# An Actor Database System for Akka

Sebastian Schmidl, Frederic Schneider, Thorsten Papenbrock[1]

**Abstract:** System architectures for data-centric applications are commonly comprised of two tiers: An application tier and a data tier. The fact that these tiers do not typically share a common format for data is referred to as *object-relational impedance mismatch*. To mitigate this, we develop an actor database system that enables the implementation of application logic into the data storage runtime. The actor model also allows for easy distribution of both data and computation across multiple nodes in a cluster. More specifically, we propose the concept of *domain actors* that provide a type-safe, SQL-like interface to develop the actors of our database system and the concept of *Functors* to build queries retrieving data contained in multiple actor instances. Our experiments demonstrate the feasibility of encapsulating data into domain actors by evaluating their memory overhead and performance. We also discuss how our proposed actor database system framework solves some of the challenges that arise from the design of distributed databases such as data partitioning, failure handling, and concurrent query processing.

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

## 1 Introduction

Today's applications need to process data at ever growing rates. Regardless of its origin and kind, data is ever growing and needs to be stored and queried. Cluster or cloud deployments and multi-core hardware architectures allow scaling application logic in terms of computational power. Traditional data management systems, however, are at risk of becoming the bottleneck in data-centric software systems, because the separation into data and application tier costs performance, impacts code maintainability, and increases error susceptibility.

The performance costs are due to the fact that relational database management system (RDBMS) model their data in terms of relations while applications usually model the data as objects. The translation of relations into objects and vise versa is known as the *object-relational impedance mismatch* and requires some additional effort. The use of object-relational mapping (ORM) tools, such as Hibernate for Java or Active Record for Ruby on Rails, is a convenient yet expensive approach to provide a middle tier for the translation. Some key-value stores solve the impedance mismatch more elegantly, but they suffer from worse join and aggregation costs. Furthermore, code maintainability decreases

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

when stored procedures are being used to push application logic closer to the data, i.e., into the data tier for performance reasons, and the error susceptibility increases, because large, monolithic RDBMSs suffer from hand-crafted, non-standardized, and inconsistent attempts to fault-tolerance, parallelization, data encapsulation, workload distribution, and replication. The actor programming model, on the contrary, offers an effective solution for all these challenges.

Using the actor programming model to fuse application and data tier is a concept originally proposed by Shah; Salles [SS17a] and tested on the Orleans actor framework. The authors call for a new paradigm by 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. 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 actors provide a lock-free concurrency model, allows for naturally scaling out applications and systems [Ve15].

We build on this idea and present an application development framework for actor-based data-centric applications. In contrast to Shah; Salles [SS17a], our actor database system targets the Akka actor framework that offers different mechanisms for fault tolerance and actor lifecycles than Orleans. In detail, 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. Since these actors are not part of a dedicated database runtime, but are defined using the application framework, data objects share the same representation throughout business logic and data storage. This approach bridges the aforementioned impedance mismatch between data and application logic tier. In contrast to reactors, Dactors are not relational entities, but employ relational structures internally. Dactor state can be manipulated via an SQL-like interface. To define application logic relying on data contained within multiple Dactors, we provide the concept of Functors, which make the usage of asynchronous and concurrent computations explicit.

To our knowledge, we present the first implementation of this concept using the Akka framework. Comparable approaches are discussed in Sect. 2. In Sect. 3, we outline our concept for an actor database system in more detail, before presenting the results of our experimental evaluation using our framework in Sect. 4. We offer a concluding statement about this and future work in Sect. 5.

## 2 Related Work

The actor model that we introduced in the previous section has been implemented as libraries for various programming languages. The most popular implementations are Erlang's in-build actors, the Orleans framework for .NET, and the Akka framework for Java [Ar07; Be14; Li18a]. Although most research in the area of actor-based database systems targets the Orleans framework, Akka is probably the most widely used actor model implementation – not least because of the popularity of Java and the fact that it is used in frameworks such as Apache Spark and Apache Flink. For this reason and because Akka differs in various aspects from Orleans, we focus on this framework in our research.

Despite their popularity for building distributed applications, all current actor programming frameworks lack database-like state management capabilities, specifically for data-centric applications. The developer has to decide how to handle state persistence and how to satisfy failure, replication, and consistency requirements of an application – the actor model implementations neither provide atomicity nor consistency guarantees for state across actors. Shah; Salles [SS17a] therefore stated the need for state management in actor systems and proposed to integrate database functionality into the actor model. The authors postulate that *Actor Database Systems* should be designed as a logical distributed runtime with own state management guarantees. More specifically, their manifesto specifies four tenets that define an Actor Database System [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 |

Our actor database system (currently) covers the first three of these four tenets: For **tenet 1**, we use actors to achieve a modular logical model for data encapsulation. Dactor instances are in-memory storekeepers for application data. They satisfy the actor definition and support high modularity. For **tenet 2**, Dactors provide a model for the concurrent computation of predefined functionality that enforces locality of data accesses. All communication between Dactors is asynchronous to leverage the advantages of increasingly parallel hardware. Our concept of Functors allows for the definition of functionality using multiple actors' data. To meet **tenet 3**, Functors and Dactor behavior can be defined in a declarative way. Due to their single-threaded computation model, Dactors basically enforce internal consistency by default. In principle, Functors also enable the implementation of further transaction protocols to ensure inter-Dactor consistency guarantees.

In contrast to the actor database system prototype introduced in [SS17a], we developed our prototype using the Scala programming language and the Akka framework (instead of .NET and Orleans). In contrast to Orleans and its convenient *virtual actors*, Akka offers more control over an actor's lifecycle, has a more explicit failure handling, and models actors in hierarchies – aspects that enable more fine-grained control over the system but also demand for more thorough architectural system design decisions.

Most related research in the field of actor database systems has been presented in conjunction with the Orleans framework and the Erlang programming language [Be17; EB16; SS17b].

Biokoda [Bi18] takes another approach and encapsulates a full relational SQL database inside an actor. Cardin [Ca17] uses actors to build a scalable key-value store and others use the actor model to build soft caching layers and cloud applications for various purposes [Er18; Li18b; NE18].

## 3 Domain Actor Database Framework

Our actor databases system consists of two building blocks: *domain actors* and *Functors*. These two concepts allow for the definition of application data within the application itself. Since both are based on actor model principles, they make the database system modular, cloud-ready, and scalable. In this section, we introduce both domain actors and Functors.

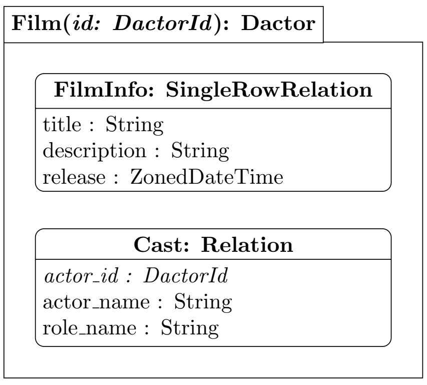
### 3.1 Domain Actors – Encapsulation of Data

Similar to Shah; Salles [SS17b], we introduce a special type of actor, called Dactor, that acts as an application-defined scaling unit. Dactors can be used to model application-domain objects and encapsulate the object's state and application logic in an actor. Using actors for this enforces technical encapsulation of state access due to the purely private state in actors and the need of explicit asynchronous messaging between the actors. The encapsulation also makes it easier to reason about state changes, bugs, and other failures, as only code within the Dactor can change the corresponding state.

In-memory data contained within a Dactor instance is managed in a data structure called *relation*. One Dactor can contain multiple relations. A relation is, similarly to a table in the relational database model, defined as a multiset of tuples following a predefined schema. Relations provide an SQL-like interface to query and manipulate the contained data, so known and proven syntax and semantics can be used to define Dactor-behavior. Relations form a typed, Dactor-internal data model. Using Dactors to implement a database leads to a modeling approach that is different to Entity-relationship modeling. The following example discusses the conceptual differences between the two in more detail.

We consider the example of a web application with information on movies similar to the imdb.com or rottentomatoes.com websites. A standard query for those websites is to display a film with its description and cast. A traditional data layout might be comprised of two entities: *Film* containing the film's ID, title, description and release date and *Actor* containing the actor's ID and name. Those two entities might be in a N-to-M relation (*Cast*) with an attribute showing the actor's role in the film. In contrast to the relational model, our model, shown in Fig. 1a, consists of one Dactor type and two relations and is denormalized. The information contained in the *Actor* entity is distributed across the `Cast` relations. This allows us to answer the standard queries from one single actor instance without needing to join the answers from different, possibly physically distributed Dactor instances.

This approach to layout an application's data results in much smaller data sizes per Dactor compared to typical database tables and enables many business-logic-driven approaches to scaling, data partitioning, and caching. The trade-off, however, is a large number of Dactor instances and a (partially) denormalized schema.



(a) Graphical representation of the `Film` Dactor type definition.

```
class Film(id: DactorId) extends Dactor(id) {
  override protected val relations = {
    Film.Info -> SingleRowRelation(Film.Info),
    Film.Cast -> RowRelation(Film.Cast)
  }
  override def receive: Receive =  //Dactor behavior
}
object Film {
  object FilmInfo extends RelationDef {
    val title = ColumnDef[String]("title")
    val description = ColumnDef[String]("description")
    val release = ColumnDef[ZonedDateTime]("release")
  }
  object Cast extends RelationDef {
    val actorId = ColumnDef[DactorId]("actor_id")
    val name = ColumnDef[String]("actor_name")
    val rolename = ColumnDef[String]("role_name")
  }
}
```

(b) Example code using our framework.

Fig. 1: `Film` Dactor type definition with two relations from the example application.

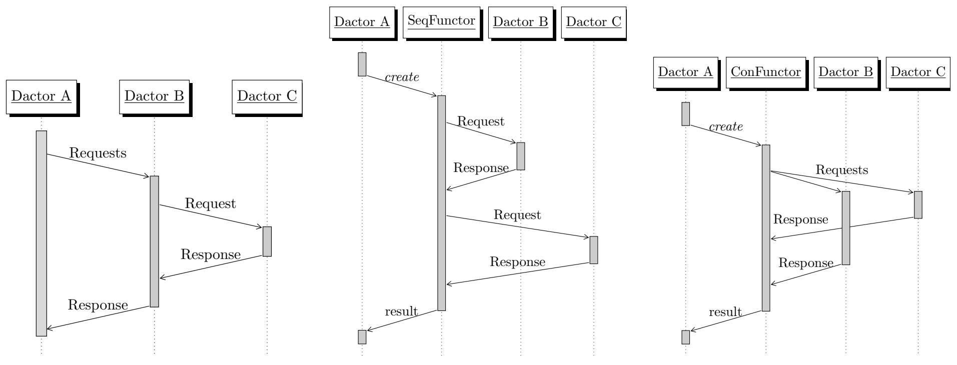
As Dactors not only contain data, but also the corresponding domain logic, computation is executed concurrently. Actors provide single-threaded semantics, which makes enforcing constraints on data stored inside one Dactor easy. While state querying and modification within Dactors is possible in a declarative way, the application developer can explicitly define the communication across all kinds of actors via asynchronous messages. The explicit messaging differentiates Dactors from Shah; Salles [SS17b]'s reactors, as reactors can be used as relational entities and hide the message passing from the developer.

To illustrate the definition of a `Dactor` in code, we show the definition of `Film`'s data model in Fig. 1b. Developers can model the application's domain objects by defining Dactor types as subclasses of the framework-provided `Dactor` class in a declarative way. Instances of such user-defined Dactor types are managed by the framework and are available for messaging in a consistent namespace. Using the column's predefined data types, all functions support compile-time type-safety. Due to the Dactor system sharing the application's runtime and programming environment, these data or object types are equal to the types handled in any application logic. Thus, this approach helps eliminate the impedance mismatch between application logic and data tier with regard to handled data types and object (de-)serialization.

## 3.2 Functors – Encapsulation of Queries

Dactors can answer queries via explicit, asynchronous messaging, i.e., they answer a query with their local data. Sometimes, however, queries need be answered by several actors. In such cases, it makes sense to encapsulate the processing in a new, short-living actor that we call Functor. Functors are the framework's concepts that enable inter-Dactor communication and computations. They communicate with (usually multiple) Dactors, track the completion of a query, handle the state of pending requests, and resolve failure cases. Every actor can create a new Functor to encapsulate multiple requests to Dactors. The Functor handles the message processing and sends the final result or a failure message back to its creator. Since all Akka actors live in hierarchical parent-child relationships, Functors are always created by an actor as a child. This Akka-specific hierarchical relationship enables notifying the calling actor even in case of unforeseen crashes of the Functor themselves, which in turn allows to trigger error handling, e.g. retrying the Functor execution.

In our actor database system, we consider three messaging patterns for inter-Dactor communication, which are shown in Fig. 2. These patterns are provided as messaging primitives by the framework and can be combined to create more complex message flows and computational models:



(a) Cascading Computation (b) Sequential Computation (c) Concurrent Computation

Fig. 2: Inter-Dactor communication patterns. Gray bars indicate that an actor holds state that is related to the showed message flow.

**Cascading Computation** is a pattern where a high-level message to an initial Dactor (Dactor A) triggers successive messages to other Dactors, which are hidden from the original requester. Following Dactors can also trigger further messages themselves. As one can see in Fig. 2a, this pattern is comparable to function calls in Object-oriented Programming. But contrary to simple function calls, messages in this pattern are sent asynchronously. This means that the requesting Dactor has to manage the state of pending responses. This clutters the domain logic in the Dactor and leads to complex and error-prone code. If used sparsely, this pattern supports separation of concerns and the tell-don't-ask paradigm.

**Sequential Computation** is used for queries that consist of consecutive steps, where each step depends on the previous step's result, such as filter chains. This pattern can be implemented via the aforementioned Functors. Using a Functor to process the consecutive steps of the computational chain relieves Dactor A from dealing with intermediate state, because it is managed by the Functor. Each Functor deals with only one request-response pair at a time, which leads to simple state and processing logic for the Functor itself.

**Concurrent Computation** is a another messaging pattern based on Functors to encapsulate the processing of multiple request-response pairs. The concurrent Functor sends messages to several Dactors in parallel and collects the results when they are finished to forward them to its creator. It allows for highly parallelized computations as all involved Dactors are messaged at the same time and calculate their responses concurrently.

In summary, explicit message handling in Dactors is used to implement the cascading communication pattern; sequential and concurrent Functors are the framework's concepts to enable inter-Dactor communication and computations. Returning to the web application example, we now want to add a new film to the database using the Functor concept. This involves changes to a new `Film` and the corresponding `Studio` Dactor instances. We can combine the concurrent and sequential Functors to implement this functionality, which is displayed in Fig. 3. Both sequential Functors are comprised of two subsequent steps: They retrieve information from one Dactor to update the other one. They are independent of each other, so they can be executed in parallel, which is done by using the concurrent Functor. It only sends a successful response to its caller after both sequential Functors have sent their responses to the concurrent Functor.

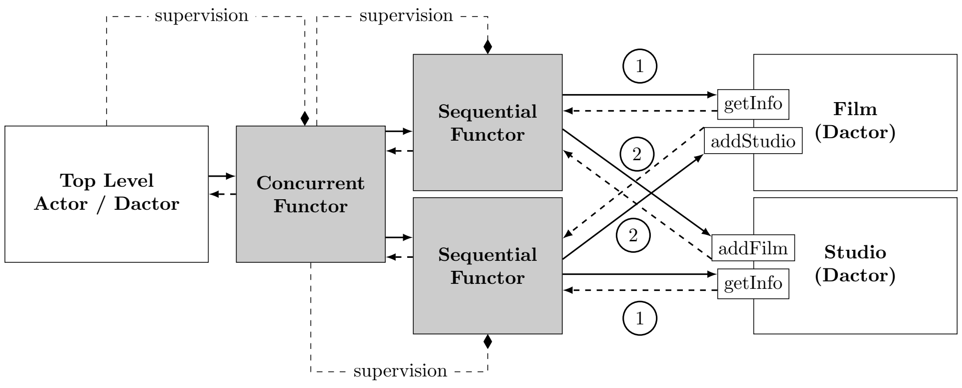


Fig. 3: Component diagram indicating the message flow through Functor objects and their supervision by the calling actor. Arrow and dashed arrow pairs indicate corresponding request and response messages. The outgoing requests of each sequential Functor are numbered to indicate their order.

### 3.3 System Details

*Data partitioning* in an actor database system differs fundamentally from common partitioning techniques used in relational databases. While large tables are typically partitioned based on a specific column's value or the hash thereof, our framework provides Dactors as entities for data encapsulation and partitioning. Dactors can be provisioned across multiple virtual runtimes and physical machines, because every Dactor instance is independent of the others and the only mean of communication is message passing. As such, they provide flexible, fine-grained data partitioning based on application needs.

The distributed nature of the database system introduces the new problem of partition or *actor discovery*. The framework maintains a unified namespace, in which each Dactor instance is identified by its Dactor type and a unique ID. In fact, querying a specific Dactor just requires obtaining the messaging address from the name-service and sending a message to it. In case of a multi-node deployment, this is complemented by Akka's Cluster Sharding component, which routes the messages to the right physical host.

Finally, *failure handling* requires careful monitoring due to Dactor distribution, especially with regard to computations relying on multiple Dactors' data. Building on Akka's parent-child supervision concept, our framework allows for transparent failure handling configurations. Failures can be handled within Dactors if appropriate. In case of multi-Dactor queries a fail-fast approach is chosen to allow calling actors to react to exceptions in a timely manner.

## 4 Dactor Memory Overhead Experiments

In our framework, a complex query comprising multiple Functors has a runtime of 25 ms while queries involving a single Dactor instance respond under 10 ms. Our concept is based on the usage of hundreds of thousands of Dactors, which all store only a small amount of data. Therefore, the question of how much memory overhead is introduced for storing data split across a multitude of actors is more interesting than the computation time.

We performed experiments using an exemplary actor database system, which is modeled based on a real-world scenario. It consists of four different Dactor types, each containing one to three relations. The data stored in one Dactor ranges from seven to about 700 kilobytes depending on its type. This small data size makes the relative memory overhead more visible as we need thousands of Dactors and relations to load our datasets into memory. We used a script to generate four different datasets emulating the scaling of the system by increasing the number of Dactor-instances and keeping the data size stored in one Dactor nearly constant. The script creates various primitive data types, such as `String`, `Double`, and `Int`, as well as complex data types, such as `ZonedDateTime`, and distributes the data across Dactors and relations. The dataset sizes are reported in Tab. 1.

For each dataset we performed three different tests:

***Single string*** Convert all data into its `String` representation and load it as a single big `String` into memory. This test serves as baseline for the other tests.

***Relations*** Load the data into their respective `Relations`, preserving the type information and using the in-memory data storage objects from our framework.

***Framework*** Use the full-fledged framework to load the data into memory. This approach stores the data distributed across Dactors in `Relation` objects.

To obtain the used memory of the objects in our test approaches, we used VisualVM[2] to create heap dumps. After the data was completely loaded into memory, we triggered a garbage collection run and created a heap dump. VisualVM is able to compute the retained sizes of object hierarchies in those dumps. This allowed us to investigate the memory usage of selected objects and their members in detail. We report the results of those experiments in Tab. 1.

| **Dataset** | **Disk size** | **# Dactors** | **Heap size** | | | **Overhead /** |
|---|---|---|---|---|---|---|
| | | | ***Single string*** | ***Relations*** | ***Framework*** | **Dactor** |
| $D_1$ | 10 MB | 829 | 11 MB | 28 MB | 29 MB | 526 B |
| $D_2$ | 25 MB | 2 578 | 18 MB | 43 MB | 44 MB | 539 B |
| $D_3$ | 50 MB | 4 373 | 47 MB | 116 MB | 119 MB | 532 B |
| $D_4$ | 100 MB | 8 618 | 101 MB | 233 MB | 237 MB | 534 B |

Tab. 1: Used heap size of our three different methods to load data into memory and the memory overhead of Dactors compared across the four datasets.

If the data is loaded into memory as a single big `String` object, it takes up around the same amount of heap as the dataset is big. Storing the data in *Relations* introduces a overhead of about 150%. Even for the smallest dataset the data is split up across thousands of relations, which each use a two dimensional `Array` to store the individual values in a non-optimized way. In addition to that, relations also store metadata about the contained data, such as column names and data types. As we can see in Tab. 1, using the full framework with Dactors does not introduce much additional memory overhead. On average, using a Dactor only needs an additional 533 B more.

Let us assume that we have a 1 TB database and we chose to store 1 MB per Dactor. This requires the actor database to instantiate about one million Dactors. Using the average overhead of 550 B per Dactor, this would yield a relative memory overhead of only 0.05 %. Doing the same thought experiment with storing 10 MB per Dactor, results in about 100 000 Dactors and reduces the relative memory overhead to 0.005%.

## 5 Conclusion

In our research, we study the question how database features can be incorporated into the actor programming model. This work presents a proof-of-concept implementation of an actor database framework, which enables developers to declaratively define a data model using Dactors. Dactors model application-domain objects by encapsulating both the object's state

[2] `https://visualvm.github.io/`

and its application logic. The framework provides a shared, distributed runtime for database functionality and application logic, mitigating the *object-relational impedance mismatch* between data and business logic tier. The introduced Functor concept, which are temporary actors that manage multi-Dactor queries, provides a transparent computation model and failure handling capabilities. First experiments with our Akka-based actor database system show that the memory overhead introduced by using actors for data management is low. Hence, the approach is feasible and pays off especially if large amounts of data need to be stored for highly concurrent data manipulation workloads. As future work, we aim to develop inter-Dactor consistency guarantees by extending Functors with a rollback and, e.g., a *two-phase commit* protocol implementation.

## References

[Ar07] Armstrong, J.: A History of Erlang. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. Pp. 6-1–6-26, 2007.

[Be14] Bernstein, P. A.; Bykov, S.; Geller, A.; Kliot, G.; Thelin, J.: Orleans: Distributed Virtual Actors for Programmability and Scalability, tech. rep., Microsoft Research, 2014.

[Be17] Bernstein, P. A.; Dashti, M.; Kiefer, T.; Maier, D.: Indexing in an Actor-Oriented Database. In: Proceedings of the Conference on Innovative Data Systems Research (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.: Transactions for Distributed Actors in the Cloud, tech. rep., Microsoft Research, 2016.

[Er18] Erlang Solutions Ltd: Case Studies & Insights, 2018, URL: https://www.erlang-solutions.com/resources/case-studies.html, visited on: 08/17/2018.

[Li18a] Lightbend, Inc.: Akka, 2018, URL: https://akka.io/, visited on: 08/15/2018.

[Li18b] 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.

[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 Actor Database Systems. In: Proceedings of the International Conference on Management of Data (COMAD). Pp. 259–274, 2017.

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