CPSC416 Capstone Project

Distributed Load Balancer System Design

Vincent Kohm

The Problem

In the domain of Machine Learning, tasks that demand considerable computational resources are prevalent, primarily due to the intricacies of model architectures, the large nature of datasets, or the inherent complexity of certain computational operations. Among the array of challenging endeavors in Machine Learning, tasks such as hyperparameter tuning, large-scale data processing, the application of ensemble methods, and the process of feature extraction and selection are particularly noteworthy. To elucidate this concept, hyperparameter tuning in Machine Learning models serves as an exemplary case. Hyperparameters, which can be analogized to adjustable controls for fine-tuning a model's performance, are integral components of most Machine Learning models. In scenarios where multiple hyperparameters exist, the task of identifying their optimal configurations becomes increasingly arduous, more so in the context of extensive datasets. The technique of grid search cross-validation is commonly employed for ascertaining the most suitable values for these hyperparameters. This method entails dividing the training segment of a dataset into 'n' partitions, where 'n-1' partitions are utilized for training purposes and a singular partition is designated for validation. This procedure is iteratively executed across all possible combinations of these partitions, thereby ensuring a comprehensive evaluation. For instance, in a scenario where a dataset is segmented into ten subsets, the cross-validation process would encompass ten distinct combinations of these subsets. Considering a Machine Learning model with two hyperparameters, each subjected to an exploration of ten possible values, the process culminates in 100 unique hyperparameter configurations. When coupled with 10-fold cross-validation, this translates into 1000 individual validations.

This example underscores that the intricacy of hyperparameter tuning is amplified in models characterized by a multitude of hyperparameters and an expansive range of potential values, particularly when coupled with extensive datasets. It is imperative to acknowledge that hyperparameter tuning represents merely a single facet of the broader spectrum of activities encompassing the training and deployment of a model. Other facets, too, contribute to the overarching complexity inherent in the entire process, especially when executed on a monolithic architecture.

In the course of my academic pursuits, I developed a foundational understanding of Machine Learning models and their applications through my engagement with the Applied Machine Learning course (CPSC330) at the University of British Columbia. This course placed a significant emphasis on the practical application of machine learning tools, encompassing a spectrum of topics including data cleaning, feature extraction, both supervised and unsupervised learning techniques, the establishment of reproducible workflows, and the effective communication of results. It was within this academic context that I became aware of the numerous complexities inherent in the handling of Machine Learning models. These challenges, particularly those related to the scale and efficiency of machine learning operations, are further elucidated in scholarly works such as "Large Scale Distributed Deep Networks" and "Machine Learning Load Balancing Techniques in Cloud Computing: A Review"[LINK],[LINK]. These papers offer a comprehensive exploration of the intricacies and technological hurdles prevalent in the field of Machine Learning, especially in the context of large-scale implementations and cloud computing environments.

During another course I followed at the University of British Columbia I had the opportunity to dive into the spectrum of Distributed Systems where I got acquainted with many relevant aspects such as replication, concurrency, failure scenarios, key-value stores, sharding, consensus protocols such as Raft or Paxos, and many more. While exploring the realm of distributed systems it crossed my mind that using a distributed architecture might help trivializing the many intricacies of deploying a Machine Learning model. Employing a distributed architecture in the deployment of machine learning models offers a substantial reduction in the complexities associated with such deployments, primarily through the dispersion of computational loads across a network of interconnected nodes. This methodology aligns with the foundational principles of distributed systems, which involve orchestrating multiple computational units to collaboratively execute specific tasks – in this context, the deployment and functional operation of machine learning models.

The advantages of a distributed architecture are diverse. Key among these is the enhancement of scalability, enabling effective handling of the demands posed by expansive machine learning models and large datasets. Parallel processing, a central feature of distributed systems, allows for the efficient training of intricate models and the processing of substantial datasets within practical time constraints. Another significant aspect is the optimization of resource usage, achieved by evenly distributing computational tasks, thereby averting the

bottlenecks typical in monolithic architectures. Distributed systems also inherently improve fault tolerance. This is particularly notable in their ability to eliminate single points of failure; if one node experiences a failure, others within the network can compensate, ensuring uninterrupted operation of the model.

Nevertheless, the deployment of machine learning models within a distributed framework introduces several challenges inherent to distributed systems. Efficient distribution and management of data across the network's nodes are crucial, necessitating robust strategies for data handling. The coordination and synchronization of tasks across various nodes are essential to maintain operational coherence. Network communication, particularly concerning data transfers and the consolidation of results, is a vital operational component. Furthermore, maintaining a balanced workload distribution across nodes is crucial to prevent overloading and ensure system efficiency.

The focus of this project is to explore the suitability of deploying Machine Learning (ML) models using distributed architectures, a topic stemming from the reasons previously outlined. Firstly, I will describe the limitations I set for this project to make this feasible in the time given. Additionally, I will examine the technical challenges involved in designing a distributed architecture to solve a trivial task. This will serve as a simplified parallel to the more complex Machine Learning tasks previously described, illustrating key concepts in a more accessible manner while decreasing the overall complexity of this project.
A significant portion of the paper will be dedicated to designing a solution to these challenges. This will involve outlining a proposed architecture for the distributed system, including the structure and operational mechanics of the system. I will also describe the individual components of this system, focusing on their roles and responsibilities and how they interact within the broader architecture. Furthermore, I will discuss potential failure scenarios that could arise within a distributed architecture and suggest strategies to mitigate these risks which could be implemented in future, ensuring system reliability and robustness. Lastly, the paper will propose potential evaluation metrics. These metrics will be designed to assess the effectiveness and efficiency of the system, providing a means to measure the success of the implementation.

Challenges

This project encompasses a variety of challenges, which are pivotal to its successful execution and will be elaborated upon in this section. A primary challenge encountered during the planning phase pertains to the transfer of a large dataset into the storage structure. While the process of sending data via put methods may appear straightforward, it becomes significantly complex with very large datasets. This complexity arises from the considerable latency induced by network bandwidth constraints during data addition to the storage unit. Consequently, this raises a critical question: Is the adoption of a distributed system justifiable in light of the high latency experienced during data movement? This dilemma is not unique to this project but is a prevalent concern in the industry. Amazon's AWS Snowball service offers an intriguing solution to this issue. AWS Snowball, a data transport solution, facilitates the secure transfer of large volumes of data into and out of the AWS cloud, effectively bypassing the limitations inherent in traditional internet-based transfers. It is particularly advantageous for substantial data migration, edge computing, and scenarios where network connectivity is suboptimal for large data transfers [LINK].

An academic perspective on the scalability issue is provided in the paper 'Scalability! But at what COST?'. This paper introduces the COST (Configuration that Outperforms a Single Thread) model as a metric for evaluating big data system performance. The paper contends that a distributed system's scalability should be affirmed only if it can surpass the performance of a single-threaded implementation on a single machine. This challenges the presumption that efficiency is inherent in a system's ability to manage large-scale data processing in a distributed format [LINK]. In light of this, it becomes challenging to ascertain whether computation times will indeed be minimized in a distributed architecture compared to a monolithic architecture.

Another significant challenge lies in data distribution. Determining which storage nodes are responsible for specific data subsets requires an understanding of the data access patterns needed for the intended task. For example, in tasks necessitating sequential access, it would be highly inefficient for a node to communicate with multiple nodes to retrieve a sequential data segment. Additionally, if a task requires both random and sequential access, the data storage strategy must be optimized for all access patterns. The decision whether data

distribution across nodes is dynamic or static, particularly for tasks involving substantial data, is crucial. Simple get methods returning single data points may introduce additional latency, potentially rendering a distributed architecture less efficient than a monolithic one. To mitigate this, a bulk get method like Redis's MGET, which allows retrieval of multiple keys simultaneously, could be a viable solution [LINK].

Furthermore, deciding the number of nodes in the system and how work and storage are distributed is essential. The goal is to find an optimal balance between distributing workload across nodes and minimizing the latency associated with inter-node communication.

It is for the above reasons I decided to impose several limitations to relax the problem and to make this a feasible project in the given time constraint while trying to evaluate the change of performance of the given task in a meaningful manner.

Limitations

While the aforementioned tasks related to Machine Learning (ML) models are of considerable relevance, the scope of this investigation will be intentionally constrained to simplify the problem. This approach is adopted to maintain a focused examination on the architecture of the distributed system, which constitutes the primary objective of this project. To elaborate, engaging with complex tasks distributed across nodes could inadvertently shift the emphasis towards resolving the intricacies of the task itself, rather than concentrating on the nuances of the distributed design.

Nevertheless, it is imperative to note that the simplified problem under consideration still maintains a conceptual parallelism with more complex tasks in the domain of ML. The task selected for this study involves traversing a dataset comprising 1 billion integers and identifying the most frequently occurring integer. This task, while simplified, effectively illustrates the key principles of distributed computing relevant to this research.

Moreover, this project is characterized by certain constraints, one of which is the absence of data replication. Consequently, if data becomes inaccessible – for instance, due to a storage

node becoming unresponsive or experiencing a failure – it cannot be retrieved from alternative sources, resulting in data loss. This study operates under the assumption of a non-Byzantine network, where the nodes are not malicious and do not seek to manipulate the data.

Additionally, the research imposes limitations on the deletion of data points post-storage, aimed at ensuring a balanced distribution of workload across the nodes. In line with this, the project does not make use of a mechanism for the redistribution of data points to different nodes once stored. This design choice is intended to obviate scenarios where a node might seek to access a data point from another node, only to find that the data point has been relocated.

Requirements

Before exploring the detailed architecture of the system, it's essential to understand its fundamental capabilities. Essentially, the system is engineered to read a dataset from a disk and then execute a series of bulk 'put' operations to insert this data into a distributed storage system. This storage system is comprised of multiple nodes. Once the data is successfully stored, the application layer will take charge of generating and distributing subtasks across a network of worker nodes. Each worker node, upon receiving its subtask, is tasked with fetching the required data from the appropriate storage node. The node then processes this data to compute an intermediate result. After a worker node completes its task and computes the result, it sends this information back to the application layer. The application layer, upon receiving all the individual results from the worker nodes, aggregates these to formulate a final outcome. This final result is then displayed in the system's console.

Component Descriptions

Storage Node: A storage node, as the name suggests, is designated with the responsibility of preserving a designated subset of data within a local key-value store. This node remains in a state of readiness to receive communication from any other network component, proficiently executing both 'put' and 'get' operations. The 'put' operation entails the insertion of a new data point into the local key value store, while the 'get' operation involves retrieving a specific value corresponding to a given key and subsequently returning it to the requestor. Notably, these operations are engineered to accommodate bulk transactions, thereby enabling the handling of numerous key-value pairs concurrently. Each storage node operates autonomously yet remains accessible over the local network through Remote Procedure Calls (RPC). Upon the instantiation of multiple storage nodes, the collective ensemble is referred to as a 'network of storage nodes.'

Worker Node: In a parallel vein to the storage node, a worker node is capable of functioning independently from other system components, remaining vigilant for incoming task assignments. Its primary function is to evaluate and process incoming tasks, involving communication with the corresponding storage nodes for data retrieval. It employs an internal algorithm to deduce solutions for the subtasks and relays these outcomes back. The emergence of multiple workers is termed as a 'network of worker nodes.'

Storage Coordinator: This class is responsible for the initial insertion of data. It uses a method to read the data from disk into local memory and keep track of the data size.
In addition, it is also responsible to take the data size into consideration as well as the number of available storage nodes and suggest an appropriate data distribution across the nodes. It does so by balancing an equal amount of the data set on each storage node. Once the distribution has been calculated it will send put request with the data to the appropriate storage nodes.

Workload Coordinator: The Workload Coordinator class is integral to the system's architecture, serving as a central orchestrator for task distribution among worker nodes. Its primary role involves interfacing with both the Storage Coordinator and Application classes to acquire essential parameters for task management. Initially, the Workload Coordinator

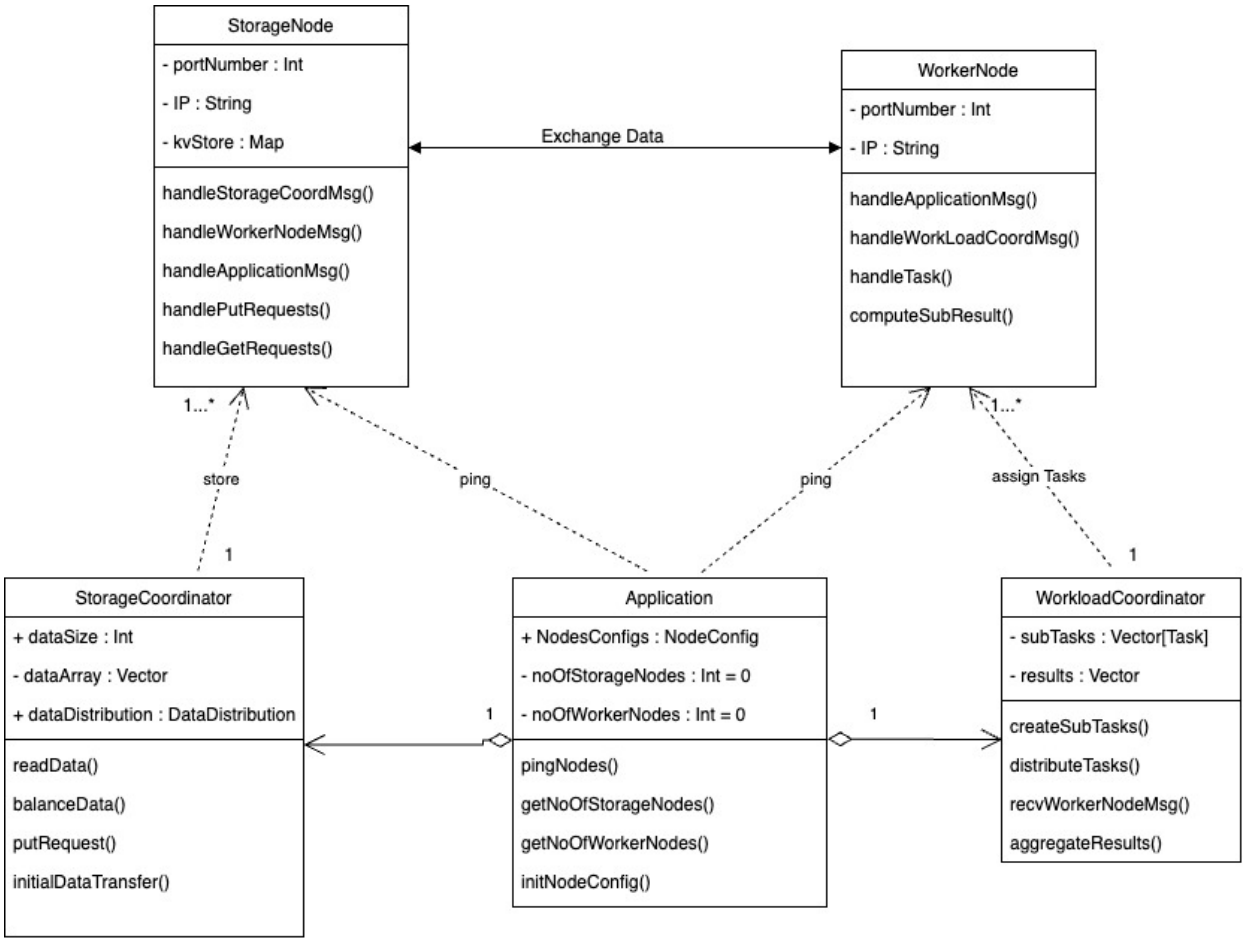
class retrieves the total data size from the Storage Coordinator class, a critical step in understanding the volume of data that needs processing. Concurrently, it acquires specific data ranges assigned to each node from the Storage Coordinator. This information is pivotal in defining the scope of data each node is responsible for handling.

Furthermore, the Workload Coordinator class engages with the Application class to determine the current count of operational worker nodes. This knowledge is fundamental in planning the distribution of workload. Subsequently, the class embarks on the equitable distribution of work. It strategically divides the overarching task into smaller, more manageable subtasks, each encompassing a designated data range and the corresponding information for the worker and storage nodes involved. This information entails the IP addresses of the appropriate storage and worker nodes.

Once these subtasks are created, the Workload Coordinator class proceeds to dispatch them to the respective worker nodes. This involves a systematic communication mechanism, ensuring each worker node receives its tailored subtask, complete with specific data range and port configurations for the necessary storage nodes. Through this coordinated approach, the Workload Coordinator class ensures a cohesive and systematic execution of tasks within the distributed system. Once the tasks have been computed by the worker nodes the Workload Coordinator will aggregate the results and print the results.

Application: The application class represents the main program that orchestrates the entire system. It is aware of all connected nodes and what type they are. It does so by sending ping messages to all nodes upon runtime. Once the Application class is aware of all connected nodes it will store that information. Once this has been completed the Application class will instantiate a Storage Coordinator to perform the initial data insertion. Once this has been completed the Application class will instantiate the Workload Coordinator and wait for its tasks to be completed.

UML Class Diagram



The class diagram presented offers an depiction of the system's architecture, detailing the attributes, methods, and the interrelations among the classes. It is designed to be aligned with the characterizations delineated in the preceding sections of the documentation. Within this schema, specialized data types have been used, including `DataDistribution`, `NodesConfig`, and `Task`. `DataDistribution` is conceptualized as a structure encapsulating information relevant to each node, including its IP address and the span of data it manages. This structure is instantiated within the `balanceData` method, operational within the `StorageCoordinator` class. Furthermore, `NodesConfig` is seen as a structure that stores the entirety of the storage and worker networks. It functions as an aggregate of nodes, recording their respective IP addresses and categorizing them by type, whether as storage or worker nodes. This construct is critical for monitoring the nodes that are active across the network. It emerges at runtime, subsequent to the `Application` class obtaining the responses from the `pingNodes` method.

Concludingly, the `Task` structure is intended to encapsulate vital information concerning the subtasks to be allocated to the worker nodes. It encompasses the designated worker node and its IP address, the specific data range required for retrieval, and the identity of the node from which this data is to be sourced. A vector of `Task` instances is to be generated, from which the `Workload Coordinator` may dispatch tasks to the appropriate worker nodes for execution.

Scalability

Within the context of this system, scalability is defined as the ability to efficiently manage an increased workload by supplementing the system with additional resources. Theoretically, the system's architecture is engineered to support horizontal scaling, which is evidenced by the capacity to add numerous independent storage and worker nodes. These components operate autonomously and are coordinated effectively by both the Workload Coordinator and Storage Coordinator, which implies a scalable design without a predefined limit to node addition. Nevertheless, the practical application of this system is confined to a single-machine environment, which introduces several tangible limitations. These constraints encompass the limited computational power provided by the CPU, which bounds the number of nodes that can perform simultaneous processing tasks. Additionally, the availability of RAM is a critical resource, with its limitations potentially leading to memory allocation issues as the number of nodes increases. Network bandwidth, shared across all nodes on this single machine, is also a factor that could result in increased latency and network congestion as the system attempts to scale. Moreover, the number of available ports on a single machine is limited, which could complicate the management of network communications for a large number of nodes.

In summary, while the design of the system theoretically supports expansion, the actual scalability is inherently limited by the physical and technical resources of the single machine on which it will be implemented. These constraints must be carefully considered to ensure the system remains functional and efficient as it scales within the parameters of the available infrastructure.

Failure Scenarios

The current design of the system inherently presents a multitude of potential failure scenarios that could lead to malfunctions or complete system crashes. These scenarios are a direct consequence of the project's scope, which was deliberately simplified to meet the constraints of the timeframe, as outlined in the limitations section. A primary concern in this design is the assumption of constant uptime and responsiveness of nodes. The system currently lacks mechanisms to address nodes crashing during runtime, which could lead to significant disruptions. Furthermore, the communication protocol assumes the successful receipt of messages without providing for the retransmission of lost or corrupted messages. This could result in the system indefinitely awaiting a message, thereby stalling the completion of tasks. Additionally, the system operates under the presumption that data is always available at the specified nodes. It does not cater to scenarios involving missing data, which could jeopardize task completion. Moreover, there is an expectation that all worker nodes will successfully complete their assigned tasks. The system, as designed, does not account for the possibility of individual subtask failures and the subsequent need for re-computation. It is important to recognize that this list is not exhaustive; other potential failure scenarios exist. These include the risk of overloading specific nodes, leading to bottlenecks, software bugs that may induce unexpected behaviors, conflicts arising from improper management of network ports, and the broader implications of network failure.

In summary, while the system's design facilitates certain primary functions, its simplified nature under the project's constraints leaves it vulnerable to a range of failure scenarios, many of which the system is currently not equipped to handle effectively. Addressing these issues would require a more complex and robust design, incorporating fail-safes, error handling, and contingency plans to enhance the system's resilience and reliability.

Remote Procedure Calls

In the proposed system architecture, considerable emphasis has been placed on the interactions among nodes within the network. However, the underlying communication protocols that facilitate this efficient interaction and data exchange have not been explicitly discussed. The system employs gRPC as the communication protocol for a multitude of

compelling reasons. gRPC leverages HTTP/2, offering multiplexed streams over a single connection, which significantly diminishes latency and enhances the efficiency of data transfers. Additionally, gRPC is particularly advantageous for systems requiring real-time data processing or handling substantial data sets due to its native support for streaming both requests and responses. While alternative protocols such as REST over HTTP/1.1 are available, they lack the sophisticated capabilities inherent in gRPC. Specifically, gRPC's prowess in effective data serialization and the advanced features provided by HTTP/2 position it as the superior choice for a messaging protocol within the context of this project. Lastly, there is a fast amount of documentation available online which aids the developer to properly use this technology.

Evaluation Metrics

In assessing the efficiency of the designed distributed architecture, the dominant metric for evaluation is correctness. Navigating the complexities inherent in constructing a distributed system for the first time presents numerous opportunities for error, making it imperative that the system produces accurate computational results. The validation of this correctness will be conducted by comparing the distributed system's results against those generated by a straightforward function on a monolithic architecture.

For this purpose, the dataset utilized will be randomly generated and persisted on disk, facilitating a straightforward assessment of the system's accuracy. Beyond correctness, the secondary goal is to enhance efficiency in obtaining the result. Specifically, the system aims to reduce the overall computation time compared to the time taken by a singular function's execution. It is important to note that this efficiency measurement will start from the moment the Workload Coordinator dispatches subtasks and conclude once an aggregate result is computed, deliberately excluding the initial data insertion phase from the metric.

Additional evaluation will involve timing individual subtasks to identify possible network bottlenecks or latency issues. Equally critical is the assessment of load distribution across storage nodes. By tracking the number of 'get' requests each storage node handles relative to

the system's total, a clear indicator of each node's workload will emerge. An equitable distribution of these requests would suggest a successful implementation of the architecture.

Furthermore, additional metrics such as latency and memory usage will also be monitored. Latency will be measured by the time elapsed for message transmission between nodes, while memory usage will be monitored during system operation. These metrics will offer comparative insights across different configurations of storage and worker nodes—for instance, evaluating the system with three storage nodes and three worker nodes, and then contrasting those results with a setup of three storage nodes and six worker nodes to test the impact of varying node quantities. In conclusion, many combinations of metrics can be used to evaluate the success of this project. However, it is important to note that all metrics apart from correctness rely on the fact that the system behaves as expected and will produce a correct result. If this will not be the case at the end of the implementation these metrics might not be of great value.

Conclusion

This document lays a solid groundwork for the implementation phase of the project, despite anticipating moments of ambiguity inherent in the realm of distributed systems architecture. Such instances should not be viewed as setbacks but as opportunities to refine and enhance the design. It is acknowledged that, despite thorough planning, certain aspects may remain unresolved at the onset of implementation. Therefore, it is deemed appropriate to allow for adaptive modifications to this document to address any unforeseen complexities or conflicting scenarios that may arise during the practical application of the design.

Moreover, the collaborative academic environment provides additional support mechanisms. Engaging in discussions with peers, who may encounter similar challenges, offers a valuable exchange of insights and solutions. Furthermore, the availability of resources such as office hours, lectures, and dedicated discussion forums like Discord channels ensures continuous support and guidance. Given these supportive elements and the comprehensive nature of the planning encapsulated in this document, there is a strong confidence in the successful implementation and functionality of the project as envisioned in the design.