Distributed Machine Learning Framework on Akka

by
Ioannis Lamprinidis

Thesis Commitee:
Prof. Samoladas Vasilis (Supervisor)
Prof. Deligiannakis Antonios
Prof. Giatrakos Nikos

A diploma thesis submitted in partial fulfilment of the requirements for the degree of Electrical and Computer Engineering

Department Name
Technical University of Crete
Date 7 October 2024

Abstract

The proliferation of data-driven applications has led to a growing demand for efficient and scalable machine learning algorithms. This thesis delves into the design and implementation of a distributed communication kernel in Akka for the Online Machine Learning and Data Mining system(OMLDM), a system that supports distributed online learning by utilizing the Parameter Server paradigm, for effortlessly deploying Online Machine Learning pipelines on streaming platforms. The objective was the implementation of an efficient, scalable, fault tolerant and robust kernel for the OMLDM, to analyze the performance overhead of Akka by comparing it to a local implementation of the OMLDM kernel, that utilizes Java Threads; To evaluate the performance speedup achieved by the kernel in a cluster environment. We demonstrate through experiments the communication overhead of Akka and the performance of the kernel in local and clustered environments.

Περιληψη

Η αυξανόμενη διάδοση των εφαρμογών που βασίζονται στα δεδομένα έχει οδηγήσει σε αυξανόμενη ζήτηση για αποτελεσματικούς και κλιμακωτούς αλγόριθμους μηχανικής μάθησης. Αυτή η διατριβή εμβαθύνει στο σχεδιασμό και την υλοποίηση ενός κατανεμημένου πυρήνα επικοινωνίας στο Akka για το σύστημα Online Machine Learning and Data Mining (OMLDM), ένα σύστημα που υποστηρίζει κατανεμημένη online μάθηση αξιοποιώντας την Parameter Server αρχιτεκτονική, για την ανάπτυξη Online Machine Learning pipelines σε πλατφόρμες ροής. Ο στόχος ήταν η υλοποίηση ενός αποτελεσματικού, κλιμακωτού, ανθεκτικού σε σφάλματα και ισχυρού πυρήνα για το OMLDM, η ανάλυση της επιβάρυνσης απόδοσης του Akka σε σύγκριση με μια τοπική υλοποίηση του πυρήνα του OMLDM που υλοποιήθηκε σε Java Threads και η αξιολόγηση της αύξησης της ταχύτητας απόδοσης που επιτυγχάνει ο πυρήνας σε ένα περιβάλλον cluster. Παρουσιαζεται μέσω πειραμάτων η επιβάρυνση επικοινωνίας που εισάγει το Akka και η απόδοση του πυρήνα σε τοπικά και clustered περιβάλλοντα.

Contents

Chapter 1

Introduction

1.1 Motivation

In the rapidly evolving landscape of data-driven decision-making,the amount of data that is generated every moment is too large or costly to send,store and then process. Distributed cluster systems have emerged as a cornerstone for processing and analyzing vast volumes of information. These systems not only enable the parallel execution of tasks across multiple nodes, enhancing computational efficiency and scalability but also move the processing power where the data is generated.

Online machine learning(OML) algorithms are Machine learning algorithms that are used whenever it is infeasible to train a model on the entire training data set at once.They train on data that becomes available in a sequential order and are able to adapt themselves whenever new data arrives. Thus combining a distributed system with OML algorithms unlock the potential for real-time predictions and adaptive responses to dynamic environments.

This thesis relied on the Online Machine Learning and Data Mining library (OMLDM), a JAVA library that implements the OML algorithms and a network of Spokes and Hubs to run the processes and it delves into the intersection of distributed cluster systems, and online ML APIs, focusing on the challenges and opportunities presented by their

integration. Specifically, we investigate the design and implementation of a distributed cluster system in Akka where each worker node leverages an online ML API for predictive tasks.The primary goal is to understand the performance implications of this architecture, exploring factors such as communication overhead, model synchronization, and resource utilization.

We anticipate that our findings will provide valuable insights for practitioners and researchers alike, informing the design and deployment of future systems that harness the power of distributed computing and OML for real-world applications.

1.2 Thesis Outline

In the following sections, we will delve deeper into the background and context of the research, outlining the objectives, methodology, and contributions of this thesis. Chapter 2 describes the architecture of the OMLDM ,Chapter 3 we analyze the frameworks that were used to implement the system, Chapter 4 delves into the design of the kernel system in Akka and Chapter 5 presents the results obtained from testing the system.Chapter 6 concludes the thesis by summarizing the main findings and discussing their implications.

Chapter 2

Related Work

2.1 Introduction

This chapter will give a quick explanation of Threads, their usefulness but also the problems that they create and will introduce a model on top of threads, the Actor model that provides a solution to the aforementioned issues.

2.1.1 Parallelism and concurency

The simultaneous execution of several tasks, known as parallel computing [10], is frequently accomplished through multiprocessing, in which each task is run on a separate processor or core. By enhancing responsiveness and enabling the effective handling of new events without waiting for ongoing computations to finish, it further enhances performance when coupled with increased processor availability. Multiple processors can be necessary, but there may be a trade-off, particularly if tasks are seldom or have a short duration and could result in processor idleness.

Parallelization, the division of a computation into independent parts that can be executed at the same time, further enhances performance when coupled with increased processor availability. However, the need for multiple processors can be a trade-off, especially when tasks are infrequent or short-lived, leading to potential processor idleness.

Concurrency, a broader concept than parallelism, allows tasks to execute in any order, regardless of whether they share a single processor or run on multiple ones. This flexibility makes concurrency a more general approach, accommodating both parallel and interleaved execution patterns. The specific execution model depends on the programming language and runtime environment.

On single-processor systems, concurrency is achieved through interleaving, where the kernel sequentially assigns the processor to different tasks, creating the illusion of parallel execution. This interleaving, often driven by timer interrupts or other events, enables multiple tasks to seemingly run in parallel despite sharing a single processor.

If two tasks are concurrent (e.g. foo1 and foo2) the order of execution is unknown. They may be executed in 1 of the 4 ways:

- foo1 executes first, then foo2.
- foo2 executes first, then foo1.
- foo1 and foo2 execute at the same time.
- alternate execution between foo1 and foo2.

2.1.2 Shared resources, Synchronization and Deadlocks

Shared resources

Synchronization mechanisms are crucial in concurrent programming to ensure the orderly access of shared resources and maintain the correctness of computations that depend on specific execution sequences. Consider the following example: Two tasks share a common variable x that acts as a counter. Every time the tasks run, they increment x by one.To achieve that, the following steps are done:

- read x
- increment by one

- write the value back to x

If the tasks(e.g. foo1, foo2) run after the other ends then x is updated successfully(x=x+2), but let's consider the following case: Both foo1 and foo2 get value x, then increment it and write back the value to x, the final result will be x=x+1.

Cycle	task foo1	task foo2	Cycle	task foo1	task foo2
0	u=x		0	u=x	
1	x=u+1		1		j=x
2		j=x+1	2	x=x+1	
3		x=j+1	3		x=x+1

Figure 2.1: Different ways of execution of two parallel tasks

Synchronization

To avoid uncertainty in a system behavior, there are multiple ways to solve this problem. One common way is with the use of mutexes. Mutex is a lock that can only be held by one task at a time. Thus, a task can be run one at a time to access a shared resource.

Deadlocks

The introduction of locks creates a new problem, deadlocks. Deadlock is a situation where no task can proceed because it awaits for another member to release the lock. A deadlock occurs in a system when all the following conditions simultaneously occur:

- Mutual Exclusion: A resource can only be used by one process at a time, and cannot be shared.

- Hold and Wait: A process holds at least one resource while waiting for additional resources held by other processes.

Cycle	task foo1	task foo2
0	lock m	
1	u=x	
2	x=u+1	lock m
3	unlock m	wait
4		j=x
5		j=j+1
6		unlock m

Figure 2.2: An example of establishing synchronization using mutexes

- No Preemption: A resource cannot be forcibly taken away from a process; it must be released voluntarily.
- Circular Wait: A chain of processes exists where each process is waiting for a resource held by the next process in the chain, creating a circular dependency.

These four conditions are known as the Coffman conditions [3].

To understand the problem of deadlocks, the following example is introduced: We want to transfer money between two bank accounts(A, B) and each account has a lock. The first attempt is to achieve that is when we want to do a transfer is to lock both accounts, transfer the money, and then unlock them. This method has an obvious problem: if one task (task A) wants to transfer money from account A to B and another (task B) wants to transfer from B to A, a deadlock may happen:

- Task A locks account A.
- Task B locks account B.
- Task A awaits for the lock of account B to become available.

- Task B awaits for the lock of account A to become available.

It becomes evident that the system cannot move forward because task A and task B wait for each other. There are ways to evade the deadlock. One simple way is to have a lock for all accounts and one better solution is, before locking the needed accounts, to sort them according to a unique variable(e.g. an ID). Thus, the use of locks is not foolproof and needs careful use when designing a system, but this problem magnifies when designing large scale systems.

2.1.3 Threads

A thread is a single sequence stream in a process and often described as light processes. They exist within a single process and allow for the execution of multiple tasks concurrently/parallelly. Each thread has its own flow of control, meaning it can execute a specific set of instructions independently of other threads in the same process.
Threads within the same process share the same memory space and other resources. This introduces the problem of two or more threads accessing the same resource. The common solution to this problem is using locks around the resources. This allows the threads to access the resource one at a time, but the following problems arise:

- Locks limit concurrency, and they are very costly on modern CPU architectures.
- The caller thread, once blocked, becomes unproductive, waiting for a resource to become available. This idle waiting is particularly problematic in contexts where responsiveness is key, such as user interfaces and backend services, as it can severely impact the user experience.
- Locks introduce the problem of deadlocks.

Furthermore, locks are primarily effective when applied within a local context. In the case of distributed environments, the only solution

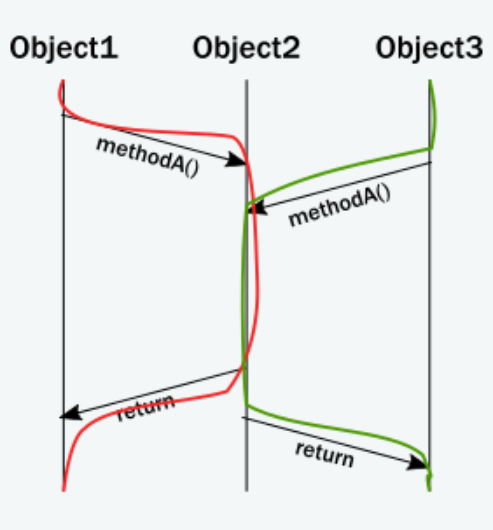

Figure 2.3: Threads resource access

is distributed locks. Unfortunately, the implementation of distributed locks requires several communication messages over the distributed network, thus becomes very expensive and not as efficient as the local lock mechanism.

The traditional call stack model, designed for single-threaded programs, falls short in accurately representing the complex execution flow of asynchronous programs that involve multiple threads. This limitation stems from call stacks being inherently confined to individual threads.

In asynchronous scenarios, when a thread delegates a task to another thread, it often involves placing an object in a shared memory location accessible to the worker thread. While the caller proceeds with other tasks, the worker thread picks up and executes the assigned job.

This raises the issue of how can the caller be notified for the completion of the task. Similar but more serious is the case of the thread failing with an exception. The caller cannot be notified through the call stack and the task is lost even though it was in a shared memory location.

Finally, when a bug happens and the thread closes, not only the current task is lost but also the state of the worker. There are ways to handle all these problems, but add up to the complexity of designing a system.

2.1.4 Actors

The actor model is an abstract paradigm that was proposed by Carl Hewitt in 1973 [6]. To solve the problems stated above, the Actor model was introduced to address parallel processing in a high performance network.Instead of focusing on low-level implementation details, the actor model encourages to approach the code from the perspective of communication and collaboration between independent units, much like individuals in a large organization.

The actor model introduces the actor as the fundamental unit of computation. An actor is an autonomous entity that encapsulates its own state and behavior. Actors interact with each other solely through asynchronous message passing, eliminating the need for shared mutable state and explicit synchronization.

Actors are able to send messages to each other. The messages have no return value. Thus sending a message to delegate work to another actor, does not transfer the thread of execution or blocks the actor. The answer is returned as a message. This enables the actor to handle other messages while it awaits a resource or an answer from another actor.

Another important aspect of the actor model is that each actor process messages sequentially and one at a time. Because the actors communicate only with messages, the sender is not able to access the internal state of the actor. Thus, any change to the internal state of the actor is an event/message for the actor.

To summarize the workflow of an actor:

1) The actor adds a message to the queue.

2) The receiving actor gets ready to execute.

3) The scheduler executes the actor.

4) The actor processes the message, changes his internal state and sends messages to other actors.

5) The scheduler puts the actor to sleep till the next message.

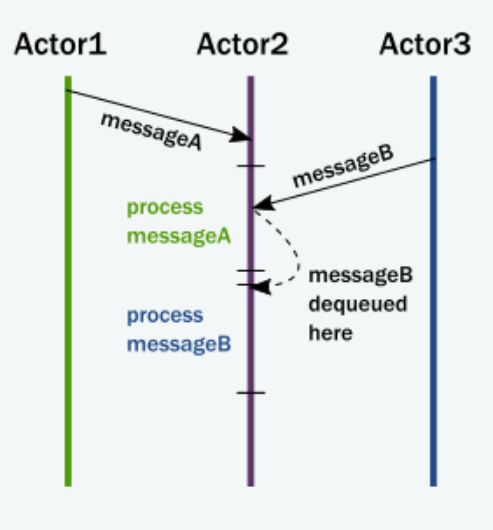

Figure 2.4: Threads resource access

To accomplish the aforementioned workflow, the actor is composed of the following:

- Mailbox: is used to store the messages received.
- Behavior: Is the state of the actor.
- Messages: The data sent to the actor.
- Address: unique identifier used to locate and interact with a specific actor within the system.

Finally, the physical location of the actors does not impact their communication, making them suitable for distributed systems.

In conclusion, the actor model offers a valuable abstraction layer on top of the raw thread architecture. It provides a powerful means to write concurrent code while effectively sidestepping many of the common pitfalls and complexities associated with direct thread management.

2.2 OMLDM

The OMLDM [8] is composed of three parts: The Online Machine learning library(OML), the Network/Middleware layer and the kernel layer.

Each layer is composed of several subsystems that are shown below:

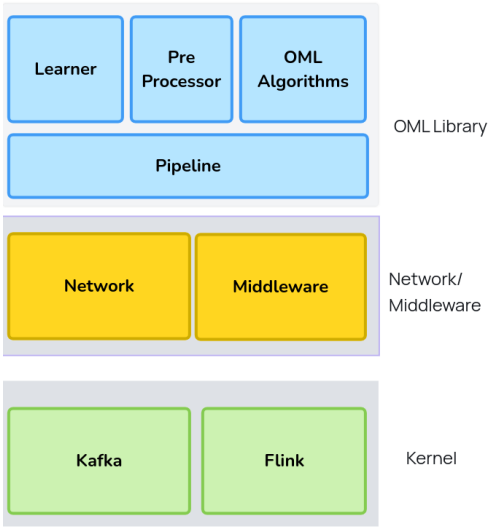

Figure 2.5: OMLDM subsystems

The OML library is organized into three separate parts: the computational layer, the Pipeline manager and the Distributed Learning Logic. his encompasses preprocessing techniques, machine learning algorithms, and data mining methodologies. At the highest level resides the computational layer, where the core machine learning operations are executed. The Pipeline Manager handles the orchestration of the lifecycle and of the computations.The Distributed Learning implements the workers and the parameter servers in a distributed learning environment.
Beneath the OML library lies the Network/Middleware layer serving as the architectural backbone. It is responsible for the creation of a network of abstract processing nodes and an API for internode communication.
At the bottom lies the kernel layer. The original implementation of the kernel layer is on Kafka and Flink.
The tripartite design of the OMLDM enables the users to modify the upper or the lower layer without impacting the rest of the system.

2.2.1 Network/Middleware

The Network layer exists between the kernel and the computational layer, thus creating a level of abstraction between them and at the same time it glues them together. This is accomplished from two subsystems:

- The Network: a group of interconnected abstract Nodes that intercommunicate.
- Middleware: implements a Remote Method Invocation(RMI) API [13], that enables an object to invoke a method on a Java Object running in another JVM (enabling remote communication).

Network

In the network exist two type of nodes, spokes and hubs.Both types of nodes are able to generate messages to other nodes. Spokes are the ones that receive the data to be processed.
The network implements a complete bipartite graph. It's a special instance of a bipartite graph where every node of the first set(e.g. spoke) is connected to every node of the second set(e.g hub). Thus, nodes of the same set cannot communicate with each other.

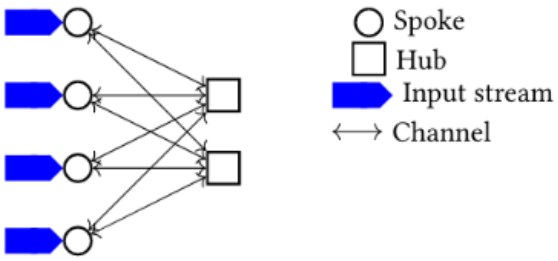

Figure 2.6: Complete Bipartite graph

Within this network architecture, communication between spokes and hubs occurs via bidirectional FIFO channels. The network also supports the atomic broadcast functionality, enabling every node of one set can send the same message to every node of the other set.

Middleware

As mentioned above, the middleware implements a Remote Method Invocation API that enables to invoke a method on a Java object that exists on either the same JVM or a remote one. Every node of the same type instantiates a common remote interface, and each node creates a proxy object that is subsequently distributed to all nodes of the opposite type. This mechanism establishes the communication channels described earlier.

Remote methods are capable of returning values, with the caller specifying a callback to manage these results, and the callback is invoked once for each response received. Combined with the inherent ability of the nodes to communicate with each other on a bidirectional channel, in a one to one or a broadcast fashion, this framework empowers nodes with a versatile suite of functionalities, including but not limited to:

- To send a one way message to a node of the opposite set.
- To send a two-way message and expect a response.The caller may also allocate a callback function to the answer.
- To broadcast a one way message to all nodes of the opposite set.
- To send a two-way message to all nodes of the opposite set and expect a response.

Furthermore, the middleware introduces flexibility for delayed responses with the use of promises. In the context of distributed systems, it's plausible that the callee might not have immediate access to the requested data. To address this, the callee returns a promise value linked to the specific call. Upon acquiring the data, the callee fulfills this promise and transmits the information to the caller node. A similar process has been implemented for broadcast promises.

The broadcast promises are used as a synchronization mechanism for the creation and initialization of the network.

2.2.2 Kernel

Description

The kernel layer is responsible for implementing the transfer of messages between the nodes and for delivering streaming data to the nodes.

For each node within the network,

the Network provides the kernel with a corresponding node object N. This object is equipped with methods that the kernel can invoke to deliver streaming tuples and incoming messages to the respective node.

Crucially, for each node, the kernel is responsible for constructing a network context object. This context object plays a pivotal role in the middleware's implementation of the RMI proxies provided to the node. The Network layer calls methods on this network context object to facilitate message transmission from the node to other nodes, and to support streaming synchronization.

Finally, it is important to note that both streaming data and messages are handled as generic objects by the kernel. This allows the kernel the flexibility to encapsulate these messages within its own message objects, enriching them with routing, timing, or any other important metadata.

2.2.3 OML

The last layer is the top layer of the system, the distributed Online Machine Learning library.

Pipeline

The computational component of this architecture is a pipeline. Each pipeline executes in a network. If i is the number of spokes and j is the number of hubs, a pipeline consists of i independent learners and j Parameter Servers. In each learner, a preprocessor is prepended in pipeline fashion.

The pipeline has the following tasks:

- To orchestrate the synchronization between local learners and the Parameter Server, various synchronization strategies have been implemented.

- To facilitate the flow of streaming input through the preprocessors and ultimately to the local learners. The pipeline decides whether to fit (train) or predict on a sample, based on the existence of a label. When training is required, the input can be organized into mini-batches, the size of which is configurable by the user for greater control over the learning process.

- To keep track of performance metrics, both for individual learners and across the entire system. This includes measures like accuracy, regret, and other relevant learning quality indicators.

- To provide a means of interaction with the environment through a dedicated control API, enabling tasks such as monitoring the execution progress and adjusting learning hyperparameters as needed.

Preprocessors

Preprocessors or data preprocessing are techniques performed on raw data to prepare it for a data processing procedure. They are placed at the start of the pipeline before the data is passed through the learner. In the current implementation have been implemented the following preprocessors:

- Feature filtering/selection

- Min/Max scaler

- Standard scaler

- Running mean and variance

- Polynomial features

Learning Algorithms

The implemented learning algorithms in the OMLDM are the following:

- Passive-Aggressive learners for binary and multi- class classification
- Online Support Vector Machine classifier
- Online Ridge Regression
- KMeans++ clustering
- "CVFDT enables the construction of Hoeffding trees for mixed attribute types (discrete and real-valued).
- Neural networks algorithms

Parameter Server

he core component orchestrating the distributed pipeline logic is the Parameter Server. In this setup, each worker has its own data stream, where it buffers data until it accumulates enough for a mini-batch. This mini-batch is used by the worker's learner to update its local model, and then discarded. The synchronization strategies are similar to the offline scenario, with the key difference that the workers have their entire portion of the dataset from the beginning. They use this local data to sample mini-batches for training. The current implemented synchronization strategies are:

- Total Asynchronous Parallel: The workers operate fully independently, without any coordination between them. This architecture prioritizes speed and performance, at the cost of the convergence of the learner.
- Bulk Synchronous Parallel: This synchronization protocol, emphasizes control over the communication and synchronization of the model, but the synchronization mechanisms may limit the throughput of the topology.

- Stale Synchronous Parallel: This synchronization model aims to balance the speed of asynchronous methods with the stability of synchronous ones.It offers a more balanced approach. While it might not achieve the absolute peak performance of either extreme, it provides a compromise for scenarios where both speed and stability are important considerations.

- Elastic Averaging: This strategy explores the parameter space more freely than traditional synchronous or asynchronous methods.

- Functional Dynamic Averaging: This strategy minimizes communication between nodes, thereby maximizing throughput. It also aims to improve model performance by updating the global model only when significant changes have occurred in the local models.

2.2.4 APIS

Due to the the Multiple Layers composing the OMLDM, there are multiple APIs(Control, Network/Middleware and Request API) to interact not only with each component, but with the system as a whole.

The Control API contains serializable Objects that are transmitted between the three layers.The system's components are effectively decoupled by separating the Control API from the core layers, which promotes a more flexible and maintainable design. With this method, individual layer development and modifications are possible without compromising the integrity of the entire system.

The user can interact with with the whole system after it has been initialized through the Request API that accepts requests in a specific JSON format. The user can request the following:

- Create: Instantiates the OML workers and Parameter Servers, sets the preprocessors, pipeline, etc.

- Query: Generates a comprehensive summary of all OML algorithms currently running.

- Delete: Deletes the current running processes.

The Middleware/Network API not only enables the instantiation of the abstract Network and Nodes that were mentioned above. It empowers users to analyze incoming data and to classify it into different categories (Message from another Spoke, Tuple to be processed, or Query to be answered). In the case of a Query or a Tuple, there may be the possibility to answer back to the Querier and how to handle this event is given as an option from the API.

The figure 2.7 is the Class Diagram of the Network layer of the OMLDM and in the figure 2.8 are the 2 core classes of the Network layer, that are needed to implement the kernel of the OMLDM. The generic Wrapper is the Node of the Network and the Network interface defines the methods that need to be implemented.

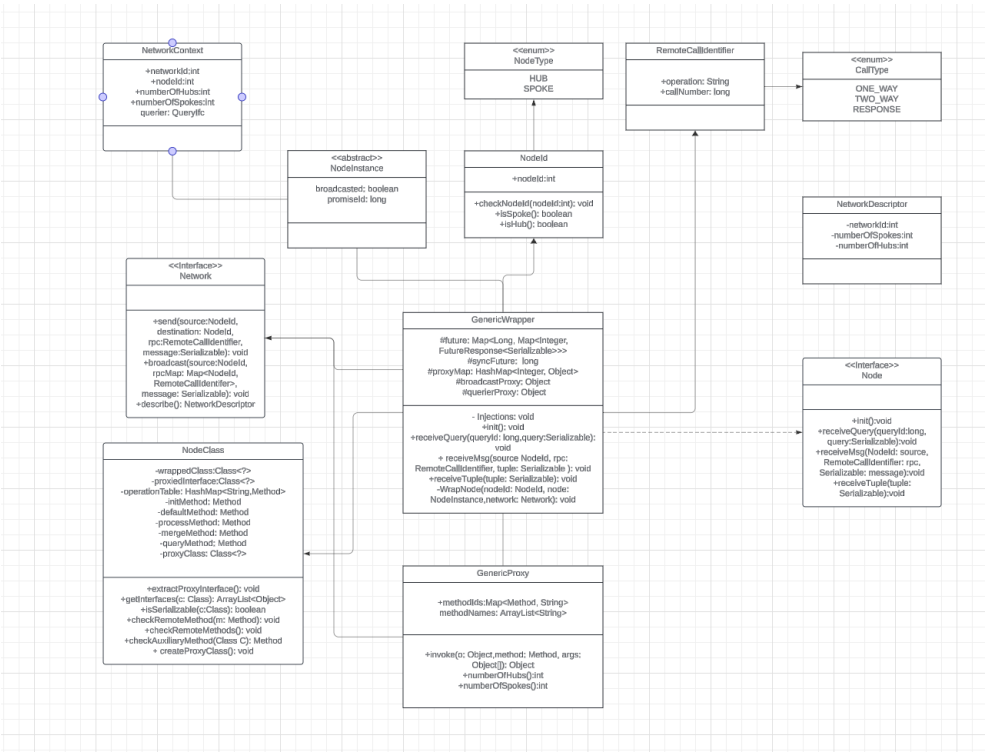

Figure 2.7: Class Diagram of Network layer

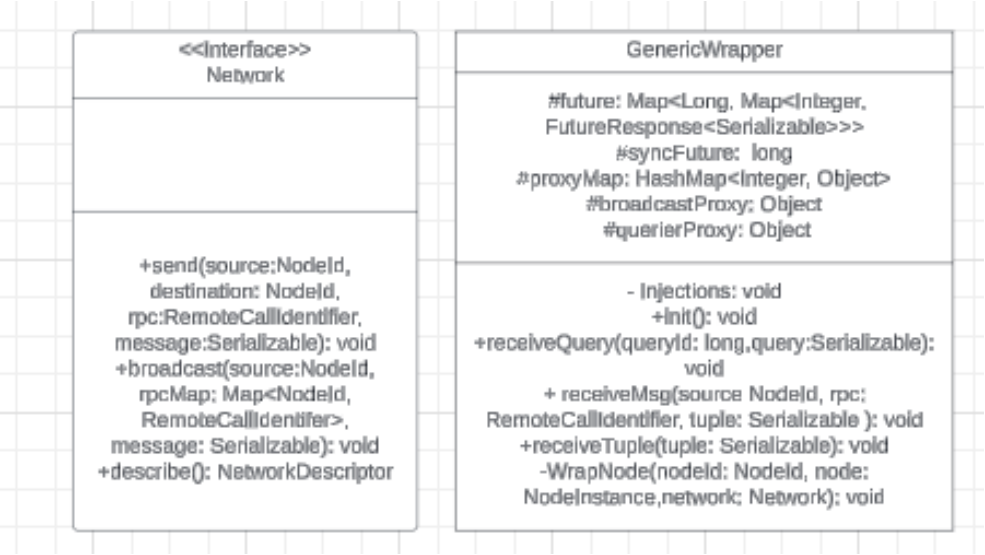

Figure 2.8: Classes needed for kernel implementation

Chapter 3

Framework Utilization

3.1 Introduction

This chapter will explain the various frameworks that were used to familiarize ourselves with the OMLDM API and implement the kernel of the OMLDM in Akka.

3.2 Akka

3.2.1 Overview

Akka [1] is a toolkit that makes it easier to create concurrent, distributed, and fault-tolerant Java Virtual Machine (JVM) applications by utilizing the actor model.

The main key benefits it offers are:

- Concurrent and Distributed: Akka's actor-based foundation supports concurrent execution without the need for explicit low-level synchronization mechanisms like locks or atomics. It extends this concurrency model, without differentiating local and distributed environments, allowing components to communicate and collaborate transparently, even when residing on different machines.

- Fault Tolerance: The supervision hierarchies of Akka offer an in-

built method for managing errors and preserving system stability. In addition, on watching their child actors' health, actors can also take remedial action when necessary, such as restarting malfunctioning components, freeing resources when an actor restarts or reporting issues to supervisors.

- Scalability: The actor model's inherent message-driven nature makes Akka systems highly responsive to changes in workload. Akka clusters can dynamically scale up or down by adding or removing nodes as needed, ensuring that the system can handle varying levels of demand efficiently.

- Reactive: Akka aligns well with the principles of reactive systems, which emphasize responsiveness, resilience, elasticity, and message-driven communication.

3.2.2 Actors

The Actor is the basic component of Akka and as described in the previous chapter it is a container for a Mailbox, Behavior and an address.In Akka they also contain Supervisor strategies and references to child actors. To create a system, actors are also capable to create other actors (child actors) to delegate important tasks or to break the functionality of a system to several components.
In Akka, an actor's unique address serves as more than just an identifier. It encapsulates crucial information, including its parent actors, the actor system it belongs to, its host machine in distributed scenarios, and its network listening port. This rich address structure facilitates seamless communication, location transparency, and effective management within the actor system. Following are two examples of an actor's address.

- local: akka://MyActorSystem/user/myActor

- cluster: akka://RemoteActorSystem@192.168.1.100:2552 /user/remoteActor

In Akka, an actor's internal state and behavior are shielded by an Actor Reference, which acts as a proxy for communication. This ensures safe interaction between actors by preventing direct access to their internal state. However, a fundamental question arises: who initiates the entire system by creating the first actor?

The answer lies in the guardian actor, denoted by "/", a special entity created at the top of the hierarchy when an actor system is instantiated. This heavyweight structure allocates threads for actor execution and delegates the crucial task of bootstrapping the application to the guardian actor. It does so by spawning essential subsystems as its children, setting the environment for the rest of the application to unfold. These subsystems are:

1) /user : The guardian for all user-created top-level actors. When you use the actor system to create an actor, it's placed under this guardian.

2) /system : Oversees system-created top-level actors. These include essential components like logging listeners or actors automatically deployed at system startup through configuration.

3) /dead Letters : The dead letter actor acts as a safety net, capturing messages sent to non-existent or terminated actors. It operates on a best-effort basis, so some messages might still be lost even within the local JVM.

4) /temp : A guardian for ephemeral, system-created actors. These actors have short lifespans and are typically employed for specific, temporary tasks.

5) /remote : An artificial path under which actors with remote supervisors reside.

3.2.3 Akka Cluster

Akka Remoting, another part of the Akka toolkit that enables actor-to-actor communication, establishes a peer-to-peer communication framework for connecting actor systems, serving as the basis for Akka Clustering. Its design is guided by two key principles: symmetric communication, enabling bidirectional connections between systems, and symmetric connection roles, where any system can both initiate and accept connections.

In Akka Cluster, a node, identified by a hostname:port:uid tuple, represents a logical member within the distributed actor system. The Cluster Membership Service manages the nodes that are in the cluster, while a designated leader node oversees cluster convergence and membership state transitions.

Vector Clock

Akka Cluster employs vector clocks to establish partial ordering and detect causality violations in its distributed environment. This data structure is instrumental in reconciling and merging differing cluster states during gossip exchanges. Each update to the cluster state is paired with a corresponding update to the vector clock, contributing to the overall system's consistency.

Cluster Membership

Akka Cluster provides a fault-tolerant decentralized peer-to-peer based Cluster Membership Service with no single point of failure or single point of bottleneck. It does this using gossip protocols and an automatic failure detector.

The cluster membership state, is implemented as a specialized CRDT, ensures eventual consistency across all nodes. Nodes within the cluster can transition through various states, including:

- joining', during cluster entry

- 'weakly up', during network partitions, if enabled
- 'up', normal operation
- 'preparing for shutdown/ready for shutdown', optional pre-shutdown states
- 'leaving/exiting', graceful removal
- 'down', unreachable but not yet removed
- 'removed', tombstone state

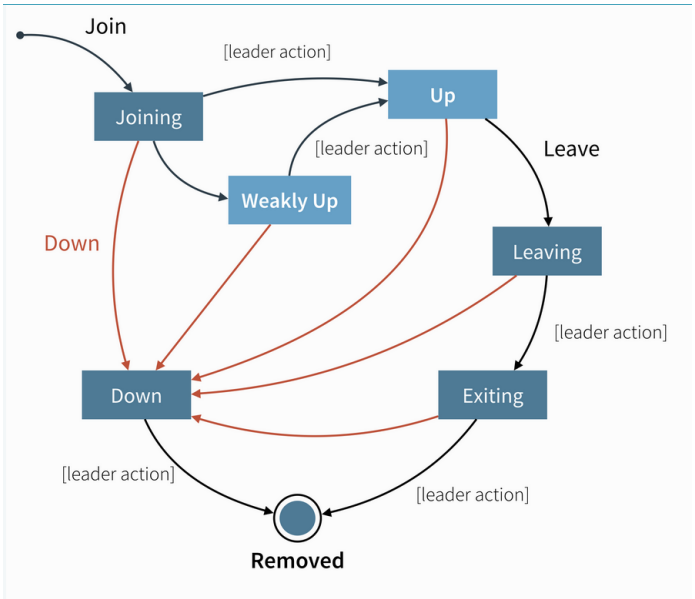

Figure 3.1: Membership lifecycle state

Leader

The leader in an Akka Cluster is responsible for confirming state changes when convergence is achieved, ensuring a consistent view of the cluster across all nodes. The leader's identity is unambiguously determined after convergence, and any node may potentially take on this role based on the cluster's current composition.

Most leader actions require convergence to guarantee agreement among nodes about the cluster's state and to ensure changes originate from a single source. However, in situations like network partitions where convergence is impossible due to unreachable nodes, the cluster might become split-brained, with each partition potentially having its own perceived leader. In these scenarios, leader actions must be carefully designed to ensure eventual consistency, even with concurrent leaders. A crucial example is the ability to down nodes, manually or automatically, to restore convergence in split-brain situations.

Gossip Convergence

Gossip convergence in Akka Cluster ensures consistent state across all nodes by verifying that every node has observed the current cluster state. This is achieved through the exchange of gossip messages containing a "seen set" of nodes that have witnessed the state. Convergence is blocked if any nodes are unreachable, impacting the leader's ability to manage cluster membership, but not affecting existing applications.

Gossip protocol

Akka Cluster employs a modified push-pull gossip protocol to optimize the spread of cluster information. It uses a shared state and vector clock for versioning, reducing unnecessary data transmission by only pushing the actual state when required. Gossip frequency increases during initial dissemination to expedite convergence. Node selection for gossip is random but biased towards those likely having older state versions, further enhancing convergence speed in the later phases.

Failure Detector

The Akka Cluster's failure detector identifies unreachable nodes by having a subset of nodes monitor each other. When a node is deemed unreachable, this information is propagated throughout the cluster via gossip, requiring only one node's detection for cluster-wide recognition.

Similarly, when a previously unreachable node is detected as reachable by all its monitoring nodes, the cluster updates its status accordingly. However, if system messages consistently fail to reach a node, it's quarantined, preventing recovery from the unreachable state.

Seed Nodes

Seed nodes in Akka Cluster act as initial contact points for new nodes, facilitating their entry into the cluster. Upon startup, a new node broadcasts a message to all seed nodes and then sends a join command to the first one that responds. The seed nodes configuration is crucial during the joining process but doesn't impact the operational cluster. New members can actually send the join command to any existing member, not exclusively to seed nodes.

Serialization

Serialization is a technique that translates an object state in a format that can be transmitted and, at a later time, reconstructed (known as deserialization) to create an identical clone of the original object

Though necessary for object transmission between systems, it can become a performance bottleneck in actor communication due to the inherent delays in converting and reconstructing object states. Akka optimizes this process by utilizing direct references for local actors within the same JVM and offering various serialization options, including Jackson, Google Protobuf, and custom serializers. The use of the Java serializer is deprecated in Akka, because Akka prioritizes security and the Java serializer is very vulnerable [11] and slow.

3.2.4 Akka Streams

Overview

Akka Streams, a powerful framework within Akka, excels at building reactive, high-performance data processing pipelines. It leverages the Ac-

tor Model's asynchronous message passing and lightweight actors to handle data streams efficiently and scalably. By implementing the Reactive Streams specification, Akka Streams ensures interoperability with other libraries and standardized backpressure handling. This prevents system overload while allowing the construction of complex pipelines where data flows through various stages for transformation or action.

Figure 3.2: Akka Streams Structure

The asynchronous and non-blocking nature of Akka Streams, along with its non-blocking IO capabilities, results in highly efficient system resource utilization, allowing concurrent processing of multiple streams. Built-in backpressure mechanisms further prevent system instability and data loss by ensuring fast producers don't overwhelm slow consumers.

A stream is composed of three elements:

- Source is the entry point of the stream where the data that is going to be processed is stored.
- Flow is the main processing building block of the stream. It has one input and one output.
- Sink is the execution environment of the stream. It is the terminal operation that triggers the computations in the entire Flow.

Flows can be connected in a pipeline fashion to each other to perform separate operations one after the other and each flow can be parallelized.

Backpressure

The backpressure mechanism is among Akka Streams' most significant features.It is an essential mechanism that serves as a feedback loop

to keep faster producers from overwhelming slower consumers, ensuring smooth and efficient data flow. Backpressure is essentially a means by which downstream elements in a stream inform upstream elements to reduce or stop producing data when they are having difficulty keeping up.

To handle temporary speed mismatches between stages, Akka Streams uses buffers. If a downstream stage is slow, the upstream stage can buffer a certain number of elements. However, if the buffer fills up, overflow strategies come into play. These strategies might include dropping elements, failing the stream, or applying other configurable behaviors.

To summarize, backpressure helps to prevent memory exhaustion and overuse of system resources and makes stream processing pipelines more stable and resilient to varying load conditions.

3.2.5 Alpakka

In the Akka ecosystem, Akka Alpakka is a potent library that makes it easier to integrate Akka Streams with a variety of outside systems and technologies. It offers a set of connectors that facilitate stream-oriented and reactive communication with different databases, message brokers, APIs, and data sources. These connectors are used as the source of a stream, thus changing the outside system does not impact the logic of the akka system.

3.3 Apache Kafka

Apache Kafka is a distributed, and scalable streaming platform designed to handle real-time data feeds between processes, applications and systems. To be more specific, by combining two messaging methods queuing and publish-subscribe, it gets the best of two worlds.

The topic is central to the architecture of Apache Kafka. It can be visualized as a pipe that enables consumers and publishers to interact with specific data streams. It is represented as a sequence of records,

where each record contains a key, a value and a timestamp.

Topics do not need a specific relationship between the consumer and the producer. Producers can write into a topic, without any knowledge about the consumer in the other side of the topic. Because of this, Kafka can be used for system to system communication to evade tight coupling, meaning that by using kafka in between two systems, they do not need to be aware of each other, but only aware of the data that is sent.

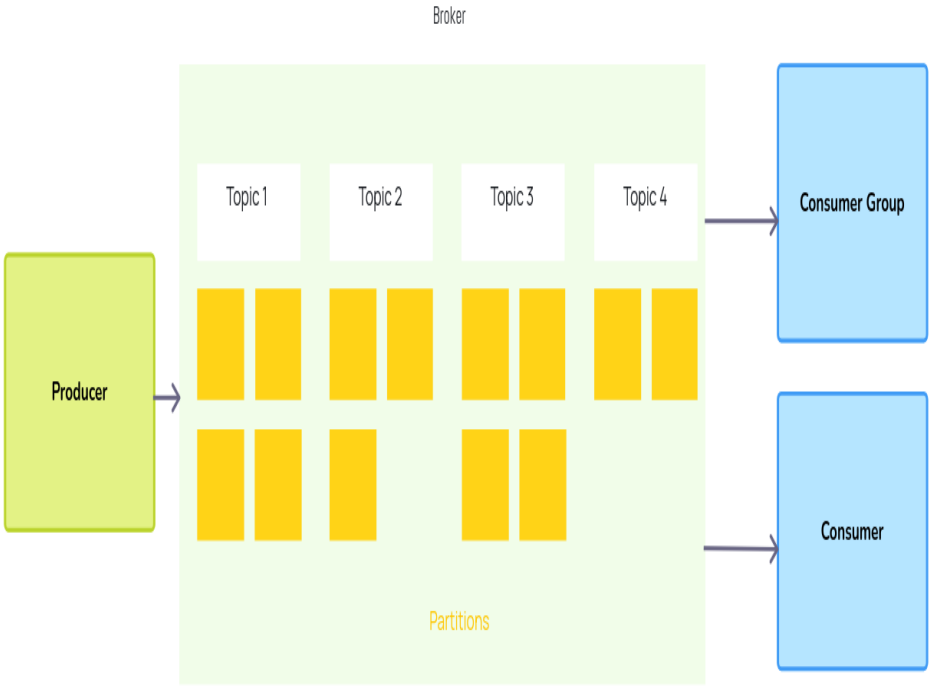

Figure 3.3: Akka Streams Structure

3.3.1 Docker

Docker is an open source platform that enables the development, shipment and execution of applications wrapped in containers. A container can be thought of as a self-contained and isolated environment that wraps an application and it's dependencies, separating the application from the host environment.

The Dockerfile is responsible for configuring the container's environment. This file allows customization, such as specifying the base image to begin with, installing required software and setting up network accessibility between the host system and the container.

Chapter 4

Implementation

4.1 Introduction

This chapter describes the implementation of the kernel of the omldm in Akka as well as the Java Thread implementation and highlights the key decisions and adaptations made during this stage.

4.2 Java Implementation

To familiarize ourselves with the Bipartite/Network component and test/debug some issues of the OMLDM component, a simple local kernel in Java was implemented using Java's Thread library. The two core objects were:

- Node: Is responsible for running an instance of either a worker or a hub within the OMLDM component. Each Node contains a queue that stores the messages sent from other Nodes. It reads three types of network events ("Query", "Message", "Tuple") and executes the appropriate response for each.

- Network: An implementation of the abstract Network class within the Middleware is responsible for instantiating and configuring nodes, as well as transmitting messages. Due to the fact that this implementation operates within a local JVM environment, the process

of sending a message is simplified to just adding an object to the appropriate queue.

The simple architecture of the system, in addition to the use of Java's Callable threads (Callable is a thread that is able to return error messages), enabled to easily pinpoint the errors in the OMLDM system.

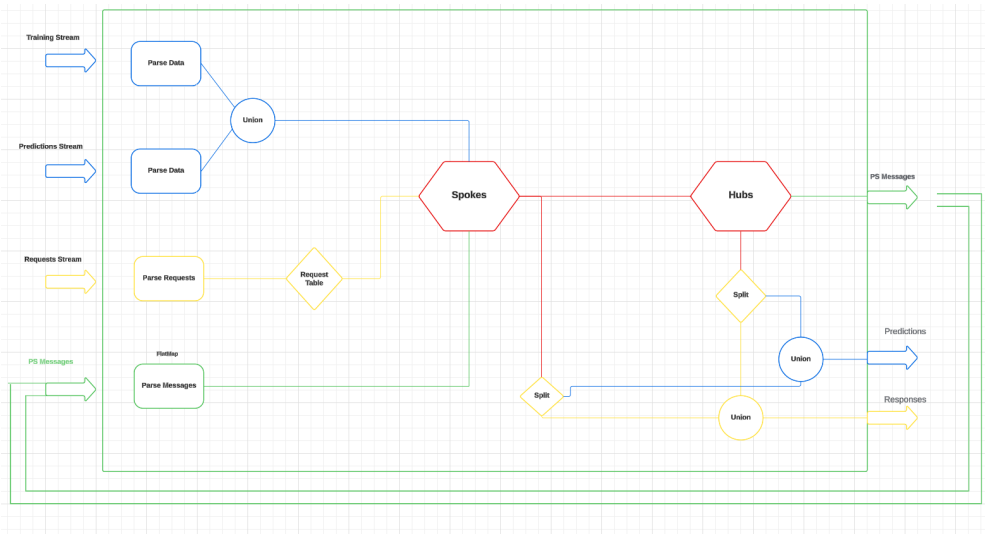

Figure 4.1: Block Diagram of Java Thread Implementation

4.3 Akka Implementation

Once the implementation and testing of the OMLDM Kernel in Java were successfully concluded, the subsequent phase involved implementing the OMLDM Kernel in Akka.

An overview of the system is: there is one NodeManager that is responsible for the state of the cluster. Each JVM (and host machine) can have 1 or more Nodes to enter the cluster. The Nodes communicate via tcp channels. Each Node reads data to be processed from a Kafka topic and may answer back in another Kafka topic. Finally, a client can enter the cluster to send requests to the system.

The system can be broken down to 2 major components:Node and Node Manager.

Node Manager

The Node Manager acts as the central control point within the kernel, overseeing the formation and operation of the node network. Its responsibilities include enabling other nodes to establish connections within the cluster, orchestrating the instantiation of the node network, managing client requests that trigger the initialization and message acceptance of nodes, and handling the graceful shutdown of each individual node.

As the seed node, the Node Manager serves as the first point of contact in the cluster and has a predefined hostname. To join the cluster network, a node that wants to start operating will first send a message to the Node Manager, signaling its intention to become part of the cluster network.

Node

The Akka Node is the implementation of the abstract Node of the Network Component of the OMLDM. Each Node must have 1 of the 2 roles:hub or spoke.

The Node processes 3 Different streams:

- Messages: These are data sent from other Nodes.
- Query: Requests or Data that is used to get additional information of the OML process.
- Tuple: Data that is used to train the OML model.

The Actor stores every message that it receives in a Mailbox and processes each message sequentially and in the order that it was received. The Tuple stream high data throughput may create the following problem: If a Node is instantiated and then a lot of tuples are sent before the synchronization of the OMLDM Network, the Mailbox will be filled with Tuples that will be processed without answering the messages that have been sent.

To solve this problem, the following method was implemented: The client does not send Tuples freely, but every time, it sends a batch of tuples and needs an answer before sending another batch. This way a "gap" is created so if any other message is received it will be processed before the next Tuple batch.

The Nodes receive a request to begin operation by the Node Manager. Upon receiving this message, each Node establishes its own instance of the Middleware Network. This Network then broadcasts its presence to the other Nodes. However, a timing issue can occur: if a Node is instructed to start its worker after another Node has already done so, it might receive a synchronization message from the already-active Network before its own worker is ready. To resolve this, the Node queues all incoming messages (except the initial start request) until its worker is fully operational, at which point it processes them.

To read from the kafka topic, 2 extra actors are created for each Node. One actor is responsible for creating an Akka Stream to read data from a source (the Kafka topic in this instance). The other actor's purpose is to store and subsequently transmit the Tuples to the Node. Both actors are instantiated when the Node receives the request to begin the operation. Additionally, by utilizing the alpakka library, the input of the Stream is encapsulated as the Source inside the stream, consequently severing the input from the rest of the application.

Every Node operates under the supervision of a designated parent (the Parent Node). This Actor has the critical responsibility of monitoring the Node's activity and, should the Node experience a crash or failure, initiating a restart process to bring it back online.

Client

To send the request to the Node Manager to initiate the work progress and to send queries to the workers, a simple Client Actor was implemented. The actor reads input from a kafka topic (Either requests or Data to be queried) and sends the appropriate message to the Node

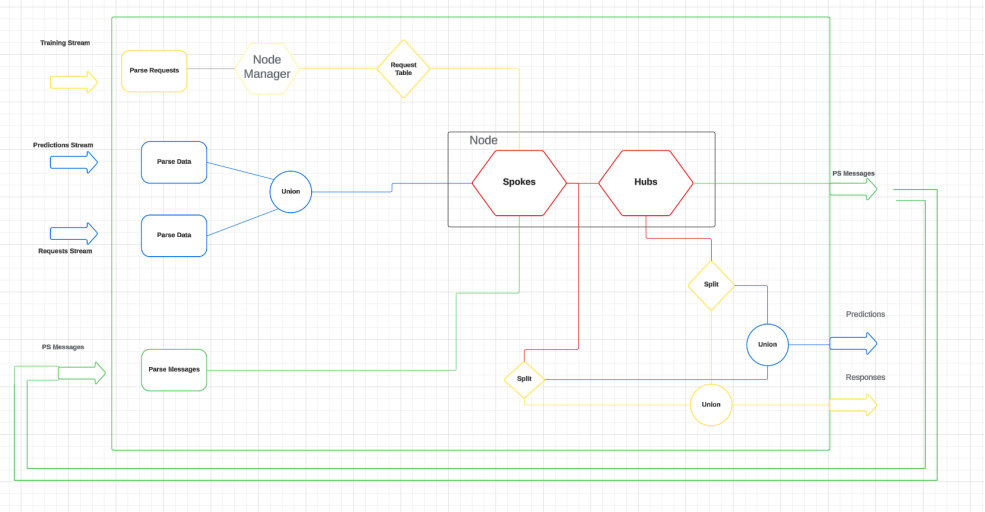

Figure 4.2: Block Diagram of Akka Implementation

Manager.

CLI

To ease the access and usage of the system, a simple CLI was implemented to configure and initiate the actors of the system. It provides the following options:

- Hostname (-h or –hostname): Specifies the hostname of the actor.
- Port (-p or –port): Sets the port number for connections.
- NodeManager (-n or –nodemanager): Creates a NodeManager instance.
- Client (-c or –Client): Creates a Client instance.
- Spokes (-s or –spokes): Defines the number of spokes to be created.
- Hubs (-u or –hubs): Defines the number of hubs to be created.

The hostname and port options are mandatory for the system to run.

Chapter 5

Results

5.1 Introduction

This chapter outlines the performance results of the OMLDM kernels written in Java and Akka. There are 2 scenarios: The first one is comparing the local, running on one machine, implementations of the two kernels and the second scenario is running the Akka kernel in a clustered environment.

5.1.1 Infrastructure Description

Before analyzing the performance results, a detailed overview of the system infrastructure and deployment process is provided to establish a clear context for the evaluation.

Our experiments were conducted on four virtual machines, each equipped with **8 CPU cores**, 16 GB of RAM and 30 GB HDD. Docker was utilized to deploy our applications and the necessary infrastructure components, including Kafka and Zookeeper.

The dataset that was used to train the OML algorithm was the EMNIST Dataset [4], a large dataset of handwritten characters, designed to be a more challenging and realistic benchmark for machine learning models than the original MNIST dataset. The digits image set, 10 different numbers/classes, and each tuple had 28×28=784 features. Because

the focus of this dissertation is not the correct training/analysis of the algorithm but the analysis of the throughput of the system and it's characteristics, we enlarged the dataset by duplicating the file to a total of $3.9 \approx 4$ GB's on Disk with a total of 1920000 elements.

There were 2 scenarios:

1 Local Scenario: We run the Java Threads and Akka OMLDM Kernels locally, to compare the difference of plain java threads vs Akka actors who run on top of java threads and compared the performance of each system for a variable number of workers: 2,3,4,6,15.

2 Distributed Scenario: We run the Akka OMLDM in a cluster, for a set number of 15 workers, while changing the number of machines and the distribution of workers in each machine.

In each run the number of hubs was 1 and the system was initialized after the dataset was imported into Kafka.

Each instantiated worker of the OMLDM (Both Akka and Java Threads) uses two threads, one for processing and one for reading from the Kafka topic. For each scenario, we run 2 types of Parameter Servers, TAP and SSP.The data was distributed evenly to every worker.

5.2 Results

5.2.1 Local

Tuples Proccesed

The figure 5.1 depicts the relationship between the number of workers and the tuples processed per second in our system. There are three key observations that can be made:

1) For each system, there is a non-linear relationship between the workers and the tuples processed per second. This can be attributed to a multitude of possible factors like: the access to the

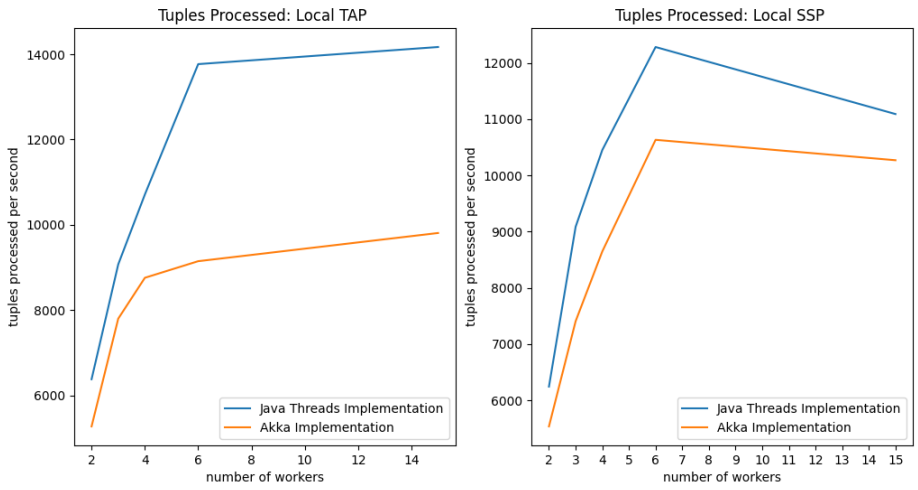

Figure 5.1: Local Experiment: Tuples Processed

HDD from multiple threads, memory constraints and in the case of SSP (Stale Synchronous Parallel) the wait of the response from the Hub for each bath processed. For the SSP specifically it is evident that it introduces an overhead when we observe the tuples processed per second from the 6 workers to 15, the throughput lowers.

2) There was a total of 8 CPU Cores to utilize, and each worker needed 2. When we increase the workers more than 3, the number of cores cannot simultaneously run all the threads of the spokes and the hub, thus the performance increase lowers, and it reaches the limit when we increase the performance more than 8 workers. This is due to the fact that even though at the start each worker needs two threads when there are no messages in Kafka, the thread sleeps thus all the cores run the process thread of the application.

3) The performance of the Java Threads implementation is, as expected, faster than Akka and that's because Akka runs on top of Java Threads and each actor is not a thread but a Callable to a thread pool. Actors are assigned to threads from a pool to handle

incoming messages. Once an actor finishes processing a message, the thread can be reused for another actor. This process introduces an overhead to the execution of each message.

Messages Hub Received

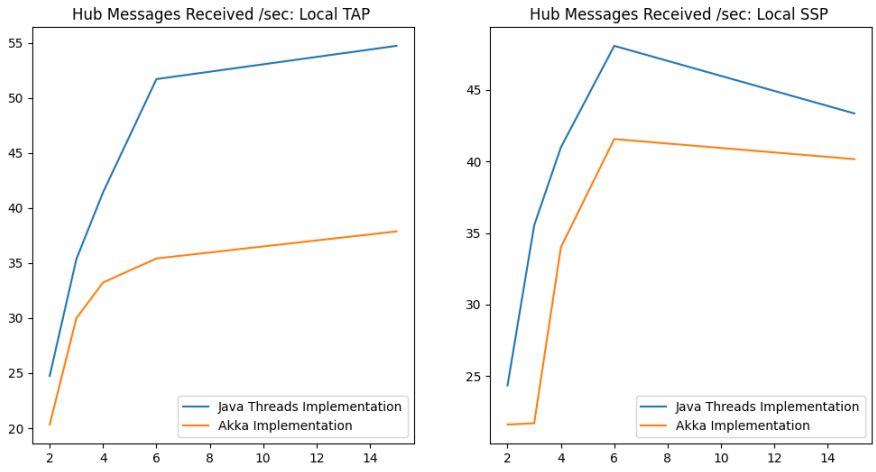

Figure 5.2: Local Experiment: Messages Received

The figure 5.2 depicts the relationship between the number of workers and the messages the Hub received per second. As expected, when we increase the number of Spokes, the number of total messages the Hub receives, increases (The Hub receives a message every time each Spoke finishes processing a mini batch of Tuples). From the figure 5.2, we can derive the same observations that were made from the figure 5.1.

Messages each Spoke Received

The figure 5.3 depicts the relationship between the number of workers and the average messages each Spoke received per second. Even though there is a decrease of the messages each Spoke received, the total messages received increase when the number of workers increase.

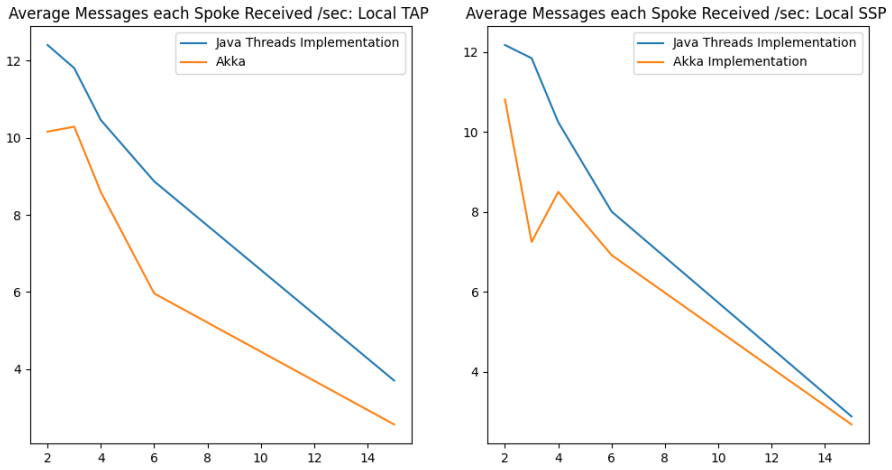

Figure 5.3: Messages Received

5.2.2 Cluster

For our cluster experiment, we deployed a total of 15 workers across 4 machines, varying the number of machines used in each trial and distributed the Spokes evenly. To isolate the impact of Hub Node placement, we conducted two distinct experiments:

1) the Hub Node was exclusively hosted on a dedicated machine, operating independently from any Spokes.

2) In contrast, this series involved the Hub Node sharing a machine with a subset of Spokes.

We run a total of 7 combination Spoke placements for each type of Parameter Server(the first Machine always run the Hub Node): 0-15-0-0, 0-7-8-0, 0-5-5-5, 3-4-4-4, 5-5-5-0, 7-8-0-0, 15-0-0-0.

Tuples Processed

From the figure 5.4, it can be observed that by increasing the number of machines and spreading to all available machines the spokes evenly the throughput is maximized. Also in the case of TAP, the placement of the

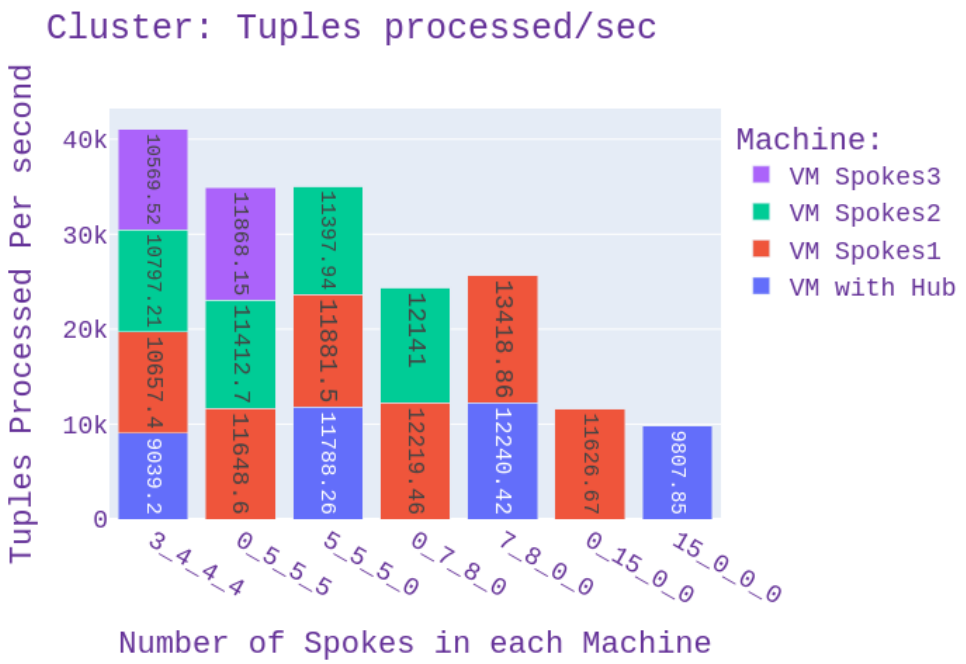

Figure 5.4: Cluster experiments TAP: Tuples Processed

Hub does not seem to impact substantially the performance of the system as a whole. Again, it can be observed that 8 workers is the maximum throughput we can get from each system, and instantiating more than 8 workers introduces overhead and slows down the system.

From the figure 5.5, it can be observed that the total throughput of each experiment with the SSP Parameter Server, is lower than the corresponding experiment with the TAP Parameter Server. Also the we can derive the same observations that were made for the TAP experiments.

Messages each Spoke Received

The figure 5.6 shows the average messages each Hub received per second and the average messages each Spoke received of each machine per second for the TAP Parameter Server. From the figure it can be observed that the existence of the Hub may slightly increase the messages that the Hub receives per second and it can be due to the fact that some of the Spokes being on the same Machine as the Hub, they do not send the messages

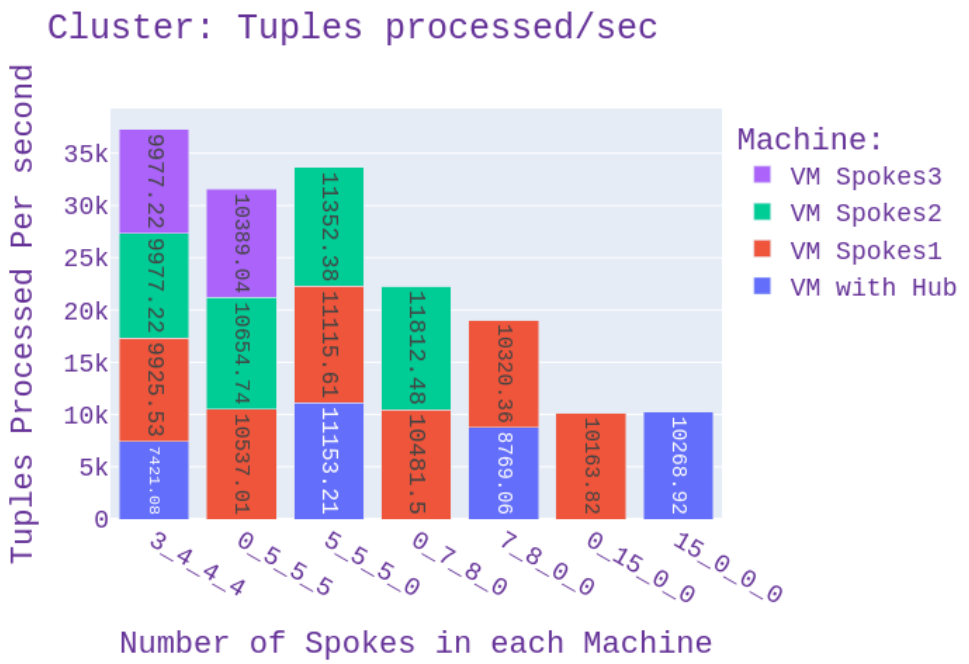

Figure 5.5: Cluster experiments SSP: Tuples Processed

via tcp, but they are passed through the Memory.

The figure 5.7 shows the average messages each Hub received per second and the average messages each Spoke received of each machine per second for the SSP Parameter Server. Again the same observations can be made as above and the SSP Hubs process less messages than the TAP Hubs.

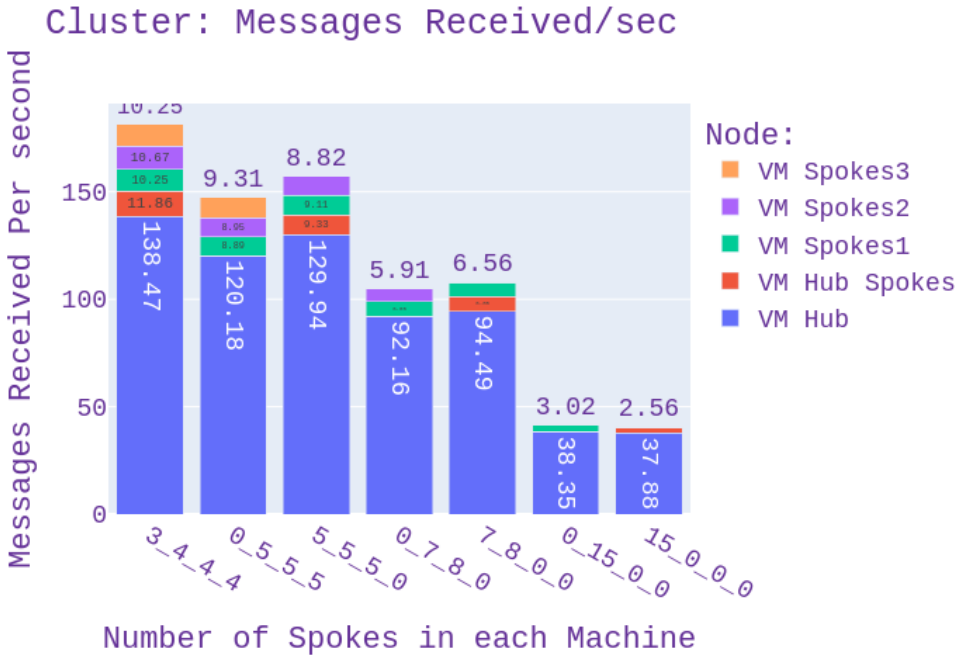

Figure 5.6: Cluster experiments TAP: Messages Received

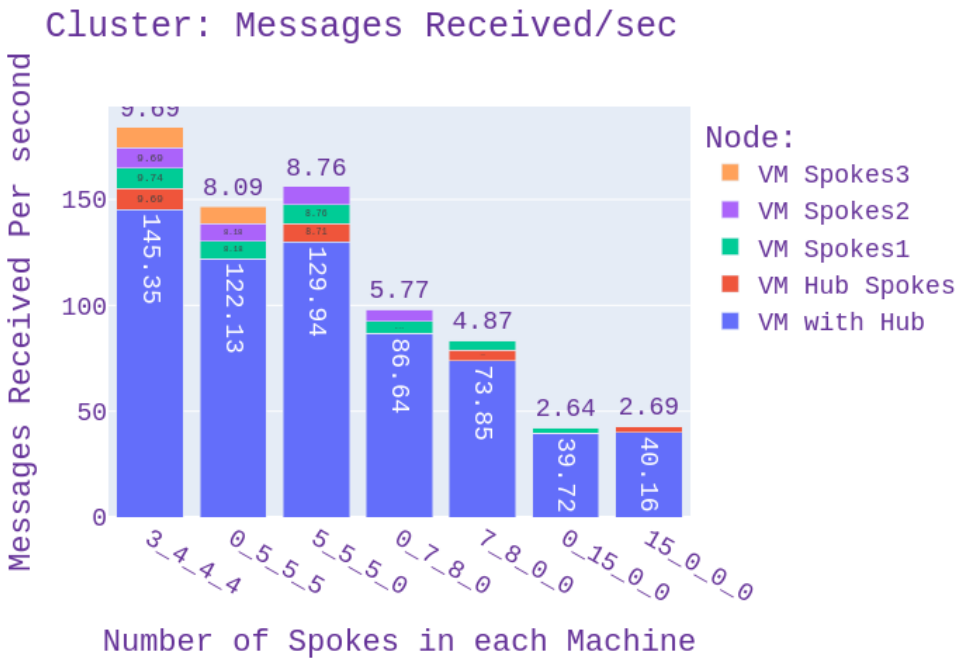

Figure 5.7: Cluster experiments SSP: Messages Received

Chapter 6

Conclusion and Future Work

For this thesis was implemented a distributed communication kernel for the OMLDM system within the Akka framework. We conducted a comprehensive analysis of the performance overhead introduced by Akka by comparing it to plain Java Threads and explored the behavior of the kernel under both asynchronous and synchronous workloads.

As of this moment of writing Akka supports only at most once or at least once message delivery(it is rare for a message to be lost but possible), thus it does not secure that a message will be delivered. The API for exactly once delivery(Reliable Delivery) is flagged as may change. When the API is ready for developing applications, it would be interesting to implement the kernel and check the performance of the at most once vs. reliable delivery. Also there are other frameworks for building distributed streaming kernels like Apache Spark that could implement the OMLDM kernel to compare the performance.

Bibliography

[1] *Actors Akka documentation*. URL: https://doc.akka.io/docs/akka/current/typed.

[2] *Apache Kafka*. URL: https://kafka.apache.org/.

[3] E. G. Coffman, M. Elphick, and A. Shoshani. "System Deadlocks". In: *ACM Computing Surveys* 3.2 (June 1971), pp. 67–78. DOI: 10.1145/356586.356588. URL: https://doi.org/10.1145/356586.356588.

[4] Gregory Cohen et al. *EMNIST: an extension of MNIST to handwritten letters*. Feb. 2017. URL: https://arxiv.org/abs/1702.05373v1.

[5] *Docker: Accelerated Container Application Development*. July 2024. URL: https://www.docker.com/.

[6] Carl Hewitt, Peter Bishop, and Richard Steiger. "A universal modular ACTOR formalism for artificial intelligence". In: *International Joint Conference on Artificial Intelligence* (Aug. 1973), pp. 235–245. URL: http://ijcai.org/Proceedings/73/Papers/027B.pdf.

[7] Qirong Ho et al. "More effective distributed ML via a stale synchronous parallel parameter server." In: *PubMed* 2013 (Jan. 2013), pp. 1223–1231. URL: https://pubmed.ncbi.nlm.nih.gov/25400488.

[8] Vissarion Berthold Konidaris. “Extreme-Scale Online Machine Learning On Stream Processing Platforms”. MA thesis. Technical University Crete, 2022. URL: https://doi.org/10.26233/heallink.tuc.92838.

[9] Dirk Merkel. “Docker: lightweight Linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (Mar. 2014), pp. 2–. URL: https://dl.acm.org/citation.cfm?id=2600239.2600241.

[10] Fred B. Schneider and Gregory R. Andrews. *Concepts for concurrent programming*. Jan. 1986, pp. 669–716. DOI: 10.1007/bfb0027049. URL: https://doi.org/10.1007/bfb0027049.

[11] *The perils of Java deserialization*. URL: https://community.microfocus.com/cyberres/fortify/f/discussions/317555/the-perils-of-java-deserialization.

[12] *Thread Objects (The JavaTM Tutorials Concurrency)*. URL: https://docs.oracle.com/javase/tutorial/essential/concurrency/threads.html.

[13] Roger Riggs nn Wollrath and Jim Waldo. “A Distributed Object Model for the Java System”. In: *USENIX* 9 (June 1996). URL: https://pdos.csail.mit.edu/6.824/papers/waldo-rmi.pdf.

[14] Wei Zhang et al. “Staleness-aware async-SGD for distributed deep learning”. In: *International Joint Conference on Artificial Intelligence* (July 2016), pp. 2350–2356. URL: http://dblp.uni-trier.de/db/conf/ijcai/ijcai2016.html#ZhangGL016.