Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2022, 9(12):170-174

Research Article ISSN: 2394 - 658X

Leveraging the Actor Model to Build Scalable and Resilient Data Platforms

Gautham Ram Rajendiran

gautham.rajendiran@icloud.com

ABSTRACT

The paper discusses an approach to simplify the implementation of data platforms using the Actor Model in order to reduce the complexity in the implementation of data platforms. Traditional data platforms [2] rely on several components, such as message brokers, stream processors and task orchestrators, which can have a high initial setup time and also operational overhead. The Actor Model encapsulates all these components into one system, wherein actors can ingest, process, and store data in a highly concurrent, distributed and fault-tolerant manner. This will reduce the effort of developers by focusing on implementation details and will provide more time to focus on other aspects of improvement.

Keywords: Distributed Systems, Data Platform, Actor Model, Concurrency, Parallelism

INTRODUCTION

This paper explores an alternative approach of building data platforms by using a mathematical model of concurrent computation in Computer Science called the Actor Model [1]. We explore how such an approach simplifies architecture and implementation by consolidating almost every component from the traditional pipeline into a single system. We will then explore the implementation by using a case study and analyze performance metrics like latency and throughput. Finally, we elaborate on how such an architecture can be used to build data pipelines for various use cases and conclude by highlighting the advantages of time saved due to native implementations, scalability and fault tolerance of data platforms built using this method versus contemporary data platforming techniques.

PROBLEM STATEMENT

Contemporary data processing platforms contain a number of components, of which a few are commonly found:

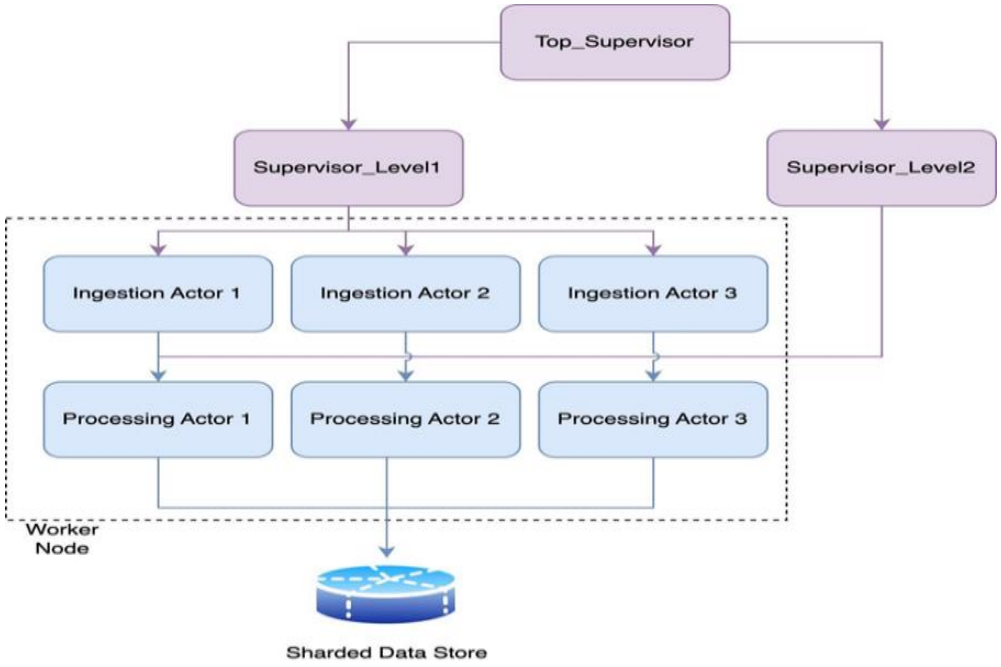
1. Message broker - Ex. SQS, Kafka
2. Message Consumer
3. Distributed Processor - Ex. Apache Flink, Apache Spark, Apache Airflow
4. Data Sink - Ex. Data warehouse

These are the bare minimum components required to build a publisher-consumer based data platform that can be extended for both batch and stream processing. Each of these components have their own implementation complexities and maintenance overheads. For example a Message broker needs to implement scaling policies, message delivery guarantees, message replay features etc. Consumers need to be aware of processing constraints, producer back pressure and late-arriving data. Such complexities increase with the number of components which increases the time required to implement these systems and the operational overhead of maintaining such a system. In this paper we aspire to reduce such complexity while also maintaining the advantages that such elaborate systems offer viz. availability, reliability and fault tolerance.

IMPLEMENTATION

A data platform can be broadly classified to have a few core responsibilities, some of which are defined below along with a brief description of how they can be modeled using the Actor Model.

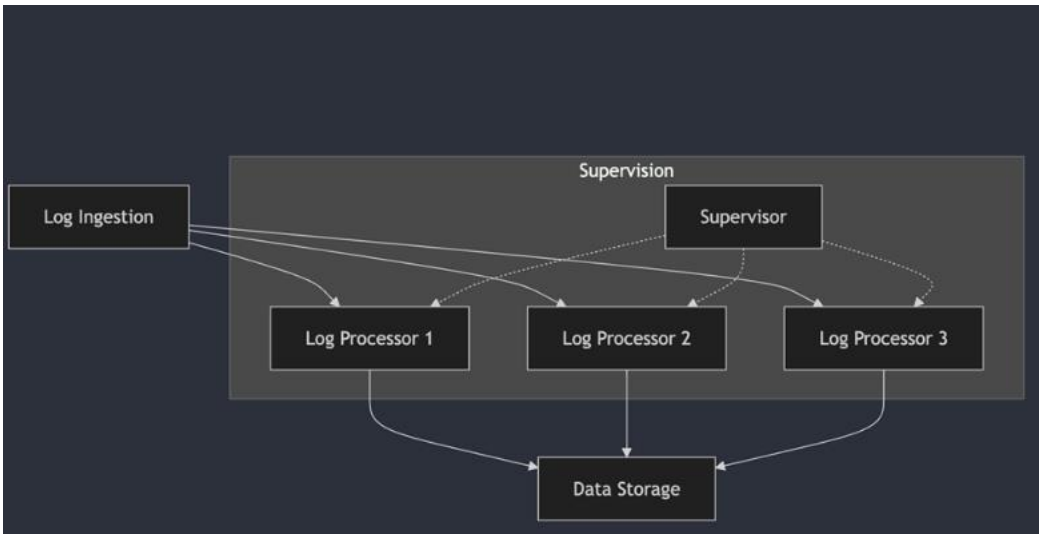
Data Ingestion: Actors receive data coming from different sources, such as IoT devices, event streams, or user activity logs. Each actor will process its assigned portion of data independently so that even a really big volume of incoming events can be sharded and processed by individual actors.
Data Processing and Transformation: Each actor in data processing may perform data transformation, aggregation, or cleaning activities. Since the Actor Model encourages isolation, failure of one part of the pipeline does not affect others.
Data Storage: After processing, the actors push the data for storage to databases or other distributed storage systems. Multiple actors may store concurrently to different partitions or shards; each shard is then scaled horizontally.
Monitoring and Recovery: Using the supervisors to monitor the actors and restart them upon failure yields a robust, fault-tolerant platform that is self-healing with no human intervention.



The diagram above depicts how multiple layers of supervisors can coordinate the lifecycle of actor processes and dynamically spin up a process for each unit of work that the supervisor receives. A sharded data store can be utilized for parallel insertions.
The Supervisor is a construct that implements fault-tolerance in the actor model; it coordinates the lifecycle of an actor process. A hierarchy of supervisors will eliminate the possibility of a single point of failure due to the presence of only one supervisor. In the next section we demonstrate the implementation of a log processing pipeline implemented using a data platform built with design principles from the model.

CASE STUDY: LOG STREAM PROCESSING PIPELINE

A log processing system has to deal with large-scale ingestion of logs from various different sources. It has to be concurrent, scalable and fault tolerant. We will implement this pipeline using principles from the Actor Model that we have covered so far, and highlight its efficiency and advantages.
The pipeline is composed of the following stages:
1. Log Ingestion: Logs are ingested from various sources (e.g., application logs, server logs) in real time. This is an external source of events.
2. Log Processing: Multiple log processors handle logs concurrently, performing tasks such as filtering, formatting, and enriching the data. This corresponds to the actor process.
3. Data Storage: Processed logs are stored in a distributed storage system, such as a database or file storage. This is the data sink.
Supervisor monitors the log processing actors and re-creates the actor process in case of premature termination due to errors during processing.



The figure above depicts how multiple actor processors are created dynamically for every log file that has been ingested. If such a system were to be implemented using contemporary data platforms, we would need to set up auto-scaling clusters using tools like Kubernetes or AWS ECS. Leveraging the Actor Model capabilities makes such a setup very easy since horizontal scaling of actors can happen within a single node or across multiple nodes, which gives us the option to scale vertically or horizontally.

Apart from ease of setup, several other advantages have been identified through this case study. Some of which are explained below:

Ingesting and Processing Logs Parallely: The Actor Model's asynchronous message-passing and process based architecture enables the creation of processes that can spin up on the fly and process a task without needing to worry about resource contention. This significantly increases throughput.

Scalability: As more log sources are added or log volume increases, new actors (log processors) can be spawned dynamically. Each actor handles its own task, and since actors don't share state, scaling out across multiple nodes is straightforward.

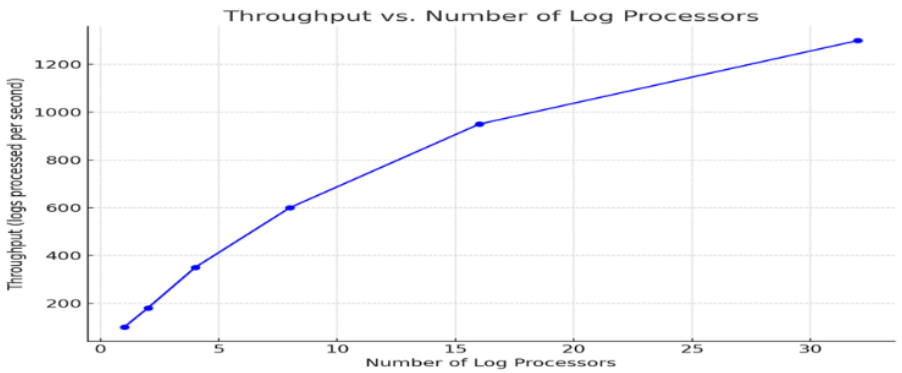
Automatic Recovery: In the Actor Model, failures are isolated to individual actors. Supervisors monitor the log processors and restart them if they fail, ensuring that the system continues to run without manual intervention. This increases the overall system's fault tolerance.

Graceful Shutdown: Since each actor operates independently, a failure in one log processor won't bring down the entire system. This design prevents cascading failures and ensures the system can shutdown gracefully, with minimal impact on performance.

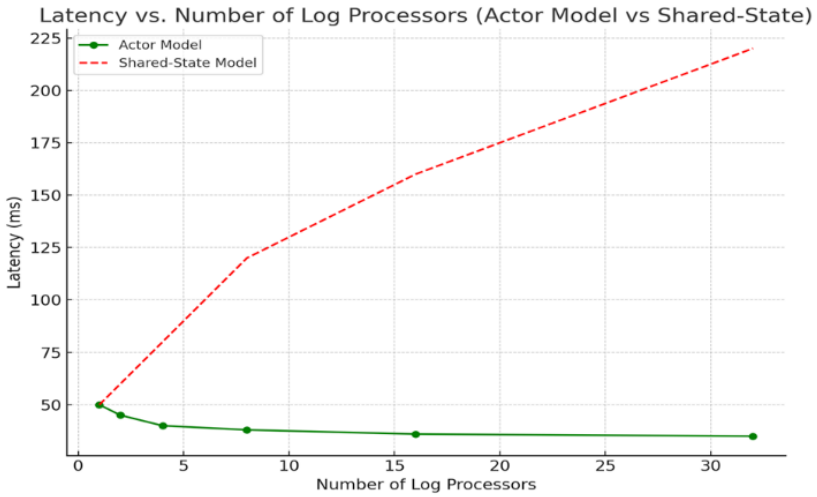
No Shared State: Each log processor maintains its own state, which reduces complexity when handling concurrency. There's no need for locks or shared memory management, which are common sources of bugs in traditional multi-threaded systems.

In addition to the performance benefits mentioned above, we also observed the following throughput and latency improvements.

The graph below demonstrates how throughput of the system increases as more log processors (actors) are added. This demonstrates the scalability of the Actor Model. By simply increasing the number of log processors, the system can handle a significantly higher volume of logs, as long as there are sufficient worker nodes available to scale into.



The graph below shows how latency behaves in the Actor Model versus a traditional shared-state concurrency model. The shared-state system was implemented using thread pools with access to the same number of cpu cores as the actor model. With the actor processes,, latency remains stable even as the number of log processors increases, because each actor operates independently. In contrast, latency increases in a shared-state system as more processes contend for shared resources, leading to performance bottlenecks. In situations like this, shared-state systems would need to vertically scale, which is not a problem for the Actor based system due to efficient resource utilization.

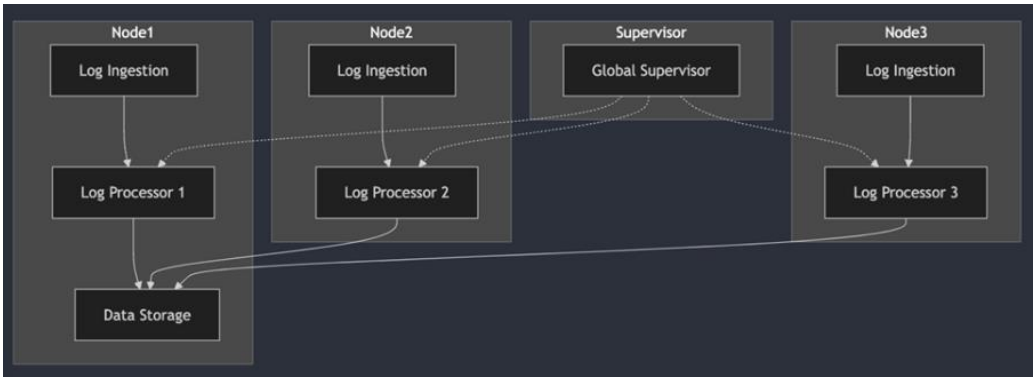


The pipeline so far has only been set up to scale the number of processes within a single node. One of the major strengths of the Actor Model is its ability to scale across multiple nodes effortlessly.

Distributed Actors: Actors (processes) are not tied to a single machine. They can be distributed across multiple nodes in a cluster, enabling horizontal scaling. Each node in the cluster can host log processors, and the supervisor tree can be distributed to monitor actors across these nodes.

Seamless Node Communication: Message based communication between processes supports transparent communication across nodes. This means that actors on different machines can send and receive messages as if they were on the same machine, making scaling nearly seamless.

Diagram: Multi-Node Log Stream Pipeline



GENERALIZED USE OF THE ACTOR MODEL IN BUILDING DATA PIPELINES

Although the case study focused on a log processing pipeline, the same principles can be applied to virtually any data pipeline. Whether it comes to processing real-time events, streaming data, or performing complex transformations, the Actor Model offers a highly scalable and resilient solution.

Here are a few other sample data pipelines that can benefit from this approach:

Real-Time Event Processing: Actors can process streams of events from IoT devices, financial transactions, or web activity in real time, without the need for traditional event-driven architectures.

Data Transformation Pipelines: Similar to the log processing system, data pipelines that require filtering, enrichment, or aggregation can be easily modeled with actors handling each stage and a dedicated actor for each transformation.

Analytics and Machine Learning Pipelines: The Actor Model is also well-suited for distributed tasks like feature extraction, model inference, and data aggregation, which can be performed in parallel by actors across nodes.

Such an ubiquitous nature is what makes this model an ideal candidate to implement data platforms.

PROGRAMMING LANGUAGES THAT IMPLEMENT THE ACTOR MODEL

Several programming languages have implementations of the Actor model, the most popular among them being Erlang, and by extension Elixir[4] which uses the BEAM virtual machine behind the scenes. Java and Scala have a library called Akka[5] which implements this. Orleans is a module that implements the actor model for .NET. There are other libraries that implement the model for other languages like Python. All of these languages have their own implementations of the actor and asynchronous messaging system. Fault tolerance is specific to the language itself, but generally adheres to a supervisor based pattern as shown in this article.

RESULT

By adopting this simplified yet sophisticated approach, building and maintaining a data platform becomes easier. This architecture has fewer components and reduced operational overhead. Scalability is enhanced as actors can be dynamically added to handle increase in load, while fault tolerance is built into the architecture through supervision trees. This leads to faster deployment, less developer effort, and a more resilient system compared to traditional data platforms that rely on multiple external services and configurations. The model is also abstract enough to be implemented for various data pipelines, which makes it an ideal candidate to build a data platform. Overall, we get a platform that is more efficient, scalable, and fault-tolerant.

CONCLUSION

We explored the use of the actor model and its advantages when used to implement a data platform. We followed this up with a demonstration of its native scalability and resilience. We then contrasted this implementation to traditional pipelines that rely on multiple services, message queues, and orchestration layers. As a result, we are able to conclude that the Actor Model provides a more resilient, scalable, and maintainable solution compared to other data platforming solutions and so, we propose that leveraging the actor model is a great method to build scalable and resilient data platforms.

REFERENCES

[1]. A. A. L. Zhang and H. Y. Xie, "A Study of Distributed Computing Models: The Case of MapReduce," arXiv preprint arXiv:1008.1459, Aug. 2010. [Online]. Available: https://arxiv.org/abs/1008.1459

[2]. B. Cheng, S. Longo, F. Cirillo, M. Bauer and E. Kovacs, "Building a Big Data Platform for Smart Cities: Experience and Lessons from Santander," 2015 IEEE International Congress on Big Data, New York, NY, USA, 2015, pp. 592-599, doi: 10.1109/BigDataCongress.2015.91.

[3]. Z. Wu et al., "Towards building a scholarly big data platform: Challenges, lessons and opportunities," IEEE/ACM Joint Conference on Digital Libraries, London, UK, 2014, pp. 117-126, doi: 10.1109/JCDL.2014.6970157.

[4]. S. G. Johnson and R. S. Burden, Programming Elixir 1.6: Functional, Concurrent, and General-purpose Programming, Pragmatic Bookshelf, 2018.

[5]. Akka: Build Powerful Reactive, Concurrent, and Distributed Applications More Easily, Akka.io. Available: https://akka.io/.