 2012, ISBN: 978-1-4503-1630-9, URL: <http://doi.acm.org/10.1145/2414639.2414641>.
- [Li18a] Lightbend, Inc.: Akka, 2018, URL: <https://akka.io/>, visited on: 08/15/2018.
- [Li18b] Lightbend, Inc.: Akka Documentation – Cluster Sharding, 2018, URL: <https://doc.akka.io/docs/akka/current/cluster-sharding.html?language=scala#cluster-sharding>, visited on: 08/16/2018.
- [Li18c] Lightbend, Inc.: Lightbend Case Studies, 2018, URL: <https://www.lightbend.com/case-studies#filter:akka>, visited on: 08/17/2018.
- [NE18] .NET Foundation: Who Is Using Orleans?, 2018, URL: <https://dotnet.github.io/orleans/Community/Who-Is-Using-Orleans.html>, visited on: 08/17/2018.
- [OO14] Ongaro, D.; Ousterhout, J. K.: In search of an understandable consensus algorithm. In: USENIX Annual Technical Conference. Pp. 305–319, 2014.

- [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.