An implementation of a fault-tolerant database system using the actor model

Ioan Cristian Schuszter[1] and *Marius* Cioca[2]

[1] University of Petrosani, Department of Computer and Electrical Engineering, Petrosani, Romania
[2] "Lucian Blaga" Faculty of Engineering, University of Sibiu, Sibiu, Romania

Abstract. Fault-tolerant systems are an important discussion subject in our world of interconnected devices. One of the major failure points of every distributed infrastructure is the database. A data migration or an overload of one of the servers could lead to a cascade of failures and service downtime for the users. NoSQL databases sacrifice some of the consistency provided by traditional SQL databases while privileging availability and partition tolerance. This paper presents the design and implementation of a distributed in-memory database that is based on the actor model. The benefits of the actor model and development using functional languages are detailed, and suitable performance metrics are presented. A case study is also performed, showcasing the system's capacity to quickly recover from the loss of one of its machines and maintain functionality.

1 Introduction

As more and more companies take part in the process of digital transformation, so too must their services be scaled in order to adapt to the growing challenges of the current IT landscape. Scalability is an extremely important characteristic of a growing company's services, as it can be a hard ceiling for the growth of revenue.

Most IT systems require something more than ephemeral storage, a means of storing data even if a power loss occurs, hence they resort to using databases. Traditional database systems are often based on the relational data model developed in the 1980's [1] but they are often difficult to scale. The CAP theorem is a highly cited theoretical proof that only two out of the three desirable features of a database system can be achieved at once [2]. As such, it is desirable to try and achieve all 3 qualities: consistency, availability, partition tolerance. The system implemented in this work strives to achieve availability and partition tolerance, while trying to attain eventual consistency via replication, maintaining appearances through the use of

This paper studies the implementation details of a distributed in-memory database leveraging a programming model known as actor systems. These systems are built to be resilient and flexible, being heavily used in the telecommunications industry and in the design of fault-tolerant systems [3]. A comparison is also drawn and presented between the developed system and alternatives such as Redis [4], showing that performance is not lost in detriment to fault-tolerance.

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

2 Technology considerations and background

To better understand the motivation for the elaboration of this work, it is necessary to understand all the concepts covered by the implementation. As such, they will be briefly discussed in the following subsections.

2.1 Actor models and modern implementations

The concept of actors as a system for general purpose computation emerged in the early 1970s, far before many practical implementations of computer science could actually be achieved. It was Hewitt, Bishop and Steiger that showed that concurrent computation could be achieved through a primitive named the actor and a small amount of permitted operations [5]:

- Spawning other actors
- Sending a message to another actor
- Answering a message by replying to an actor

Throughout the years, many improvements were added to the model in such a way that an actor theory was formed, drawing heavy inspiration from physics and functional languages like Lisp. Actors may modify their own internal state but can be interacted with only via messages, thus locks are not needed at all in their programming.

Akka is an actor programming library that has garnered a large amount of following in the past years [6], being used in various highly-distributed environments with great success, such as in the development of a fog computing application [7]. The library uses the Scala programming language for flexibility and leverages concepts from functional programming, such as immutability, for implementing an efficient actor system. It forms the core of the proposed architecture in this paper. Another

Akka Http is a complement to the Akka library, allowing fast concurrent APIs to be built and contacted from the web, using the underlying actor model to implement an extremely scalable web server [8]. It is used as the wrapper web service for the database, as it is meant to be accessed over the web.

2.2 NoSQL databases

The paper presents an efficient design of a NoSQL in-memory database. As such, familiarity with the terms is achieved through the following overview.

The 2000s and 2010s have led to a revolution in terms of storage capacities as well as needs. SQL databases were not able to cope with some of the loads that services were facing, leading to a new generation of databases coined NoSQL. There are in-memory databases that are often used for caching requests or queries, such as **memcached** or **Redis**, or document-based databases such as **MongoDb**, as well as columnar databases such as **Cassandra**. Han et al present a very structured review on all the different flavors of available NoSQL databases in their paper [9].

2.3 In-memory databases

These type of databases store values in memory and are often used for high-volume tasks such as storing user sessions or caching server responses so that the server's CPU is spared several cycles. Internet-scale services have extreme performance requirements, making in-memory databases a great tool for scaling services.

There are many implementations, both commercial such as those provided by Oracle TimesTen and open-source such as Redis or memcached [10].

3 Architecture and implementation

The entire database system acts as a web service to the eyes of the outside world. Complexities such as the distribution of the actor system and the messages being passed between them are hidden from the clients.

There are several actors involved in the composition of the database:

- **MasterActor**: responsible for managing the rest of the actors, created on system launch
- **HeartbeatActor**: responsible for communicating information about live actors to the master actor and notifying the master actor in case any of them goes missing
- **DataPartitionActor**: responsible for the management of the actors which actually store the data, this data is replicated across several different partitions and actors that store the data in maps.
- **DataStorageActor**: the base actor that knows how to handle a small amount of messages such as "Put", "Get" and "Delete". It reports to the HeatbeatActor every second using a scheduled heartbeat.

For a visual interpretation of the entire system, **Fig. 1** presents the major information captured by the description above.

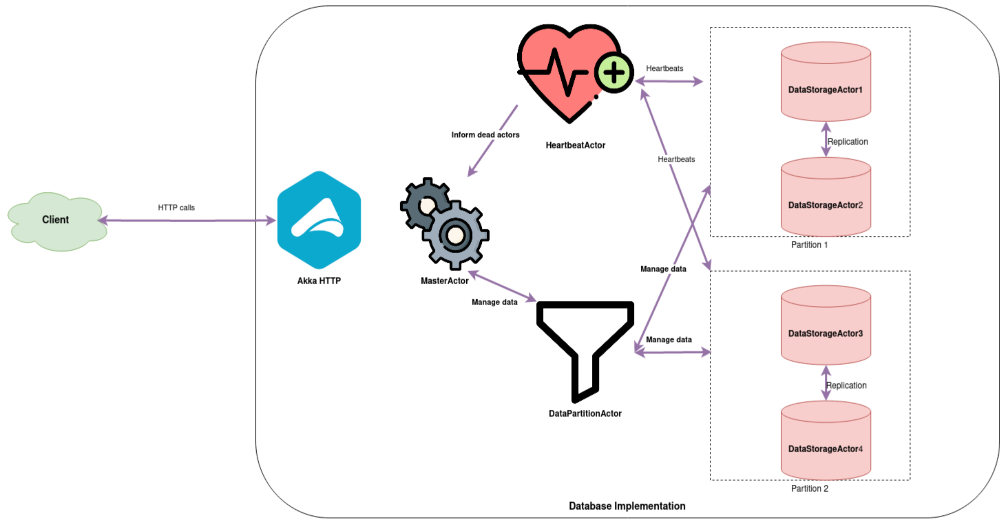

Fig. 1. Full architecture diagram

The data storage actors are organized in partitions, with a configurable replication factor (defaulting to 3). Every key-value pair stored in the database by an external client is propagated in a round-robin fashion to every partition. Due to the advantages given by **Akka Cluster**, the data partitions can live on different machines and gracefully handle the loss of one or more actors. As long as one of the actors in the partition is still alive, the system is guaranteed to not have any data loss.

3.1 Data recovery case study

Once an actor is removed from the system, the **HeartbeatActor** realizes when it doesn't send any status update within 5 seconds. If no response is received, the **DataSorageActor** is evicted and replaced with a fresh, empty one. The new actor then receives a message to replicate the data from the other actors in the partition and re-achieve equilibrium.

An endpoint for simulating a failure in one or more of the actors was also implemented, to test out the automatic re-balancing algorithm of the partitions. When the **HeartbeatActor** discovers issues with the service, a new **DataStorageActor** is started and it replicates all the missing data. An example of the log output from the service can be seen below. As the data is stored in-memory, the replication process takes only a few seconds.

```
    Apr 11, 2021 2:05:53 PM com.cristis.actors.HeartbeatActor$$anonfun$receive$1 applyOrElse
    INFO: These actors seem to be dead:
Map(Actor[akka://default/user/master/dataStorage_1#2081017385] -> 1618142747087), telling the
master!
    Apr 11, 2021 2:05:53 PM com.cristis.actors.DataStorageActor$$anonfun$receive$1 applyOrElse
    INFO: Distributing data to
Vector(Actor[akka://default/user/master/dataStorage_10#1098448318])
    Apr 11, 2021 2:05:53 PM com.cristis.actors.DataStorageActor$$anonfun$receive$1 applyOrElse
    INFO: Loading data from Actor[akka://default/user/master/dataStorage_0#1823960673]
    Dead actors: Vector(Actor[akka://default/user/master/dataStorage_1#2081017385])
    Affected partitions: ListBuffer(0)
```

4 Results

Redis with a small Spring Boot HTTP API wrapper was chosen as a comparison to the Akka implementation of the scalable in-memory database. Several tests were performed in order to better understand how the system scales under real-time load. Being a distributed service, concurrent calls were used in order to simulate multiple simultaneous users accessing data and modifying it. The systems were tested on a machine running Ubuntu 20.04 with a Ryzen 5 1600 CPU and 16GB of RAM. All values come from tests performed on the same machine.

As can be seen from **Fig. 2**, both systems behave very well when it comes to setting various values, with the distribution of response times being around the 3ms mark. The bins do show that the Akka DB tends to have slightly faster response times than the Redis one, likely due to the usage of a single thread to handle all queries and locking while the value is being populated.

In **Fig. 3** however, a completely different story is presented. Both databases were populated with 10000 random keys as GUIDs, in order for the distribution in-memory to be completely random. Querying both the Spring Redis service and the Akka DB worked fine for individual queries, but when sustained with concurrent loads, the single-threaded nature of Redis could be seen quite well. While the response times of the Akka DB were on average around 500ms for the entire dataset, the response times for bulk queries in Redis averaged around 10 seconds. The single-threaded nature of the Redis instance causes all the concurrent requests to stall and increases the wait time for each of the clients significantly. One way of avoiding such an issue is to setup a clustered Redis instance, but that would increase the time spent waiting in this particular case, as network calls are generally expensive and they would be needed in order to ensure the retrieval of all the keys.

Without doubt, the usage of Akka HTTP and immutable actor messages ensures a no-blocking situation which greatly benefits the response times of the service overall. This property, combined with the inherent fault tolerance of the distributed data in the actor system, shows this architecture and implementation as something suitable for a production service.

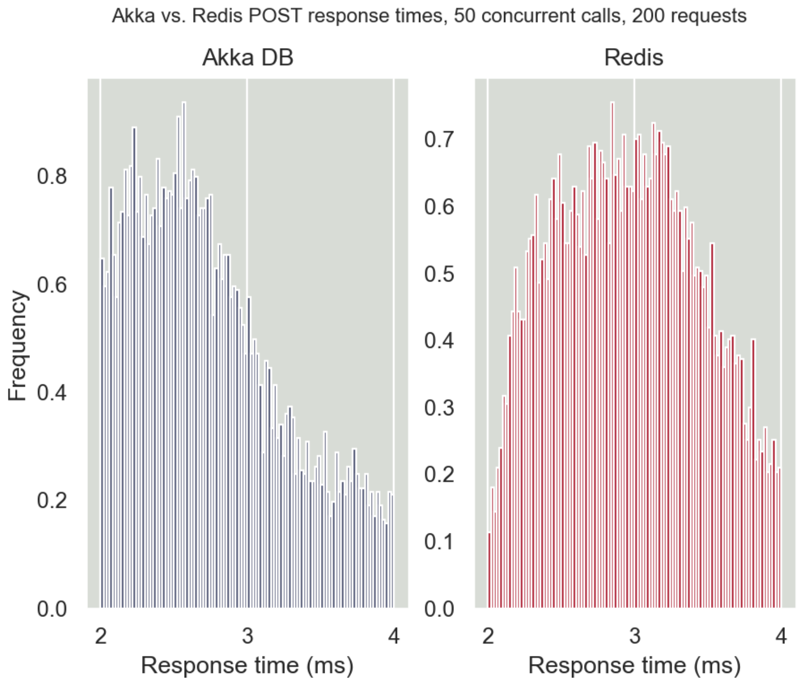

Fig. 2. Write performance for 50 concurrent calls. 10000 keys loaded in the database.

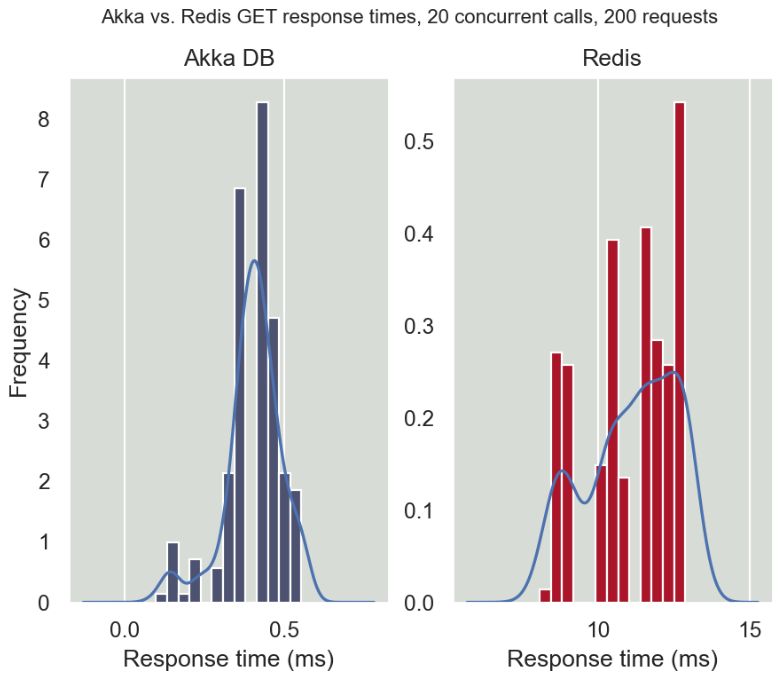

Fig. 3. Read performance for 20 concurrent calls. 10000 keys loaded in the database.

5 Conclusions

The service architecture and implementation discussed in this paper has shown great promise as a replacement for other conventional in-memory database implementations that are out there on the market, as their capacity to scale (especially for enterprise solutions) is limited by their lack of fault-tolerance. Although it is a valid argument that the data put in in-memory DBs, like Redis, is inherently volatile, there are many instances in which volatile data should be persisted at all costs to avoid inconveniences, such as session data for user logins. In completion to the above, there is a body of research covering the leveraging of new NVM technology to recover from lost data, even in the case of power failures, but this is still just a research topic [11].

Another advantage of the system presented is the use of the actor library itself. Pure functional programming in the mathematical sense, which uses immutable data structures, is a powerful tool for building distributed lock-free systems [12]. As such, extending the existing Scala code of the application is both safe and efficient for such a database system implementation.

Future work will cover more extensive research on the way actors are split across various machines using Akka Cluster would influence the performance and response time of the service. Calls to agents on another machine would generally be much slower than to those in a system running on the same machine, but it is important to understand exactly how large the overhead is. As such, an extension to this paper can be the study of various network architectures and partition strategies for the in-memory database system and their influence on the system's performance.

References

1. E.F. Codd, Communications of the ACM **26.1**, 64-69 (1983)
2. E. Brewer, *29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 335-335 (2010)
3. K. Ozaki et al., *20th International Conference on Advanced Information Networking and Applications*, vol. 2, 5 (2006)
4. Redis main site. Available on https://redis.io/ (accessed 04.2021)
5. C. Hewitt, P. Bishop, R. Steiger, *3rd International Joint Conference on Artificial intelligence*, 235-245 (1973)
6. M. Gupta, *Akka essentials* (Packt Publishing Ltd, 2012)
7. S. N. Srirama, F. Dick, M. Adhikari, Future Generation Computer Systems **117**, 439-452 (2021)
8. Akka HTTP documentation, Lightbend. Available on https://doc.akka.io/docs/akka-http/current/index.html (accessed 04.2021)
9. J. Han et al., *6th international conference on pervasive computing and applications*, 363-366 (2011)
10. G. Mohit, V. Vishal, V. Megha, International Journal of Engineering Trends and Technology **6**, 333-336 (2013)
11. J. Arulraj, A. Pavlo, S.R. Dulloor, *ACM SIGMOD International Conference on Management of Data*, 707-722 (2015)
12. C. S. Gordon et al., ACM SIGPLAN Notices **47.10**, 21-40 (2012)