Distributed Resource Management for YARN

KUGANESAN SRIJEYANTHAN

Master’s of Science Thesis
KTH Royal Institute of Technology
School of Information and Communication Technology
Supervisor: Dr. Jim Dowling
Examiner: Dr. Johan Montelius
Stockholm, Sweden, September 2015

TRITA-ICT-EX-2015:231
Abstract

In the last year, Hadoop YARN has become the de facto standard resource management platform for data-intensive applications, with support for a wide range of data analytics platforms such as Apache Spark, MapReduce V2, MPI, Apache Flink, and Apache Giraph. The ResourceManager fulfills three main functions: it manages the set of active applications (Applications service), it schedules resources (CPU, memory) to applications (the FIFO/Capacity/Fair Scheduler), and it monitors the state of resources in the cluster (ResourceTracker service). Though YARN is more scalable and fault-tolerant than its predecessor, the JobTracker in MapReduce, its ResourceManager is still a single point of failure and a performance bottleneck due to its centralized architecture. Single point of failure problem of YARN has been addressed in Hops-YARN that provides multiple ResourceManagers (one active and others on standby), where the ResourceManager’s state is persisted to MYSQL Cluster and can quickly be recovered by a standby ResourceManager in the event of failure of the active ResourceManager. In large YARN clusters, with up to 4000 nodes, the ResourceTracker service handles over one thousand heartbeats per second from the nodes in the cluster (NodeManagers), as such become a scalability bottleneck. Large clusters handle this by reducing the frequency of heartbeats from NodeManagers, but this comes at the cost of reduced interactivity for YARN (slower application start-up times), as all communication from the ResourceManager to NodeManagers is sent in response to heartbeat messages. Since Hops-YARN is still using a centralized scheduler for all applications, distributing the ResourceTracker service across multiple nodes will reduce the amount of heartbeat messages that need to be processed per ResourceTracker, thus enabling both larger cluster sizes and lower latency for scheduling containers to applications. In this thesis, we will scale-out the ResourceTracker service, by distributing it over standby ResourceManagers using MySQL NDB Cluster event streaming. As such, the distributed Resource Management for YARN that is designed and developed in this project is a first step towards making the monolithic YARN ResourceManager scalable and more interactive.
Acknowledgement

I want to thank my adviser, Professor Jim Dowling, for his insightful guidance, his constant encouragement, and his patient and prompt replies to my suggestions and scribblings, despite his busy schedule. I want to address my thanks to Gautier Bethrou, for his commitment and the discussions that generated many ideas described in this work. I would also like to thank Professor Luís Manuel Antunes Veiga for approval of this work and providing me a constructive suggestions to improve this thesis. This thesis could not have been completed without their generous and professional assistance. Moreover, I would like to thank all the professors who taught us at IST and KTH Royal Institute of Technology during my master studies.

I would also like to thank my family for their support and my wife for her immense patience while trying to finish this thesis. Further, I am extending my appreciation to the researchers in Distributed System group projects, Their advices and careful guidance were invaluable during the course of the whole work.

To all my friends, thank you for your understanding and encouragement in my moments of crisis. Your friendship makes my life a wonderful experience. I cannot list all the names here, but you are always on my mind.

Thank you, Lord, for always being there for me.

This work was partially supported by Erasmus Mundus Master Courses Scholarship (Category A), provided by EACEA\(^1\).

---
\(^1\)Education, Audiovisual and Culture Executive Agency  
This thesis is dedicated to my wife and parents. For their endless love, support and encouragement.
Contents

Acknowledgement

List of Figures

1 Introduction
   1.1 Problem statement ........................................ 2
   1.2 Thesis Objectives and Contributions ......................... 3
   1.3 Organization of the Thesis ................................ 4

2 Background
   2.0.1 YARN .................................................. 5
   2.0.2 Components and Services ................................ 7
   2.0.3 ApplicationMaster ..................................... 10
   2.0.4 NodeManager ......................................... 10
   2.0.5 RPC Communication ................................... 11
   2.0.6 Fault tolerance ....................................... 11
   2.0.7 Single point of failure ................................. 11
   2.0.8 ResourceManager the Scalability Bottleneck ............ 12
   2.1 MySQL cluster database ................................ 13
      2.1.1 The Architecture of MySQL Cluster .................... 14
      2.1.2 Data nodes ......................................... 14
      2.1.3 Application nodes .................................. 15
      2.1.4 Management nodes .................................. 16
      2.1.5 MySQL NDB cluster event streaming ................... 16
   2.2 Hops YARN Architecture ................................ 19
      2.2.1 Data Access Layer .................................. 20
   2.3 ClusetJ .................................................... 21
   2.4 Related work ............................................. 23
      2.4.1 Mesos .............................................. 23
      2.4.2 Google Omega ...................................... 24
   2.5 Summary .................................................. 26

3 Design
   3.1 Architecture of Event Streaming library .................... 27
List of Figures

1.1 Sequencing cost variation per genome from 2001 to 2014 ............... 2
2.1 Difference between MapReduce framework and YARN framework .... 6
2.2 Architecture of YARN framework ........................................ 7
2.3 Scheduler ........................................................................... 9
2.4 Sample queue hierarchy ...................................................... 10
2.5 YARN internal communication strategy ................................. 13
2.6 Architecture of MySQL NDB cluster .................................... 14
2.7 The MySQL NDB cluster data partition ................................. 15
2.8 Diversity of application connectivity ................................. 16
2.9 Transactions at different data nodes and possible event streaming ... 17
2.10 Merge events and ordering the transactions .......................... 18
2.11 MySQL Cluster-to-Cluster Replication Layout ........................ 19
2.12 Java access to MySQL Cluster ......................................... 21
2.13 ClusterJ interface annotations ........................................... 22
2.14 Cluster resource is statically partitioned ............................... 25
3.1 Loading the C++ library into YARN ....................................... 28
3.2 Architecture of C++ Event streaming library .......................... 29
3.3 Time distribution of C++ processing and JNI invocation calls ........ 30
3.4 Time line of multithreaded processing ................................... 31
3.5 Memory layout of streaming library and YARN ...................... 32
3.6 Hops distributed YARN Architecture .................................... 33
3.7 Component diagram of YARN Scheduler Load Simulator ........... 34
3.8 Architecture of standalone simulator ..................................... 37
3.9 Architecture of distributed simulator .................................... 37
4.1 Communication queue between Event polling thread and JNI dispatcher thread ........................................ 41
5.1 MySQL NDB cluster deployment ........................................... 47
5.2 Streaming library through-put evaluation with various number of threads ........................................... 49
5.3 Simulated application load ................................................... 50
5.4 Simulated application load ................................................... 51
5.5 Hadoop-2.4.0/2.6.0 deployment ............................................. 52
5.6 Hops distributed YARN deployment ................................. 52
5.7 Scalability performance of Hadoop-2.4 .............................. 53
5.8 Scalability performance of Hadoop-2.6 .............................. 54
5.9 Scalability performance of Hops distributed YARN ............... 54
5.10 True heartbeat percentage variation with ResourceManager and Re-
sourceTrackers transactions ............................................. 55
5.11 Scheduler operation: Node update time cost .................. 56
5.12 Scheduler operation: Application master allocate time cost .... 56
5.13 CPU utilization of all the components of Hops distributed YARN, Hadoop-
2.4.0 and Hadoop-2.6.0 .................................................... 57
5.14 Memory usage of Hops distributed YARN components ........... 57
Chapter 1

Introduction

We are currently witnessing a huge growth in data volumes and number of connected devices. Demanding of big data processing is extremely growing in enterprise domain, medical research, large scale scientific research etc. The term Big Data has gone through from very simple concept to more complicated in these days. Historically, the data was mainly being produced from workers. In other words, the companies will have to manually enter data into computer systems. After the evolution of the Internet, users and machines are started to accumulate data. So, data is generated at such a higher rate that it is impossible to store, transfer and analyze it with limited computing power, storage and bandwidth.

Advancement in the field of Distributed Systems allows researchers to store data in a distributed manner over multiple machines and networks empowered by programming models that have been developed to process data efficiently over such distributed storage. Meanwhile in the research domain, the need to develop such computer systems is driven by a plethora of scientific areas, one of which is analyzing and storing biobank information that can be later used by medical and bioinformatics applications. One of the first such applications was the Human Genome Project[1] which in 2003 managed to successfully decode the human genome, and, interestingly, it cost nearly 3 billion dollars. Genome sequencing cost, however, has decreased nearly by half every four months since 2007. As of now, sequencing a whole human genome would cost only 1,000 dollars. The sequencing cost variation can be clearly seen in Figure 1.1. This has been considered as the paradigm shift in sequencing technology because of falling the cost per genome dramatically in the beginning of 2008 [2]. The main reason for this was moving the sequencing technology from first generation platforms to second generation platforms, which is the Next-Generation sequencer. In order to expand the whole-genome sequencing up to the population-scale, speed and cost effective data processing platforms have become the most important driving force for future developments. This thesis was conducted in the context of the BiobankCloud project, with its area of work being
focused on how to scale up the aforementioned compute platform to handle very large workloads by distributing YARN services over multiple nodes.

![Figure 1.1: Sequencing cost variation per genome from 2001 to 2014](image)

### 1.1 Problem statement

Despite the fact that distributed file systems are continuously becoming faster and more robust, the increasing demand for processing very large datasets poses a challenge to these systems. Users of these compute platforms require their applications to run without unexpected interruptions and in the least amount of time possible. Although a lot of progress has been made in storing large datasets over a distributed file system, there are still hurdles to overcome when it comes to managing cluster resources and scheduling application execution on top of them. Such tasks can be extremely demanding in terms of memory usage and CPU utilization which means that the compute platform must make sure that all available resources distributed over the participating machines are utilized in the most efficient way. In general, the entity that is responsible for managing cluster resource allocation to applications is called ResourceManager(RM). This work is based on the recent paradigm set by YARN [3] which means that we consider a ResourceManager that only performs cluster resource management. In large YARN clusters, the ResourceTracker service which is part of the ResourceManager handles large number of heartbeats which carry application related information; node status etc., from the nodes in the cluster has become a scalability bottleneck. Large clusters are reducing heartbeat frequency to handle such a huge load from NodeManagers, but it would be frustrating from a user’s point of view to have the slower application execution and interacting with YARN clusters.

The Hops-YARN increases the scalability and provides High Availability of
Apache YARN by externalizing the ResourceManager state from the heap of a JVM\(^1\) to MySQL Cluster\(^2\). The contribution of this thesis is the design, implementation and experimental evaluation of Hops distributed YARN that allows the ResourceTrackers to run on multiple stateless physical nodes simultaneously to increase the scalability of the cluster. These ResourceTracker nodes contact the database each time they want to update the state, and our implemented C++ NDB Event streaming library constructs appropriate Java objects for the scheduler to operate as normal YARN. The same procedure will be followed by ResourceTracker when scheduler updates the state in database. This approach can improve the Hops-YARN in the following two ways.

First, it increases the scalability of Hops-YARN as the ResourceTrackers will be able to run on multiple machines all of which can serve requests simultaneously and, secondly, to make Hops-YARN more interactive as the scheduler updates its cluster view without any delay, so that application schedule and execution time is reduced.

### 1.2 Thesis Objectives and Contributions

The purpose of this work is to demonstrate that by using NDB Event streaming as an efficient communication mechanism, which drives Hops distributed YARN becomes more scalable and interactive. The major contributions of this thesis are further detailed below:

- Multi-threaded architecture of C++ NDB Event streaming library:
  1. The design and implementation of the Multi-threaded C++ NDB Event streaming library.
  2. Throughput analysis of implemented library and compare with YARN requirements.
  3. Integration of the newly implemented Event streaming library with distributed ResourceTrackers, and scheduler.

- Design and implementation of a realistic YARN distributed load simulator:
  1. Extracting the features from existing YARN Scheduler Load Simulator.
  2. Designing a new architecture to support distributed ResourceTrackers, and scheduler over the network simulations.

---

\(^1\)Java Virtual Machine  
\(^2\)MySQL cluster  
3. A detailed model and study about the ApplicationManager and NodeManager simulation in multiple instance simulator mode.

- The synthetic load generation and evaluation:
  1. Implement a new trace generator to generate different configurations of NodeManagers and Applications
  2. The scalability evaluation on Hops distributed YARN, Hadoop-2.4.0 and 2.6.0

1.3 Organization of the Thesis

Chapter 1 introduces the reader to the general area of the problem and states what is the exact problem to be investigated and solved. It continues by briefly describing the approach to the solution and mentions the goals of this work. Chapter 2 provides the reader with all the necessary background knowledge in order to fully comprehend the proposed solution and experimental results. Emphasis is given on the frameworks that are used to implement this work. In addition, related work is referenced. Chapter 3 demonstrates the model and architecture of the solution while chapter 4 provides the implementation details. Chapter 5 presents the various evaluations of proposed methods and analysis of experimental results. Chapter 6 concludes the presented work, and proposes future work based on this report. Chapter 7 gives an extra detail of library configuration parameters for easy operation.
Chapter 2

Background

This chapter presents all the necessary background information and references work that reader needs to have in order to fully understand and comprehend this work. The first section presents an operational detail about YARN, and its components. Emphasis is put on YARN as it is the technology this thesis was based and implemented on. Subsequently, the MySQL cluster\[4\] distributed database and the ClusterJ\(^1\) library functionalities are also presented, as the Hops YARN echo systems which are widely used technologies to archive high availability, scalability and reliability.

2.0.1 YARN

To perform a data analysis effectively and efficiently on big data, tools must be able to absorb data quickly, but they also must be able to analyze this data for value, providing analysts to ask and iterate their business question quickly. The legacy tools are inadequate to process incoming data volumes which are exploding in complexity, speed and size. The Apache Hadoop Mapreduce\[5\] framework was entered in to the Big Data analytics scene to solve big data problems in some extent. The purpose of Hadoop\[6\] is to facilitate the scaling of certain forms of batch-oriented data processing built on fundamentals which severely bound its ability to act as an analytic database. Generally, the Hadoop MapReduce framework was conceived to resolve a very specific problem which is to enable distributed MapReduce processing on fixed sized clusters comprised of low-cost hardware. The Hadoop Distributed File System\[7\] was created to store actual data to enable distributed MapReduce processing.

As shown in Figure 2.1, the MapReduce framework, JobTracker\(^2\) is mainly re-

\(^1\)Using clusterJ

\(^2\)JobTracker
http://wiki.apache.org/hadoop/JobTracker
sponsible for managing cluster resources and controls the execution of the MapReduce job. The JobTracker reserves and schedules slots for all Jobs, configure, execute and monitor each task, and if a task fails or crash, it allocates a new slot and reattempts the job. Once a job is done, the JobTracker cleans up temporary resources and releases the job’s slot to make it available for other jobs. Each task runs on the DataNode which in one of the machines on cluster, and each machine has a predefined number map slots and reduce slot for running tasks concurrently.

The MapReduce framework suffers from the following limitations

1. Scalability bottleneck: As we have seen in previous discussion, the JobTracker runs on a single machine and executing multiple tasks like resource management, task scheduling and monitoring increases the complexity for the JobTracker. This prevents larger cluster data processing and limits the scalability.

2. Resource utilization: Each TaskTracker has predefined number of map slots and reduce slots. It is possible for either map slots or reduces slots to become empty, while other slots become full. The DataNode reserved for map slot or reduce slot could sit idle when there is an immediate need for those resource to be used as Mapper slots or Reducer slots.

3. Availability: The JobTracker is a single point of failure, which means if the JobTracker fails, all the jobs must be restarted.

YARN was developed as a means of dealing with the aforementioned issues. The YARN took over the task of cluster management from MapReduce by splitting two major responsibilities of the JobTracker which are; resource management and job scheduling/monitoring. YARN has a centralized ResourceManager component, which manages resources and allocates resources to the application. The detailed architecture of the YARN is shown in Figure 2.2. Due to the central resource management capabilities, YARN performs efficient utilization of the resources which

---

3 TaskTracker

http://wiki.apache.org/hadoop/TaskTracker
allows multiple applications in Hadoop to share common resources. As YARN decouples resource management and scheduling capabilities from data processing components, YARN now supports wide spectrum of data analytic tools such as interactive querying, streaming applications simultaneously with MapReduce batch processing.

![YARN Architecture Diagram](image)

**Figure 2.2: Architecture of YARN framework**

### 2.0.2 Components and Services

As shown in YARN architecture in Fig 2.2, first a list of its components is presented and then each one is discussed in detail.

1. ResourceManager
   a) ResourceTracker service
   b) ApplicationMaster service
   c) Scheduler

2. ApplicationMaster

3. NodeManager

1. **ResourceTracker service**: The ResourceTracker service is responsible for handling NodeManager registration requests and the periodic heartbeats each
NodeManager sends to the ResourceManager. Registration means that a NodeManager notifies the ResourceManager that it is available to receive applications and provides information about its available physical memory and CPU cores. The periodic heartbeats serve two purposes. Firstly to notify the ResourceManager via the ResourceTracker service of its health, and secondly, about the current state of the containers available in the NodeManager. This information is used by the scheduler to update its global view of the cluster resources so that it efficiently allocates them upon future requests from the ApplicationMaster.

2. **ApplicationMaster service**: The requests by the ApplicationMasters are handled by the ApplicationMaster service. All requests follow the same format and can contain information such as number of containers required, resources per container in terms of memory and CPU cores, locality preferences and priority of requests from within the application. In addition, the ApplicationManager handles requests for new jobs, allocates the container on which the ApplicationMaster of the requested application will execute and can restart the ApplicationMaster in case of failure.

3. **Scheduler**: The scheduler is doing the most prominent job in distributed systems, and normally they operate in between jobs and machines. In a cluster, tens of thousands of machines are running jobs which are collections of tasks. A task, we could imagine as Linux process or process group, and each task runs on one machine. We can have many tasks running on any one machine; in fact we share machines for different kind of tasks. For instance, machine running on Gmail or map reduce jobs. Typically ten or fifteen tasks run on each machine and the purpose of the cluster management system known as the scheduler which is to decide which task should be executed on which machine.

Things are getting more complex, because the times of tasks is actually remarkable variable due to whole different kind of workload such as task running shorter period of time, tasks running for a month, tasks consuming more CPU core, task has incredibly higher reliability requirements, tasks with latency intensive jobs. All these things are running on a same universe. The size of the clusters are getting larger, and the rate which work arrive turn out be roughly proportional to number of core in to the cluster. So, the scheduler has to do more work. As shown in Figure 2.3, the small boxes represents machines, 1000 of arriving jobs which may have various number of identical tasks, and the cluster scheduler is middle between jobs and machines which places the task in machine identify what kind of jobs is coming in and process correctly and put it right place and give it right proprieties.

The YARN monolithic scheduler has a single scheduler, handle all kind of workloads and placing tasks according to scheduler modes such as FIFO, Fair and capacity[6].
a) **Capacity scheduler**: The capacity scheduler is designed to run Hadoop applications as a shared, multi-tenant cluster in an operator-friendly manner while maximizing the throughput and the utilization of the cluster \(^4\). Using a capacity scheduler allows sharing a large cluster while providing capacity guarantees for participating organizations. This is accomplished by hierarchical queues which are the fundamental unit of scheduling in YARN. Figure 2.4 shows an example of queue hierarchy of cluster resource management. Hierarchy of the queue set up can be based on database structure, resource requirements, and access limitations by various organizations. The capacity of each queue denotes the percentage of cluster resources which are accessible for submitted applications in to that queue.

Queues are mainly classified into two types: parent queues and leaf queues. Parent queues allow management of resources among organizations and sub-organizations, and they do not accept application submissions directly. Applications should be placed in leaf queues. Capacity scheduling ensures that guaranteed resources are first shared among the sub-queues of an organization prior any available resources are share with queues belongs to other organization.

b) **Fair scheduler**: The fair scheduler organizes jobs into pools, and di-

---

\(^4\)Hadoop MapReduce Next Generation - Capacity Scheduler
https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html
vides resources fairly between these pools. Fairness property can be determined by several factors such as memory, CPU, application priorities etc. By default, Fair scheduler bases scheduling decisions only on memory. But it can be extended to support both memory and CPU, using the notion of Dominant Resource Fairness developed by [8]. Fair scheduler works as following; if a single application is running, that Application uses the entire cluster resources. When new applications are submitted, resources that free up are assigned to the new applications, so that all the applications eventually get roughly equal amount of resources. Apart from that, Fair scheduler allows assigning guaranteed minimum resource share to queue, which is useful for ensuring that certain users, groups or production applications always get sufficient resources.

c) FIFO scheduler: As name implies, applications are scheduled in an order as they arrived in the scheduler. The FIFO does not need any extra configuration as Fair and Capacity. FIFO scheduler are not suitable for shared cluster as large application will use all the resource in a cluster, so all the applications have to wait until its turn. The FIFO scheduler allows long-running jobs to complete in a timely manner, so that concurrent small queries would be able to get results back in reasonable time.

2.0.3 ApplicationMaster

The ApplicationMaster(AM) [6] is spawned by the scheduler in its own container and its main job is to monitor and manage application execution. It also requests resource allocation (containers) from the scheduler and periodically reports its status back to the scheduler. In addition, the ApplicationMaster can give up a container if the application no longer needs it which frees up system resources.

2.0.4 NodeManager

The NodeManager(NM) [6] in YARN is the equivalent of a worker in MapReduce. It communicates with both the ResourceManager and the ApplicationMaster and transmits status of currently running containers and available resources in terms of
memory and CPU on its machine by sending heartbeats to the ResourceManager. It is also responsible of terminating containers based on requests made by either the ResourceManager of the ApplicationMaster.

### 2.0.5 RPC Communication

Entities in YARN that need to exchange messages with each other remotely, do it by employing the Protocol Buffer RPC technology\(^5\). Developed by Google, protocol buffers describe a mechanism to serialize structured data. The programmer only needs to once provide information about what type of data the message must contain and the compiler generates the required code used to actually transmit the message to a remote process. Its main advantages over other similar types of transmitting structured data such as XML, is its simplicity and performance. Protocol buffers make it easy to create new RPC messages with writing minimal code.

### 2.0.6 Fault tolerance

YARN was developed knowing that fault-tolerance integrated in every one of its components and the goal was for these components to be loosely coupled. That means that when an ApplicationMaster fails this does not affect performance of the ResourceManager, although the latter needs to reinitialize the ApplicationMaster so that the application eventually finishes successfully. However the ApplicationMaster state is not retrieved which means that tasks running while ApplicationMaster was restarting will have to be all re-run. If a NodeManager fails, the ResourceManager detects that by not receiving a heartbeat and marks all containers running on that NodeManager so that they are not used in the future. Tasks running on the failed NodeManager, are reassigned to other NodeManagers by the responsible ApplicationMaster. When the ResourceManager fails and restarts, the applications need to be restarted in order to complete. Clients however do not have to resubmit the applications.

### 2.0.7 Single point of failure

In order to solve the single point of failure problem, a replication mechanism has recently been introduced in YARN. The principle of this mechanism is to run two ResourceManagers: a main ResourceManager and a standby ResourceManager. The main ResourceManager run the tasks described above and stores a copy of its state in ZooKeeper[9]. As long as the main ResourceManager is running, the standby ResourceManager does nothing. When the main ResourceManager crash the standby ResourceManager pulls the state of the crashed ResourceManager from ZooKeeper and becomes the new main ResourceManager. This solution is limited by the small

---

\(^5\)Protocol Buffers  
https://developers.google.com/protocol-buffers/
throughput of ZooKeeper which results in the impossibility for the main ResourceManager to store its full state. As a result the main ResourceManager only store a partial state in Zookeeper and the standby ResourceManager must restart all the running applications, when a failover happens. This can result in the loss of hours of computation. In order to solve this problem, the Hadoop community is working on a mechanism that would allow the standby ResourceManager to rebuild the missing parts of the state by getting information from the NodeManagers. Getting the information and rebuilding the state is a slow process and can result in the cluster being frozen during a long time after a fail over.

2.0.8 ResourceManager the Scalability Bottleneck

The second problem of Hadoop YARN architecture, the bottleneck, is not addressed by the solution mentioned above: as long as the main ResourceManager does not crash the standby ResourceManager stay inactive and is a waste of resources. This bottleneck problem has two results: The size of the cluster is limited and the responsiveness of the cluster is limited. Even if most of the existing Hadoop clusters are too small to reach the size limitation, there exist some large clusters, such as Yahoo’s clusters, that reaches the limits of this architecture. The responsiveness problem comes from the mechanism used to exchange messages between the ResourceManager and the NodeManagers. In order to ensure that the ResourceManager will never be overloaded all the exchanges between the ResourceManager and the NodeManagers are following a heartbeat mechanism. When setting up a cluster, the administrator decides on a frequency at which the NodeManagers will send messages to the ResourceManager. This way, there is no risk of spike utilization of the ResourceManager and it is easy to modulate throughput at which the ResourceManager will have to handle messages by modulating the frequency of the heartbeats. In order to avoid overloading the ResourceManager the administrator must choose a low frequency. This results in the ResourceManager needing time to be informed of any change in the state of the cluster and to propagate resource attribution to applications. As a result, starting an application and getting resources for this application is slow and the cluster is not interactive.

A bottleneck problem also appears in the internal architecture of the ResourceManager. This architecture is represented in figure 2.5. In this figure, each square represent a single thread. Each thread is equipped with a queue and the threads communicate with each other by putting events in each other queue. This system provides some parallelism. But, as we can see, the RMContext Queue plays a central role: all the messages received by the ResourceManager result in at least one event in the RMContext Queue. As a result the RMContext Queue is a bottleneck inside of the ResourceManager which is itself a bottleneck in the YARN Architecture. In order to reduce the bottleneck aspects of YARN we propose to give an active role to the standby ResourceManager, to offload the load on the main ResourceManager.
2.1 MySQL cluster database

The Hops-YARN [6] increases the scalability of Apache YARN by externalizing the ResourceManager state from the heap of a JVM to a distributed, in-memory, replicated, database: MySQL Cluster [7]. This has enabled us to partition the ResourceManager services, and separately scale-out the ResourceTracker service, which is concerned with processing heartbeats from and commands to NodeManagers. The MySQL Cluster is a real-time, ACID-compliant transactional, relational database, with no single point of failure[4]. It integrates the standard MySQL server with an in-memory clustered storage engine called NDB (which stands for "Network DataBase"). MySQL cluster’s most notable features are

1. Auto-sharding for reads and writes scalability
2. Cross-data center geographic synchronous and asynchronous replication
3. Online scaling and schema upgrades
4. SQL and NoSQL interfaces

---

2.1.1 The Architecture of MySQL Cluster

As illustrated in the Fig 2.6 below, MySQL Cluster comprises three types of node which collectively provide service to the application: Data nodes, Application nodes, Management node. The MySQL cluster’s real-time design provides negligible latency response with the power to service millions of operations per second. It also provides load balancing and the ability to add more nodes in to a running cluster with zero downtime which allows linear database scalability to handle peak workloads. In scalability tests with 48 nodes, MySQL Cluster 7.4 delivers massively concurrent NoSQL access - 200 Million reads per second using the FlexAsync benchmark.

The MySQL Cluster provides both SQL and native NoSQL APIs for data access and manipulation. Hops-YARN mostly uses the NoSQL APIs because it allows us to reach a high throughput by bypassing the MySQL Server and directly communicate with the storage layer.

![Figure 2.6: Architecture of MySQL NDB cluster](image)

2.1.2 Data nodes

The Data nodes manage the storage and access to data. Tables are automatically sharded across the data nodes which also transparently handle load balancing, repli-

---

8 Performance Testing of MySQL Cluster: The flexAsynch Benchmark
https://blogs.oracle.com/MySQL/entry/performance_testing_of_mysql_cluster
2.1. MYSQL CLUSTER DATABASE

cation, fail-over and self-healing. Data nodes are divided into node groups\(^9\). The number of node groups is calculated as:

\[
\text{Number of Node Groups} = \frac{\text{Number of Data Nodes}}{\text{Number of Replicas}}
\]

The partitions of data are replicated within the same node group. For example, suppose that we have a cluster consisted with 4 data nodes with replication factor of 2, so there are 2 node groups as shown in Figure 2.7. All the rows for table are sliced into partitions and each data node keeping the primary fragment for one partition and backup fragment for another partition. Query from MySQL server needs lots of network hops to the data nodes or in between data nodes in order to fetch the required data which would affect performance and scalability will be slashed down. So, minimizing the number of network hops would yield best performance in MySQL cluster query lookup. By default, Sharding is based on hashing a table’s primary key, however the application level can include a sub-key for improving performance by grouping data commonly accessed together in the same network partition, hence reducing network hops.

![Figure 2.7: The MySQL NDB cluster data partition](https://dev.mysql.com/doc/refman/5.0/en/mysql-cluster-nodes-groups.html)

2.1.3 Application nodes

The Application nodes are allowing connectivity from the application logic to the data nodes. There are several well-known APIs are presented to the application. MySQL provides a standard SQL interface which provides connectivity to all of the leading web development languages and frameworks. MySQL cluster supports whole range of NoSQL interfaces including Memcached, REST/HTTP, C++ (NDB-API), Java and JPA. MySQL server internally uses the low level NDB API to access the data nodes. The performance gain will be increased by accessing the data

\(^9\)MySQL Cluster Nodes, Node Groups, Replicas, and Partitions
https://dev.mysql.com/doc/refman/5.0/en/mysql-cluster-nodes-groups.html
nodes directly using the C++ NDB API. But, this would require more attention on
development and maintenance of implemented application. It is also possible to mix
SQL and NDB API based on application requirements, specifically, performance
critical workload handling can be implemented by using NDB API. Our Event
streaming shared library has used C++ NDB API to deliver best throughput for
Hops-YARN.

![Figure 2.8: Diversity of application connectivity](image)

2.1.4 Management nodes

The Management nodes are used to configure the cluster and provide arbitration
in the event of network partitioning. It also provides rich functionalities to admin-
istrators to monitor the cluster usage. It also provides administrative services for
the cluster. These include starting and stopping MySQL cluster nodes, handling
MySQL cluster logging, backups, and restoration from backups, as well as various
other management tasks.

2.1.5 MySQL NDB cluster event streaming

The rows of tables stored in different data-nodes in NDB are being updated by
transactions which are handled by multiple internal transaction coordinators. Each
transaction coordinator is starting, executing and committing transactions indepen-
dently in different data nodes. Concurrent transactions only interact where they
attempt to lock the same row. This design minimizes unnecessary cluster-wide
synchronization which is required by transaction coordinators to obtain consistent
point in time to commit transactions. This enables linear scalability of reads and
writes in NDB cluster.

The stream of changes made to rows stored at a different data node is written to
a local Redo log for node and system recovery. The change stream is also published
to NDB API event listeners, including MySQLD servers recording Binlogs which
2.1. MYSQL CLUSTER DATABASE

then will be used for cluster replication. Each data-node's stream comprises the row changes it was involved in, as committed by multiple transactions, interleaved in a partial order which cannot be used to create meaningful transaction object. This is because of all changes by transaction are not reflected in a single data-node’s stream.

The disseminated stream from each data node is called event streams as transactions are fundamentally update/delete/insert events on the database. These row event streams are generated independently by each data node in a cluster, but to be useful they need to be correlated together. During a system-wide crash recovery, all the data nodes need to recover to cluster-wide consistent state which contains only whole transactions existed at some point in time. One possible way to do is an analysis of the transaction ids and row dependencies of all recorded row modifications to find a valid order for the unified event streams. This process would add an extra overhead and non-deterministic time of recovery. NDB cluster uses a distributed logical clock known as the *epoch* to group large sets of committed transactions together to slash down this problem. The *epoch* atomically increases every 100 milliseconds by default and it can be represented by 64 bit number. So, each 100 milliseconds, a Global Commit Protocol which is known as distributed consensus protocol results in all transaction coordinators in the cluster mutually agreeing on a single point of time at which to change *epoch*. Figure 2.9 shows an example of *epoch* counter and corresponding event streams caused by several transactions. Epoch boundary has been defined by cluster-wide consistent point in time; the difference between commit of the first transaction in *epoch* $n+1$ and commit of the last transaction in *epoch* $n$.

![Figure 2.9: Transactions at different data nodes and possible event streaming](image)

Figure 2.9: Transactions at different data nodes and possible event streaming
CHAPTER 2. BACKGROUND

The fundamental property of \textit{epoch} is that, each \textit{epoch} carries zero or more committed transactions and every committed transaction belongs to one \textit{epoch}. This property was mainly used to build our Event Streaming library discussed in following next sections.

As we have discussed before, all the row event streams are produced independently by each individual data node in the cluster; it is impossible to use those individual events from the data nodes to create a meaningful transaction state. Once all this independent streams are combined and grouped by \textit{epoch} number, meaningful unified stream could be constructed. As epochs give only a partial order, one of the interesting facts about transaction ordering is that, within an epoch, row changes may be interleaved in any way, except that multiple changes to the same row will be recorded in the order they were committed. This means that applications that are constructing these transactions related objects should use some special flag to order them. Other than that, intra \textit{epoch} bounty transactions can be ordered as NDB API internally ordered the \textit{epoch} in sorted manner. For instance, as shown in Figure 2.10, T4 $\rightarrow$ T1(Happened before), and T7 $\rightarrow$ T5. However, we cannot infer whether T4 $\rightarrow$ T3 or T3 $\rightarrow$ T4. In \textit{epoch} 2, we see that the row events resulting from T1, T2 and T7 are interleaved. Figure 2.10 also shows the possible ordering of transactions in between \textit{epoch} boundaries.

![Figure 2.10: Merge events and ordering the transactions](image)

If an application has a strict requirement of transaction ordering in a \textit{epoch} interval, NDB API provides special function call \texttt{setAnyValue} which applications can pass any values (ex: atomic auto increment value) in NDB transaction operation. So, during an event processing, this value could fetch from streaming to order the transactions before delivered to the application.

As mentioned before, the row changes in data nodes are recorded by MYSQLD process for replication. Figure 2.11 represents non-cluster replication which master and slave servers are participating during the replication process. The MySQL supports other advanced type replication called multi master replication or circular replication where all the masters are replicating data among different cluster. Fundamentally MySQL replication process is asynchronous which means that replication process in the slave server can be started in a different time than the one taking place in master server. This would make replication more flexible and make sure that even if connection lost in between master and slave, replication will proceed.
2.2. HOPS YARN ARCHITECTURE

The Hops is a new next-generation distribution of Apache Hadoop, the Open-source software for storing enormous data sets across distributed commodity servers and then running distributed analysis applications in parallel over the data. Hops provide very flexible deployments of several big data analytics tools through a WebUI which can be used to manage projects, search, visualize DataSets, and interactively analyze data. The Hops-YARN is part of Hops platform which provides a distributed stateless ResourceManager, whose state is migrated to MySQL Cluster, in-memory distributed database. This approach makes the YARN highly available, with failover of a ResourceManager happening in a few seconds. Hops-YARN stores all its internal states in an external database, this opens up new possibilities for analyzing the performance of both the ResourceManager and applications submitted to it. The state information is passed through newly added Data Access Layer which is an interface for multiple data storage with YARN.
The YARN is based on event based architecture, and it is extremely difficult to predict the order in which each event will be handled. As a result, it is nearly impossible to decide in advance when the modification to the ResourceManager can be committed to the database without committing a partial state from which it would be difficult to recover. In order to find where to commit, we proceed as follow. Each time a RPC is received and saved to the database we create a thread safe object that contain a counter and that will be filled with all the ResourceManager variables modifications that have to be committed to the database. The counter is initialized with the value one and the object is passed with each event that is triggered by the RPC handler and recursively with each event that is triggered by one of these events handlers. Each time this object is passed with an event its counter is incremented by one. Each time an event has finished to be handled its counter is decrease by one. This way the counter counts the number of events that have not finish to be handled and that are related to the initial RPC. When the counter reach the value 0, it means that the entire event related to the RPC have been completely handled, as a result all the modification to the ResourceManager state due to the RPC have been made and can be committed. Each time a dispatcher has finished calling the handle function on an object it decrease the counter by one. If the counter does not reach 0, do nothing. If the counter reach 0, commit the modification to the database and remove the original RPC from the database.

2.2.1 Data Access Layer

The Hops YARN introduced new layer called Data Access Layer to interact with NDB cluster for store and retrieve the data due to licensee incompatibility. NDB cluster is licensed under GPLv2 and YARN is licensed under APLv2, we may not distribute the combined daemon using GPLv2 and APLv2 licensed material. Explicitly calls out that combination as not allowable for public distribution. Basically, this license incompatibility applies only when some Apache project software becomes a derivative work of some GPLv3 software, because then the Apache software would have to be distributed under GPLv3. ASF’s main requirement which all the Apache software must be distributed under the APLv2.

The main advantage of having DAL layer is the extendibility. Because, DAL is the abstraction to MySQL Cluster so that YARN’s ResourceManager can use a simple interface to persist and retrieve its state. A set of interfaces define the access parameters to the database tables such as table name and columns and methods performed on the database by the ResourceManager. The new set of definitions and implementations can be easily added this layer without changing any logic of YARN operation. Another set of classes provide the implementation of these interfaces by using ClusterJ. In addition, some classes work as adaptor classes between the objects represented by the database table and the ones in YARN.
2.3 ClusetJ

As the entire projects implementation is done in Java, a fast and reliable library would have to use to connect the application with the database. To make the best use of an immensely fast database such as MySQL Cluster, the application needs to be able to submit queries and retrieve results quickly. It would be inefficient if the application layer proved to be the bottleneck in the architecture. The outcome of these considerations is to use the Java library ClusterJ\textsuperscript{10} to accomplish transactions with MySQL cluster.

![Figure 2.12: Java access to MySQL Cluster](image)

Figure 2.12 explains in detail how applications are connecting through MySQL cluster using ClusterJ. ClusterJ provides a high performance library that is fairly easy to use due to its intuitive interface. It follows the same style of technologies such as Hibernate and Java Persistence API (JPA), all of which make use of the Domain Object Model DataMapper pattern. Essentially this means that an abstraction layer is inserted between business logic and domain objects. These objects are directly mapped to the database tables which are made possible by adding special annotations to the Java classes. This makes it easy for the programmer to persist Java classes members without having to write complex insert/update SQL-like queries.

However, a drawback of ClusterJ is its lack of support for inheritance, something that did not affect the architecture of this project. Furthermore, ClusterJ supports transaction functionality and different row locking levels, concepts fundamentally necessary for the distributed ResourceManager architecture. ClusterJ does all that while being able to demonstrate very high throughput compared to other persistence APIs such as JPA\textsuperscript{11} as ClusterJ for an insert statement was benchmarked as 10 times faster. Figure 7 shows the ClusterJ session factory concepts in detail.

\textsuperscript{10} MySQL :: MySQL Cluster API Developer Guide :: 4.2.2 Using ClusterJ
\url{https://dev.mysql.com/doc/ndbapi/en/mccj-using-clusterj.html}

\textsuperscript{11} Using JPA with MySQL Cluster
\url{https://dev.mysql.com/doc/ndbapi/en/mccj-using-jpa.html}
CHAPTER 2. BACKGROUND

1. SessionFactory: There is one instance per MySQL Cluster instance for each Java Virtual Machine (JVM). The SessionFactory object is used by the application to get hold of sessions. The configuration details for the ClusterJ instance are defined in the Configuration properties which is an artifact associated with the SessionFactory.

2. Session: There is one instance per user (per Cluster, per JVM) and represents a Cluster connection.

3. Domain Object: Objects representing the data from a table. The domain objects (and their relationships to the Cluster tables) are defined by annotated interfaces.

4. Transaction: There is one transaction per session at any point in time. By default, each operation (query, insert, update, or delete) is run under a new transaction. The Transaction interface allows developers to aggregate multiple operations into a single, atomic unit of work.

Figure 2.13: ClusterJ interface annotations

Hops Data access layer has several configurable parameters for ClusterJ library. Following configurations variables are widely using by Hops echo systems.

- com.mysql.clusterj.connectstring: Address of management server of MySQL NDB Cluster.
- com.mysql.clusterj.database: Name of the database that contains the metadata tables.
- com.mysql.clusterj.connection.pool.size: This is the number of connections that are created in the ClusterJ connection pool. If it is set to 1 then all the sessions share the same connection; all requests for a SessionFactory with the same connect string and database will share a single SessionFactory. A setting of 0 disables pooling; each request for a SessionFactory will receive its own unique SessionFactory. We set the default value of this parameter to 3.
2.4. RELATED WORK

- com.mysql.clusterj.max.transactions: Maximum number transactions that can be simultaneously executed using the clusterj client. The maximum support transactions are 1024.

- io.hops.metadata.ndb.mysqlserver.host: Address of MySQL server. For higher performance we use MySQL Server to perform a aggregate queries on the file system metadata.

- io.hops.metadata.ndb.mysqlserver.port : If not specified then default value of 3306 will be used.

- io.hops.metadata.ndb.mysqlserver.username : A valid user name to access MySQL Server.

- io.hops.metadata.ndb.mysqlserver.password : MySQL Server user password

- io.hops.metadata.ndb.mysqlserver.connection_pool_size : Number of NDB connections used by the MySQL Server. The default is set to 10. Session Pool For performance reasons the data access layer maintains pools of preallocated ClusterJ session objects. Following parameters are used to control the behavior the session pool.

  1. io.hops.session.pool.size: Defines the size of the session pool. The pool should be at least as big as the number of active transactions in the system. The default value is set to 1000.

  2. io.hops.session.reuse.count: Session is used N times and then it is garbage collected. The default value is set to 5000.

2.4 Related work

Other frameworks have recognized the same limitations in the standard Hadoop architecture, and have proposed/implemented new frameworks which can be compared to YARN architecture. The most closely fit to YARN are: Omega[10] and Mesos [11], maintained and used by Google and Twitter respectively. All these systems are sharing a common underlying principal and the high-level goal of improving programming model flexibility, scalability and performance enhancement. The primary difference between all these frameworks are around their design priorities and how they approach scheduling work.

2.4.1 Mesos

The Mesos, a platform for sharing clusters between many different cluster computing frameworks, such as Hadoop and MPI\textsuperscript{12}. Mesos accomplishes data locality by

\textsuperscript{12}Message Passing Interface

\url{https://computing.llnl.gov/tutorials/mpi/}
sharing resources in a fine-grained way. In order to achieve these qualities, Mesos introduced a two-level scheduling execution called resource offers that Mesos determines which resources are available, and it makes offers back to a framework. This model is considered a non-monolithic model because it is a "two-level" scheduler, where scheduling algorithms are pluggable. Mesos allows multiple scheduler algorithms to be performed, each of them may have different functionalities which offers to accept or decline, and can accommodate thousands of these schedulers running multi-tenant on the same cluster. Since Mesos has two level scheduler, each of the scheduler, have them access to control arbitrate make decision above which scheduler have access to which resources. So, in the Mesos world, Mesos master dynamically decides how many resources to offer each framework, while frameworks decide which resources to accept and which computations to run on them. The resource Offers come in, and the framework can then execute a task that consumes those offered resources. Or the framework has the option to decline the offer and wait for another offer to come in.

Even though, the Mesos and YARN both have similar scheduler architecture, But, there are mainly two differences among these systems. First, YARN has a request-based approach, whereas Mesos is a resource offer-based ResourceManager. Second, Mesos uses framework schedulers for inter-job scheduling, whereas the YARN uses per-job optimization through ApplicationMaster. But, Mesos per-job ApplicationMaster has a significantly lower overhead compared to the YARN. The YARN ApplicationMaster is dynamically changing resource consumption, based on function of the containers it receives from the ResourceManager and ApplicationMaster requests are late-binding which is process spawned is not bound to the request but to the lease. The conditions that caused the ApplicationMaster to issue the request may not remain true when it receives its resources. But, monolithic schedulers are having problems such that, hard to diversity, code growth and scalability bottleneck.

2.4.2 Google Omega

The Omega’s main design principal is to provide distributed, multi-level scheduling known as shared state optimistic concurrency control scheduling approach. This approach would yield higher scalability, but it is preventing the scheduler to enforce global properties such as fairness/capacity which is the main advantage of monolithic schedulers like YARN. Omega has a description of state of machines such as how busy there are, how much resource are available etc. The scheduler has a local copy of that state in memory; so that we can operate quickly make decision of information that is the current best estimate of cell. Once the scheduler has taken a decision, then it pushes that decision down to centralized state that everybody has access to.

Let’s see an example scheduling strategy of the Omega. As shown in Figure
2.4. RELATED WORK

Figure 2.14: Cluster resource is statically partitioned

2.14, each scheduler will get own local copy which is the mimic of what stand in the central state, the first thing the scheduler does is to update its own local copy and send the delta to central state. In order to increase the performance, Omega only sends the difference to central state. Same as second scheduler also get local copies. When the new work arrives, according to local copies, each scheduler decides the resource allocations from the entire cluster resource. The right side scheduler is allocating some resources in its local copy and the same time, left side scheduler is also allocating some resources in its local copy in parallel. In this particular case, they overlap, so what Omega does, is to detect the conflict in central state. Subsequently, one of the scheduler loses and one will win. It is like transaction atomic operation where all the changes are made or none of them are happened. Right hand side scheduler refresh its state and try again, this approach would give greater performance when machine have large number of resources, but, when cluster is getting smaller, the rate of conflict will increases, But, all the decision are being made is quite fast, so the matter of time scheduler spending waiting time will be very short . The churning is also another factor which will impact the decision making in this approach. But, results have shown that this approach has made huge impact on Google production cluster with vast number jobs.
2.5 Summary

In this chapter, we discussed the architectures of YARN and Hops YARN in detail and as well as an in-memory, replicated, distributed database management system MySQL Cluster. We presented the related knowledge on existing frameworks which are almost related to YARN functionalities. Finally, we introduced ClusterJ library which is used by Hops YARN for persisting state information into MySQL NDB cluster.
Chapter 3

Design

The purpose of this chapter is to show the model that was developed and the architecture designed for the Hops distributed YARN. Section 3 provides the reader with insight details of NDB Event streaming and the library we implemented by absorbing ideas from MySQL NDB cluster replication. Section 4 explains the extended version of YARN Scheduler Load Simulator that we have modified to support over the network simulation in two different modes such as standalone simulation and distributed mode simulation in order to evaluate Hops distributed YARN.

3.1 Architecture of Event Streaming library

The MySQL cluster’s epoch-driven Event API which provides streaming functionalities inspired us to build a high frequent event stream processing library to improve the scalability of Hops YARN. The purpose of this library to efficiently handle and process event streaming, and converting to Java objects that Hops distributed YARN can process. Further, the library should be versatile that can be dynamically loaded in each Hops distributed YARN components as a normal native library loading with minimal source code modification in YARN. Hops YARN derives from the Hadoop and, as such, it is available under the Apache version 2.0 open source license mode, and same time our C++ library using MySQL NDB cluster API functions which are available under the GPL version 2.0 licensing model. Our native library that implements the C++ API for MySQL Cluster, however, is licensed under the GPL v2.0 model, but static linking of Apache v2 code to GPL V2 code is allowed, as stated in the MySQL FOSS license exception. The FOSS License Exception permits use of the GPL-licensed MySQL Client Libraries with software applications licensed under certain other FOSS licenses without causing the entire derivative work to be subject to the GPL. As shown in Figure 3.1, the Hops distributed YARN calls Java API functions which allows native library to execute necessary functions implemented using NDB Event API.
Figure 3.2 shows the schematic diagram of the newly implemented event processing library. The design consists of single Event Polling thread and one or more dispatcher threads, both respectively pooling events from NDB Event API buffer and creating Java objects for the Hops distributed YARN. The library could be operated in two different set-ups; single threaded and multithreaded. The single thread processing has only one dispatcher thread and multithreaded processing has several number dispatcher threads. The dispatcher threads should follow a basic protocol which is orders the events prior to dispatch. As any row event recorded in \( \text{epoch } n+1 \) logically happened after every row event in \( \text{epoch } n \), independent thread dispatching is restricted, and all threads are bounded in to a specific condition where all the threads should communicate prior to dispatch Java objects. The multithreaded design addresses the following limitations that have been taken into consideration to increase the dispatching throughput of the library.

1. Each epoch carries zero or more committed transactions. Since ordering the event is the key point, allowing multiple threads to process each individual event in single epoch would increase an extra overhead by blocking each other threads during the dispatch.

2. Need an efficient communication mechanism among threads to avoid frequent locking of common data structures inside the library.

The first problem is solved by introducing per \textit{epoch} transaction processing by each threads. When all events for a given epoch have been received, each \textit{epoch} transactions are placed into dispatcher thread queue for further processing. Queue placement of each \textit{epoch}’s transaction objects are arranged into round-robin method. For instance, if the library is configured to run 5 threads, 1\textsuperscript{st} \textit{epoch} transaction objects shall be placed into queue 1, \((1\% 5=1)\), and it will be taken by

\footnote{modulo operation} \url{https://en.wikipedia.org/wiki/Modulo_operation}
3.1. ARCHITECTURE OF EVENT STREAMING LIBRARY

Figure 3.2: Architecture of C++ Event streaming library

thread 1 for process and dispatch. 2nd epoch transactions objects shall be placed into queue 2, \(2\%5=2\), and it will be taken by thread 2 for process and dispatch. Likewise, epoch 6th transaction objects shall be assigned to queue 1 to and taken by thread 1 for process and dispatch. This approach gives more freedom to threads to process the data in parallel but not in dispatching. This means, random dispatching is prohibited as applications receiving Java objects are not aware about the ordering, and handling the ordering in application level would be more complicated. So, second dispatcher thread is waiting for the first dispatcher thread to dispatch, third dispatcher thread is waiting for second dispatcher thread to dispatch, and so an eventually, all threads are waiting for their neighbours to dispatch first. This is solved by thread signals, which each thread uses signal to notify neighbouring thread to 'Go and dispatch'. In this approach, we are making sure that threads are dispatching in consistent order as they arrived into the Event polling thread. Further, thread signal is reduced thread communication overhead and improved the synchronization among threads as they only need to communicate among neighbors.

Initial evaluation has demonstrated the library’s throughput performance did not increase with number of threads. This evaluation was carried by generating a larger load with 8000 transactions per epoch for the library processing. Throughput has been measured with several number of thread configurations. For instance, 8000 transactions objects have taken nearly 800 ms to dispatch which resulted 10 Java objects/ms. Normally, we would expect the higher throughput when numbers of threads are processing the events in parallel. The profiling result has shown that dispatching process known as JNI\(^2\) invocation has taken more time than the processing C++ NDB events row data. Figure 3.3 shows the time distribution for

\(^2\)Developer Android -JNI Tips
http://developer.android.com/training/articles/perf-jni.html
CHAPTER 3. DESIGN

Each method. This clearly indicates, 80% of time is taken by JNI invocation calls and parallel data processing does not help to boost the throughput. The architecture had been gone through some modification, such that, instead of parallelizing the data processing operations which is too small and negligible according profiling, we processed JNI invocation calls concurrently as shown in 3.4.

As shown in Figure 3.5, in multithreaded environment, each thread is creating thread specific object from Java class provided by Application programmer and keeping the reference in native side. Whenever a dispatcher thread receives the job from Event polling thread, dispatcher threads are parallely building Java objects by earlier reference created by the library. All the newly created Java objects are placed in Java heap memory and memory locations of the objects are passed to the library. In this approach, all the threads are concurrently building Java objects and waiting for their neighbours thread for signal as we discussed above. Now, imagine a situation where all the threads are concurrently building Java objects, how do they start dispatching into application? Initial firing signal is sent from Event polling thread and the rest of the invocations are being performed by dispatcher thread sending signal among neighbours. So, in this way, a thread can build a Java objects without interfering with about other threads and keeping the built objects inside the thread local data structures. This approach enables to avoid frequent memory allocation in native side, and this responsibility is given to Java garbage collectors once native side pushed the Java objects into the application.
3.2. HOPS DISTRIBUTED YARN

A Working model of the Hops distributed YARN is shown in Figure 3.6. Each NodeManager is associated with a ResourceTrackers in a load balanced way. The NodeManagers send their heartbeats to the ResourceTrackers they are associated to. When the ResourceTracker receives a heartbeat, it handles it, creates an answer based on its in-memory view of the cluster, sends this answer to the NodeManager and commits to the database modified by the heartbeat. Then, the database streams the updates to the Scheduler which then will be processed by Event Streaming library and updates the scheduler in-memory view of the cluster. When the Scheduler receives a request for resources, either from a client or from an ApplicationMaster, it handles it, takes resource allocation decisions based on its in-memory view of the cluster, answers to the request and commits to the database the new state of the affected resources. When the commit is done, the database uses the NDB event API to stream the updates to the ResourceTrackers. The ResourceTrackers use this information to update their in-memory view of the NodeManagers they are associated to. These updated views are then used to answer to the next NodeManagers heartbeats.

The YARN ResourceTracker and the Scheduler are communicating using events through concurrent queue called nodeUpdateQueue. When the ResourceTracker receiving a heartbeat from a NodeManager N, it checks the queue, this event will
then be handled by the scheduler. If it is not the case, the schedule updates the
view of N but does not send any event to the scheduler. In order to provide the
same functionalities as Haop YARN, the Hops distributed YARN is using two
types of events to notify the Scheduler, when to process the heartbeat and when
to update the view. According to event type dispatched by the Event Streaming
library, scheduler will carry out view update or process the event to update its
internal state.

Since ordering the updates is really important in the Scheduler side, these two
types of events have an atomic counter which increases every time the Resource-
Trackers handle an event. The value of this counter is committed to the database
at the same time as the resources update. This counter value will be used by the
Event Streaming library to order the events before to dispatch to the Scheduler.
As each NodeManager is associated to one ResourceTracker, ordering by this value
guarantees that for each NodeManager, the event will be processed in the correct
order, for example; the NodeManagers’s registration updates will be handled by the
scheduler before any heartbeat updates.

Once events are fetched from nodeUpdateQueue by the Scheduler, it processes
the events one by one and then it commits the database including nextheartbeat
flag as true. This true value indicates that scheduler has successfully handled the
3.2. HOPS DISTRIBUTED YARN

Event sent by corresponding NodeManager. This update then will be streaming by NDB API and captured by Event Streaming library located in ResourceTrackers. Unlike scheduler, ResourceTracker’s Event Streaming library doesn’t order the transaction objects belongs to epoch as ResourceTrackers only just updating the NodeManager’s nextheartbeat flag and, order there are getting updated is not important. Since each NodeManager is associated with a ResourceTracker in a load balanced way, each ResourceTracker will be receiving NodeManagers updates which they are not intended. Current design of Event Streaming library let them pass through in Application level for further filtering. Since this approach would increase an extra overhead in YARN ResourceTracker, this filtering process will be transferred in native level to discard updates in future design.

Using the event API is efficient and implies a low charge on the Scheduler and the ResourceTrackers. When number of the NodeManagers increase in the cluster, heartbeat messages would be drastically increased. During the evaluation Event Streaming library has handled high frequencies of data from multiple ResourceTrackers and scheduler, as a result, Hops distributed YARN is providing more scalable and more responsive service.
3.3 YARN scheduler load simulator

The Scheduler Load Simulator (SLS) is an in-built tool shipped with the Hadoop package which is capable of simulating large-scale clusters and applications loads in a single machine. Without having real clusters, developers would be able to measure scheduler performance by simulating loads from this tool. Figure 3.7 illustrates the implementation architecture of the simulator. One of the major advantages of Scheduler Load Simulator is to simulate real Yarn ResourceManager removing the network factor by simulating NodeManagers and ApplicationMasters via handling and dispatching NodeManager/ApplicationMaster’s heartbeat events from within the same JVM.

There are two different methods, which Scheduler Load Simulator is getting simulation feeds for simulations; Apache Rumen \(^3\) format and own input traces. The Hadoop has been using Rumen as a data extraction and analysis tool. Rumen analysis JobHistory logs to extract useful information and keeping them in an easily-parsed format. Once the SLS initialized with an input workload traces in any of these two methods, for each NodeManager and ApplicationMaster, the simulator builds a single thread task to simulate. All NodeManager/ApplicationMasters simulators are run in a thread pool. The simulator reuses YARN ResourceManager, and builds a wrapper out of the scheduler. The Scheduler Wrapper can track the scheduler behaviours and generates several logs, which are the outputs of the simulator and can be further analysed. For instance, node heartbeat request are initially sending to Scheduler Wrapper and getting time stamped. Then, requests will be forwarded to the actual scheduler for the processing. Once a request is processed, scheduler wrapper is time stamped again and recording the node update time cost in a relevant files.

![Figure 3.7: Component diagram of YARN Scheduler Load Simulator](image)

The current architecture of Scheduler Load Simulator has limitations of over the network simulation as it has been implemented to directly use ResourceMan-

\(^3\)Apache Rumen
http://hadoop.apache.org/docs/r1.2.1/rumen.html
3.3. YARN SCHEDULER LOAD SIMULATOR

ager instance. So that NodeManagers and ApplicationMasters are directly calling ResourceManager instance to communicate instead of RPCs. This prevents us to do an evaluations in following ways.

1. Scalability performance of Hops distributed YARN: In order to measure the scalability of Hops distributed YARN, load must be generated to the multiple ResourceTrackers and scheduler. This could only be achieved by separating the load generation logic from existing Scheduler Load Simulator.

2. Multiple instance of simulator [12]: Single instance of simulator is inadequate to generate higher load as NodeManager instances are handled by pool of individual thread task. In fact, Number of NodeManager simulation can be increased by lifting up the number of threads in thread pool. Due to thread context switching, generating a heartbeat in a configured heartbeat interval is impossible when number of threads exceed the limit.

Both of these limitations have been taken into consideration to extend existing Scheduler Load Simulator to support above mentioned requirements. The new load simulator is designed to simulate in three different modes such as standalone which is shown in Figure 3.8, distributed mode which is shown in Figure 3.9 and standard mode which current Hadoop YARN is implemented. The NodeManager and ApplicationMaster are connecting to ResourceTracker service and ApplicationMaster service respectively to send data over the network. After several rounds of experiments with the simulator, we figured out the bottleneck which limits the simulator to simulator more than 5000 NodeManagers with the experimental load. This limitation has been solved by creating multiple instance of simulator, each simulators divide the load among them and simulate to designated ResourceTrackers and scheduler.

Simulator has two main entities; Unmanaged applications masters and NodeManagers.

1. Unmanaged ApplicationMasters : The ResourceManager is directly manages the ApplicationMaster of the YARN application. To me more specific, the ResourceManager stores and manages the ApplicationAttempt object for each application attempt of the YARN application. Subsequently, the ApplicationMasterLauncher service in the ResourceManager creates an AMLauncher to launch the ApplicationMaster. After that, ApplicationMaster negotiates resource allocation with ResourceManager and launches tasks to the actual work. Those procedures are not possible in a simulation world as ApplicationMasters are not managed by ResourceManager. The Hadoop YARN

35
facilitates a new feature called unmanaged ApplicationMaster which the ApplicationMaster that is not launched and managed by the ResourceManager, and is managed outside the cluster. This feature has been used in the simulator which creates an individual ApplicationMaster simulator to submit an application to ApplicationMaster service. Each ApplicationMaster is handled by dedicated unmanaged ApplicationMaster launcher which will be executed by separate Java process. Each ApplicationMaster launcher processed will be killed by the simulator during the application tear down process to avoid unnecessary processes overhead.

2. Nodemangers: All the simulated NodeManagers are assigned to an individual thread in a thread pool. According to thread pool configuration and heartbeat time interval, NodeManagers are sending heartbeat to ResourceTrackerService.

3.3.1 Standalone simulator

As we have seen in Figure 3.8, standalone mode is a single instance of simulator generating a load to ResourceTrackers and Scheduler. Since un-managed application launchers are separate processes, each of them should use any of interprocess communication mechanism to update NodeManager’s internal data structures which then will be used by NodeManager to send a heartbeat. Since Remote Method Invocation performs the object-oriented equivalent of remote procedure calls (RPC), with support object serialization to marshal and un-marshal parameters, Java RMI has been chosen for communication from ApplicationMasters. The experiments have demonstrated that 5000 NodeManagers is the Maximum that standalone simulator can simulate with current application load.

3.3.2 Distributed simulator

Creating multiple independent standalone simulator is not going to work in simulation environment as random ApplicationMaster container allocation by the scheduler. For instance, we have two identical simulator simulating independent NodeManagers, but when first time application register with scheduler, the scheduler is randomly allocating ApplicationMaster container in any of the active NodeManager in the cluster. So, it is possible that Simulator 1 may get response from scheduler which is actually belongs to the NodeManager in simulator 2. For this reason, there should be a communication in between distributed simulators. As shown in Figure 3.9, Simulator 1 is dedicated to applications and half of the NodeManager load and simulator 2 is only simulating rest of the NodeManagers load. So, ApplicationMasters are updating Simulator 1 data structures, and it may update Simulator 2’s data structures depends on response from the scheduler.

---

4 Java Remote Method Invocation

3.4 Summary

In this chapter, we discussed about MySQL cluster replication using Event Streaming, proposed architecture of Stream processing library and distributed scheduler load simulator. Event stream processing library is designed to generate large Java objects as fast as possible from incoming high frequent event streaming from NDB.
cluster data-nodes. Finally, we extended and added new features for Hadoop Scheduler Load simulator to support over the network simulations and large cluster job simulations.
Chapter 4

Implementation

Based on the design, library was implemented as a multithreaded event processing shared library. Section 4.1 looks into the implementation details of the multithreaded more thoroughly by explaining internal communication mechanism and multithreaded JNI techniques used to achieve higher throughput. Section 4.2 presents integration of the implemented library with YARN including both entities Resource-Trackers and scheduler.

4.1 Multithreaded implementation

The implementation of the Event streaming library mainly consists of multi-thread queue communication and Java Native Interface (JNI) invocations. This section begins with the fundamental of threads, multithreaded implementation and design considerations of thread communication. The POSIX thread library (pthread) ¹ has been used for all threading support inside the Event Streaming library.

The major objective of using the POSIX thread library is to reduce the overhead and complexity of our Event Streaming library. The POSIX thread libraries let us allows to build a very robust multithreaded parallel processing applications in shared memory multiprocessor architectures. Another advantage of pthread is the compatibility with Java threads, which allows us to attach all our native pthread threads directly to the Java threads. When the library boots up, a configured number of dispatcher threads which is designated for Java object creation and dispatch are attached to Java thread and will not be detached until application shut-down the library. Because of continuous attach and detach native threads, the JVM must continually associate threads with Java objects. Some JVMs may re-use threads, or temporarily cache mappings to improve performance, but attached one would give better and more predictable behavior if we do not rely on the JVM to do it for us.

¹POSIX Threads Programming
https://computing.llnl.gov/tutorials/pthreads/
Apart from that, all other implementations are using low-level C/C++ functions for string manipulation, memory management, etc., rather than other high-level wrapper C/C++ function libraries to avoid unnecessary complexities.

The library initializes itself by subscribing to tables and columns which library is configured to listen. Upon successful subscription, events are started to flow into the library. Initially, incoming events are converted to C++ data structures as NDB row format events are unable to process inside the library. Because, the API function which converts stream events to NDB record attributes allocates memory for an NDB record attribute object that is later used to obtain the attribute value. This storage is managed by the NDB record instance; it is freed when the operation is released, such as at transaction close time; any data written here that we wish to preserve should be copied elsewhere before this freeing of memory takes place. For instance, database VARCHAR records are given by C++ char pointers to calling functions, so that applications should take responsibility to copy the content from given memory location to application’s own data structures. In order to do that, the Event streaming library has a predefined record structure to hold all the table related information which can be easily manipulated and processed by dispatcher threads. All dispatcher threads have dedicated queue to receive the C++ stream object for processing and dispatch.

4.1.1 Message passing queues

The concurrent FIFO queues are widely used in all multithreaded implementations. We have introduced a new memory optimized and efficient queue implementation mechanism for multithreaded communication. The queue implementation is composed of three different entities; Queue holder, Queue container and queues. The queues are logical representation of the linked-list.

The queue holder is composed of three different types of queue sections such as owner queue, common queue and foreign queue. The queue containers are the actual transporters of the messages in between threads. The queue container class representation is shown in following code snippet 4.1. Since QueueContainer class is accepting void pointer as a parameter, developers can simply create user-defined data structures to encapsulate the actual messages. Here, pNext pointer will be used to link the data structures in FIFO format.

Having three types of queues inside the single queue holder decreases the frequent thread locking which gave us a higher performance during the high frequent event processing. The algorithm uses two separate locks to distinguish synchronization for queuing and dequeuing threads. The locking will be taking place in two places where owner’s queue shifts the message to the common queue and common queue is shifts the message to the foreign queue. If the consumer thread is busy doing processing, the producer thread can transfer the messages to common queue with
4.1. MULTITHREADED IMPLEMENTATION

a single lock without any interruption to consumer thread. This approach enables the larger message transfer to queues with minimal thread synchronization.

When the application starts, the pre-allocated container pool will be created and responsibility of maintaining them would be assigned to owners and foreign queue. Every processed C++ NDB objects have been transferred by selected free containers from container pool. The container pool has pre-allocated nodes in order to avoid frequent memory allocation and deallocation which can result in performance degradation. Because dynamic memory allocation tends to be non-deterministic; the time taken to allocate memory may not be predictable and the memory pool may become fragmented, resulting in unexpected allocation failures. After several tests, the maximum number of pre-allocated containers have been obtained which is totally depends on number of active threads in the library.

The Figure 4.1 represents the memory allocation of each container node that is residing in the queue. So, Queues are actually carrying a memory location once it has been allocated by Event polling thread. The Event polling thread is polling every 1 ms from NDB Event buffer and reading the row NDB Event records from the buffer. Once it has been polled from buffer, the memory will be deallocated or overwritten by fresh events by NDB events. In order to handle this situation, we created predefined C++ struct to hold all the subscribed tables related information. Whenever Event polling thread wants to send a C++ object to dispatcher thread, the new container will be created with C++ NDB object pointer. This struct pointer will be assigned QueueContainer void pointer to pass through the queue.

Figure 4.1: Communication queue between Event polling thread and JNI dispatcher thread

The conventional way of processing messages from queue is continuously polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity.
CHAPTER 4. IMPLEMENTATION

A condition variable is a way to achieve the same goal without polling. To avoid such a problem, we used pthread condition signal mechanism to notify consumer threads, when to process the data. Condition signal provides yet another way for threads to synchronize, and allow threads to synchronize based upon the real value of data meanwhile conventional mutexes implement synchronization by controlling thread access to data. Each time Event polling thread is sending wake-up signal to all the dispatcher thread to process the data. If the dispatcher thread is busy and couldn’t able to process the data; the total memory configured for dispatcher queue is grows. This situation also handled by condition signal from dispatcher thread 'stop sending data' to Event polling thread until queue size become configured size.

Listing 4.1: Queue container class

```cpp
class QueueContainer
{
public:
    QueueContainer(void *) ;
    QueueContainer() ;
    virtual ~QueueContainer()
private:
    void *pData ; //Pointer to any object memory location
    QueueContainer *pNext // Pointer to next container;
};
```

4.1.2 JNI invocation

The Second part of Event Streaming library is using Java Native Interface known as JNI\(^2\) to interact Hops distributed YARN components. The JNI is a native programming interface which is a part of Java SDK. The JNI invocation API can be used to embed a JVM into native application, thereby allowing programmers to call Java code from within native code. It is vendor-neutral and has support for loading dynamic shared libraries.

The JNI defines two key data structures; **JavaVM** and **JNIEnv**. The JavaVM provides the "invocation interface" functions, which allow us to create and destroy a JavaVM while the JNIEnv provides most of the JNI API functions. All the native functions are receive a JNIEnv pointer as the first argument.

The Event streaming library provides two main JNI API functions; **startEventAPISession()** and **teardownEventAPISession()**. The library is initialized by YARN ResourceManager starting services by passing JNIEnv pointer to library. The JNIEnv is used for thread-local storage. So, we cannot share JNIEnv pointer in between threads. Our multithreaded library initially passing JavaVM pointer to all threads to obtain a own JNIEnv pointer, so that they can operate in parallel to create necessary Java objects.

\(^2\)developer android

4.2. YARN INTEGRATION

All the dispatcher threads attach themselves to the JVM during the initialization phase and each of them creates a thread specific object for parallel object building that we have discussed in previous sections. When an application issues a `teardownEventAPISession()` function call, first all the dispatcher threads call `DetachCurrentThread` to free JNIEnv pointers and other references that threads were keeping inside. Then upon successful termination of dispatcher threads, Event polling thread will be closed down.

The next important JNI functionalities which has been used is the Event streaming library and the local and global references. In most cases, each function argument passed to native method, and nearly every object returned by a JNI function is a local reference, which means that it is only valid for the current period of the current method in the current thread. Once native methods returned, and object given by native method is not valid if it is local reference. For instance, the functions using `jobject`, `jclass`, `jstring` should get a new global reference by invoking JNI `NewGlobalRef` functions. Since most of the variables coming from streaming are C++ string, our library frequently calling new global reference function to create Java String. In order to avoid memory leak in native side, all these new global references should be let go after some time, so that Java garbage collectors can deallocate the memory. Otherwise Java garbage collector assumes, somebody has reference to newly created object and can not free the memory.

We used an optimization in native side by caching Java class `fieldId` and Java `methodId` when library initialize. As long as class is loaded in the memory, address of class `fieldId` reference and `methodId` reference are static and can be used without fetching every time there is dispatching object to Java side. This approach has eliminated the frequent memory lookup time during the higher frequent Java object building.

4.2 YARN integration

Upon successful library loading by Hops distributed YARN ResourceTrackers, and scheduler, Event streaming library reads special classes dedicated to the ResourceTrackers, and scheduler for JNI dispatching. These classes have all Java methods that has to be filled from native library. If the native thread is directly calling and processing Java Objects into the application level, native queue will be simply growing up, and it will affect the overall throughput performance. In order to avoid this situation, native JNI dispatcher threads are pushing the Java objects into concurrent queue which is maintained by scheduler and ResourceTrackers inside. In this approach, native threads can push the data into the concurrent queue as fast as possible without waiting for Java processing thread.
CHAPTER 4. IMPLEMENTATION

4.3 Distributed load simulator

As we have learned in the previous system design section, the distributed load simulator has two main components: NodeManager simulator and ApplicationMaster simulator. All the running ApplicationMasters are communicating to NodeManager through RMI channels to share launched ApplicationMaster containers, job containers to execute jobs and completed containers data structures. These individual ApplicationMaster process are update thread-safe data structures inside the simulator which then will be processed by NodeManagers.

Listing 4.2: Interface class

```java
public interface AMNMCommonObject extends Remote {
    void cleanupContainer(String containerId, String nodeId) throws RemoteException;
    boolean isNodeExist(String nodeId) throws RemoteException;
    void addNewContainer(String containerId, String nodeId, String httpAddress,
                          int memory, int vcores, int priority, long lifeTimeMS) throws RemoteException;
    void decreaseApplicationCount(String applicationId) throws RemoteException;
}
```

The above 4.2 remote interface that defines the methods the ApplicationMasters can invoke on the NodeManagers which are handled by single Java process. This interface itself supports four methods: cleanupContainer, isNodeExist, addNewContainer and decreaseApplicationCount. Whenever the Scheduler allocate new container to the ApplicationMaster, addNewContainer function will be called to inform NodeManagers. Before it invoke addNewContainer function, ApplicationMasters are checking corresponding NodeManagers are existing in their own machine or in remote machine using isNodeExist. Depending on the result, the ApplicationMasters choose the remote connections to update container information at NodeManagers. This interface will be throwing an exception if the request is malformed for any reason.

The example usage of addNewContainer and isNodeExit is shown in following 4.3 code snippet.

Listing 4.3: ApplicationMaster adding new container

```java
Container container = response.getAllocatedContainers().get(0);
if (primaryRemoteConnection.isNodeExist(container.getNodeId().toString())) {
    primaryRemoteConnection.addNewContainer(
        container.getId().toString(),
        container.getNodeId().toString(),
        container.getNodeHttpAddress(),
        container.getResource().getMemory(),
        container.getResource().getVirtualCores(),
        container.getPriority().getPriority(), -1L);
} else {
    secondryRemoteConnection.addNewContainer(
```
4.4 Summary

In this chapter, we discussed about various implementation aspects which we have been used to achieve our goals. Specifically, how multiple threads resided inside the Event Streaming library communicate YARN Java classes through JNI is elaborated in detail. We found out having multiple independent JNIEnv pointer and building Java objects in parallel would give a tremendously higher throughput and avoid unnecessary memory allocation at native side. All the components that have been used to implement the Event Streaming library has implemented from scratch, and all are using standard C/C++ data structures and avoid any external libraries. (Ex: boost\textsuperscript{3})

\textsuperscript{3}boost C++ libraries

http://www.boost.org/
Chapter 5

Evaluation

In this section we evaluate both experimentally and analytically the Event Streaming library and Hops distributed YARN. All the experiments were performed on nodes behind a single 10 Gbit switch, where the network round trip time between any two nodes is in single digit millisecond range. The cluster consist of 9 nodes which all are interconnected with 10 Gbit switch have following configurations. Each machine has an Intel(R) Xeon(R) processor with 24 cores at 2.6GHz, 250GB of RAM and Oracle JRE version 1.7.0_79 installed on the 64-bit version of CentOS 6.6.

5.1 NDB cluster deployment

Figure 5.1: MySQL NDB cluster deployment

Figure 5.1 shows the deployment of MySQL NDB in the cluster. NDB cluster is composed of 4 data nodes and one management node which are denoted as ndbd, ndb_mgmd respectively. The machine named in bbc7.sics.se is running management server and bbc4.sics.se, bbc5.sics.se, bbc6.sics.se, bb7.sics.se are running data nodes daemons. NDB cluster is configured to have replication factor 2, and the DataMemory which defines the amount of space available for storing database records, specifically memory allocated by DataMemory which is used to store both the actual records and indexes was set to 5000M.
5.2 C++ Event streaming library throughput evaluation

Our main goal in this evaluation is to find out the maximum throughput the C++ Event streaming library can provide to the ResourceTrackers and scheduler in Hops distributed YARN. This is needed to make sure the implementation exhibits the desired behavior of the system. Only after the system managed to successfully provide expected throughput, we proceeded with the Hops distributed YARN evaluation described in the next sections.

The above deployment of NDB cluster with 4 data nodes did not allow us to generate enough streaming input to evaluate the throughput limits of our Event Streaming library. Even though, there were several clients parallely executing transactions to maximize the number transaction per epoch, the result was either a crash of NDB or a low numbers of transactions per epoch. We implemented a load generator to overcome this limitation which generate the same input as NDB streaming to our C++ Event streaming library. One of the main advantage that generator delivers is to provide consistent streaming feed to our library, and also provide accurate measurements for different thread configurations.

In order to generate the same application scenarios that Hops distributed YARN is expected to process; a scenario simulating transaction persisting the state of 2 Java classes, a scenario simulating transaction persisting the state of 5 Java classes, and the last scenario simulating transaction persisting the state of 9 classes which is considered as a worst case. In each of the scenarios, the classes are composed of 10 string attributes. Note that it is worst when compared with actual YARN Java classes where most of the attributes are integers.

We run the evaluation on one of the cluster machine. Simulation was prepared to generate 8000 transactions per epoch, and repeated up to epoch count is 100. The evaluation was carried out in following four phases.

1. Load preparation - The generator which inside the Event Streaming library is allocating memory to accommodate 8000*100 (800000) transaction objects. The memory locations of allocated objects are passed to each dispatcher thread’s queue.

2. Load transformation - Each thread dispatcher’s queue objects are shifted in high speed, and let all the dispatcher thread to process incoming event streams.

3. Dispatching - Once a library finished processing the streaming, built Java objects are pushed to the application’s concurrent queue.
5.2. C++ EVENT STREAMING LIBRARY THROUGHPUT EVALUATION

4. **Throughput measurement** - Java application queue is processed by another Java thread, and throughput measurement is taken by measuring total number Java objects and time took to process them.

The experiment was started from single dispatcher thread and increased up to 24 threads to monitor the throughput fluctuation. Figure 5.2 shows the throughput performance of our Event Streaming library.

![Figure 5.2: Streaming library throughput evaluation with various number of threads](image)

The throughput results have demonstrated the increasing performance when number of thread increases up to 5, and then the throughput was almost saturated. This behavior due to a high number of thread count causes more memory to be used and increases context switching, which can degrade thread processing performance.

### 5.2.1 Case analysis - Scheduler

In Scheduler case, we obtained maximum 60 Java Objects /ms. This means, Event Streaming library is capable of handling 60000 Java object building per second. These Java objects could be directly correlated with NodeManager’s heartbeat information that Scheduler need to be processed. If we have cluster size of 1000 NodeManagers, then, we would expect 1000 Java Objects per second at Scheduler side. According to our results, maximum 60000 NodeManager’s heartbeat could be handled by Event Streaming library.

### 5.2.2 Case analysis - ResourceTrackers

Since ResourceTrackers are interested in few number of state changes from Scheduler, our library’s throughput performance has climbed up to 200 Java Object/ms.
In multiple ResourceTrackers environment, all the participating ResourceTrackers would receive state information about NodeManagers which they are not interested, as NodeMangers are connected to ResourceTrackers in load balanced way. But, the results exhibit ResourceTrackers are handling higher number of Java Object than Scheduler. So, this indicates distributed ResourceTrackers are not the bottleneck in larger cluster size environment.

5.3 Distributed YARN simulator load preparation

The application trace generation for the distributed load simulator is shown in Figure 5.3. Each application is positioned with 10 sec offset and they are expected to finish the execution in 400 sec interval. An ideal total execution time of all 30 application is 690 sec which can be formulated using following equation.

\[
\text{Total execution time} = \text{Number of application} \times \text{offset time} + (\text{single application execution time} - \text{offset time}).
\]

In this experiment, all the applications are designed to acquire fixed number of containers during their execution. The number of containers per application can be calculated by following way.

\[
\text{Single application containers} = \left(\frac{\text{Number of NodeManagers} \times \text{load factor}}{\text{Number of applications}}\right).
\]

Figure 5.3: Simulated application load

The load factor parameter is changeable and it will influence the overall cluster load percentage. Evaluation was carried out by choosing 0.95 as the load factor. Generated application specific containers, duration of the containers and NodeManager location are completely random. So, all the NodeManagers are uniformly receive the same amount of containers to execute the application. Figure 5.4 demonstrates the distribution of concurrent running containers during the total experi-
5.3. DISTRIBUTED YARN SIMULATOR LOAD PREPARATION

This shows that a number of concurrently running container at any given point of time in simulated load. When the number concurrently running containers increases in the cluster, NodeManager’s heartbeat will carry lots of information to the scheduler, and this will increase the stress to the scheduler.

![Simulated application load](image)

Figure 5.4: Simulated application load

5.3.1 Experimental test-bed for Hops distributed YARN

An evaluation has been performed on the Hops distributed YARN, and results were compared with two different flavours of the Hadoop distribution: Hadoop-2.4.0, which is used to build Hops-YARN, and Hadoop-2.6.0, which is the latest release of the Hadoop distribution. As we explained in the load preparation section, NodeManagers were configured to start from 500 to 10000 and number of applications were set to 30, each of them having a fixed number of containers in each NodeManager iteration. The number of application containers in every NodeManager iteration has been determined as we saw in the previous formula.

The simulator was configured to run in distributed mode where two separate simulator instances were generating the load in the following manner. The NodeManagers load was equally divided among two simulators, and all the applications were set up to run on any single simulator to avoid unnecessary communication overhead. Since each application was executed by a separate Java process and all of them were updating the NodeManager’s common data structures, which runs on a separate machine, dividing the load would increase the two way communication between simulators. In order to avoid communication overhead between simulators, only a single instance of the simulator was allowed to simulate applications load. Figure 5.5 depicts the deployment set-up of Hadoop-2.4.0/Hadoop-2.6.0 distribution which has been configured to run on a single dedicated machine; bbc1.sics.se.

51
Figure 5.5: Hadoop-2.4.0/2.6.0 deployment

Figure 5.6: Hops distributed YARN deployment

Figure 5.6 shows the Hops distributed YARN deployment which is composed of two ResourceTrackers and one Scheduler. All ResourceTrackers and the Scheduler were running on individual machines. No two ResourceTrackers shared the same hardware to avoid interference in the system’s performance evaluation.

5.3.2 Scalability analysis

Size is just one the characteristics of scale that needs to be considered when it comes to any large scale distributed system. Commonly, the scalability of the system is measured by its capacity to handle heavy loads. Scalability could pertain to several different parameters of the system: capability of handling transactions, how much additional network traffic it can handle, or even how simple it is to add storage capacity. The Hadoop YARN scalability mainly depends on the number of NodeManagers that can be effectively handled by the ResourceManager.

Each NodeManager periodically sends heartbeats, which carry several types of information to ResourceTracker with a specific interval. The scalability factor of YARN could be determined more precisely by measuring the amount of handled heartbeats by the ResourceTracker. If the previous heartbeat of NodeManager is still being handled by the scheduler, then subsequent heartbeats are not sent to the scheduler for handling. Those heartbeats simply update the view of the cluster inside the Scheduler, and returned to the NodeManager as a false which denotes the term called "Not Handled". All other heartbeat responses are sent as true, 'Handled', by the Scheduler to the NodeManagers. There are two important things to be considered during the scalability measurement: The number of processed heartbeats, and the number of true heartbeats.

The processed heartbeat and true heartbeat percentage is calculated as follows:

\[
\text{Total heartbeat generation time by simulator} = t \\
\text{Total number of generated heartbeats} = m \\
\text{Number of NodeManagers} = n \\
\text{Total number of true heartbeats (true response of ResourceTracker)} = Z
\]
5.3. DISTRIBUTED YARN SIMULATOR LOAD PREPARATION

Ideal total processed heartbeat \( X = n \times \left( \frac{t}{\text{heartbeat interval}} \right) \)
Real processed heartbeat percentage = \( \frac{m}{X} \times 100 \% \)
True heartbeat percentage = \( \frac{Z}{X} \times 100 \% \)

Figure 5.7, 5.8 and 5.9 show the scalability of the Hadoop-2.4.0, Hadoop-2.6.0 and Hops distributed YARN respectively. There is a clear difference between the Hadoop YARN distributions and the Hops distributed YARN in scalability performance. As the number of NodeManagers increases, the number of processed heartbeat request by the ResourceTracker tremendously decreases for the Hadoop distributions. Inside the simulator, the thread which simulates the NodeManager waits for response until its previous heartbeat is processed. If the ResourceTracker is overloaded by a large number of heartbeat requests, handling performance will be drastically reduced. After 6000 NodeManagers, the performance of both distributions of Hadoop YARN starts to go down and it gets worse during the large NodeManagers simulation.

Our Hops distributed YARN has shown an incredibly higher scalability during a heavy load simulation. We managed to reach 80% true heartbeat in 10000 NodeManagers compared with standard Hadoop distribution in a configuration with two ResourceTrackers which each of them handling 5000 NodeManagers. Throughout this evaluation, distributing the ResourceTracker service has given a greater scalability for our Hops YARN. The Hops distributed YARN has increased the Scheduler’s performance into maximum level by distributing the workload to several ResourceTrackers.

Figure 5.10 shows the fluctuation of transactions on each components of the Hops distributed YARN and correlation with true heartbeat percentage. Since the heartbeat interval is configured to 3 sec, the expected average heartbeat from each
iteration of NodeManagers would be Number of NodeManagers/3. As each heartbeat is directly mapped to a single transaction, the expected average number of transactions linearly increases along with the number of the NodeManagers, and this has been taken as a reference line for rest of the discussion. As a matter of fact, true heartbeat percentage would be 100%, if the average number of transactions at the Scheduler and average number of transactions at the ResourceTrackers are equal to the ideal/expected average number of transactions. The deviation of average number of transaction starts to emerge for 5000 NodeManagers where the Scheduler receives more transactions from the ResourceTrackers, and the ResourceTracker receives significantly less transactions than expected. During the 10000 NodeManagers experiment, the ResourceTrackers sent nearly 3333 transactions/sec which is equal to the expected number of transactions. In the meantime, the scheduler only sent around 2500 transactions/sec to the ResourceTracker, which eventually affects
5.3. DISTRIBUTED YARN SIMULATOR LOAD PREPARATION

Figure 5.10: True heartbeat percentage variation with ResourceManager and ResourceTrackers transactions

the true heartbeat percentage. We can clearly conclude that the Scheduler is getting overloaded when the number of NodeManagers in the cluster increases. Since the load is distributed across several ResourceTrackers, the monolithic Scheduler is the performance bottleneck, and it would hamper the responsiveness of the applications submitted by users.

5.3.3 Scheduler operation time cost

Scheduler operations like allocate and node update, manipulate the Scheduler’s state from the ApplicationMaster service and the NodeHeartbeat update respectively. Each node’s heartbeat information is processed by the ResourceTracker, which sends an event to the Scheduler to handle if the previous heartbeat from the NodeManager was successfully finished. This node update event from the ResourceTracker played an important role at the Scheduler side. Once a Scheduler has fetched the node update event from an asynchronous dispatcher queue, it directly sends it to our Scheduler wrapper where it gets time stamped for a measurement. Then, that event is passed to an actual Scheduler to update the internal state of the Scheduler. Subsequently, the handled event is sent back to the Scheduler wrapper to get the duration of the node update event spent inside the scheduler. Figure 5.11 demonstrates the time distribution of the Hadoop-2.4.0, Hadoop-2.6.0 and Hops distributed YARN node update operation.

As we expected, the Hops distributed YARN finished the Jobs in nearly an ideal time of simulation, which was 690 sec. The Hadoop distributions took nearly 900 sec to finish the Jobs. This is the one reason our distributed YARN handled 95% of heartbeats in the configuration with 8000 NodeManagers. Apart from that, the Hadoop distributions take longer to handle each node update event. The node up-
update events internally update the application resources. For example, they update
the applications’ headroom to correctly take into account the containers assigned in
this update. However, in the meantime the ApplicationMasterService also accesses
the application to allocate requested containers from the ApplicationMasterClient.
Frequent allocate requests from the client due to the lack of resource updates from
NodeManagers make node update events slower in Hadoop distributions. Hops
distributed YARN improves the node update event by distributing the Resource-
Tracker service, which results in faster resource updates, and quicker allocations for
ApplicationMaster clients. Figure 5.12 depicts the allocate operation time for all
distributions. An allocation time reduction in the Hops distributed YARN affects
the node update operation as we have discussed above.
5.3. DISTRIBUTED YARN SIMULATOR LOAD PREPARATION

5.3.4 CPU usage analysis

This section demonstrates the user CPU usage as a percentage of the entire node’s CPU. By user CPU usage we mean that CPU time spent on kernel code is not shown. All the ResourceManager processes in these experiments were started as the local operating system user. Figures 5.13 depicts the CPU utilization of the Hadoop-2.4.0, Hadoop-2.6.0 and Hops distributed YARN respectively.

![Figure 5.13: CPU utilization of all the components of Hops distributed YARN, Hadoop-2.4.0 and Hadoop-2.6.0](image)

The Hops distributed YARN ResourceManager’s and ResourceTracker’s CPU usage keeps increasing along with number of NodeManagers. As soon as heartbeat information is persisted by ResourceTrackers, larger number of updates are streaming to the Scheduler to process. The same procedure is followed by the Scheduler which updates the database once it has handled an event from the ResourceTrackers, subsequently, all the updates are broadcast to participating Re-
sourceTrackers in Hops distributed YARN. The CPU utilization of the Scheduler is significantly higher than ResourceTrackers in each NodeManager’s configuration as a single Scheduler is processing larger information from multiple ResourceTrackers. Contrary to the Hops distributed YARN, the Hadoop distributions consume very low CPU in all NodeManager’s experiments. After 7000 NodeManagers, Hadoop distribution’s CPU consumption decreases as ResourceTrackers are overloaded, and the NodeManager simulator slowed down drastically. This affects the Hadoop distribution to process low number of heartbeat from NodeManagers, and eventually CPU utilization went down.

5.4 Summary

We conducted two different type of evaluation such as Throughput performance of Event Stream processing library and scalability analysis of our distributed YARN and other Hadoop distributions. Before we integrate our Stream processing library with distributed YARN, we did a preliminary evaluation on our Event Streaming processing library to make sure, it is building Java objects as we expected. The results have demonstrated, the library is handling higher streaming input and building Java objects more than we expected. Second evaluation was carried out by newly implemented distributed scheduler load simulator for distributed YARN and standard Hadoop distributions. Our distributed YARN is handling large clusters effectively (80% at 10000 NodeManagers) compared with standard Hadoop distribution which is nearly 14% in scalability analysis. This indicates that our Hops distributed YARN is performs 2.5x higher scalability than other standard Hadoop distribution.
Chapter 6

Conclusion

Hadoop YARN and Hops-YARN ResourceManager, which is potential performance bottleneck in letting application compete for cluster resources when cluster size increases in massive scale. In this thesis, we provide a solution for the Hops-YARN based on NDB cluster event streaming to improve scalability and make Hops-YARN more interactive by scale-out the ResourceTracker service over standby ResourceManagers. The implementation of this work is focused on making the ResourceManager more scalable by migrating the state of one of its main services, the ResourceTracker service, which is primarily responsible for managing NodeManager heartbeat requests. A novel architecture is proposed in which the NDB Event streaming can be used in all distributed ResourceTrackers, and scheduler to perform their operations in large number of cluster.

Our evaluation of the Event Streaming library has demonstrated, it is capable of processing highly frequent streaming and dispatching with minimal latency overhead. Integration of streaming library with Hops-YARN system handled large cluster size with two distributed ResourceTrackers while Hadoop YARN took more time to execute applications due to slow rate of heartbeats handling from large cluster size NodeManagers.

6.1 Future work

The Hops-YARN distributed version only persists the relevant state information both ResourceTrackers and Scheduler need to operate. Hops-YARN has already been proposed as a generic relational model schema where the state of the ResourceManager can be migrated to an external relational database. However, due to the limited time available to complete this work focus was given on designing an architecture for distributing functions related to the ResourceTracker and scheduler management. The next steps of this work are described in the following sections.
6.1.1 Persisting whole state

In the distributed Hops-YARN, the ResourceTrackers are only store important scheduler events and streaming to the scheduler. The same procedure is following by scheduler by persisting only restricted state which ResourceTrackers need to process the events. The Hops-YARN implementation allows single thread transaction commit which is not feasible when either number transaction per second or the size of the objects per transaction increases. Lots of modification have been made to support multithreaded transaction commit where several threads are parallely persisting ResourceManager state into the database. But, initial system evaluation has shown that new solution also having deadlock problem such that thread are waiting for lock when common tables are updated by the NDB. The new approach should be implemented to overcome this problem to provide the High Availability.

6.1.2 Workload simulations

The Hops distributed YARN, Hadoop-2.4 and Hadoop-2.4.6 were tested against the trace generated by our synthetic trace generators. The load simulated to evaluate the scalability performance of each components is not realistic as a production workload at Yahoo! [3]. This work can be extended to create real workload by directly compared with production workload by mimicking some characteristics such as various number of containers per applications, number of concurrent containers, application starting time etc.

6.1.3 Asynchronous transactions

The NDB API on the client side is a crucial part of the performance of MySQL Cluster. As we have discussed in MySQL cluster database section every API that is used to access the Data Server in MySQL Cluster uses the NDB API. NDB storage handler is using NDB API to access data from MySQL APIs which is residing in MySQL Cluster. Hops-YARN is using synchronous transactions supported by ClusterJ connector library which directly calling NDB API to store and fetch data. During the evaluation, multithreaded approach in application level is not enough to drive higher load from Hops-YARN.

The base of the good performance of the programming API is the ability to batch operations in various manners. NDB asynchronous API is fully supporting batching transactions in two different levels. The first level is that it is possible to batch inside one thread, which means single thread can handle multiple transactions in parallel and execute them in parallel with one execute call. In addition it is also possible to have several threads working in parallel and it is possible for every one of those threads to also be executing multiple transactions in parallel. In this approach, Hops-YARN can continuity process RPCs while NDB asynchronous API taking care about batching transactions and execute them. Another advantage of using asynchronous NDB API is to reduce the networking cost greatly by making
6.1. FUTURE WORK

TCP/IP packets larger. Furthermore, batching transaction thorough asynchronous API will reduce the context switches since handling of several message in parallel can handled without context switches.
Bibliography


Appendix A

A.1 Event streaming library

Event streaming library used in ResourceTrackers and Scheduler has different configuration setting to support message construction for Hops distributed YARN. The following sections explain the different category of setting that library can configure.

A.1.1 NDB setting

The NDB cluster setting properties section in the configuration file provides the connection setting and as well as the NDB API parameters to obtain a streaming. The NDB connect string is an address of management server of MySQL NDB Cluster and database name is where ResourceManager is persisting the state information. The NDB API has several setting that application should configure before proceed the subscription.

1. NUMBER_RETRIES_ATTEMPT: Specifies the number of retries to attempt in the event of connection failure. A negative value will result in the attempt to connect being repeated indefinitely

2. RETRY_DELAY_IN_SECONDS: specifies how often retries should be performed

3. RETRY_DELAY_IN_SECONDS: The delay represents the number of seconds between reconnect attempts; the default is 1 second

4. TIMEOUT_FOR_FIRST_ALIVE: Number of seconds to wait until first live node is detected

5. TIMEOUT_AFTER_FIRST_ALIVE: Number of seconds to wait after first live node is detected
## cluster setting
NDB_CONNECT_STRING=bbc7.sics.se:1186
NDB_DATABASE_NAME=hop_sri

## API setting
NUMBER_OF_RETRIES_ATTEMPT=5
RETRY_DELAY_IN_SECONDS=3
TIMEOUT_FOR_FIRST_ALIVE=30
TIMEOUT_AFTER_FIRST_ALIVE=30

### A.1.2 JNI setting

The library is designed to hide complex JNI functions that allows Java programmer to easily implement necessary Java functions to get C++ functionalities. Whenever Java functions or class names are modify by programmer, Event Streaming library source code remain intact except corresponding changes in the configuration parameters.

1. **TOTAL_NO_OF_COLUMN** : The total number of columns which you need to listen for, if the number is mismatch with an actual total columns, library initialization will crash

2. **TABLE_NAMES** : The comma separated table names that library is interested to listen events

3. **TOTAL_NUMBER_OF_CLASSES** : The number of Java Class to be created

4. **JAVA_CLASS_NAME_<number of classes>** : The description of each individual class name. This class name should follow the syntax as, function name | db column name, function name which manipulate this field in your Java class, function signatures

5. **CALLBACK_CLASS_NAME** : C++ will call this callback function whenever the object is ready at native side. Actual Java object creation is totally depends on programmer. This parameter has the following syntax, callback class-name, if it is in package, we need a full path | callback method. Failure to provide correct path will cause a Class not found exception from JNI

6. **SINGLE_THREAD_CALLBACK_METHOD** : The callback method in Java class that C++ should know about.

7. **MULTI_THREAD_CLASS_BUILDER_NAME** : If the multithreaded is enabled, C++ will use this Java method signature to create thread specific object, so that each thread can paralley build the Java object for later dispatch.

A.1. EVENT STREAMING LIBRARY

9. **JAVA_CLASS_NAME_LISTS**: This configuration will be used to create the lists of objects we need. If you don't want to build all the classes, just specify the classes which you want to create.

10. **REFERENCE_TABLE_NAME**: This is the table name we should use as a reference to order the Java objects

11. **PROCESSING_THREADS**: The number of processing threads

```
TOTAL_NO_OF_COLUMN=45
TABLE_NAMES=Table1,Table2, ...
EVENT_COL_NAMES=Attribute1,Attribute2...|Attribute5,Attribute6...
TOTAL_NUMBER_OF_CLASSES=2
JAVA_CLASS_NAME_1=functionName|fieldId1,setField1,(Ljava/lang/String;)V|..
JAVA_CLASS_NAME_2=functionName|fieldId5,setFieldId5,(Ljava/lang/String;)V|..
CALLBACK_CLASS_NAME=org/apache/hadoop/yarn/server/resourcemanager/NdbEventStreamingReceiver
SINGLE_THREAD_CALLBACK_METHOD=onEventMethod()V
MULTI_THREAD_CLASS_BUILDER_NAME=buildCompositeClass|()Lse/sics/hop/metadata/hdfs/entity/yarn/HopRMNodeComps;
MULTI_THREAD_CALLBACK_METHOD=onEventMethodMultiThread|(Lse/sics/hop/metadata/hdfs/entity/yarn/HopRMNodeComps;)V
JAVA_CLASS_NAME_LISTS=buildHopContainerId
REFERENCE_TABLE_NAME=ha_pendingevents
PROCESSING_THREADS=1
```

A.1.3 NDB API setting

The following configurations are specifically using by NDB API to control an event flow.

1. **MAX_EVENT_BUFFER**: This sets the maximum NDB API event buffer memory in bytes that can be used for the buffering the events for C++ library

2. **EVENT_TYPE**: Type of events that library is interested to listen for processing. (0 - all events, 1 - insert events, 2 - update events, 3 - delete events, 4 - insert and update events together)

```
MAX_EVENT_BUFFER=4294967295
EVENT_TYPE=4
```