Interim Report

Mark Nemec

1 Introduction

Describe Apache Storm but only pieces relevant to the thing.

2 Description

2.1 Motivation

In recent years, there has been an **explosion** in cloud computation software. Starting with Google publishing their paper on MapReduce,[1] many new open-source frameworks for distributed computation have emerged, most notably Apache Hadoop for batch processing and Apache Storm for processing data streams. The main idea of these frameworks is to split the work that needs to be carried out and distribute it across multiple nodes in a cluster.

Commercial companies and researchers have been able to utilise these frameworks and create distributed systems which can accomplish things that would not be possible on a single computer. This has mostly been allowed by the low price of commodity hardware. While the price of such systems is perhaps lower than price of a supercomputer with equal power there are certain limitations:

- The nodes of a cluster need to communicate through a network. This limits the speed of communication between processes that live on different nodes.
- Distributed systems waste resources due to data replication which enhances reliability and possibly performance. However, with this comes the problem of consistency.
- To run a distributed computation on commodity hardware one would usually need a data centre or to rent out instances on cloud computing services such as Amazon EC2 or Rackspace. This is not ideal for some use cases which require full control over the system or a heightened level of security.

On the other hand, even though Moore's law still holds true, processor makers now favour increasing the number of cores in CPU chips to increasing their

[1] MapReduce

frequency. This trend implies that the "free lunch", getting better software performance by upgrading the processor, is over and programmers now have to design systems with parallel architectures in mind to gain performance boosts. However, there are some limitations to this as well:

- It is generally believed that writing parallel software is hard. The traditional techniques of message passing and parallel threads sharing memory require the programmer to manage the concurrency at a fairly low level, either by using messages or locks.
- Mixing components written in several languages into one system that executes in parallel could be quite challenging.

The main idea of this project is to apply ideas from distributed system not in the context of clusters but in context of multi-core CPUs. This way, we could bring some of the knowledge from distributed systems to a single computer operation. To do this, this project presents a port of the Apache Storm project for multi-core CPUs.

2.2 Apache Storm Overview

Apache Storm is a distributed real-time computation system. It has a simple API that allows users to ...

2.2.1 Representation

Storm breaks down a computation to a network of components which pass tuples between them. Storm represents this network as a directed acyclic graph (also called a topology) and the components can be of two types:

Spout: A spout is a source of the data-stream of the network. Typically, a spout reads from a message broker such as RabbitMQ or Kafka but can also generate its own stream or read from somewhere like the Twitter streaming API.

Bolt: A bolt takes a data stream as an input and transforms it into a different stream as its output. A bolt can perform a range of functions e.g. filter out tuples based on some criteria or perform a join of two different input streams.

2.2.2 Architecture

A Storm cluster consists of a master node called Nimbus, a number of Zookeeper nodes handling the coordination of the cluster and worker nodes which perform the computation.

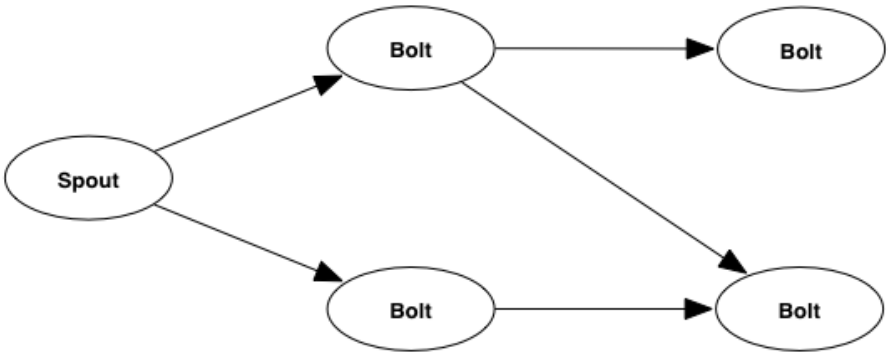

Figure 1: Example Topology

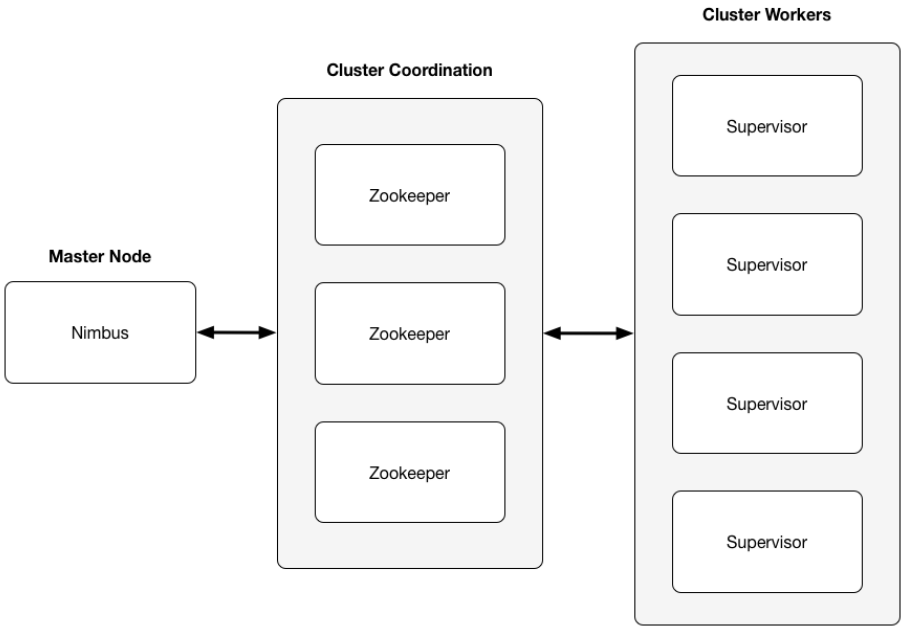

Figure 2: Architecture of a Storm Cluster

Nimbus: The Nimbus node is responsible for receiving commands from the user and informing the rest of the cluster. Nimbus is the node the user submits a topology to as well as the node that assigns workers components they will be working on.

Zookeper nodes: These nodes are the glue between the Nimbus node and the workers within the cluster. Nimbus uses the Zookeeper nodes to communicate with the worker nodes.

Worker nodes: A worker node is running a daemon called Supervisor which spawns worker processes that actually perform the computation.

2.2.3 Inside a worker node

There are three main entities used to execute a topology on a Storm cluster:

Worker: A worker is a process that is assigned a port and listens to tuple messages on a socket associated with the port. Whenever it receives a tuple, it puts it on a queue where it is picked up by one or more executors of the worker process.

Executor: An executor controls the parallelism within a worker process. Every executor runs in a separate thread, picks up tuples from the receiving queue performs the function of a component it represents and puts the transformed tuples on the transfer queue, ready to be sent along to the next component. An executor runs one (the usual case) or more tasks.

Task: A task performs the actual data processing. However, within an executor thread all the tasks are executed sequentially. The main reason for having tasks is that the number of tasks stays the same throughout the lifetime of a topology but the number of executors can change (via rebalancing). Thus if some worker nodes in the cluster go down, the topology can continue executing with the same number of tasks as before.

3 Done so far

3.1 Stripped Down the Nimbus process

A Nimbus node is the master node of a cluster. Usually the process of submitting a topology to Nimbus follows the following steps:

- User starts up the Nimbus node, supervisor nodes, and Zookeeper nodes.
- User submits a new topology to the Nimbus node.
- The Nimbus node uploads code to all the supervisors and makes assignments which it writes to Zookeeper.
- Workers running on a supervisor read their assignments from Zookeeper, create executors that execute the code.

- Nimbus keeps running and listens for commands to either rebalance, activate, disactivate, or kill a topology.

In multi-core setting some of this stuff.

3.2 Stripping Down Parallelism Abstractions

There are three main entities used to run a topology on a Storm cluster:

- Worker
- Executor
- Task

In addition to these another important abstraction is the Supervisor daemon. A supervisor is what the user starts on a particular node. The supervisor then launches a number of workers specified in the config file. Each worker is assigned a port and listens to tuple messages on a socket associated with the port. Whenever it receives a tuple, it puts it on a queue where it is picked up by one or more executors of the worker. Each executor runs on a separate thread and is associated with one (usual case) or more tasks.

3.3 Ported Thrift objects to Java objects

Since Storm runs on a cluster, serialisation plays an important role. Objects need to be serialised whenever they are going to be sent through the network and deserialised when received by a worker. Storm uses the Thrift framework for this.[2] Given a Thrift definition for a class, Thrift generates serialisation code automatically.

While serialisation is important in a distributed system, there is no need for it in a multi-core setting. Hence all the code generated automatically by Thrift had to be rewritten as well as portions of code that referenced it.

3.4 Became familiar with Clojure

According to the Storm documentation "the great majority of the implementation logic is in Clojure". Clojure is a dynamically typed LISP dialect that targets the Java Virtual Machine. Worried that not knowing the language would hinder progress, a lot of time was spent getting comfortable with the language and reading documentation. This proved to be good decision in hindsight as building up on the existing Clojure code proved to be more efficient than rewriting the implementation in Java.

[2] Thrift

3.5 Reimplemented Worker and Executor for Multi-core

In Storm, workers run within a supervisor process. Every worker potentially runs several executors which are implements as threads. Thus, a sensible way to transform this into a multi-core setting is to force all the executors of a topology to run within one worker. This way, the only part that needed to be stripped from a worker is the code that handles inter-working messaging.

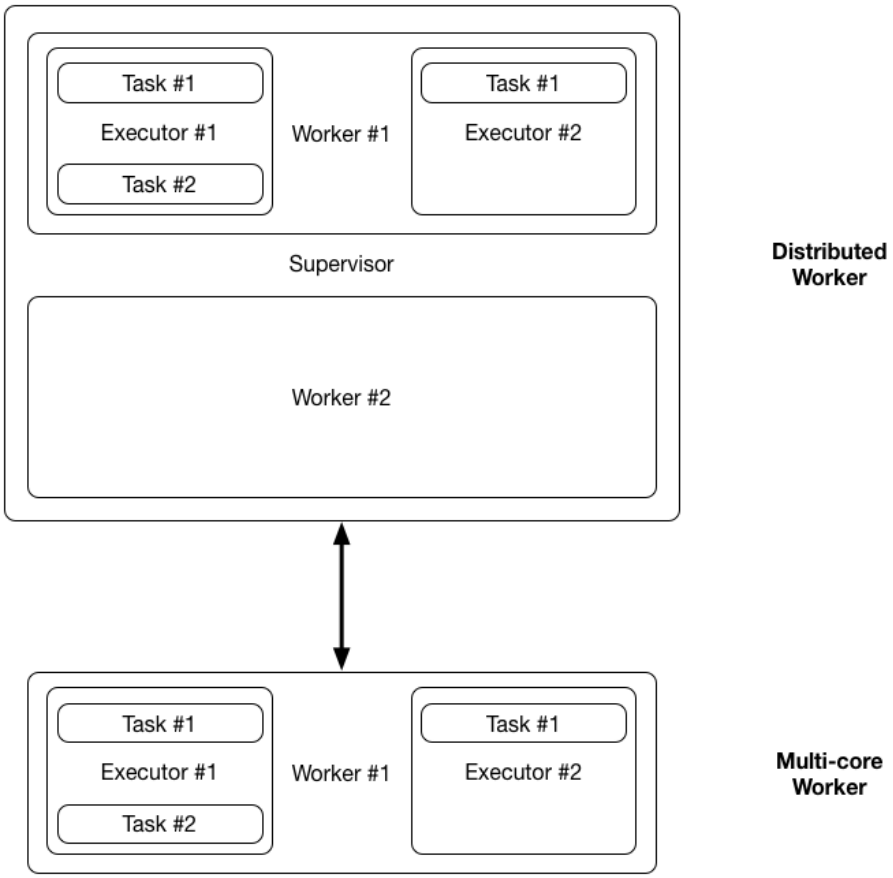

Figure 3: Worker in a distributed vs multi-core setting

4 Remains to be done

4.1 Test speed vs original storm

4.2 Remove even more code

4.3 CPU Affinity

5 Timeline

6 Conclusion