Storm on Multi-core

Mark Nemec

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2015

Abstract

This is the abstract.

Acknowledgements

Acknowledgements go here.

Table of Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Main Idea . . . 4
- 1.3 Structure of the Report . . . 4

2 Literature Review **5**
- 2.1 Data Stream . . . 5
  - 2.1.1 Comparison to a Database Management System (DBMS) . . . 5
- 2.2 Apache Storm . . . 6
- 2.3 Ports to Multi-core . . . 6
- 2.4 Similar Efforts . . . 6
- 2.5 Summary . . . 7

3 Background **9**
- 3.1 Storm Overview . . . 9
- 3.2 Storm Concepts . . . 9
  - 3.2.1 Core Concepts . . . 9
  - 3.2.2 Additional Concepts . . . 10
- 3.3 Example Topology . . . 11
- 3.4 Storm Architecture . . . 12
  - 3.4.1 Nimbus Node . . . 13
  - 3.4.2 Worker Nodes . . . 14
  - 3.4.3 Zookeeper Nodes . . . 15
- 3.5 Serialisation . . . 15

4 Bringing Storm to Multi-core **17**
- 4.1 Apache Storm on Multi-core . . . 17
  - 4.1.1 Tuple processing . . . 17
  - 4.1.2 Heartbeats & Timers . . . 18
  - 4.1.3 Acker & System Bolts . . . 18
  - 4.1.4 Zookeeper . . . 18
- 4.2 Storm-MC Architecture . . . 20
  - 4.2.1 Nimbus . . . 20
  - 4.2.2 Worker . . . 21
  - 4.2.3 State . . . 22
- 4.3 Tuple Processing . . . 22

4.4 Differences between Apache Storm and Storm-MC 22

5 Storm-MC Implementation 23
5.1 23

6 Evaluation 25
6.1 Evaluation Metrics 25
6.2 System Configuration 25
6.2.1 Software Setup 25
6.2.2 Hardware Setup 26
6.3 Performance 26
6.3.1 WordCount Topology 27
6.3.2 Enron Topology 28
6.3.3 RollingSort Topology 28
6.4 Challenges 28

7 Conclusion 29
7.1 Future Work 29

A Listings 33

Bibliography 37

List of Figures

2.1 Stream Querying. 6

3.1 WordCount topology. 11
3.2 Apache Storm Architecture. 12

4.1 Tuple processing in Apache Storm. 18
4.2 Threads over time 20
4.3 Comparison of a worker in Storm and Storm-MC 21
4.4 Tuple processing in Storm-MC. 22

6.1 Improvement of Storm-MC over Apache Storm in number of tuples processed 27
6.2 Number of threads used by Storm and Storm-MC 27

List of Tables

4.1 Break-down of threads used by Apache Storm to execute a topology with 3 components. 19
4.2 Recorded thread states over five minute period. 19

List of listings

1 WordCountTopology.java 12
2 RandomSentenceSpout.java 34
3 SplitSentence.java 35
4 splitsentence.py 35
5 WordCount.java 36

Chapter 1

Introduction

In recent years, there has been an explosion of cloud computing software. After Google published their paper on MapReduce [?], many new open-source frameworks for distributed computation have emerged, most notably Apache Hadoop for batch processing and Apache Storm for real-time data stream processing.

The main idea of these frameworks is to split the work that needs to be carried out and distribute it across multiple nodes of 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 and good horizontal scaling properties.

This project is about taking the ideas from the distributed system Apache Storm and applying them in the context of multi-core machines instead of clusters.

1.1 Motivation

While the cost of a commodity hardware cluster might be lower than the price of a single computer with equal power there are certain limitations:

- The nodes of a cluster communicate through network. This limits the speed of communication between processes that live on different nodes.
- Distributed systems waste resources by replicating data to ensure reliability.

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 frequency. This trend implies that the "free lunch" of getting better software performance by upgrading the processor is over and programmers now have to design systems with parallel architectures in mind. 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.

- Apache Storm has become the de facto tool used in stream processing on a cluster and according to their "Powered By" page [?] there are tens of companies already using Storm to process their real-time streams. It would be nice if they could keep that code.

1.2 Main Idea

The solution proposed in this paper is to take the existing Apache Storm project and port it for multi-core machines. This is implemented in a library Storm-MC with an API compatible with Apache Storm. This allows us to take an existing application written with Apache Storm in mind and run it in a multi-core setting. This way, we can avoid network latency and enjoy the significant performance improvements of a shared-memory environment.

- Prices of high-end server have decreased and one can get a 32-core machine for 10,000 USD.

1.3 Structure of the Report

In chapter 1, blah blah.

Chapter 2

Literature Review

The following chapter blah blah blah...

2.1 Data Stream

Golab and Ozsu define data stream as "a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety". [15]

2.1.1 Comparison to a Database Management System (DBMS)

Historically, data has been stored into database management systems (DBMS) where it was later analysed the assumption being that there would be enough disk space to contain the data. This approach fits many purposes but recently applications started "feeling the need" to analyse rapidly changing data on-the-fly.

This has brought on an advent of data stream processing. Several stream-processing frameworks have emerged such as Apache Storm [1], Apache Spark [2], and Onyx [3]. These frameworks, usually ran on a cluster, provide the user with abstractions which greatly simplify writing a real-time data stream processing application.

Whereas DBMSs excel at getting an exact answer to a query, data streams usually provide an approximate answer. The answer is approximate because it is usually correct only within a certain window of time, the query is simplified because it can only be ran in one pass, or because it is used with a sampling rate which does not include all events. A typical data stream analysis using windows and sampling is depicted in figure 2.1.

The assumption behind using a time window is that users are most likely interested in the most recent events. That way they can react to change quickly. Sampling on the other hand is used to reduce the number of events used for a query. [14]

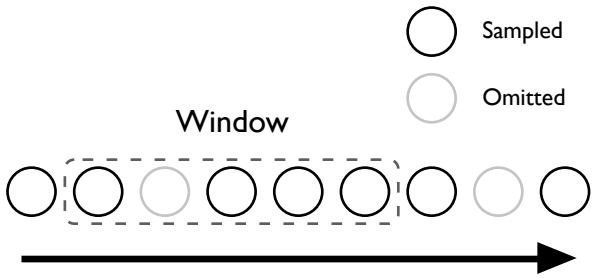


Figure 2.1: Stream Querying.

Even though the answer might only be approximate it can have great value because the query is answered at the right time. Furthermore, even though the query may run only on a subset of data it is still possible to detect trends or system failures. For example, Twitter are using Apache Storm to run real-time analysis on millions of events per second for their analytics product. [18]

In research, several techniques have been developed to enable real time data stream mining. For example, the MOA environment [12] which enables real time machine learning using the WEKA machine learning workbench [16].

2.2 Apache Storm

What is Storm

Apache Storm is an open source distributed real-time computation system. Storm was Originally created by Nathan Marz while working at BackType. [4] BackType was later acquired by Twitter which is when Storm became open source.

Who uses Storm

Storm is reportedly used by 81 companies listed on their website [?] and possibly many others.

What research has there been on Storm

2.3 Ports to Multi-core

2.4 Similar Efforts

Has anyone ported Storm somewhere

Real time on multi-core

What has been ported

2.5 Summary

Chapter 3

Background

In this chapter we give background information necessary to understand the design of Storm-MC. We give a quick overview of Apache Storm 3.1, explain the concepts (3.2) used in Storm, show an example Storm program (3.3), give details about the underlying architecture of Storm (3.4), and finally describe the serialisation used by Storm.

3.1 Storm Overview

Apache Storm was developed in a mix of Java and Clojure. As mentioned by the author of Storm in [17], writing the Storm interfaces in Java ensured large potential user-base while writing the implementation in Clojure increased productivity.

To ensure API compatibility with Storm, Storm-MC was developed using the same set of languages. This allowed for code reuse and not having to re-implement functionality already present in Storm. Hence, in the following sections we describe Storm in greater detail in hope that this will later clarify design choices made for Storm-MC.

3.2 Storm Concepts

3.2.1 Core Concepts

There are several core concepts used by Storm and hence by extension Storm-MC as well. These concepts are put together to form a simple API that allows the user to break down a computation into separate components and define how these components interact with each other. The three core concepts of Storm are:

Spout
A spout is a component that represents the source of a data-stream. Typically,

a spout reads from a message broker such as RabbitMQ [11] or Apache Kafka [7] but can also generate its own stream or read from somewhere like the Twitter streaming API.

Bolt
A bolt is a component that transforms tuples from its input data stream and emits them to its output data stream. 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.

Topology
The programmer connects spouts and bolts in a directed graph called topology which describes how the components interact with each other. The topology is then submitted to Storm for execution.

3.2.2 Additional Concepts

Stream
A stream is defined as an unbounded sequence of tuples. Streams can be thought of as edges of a topology connecting bolts and spouts (vertices).

Tuple
A tuple wraps named fields and their values. The values of fields can be of different types. When a component emits a tuple to a stream it sends that tuple to every bolt subscribed to the stream.

Stream Grouping
Every bolt needs to have a type of stream grouping associated with it. This grouping decides the means of distributing the tuples coming from the bolt's input streams amongst the instances of the bolt task. Following are the possible types of stream grouping:

Shuffle Randomly partitions the tuples among all the bolt tasks.

Fields Hashes on a subset of the tuple fields.

All Replicates the entire stream to all the bolt tasks.

Direct The producer of the tuple decides which task of the bolt will receive this tuple.

Global Sends the entire stream to a single bolt task.

Local or Shuffle Prefers sending to executors in the same worker process, if that is not possible it uses the same strategy as shuffle.

Users are also able to specify their own custom grouping by implementing the `CustomStreamGrouping` interface.

Explain why you would want to do that.

All the components of a Storm topology execute in parallel. The user can specify how much parallelism he wants associated with every component and Storm spawns the necessary number of threads. This is done through a configuration file, defined in YAML, which is submitted along with the topology.

There are two additional bolts running for every topology:

Acker
The Acker bolt guarantees fault tolerance for the topology. It tracks every tuple that was produced and ensures that the tuple has been acknowledged by every bolt of the stream.

System Bolt
The System bolt is useful in two ways:

Metrics System bolt collects metrics on the local Java Virtual Machine (JVM). Other components can subscribe to these metrics and receive their values at regular intervals.

Ticks Components of a topology can subscribe to receive tick tuples in regular intervals. These tuples can be used to trigger some event of a component.

3.3 Example Topology

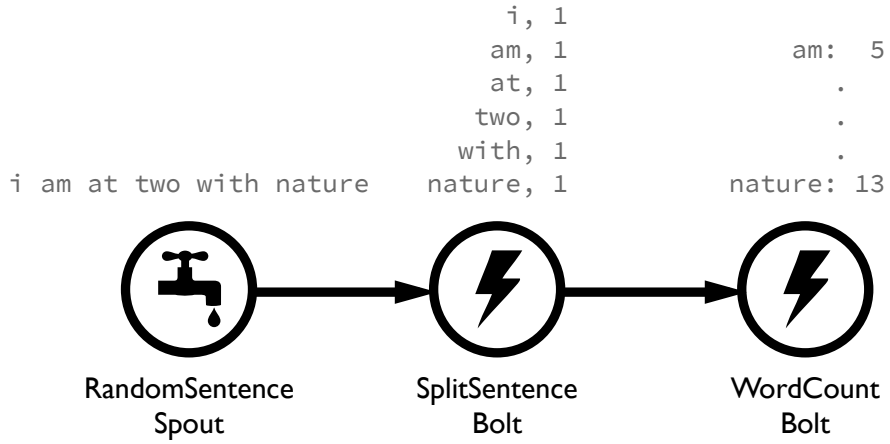


Figure 3.1: WordCount topology.

A classic example used to explain Storm topologies is the WordCount topology. In this topology, there is a spout generating random sentences, a bolt splitting the sentences on white space, and a bolt counting occurrences of every word. Figure 3.1 shows how we could represent this topology graphically.

This may seem as a simplistic example but it is useful when demonstrating how easy it is to implement a working topology using the Storm API.

```
public class WordCountTopology {
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new RandomSentenceSpout(), 5);
        builder.setBolt("split",
                new SplitSentence(), 8).shuffleGrouping("spout");
        builder.setBolt("count",
                new WordCount(), 12).fieldsGrouping("split", new Fields("word"));
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("word-count", conf, builder.createTopology());
  }
}
```

Listing 1: WordCountTopology.java

Listing 1 shows how the topology is put together to form a graph of components. Storm uses the Builder design pattern to build up the topology which is then submitted to Storm for execution. The last argument to the setBolt/setSpout method is the number of parallel tasks we want Storm to execute for the respective component. For implementation of the spout and bolts used in this topology, refer to appendix A.

3.4 Storm Architecture

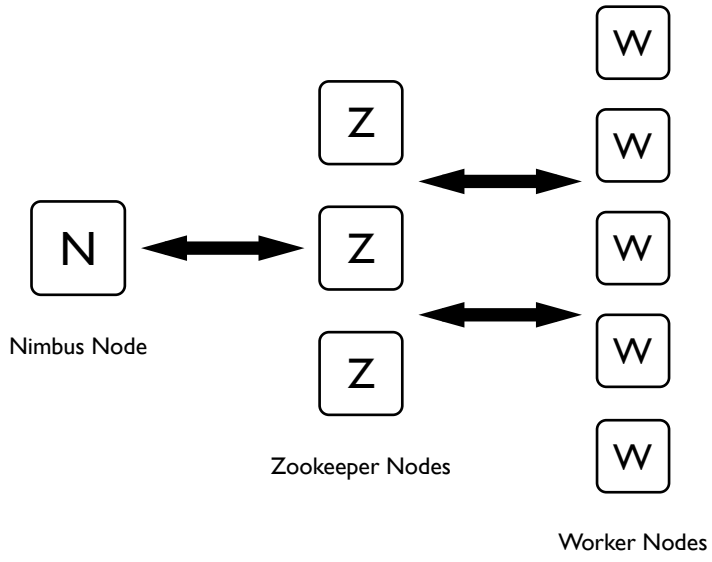


Figure 3.2: Apache Storm Architecture.

Maybe highlight similarities to Hadoop

A Storm cluster adopts the Master-Worker pattern. To set up a Storm topology, the user launches daemon processes on nodes of the cluster and submits the topology to

the master node, also called Nimbus. The worker nodes receive task assignments from the master and execute on them. The coordination between the master node and the worker nodes is handled by nodes running Apache Zookeeper [9]. Figure 3.2 shows a graphical representation of Storm Architecture.

3.4.1 Nimbus Node

The master node runs a server daemon called Nimbus. The main role of Nimbus is to receive topology submissions from clients. Upon receiving a topology submission, Nimbus takes the following steps:

Validate the topology
The topology is validated using a validator to ensure that the submitted topology is valid before trying to execute it. The user can implement his own validator or use the default validator provided by Storm.

Distribute the topology source code
Nimbus ensures that the workers involved in the topology computation have the source code by sending it over the network.

Schedule the topology
Nimbus runs a scheduler that distributes the work among workers of the cluster. Similarly to validation, the user can implement his own scheduler or use the default scheduler provided by Storm. The default scheduler uses a simple Round-robin strategy.

Activate the topology
Nimbus transitions the topology to active state which tells the worker nodes to start executing it.

Monitor the topology
Nimbus continues to monitor the topology by reading heartbeats sent by the worker nodes to ensure that the topology is executing as expected and worker nodes have not failed.

Nimbus is an Apache Thrift [8] service (more on Thrift in section 3.5) that listens to commands submitted by clients and modifies the state of a cluster accordingly. Following are the commands supported by Nimbus:

Submit a topology
Clients can submit a topology defined in a Java Archive (JAR) file. The Nimbus service then ensures that the topology configuration and resources are distributed across the cluster and starts executing the topology.

Kill a topology
Nimbus can stop running a topology and remove it from the cluster. The cluster can still continue executing other topologies.

Activate/deactivate a topology
Topologies can be deactivated and reactivated by Nimbus. This could be

useful if the spout temporarily cannot produce a stream and the user does not want the cluster to execute idly.

Rebalance a topology
Nimbus can rebalance a topology across more nodes. Thus if the number of nodes in the cluster changes the user can increase or decrease the number of nodes involved in the topology.

3.4.2 Worker Nodes

The worker nodes run a daemon called Supervisor. There are 4 layers of abstraction which control the parallelism of a worker node.

Supervisor
A supervisor is a daemon process the user runs on a worker node to make it part of the cluster. It launches worker processes and assigns them a port they can receive messages on. Furthermore, it monitors the worker processes and restarts them if they fail. A worker node runs only one supervisor process.

Worker
A worker process is assigned a port and listens to tuple messages on a socket associated with the port. A worker launches executor threads as required by the topology. 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.

Furthermore, the worker has a transfer queue where its executors enqueue tuples ready to be sent downstream. There can be multiple workers processes running inside one supervisor.

Executor
An executor controls the parallelism within a worker process. Every executor runs in a separate thread. An executor's job is to pick up tuples from the receiver queue of the worker, perform the task of a component it represents, and put the transformed tuples on the transfer queue of the worker. There can be many executors running inside one worker and an executor performs one (the usual case) or more tasks.

Task
A task represents the actual tuple processing function. 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 (by rebalancing). Thus if some worker nodes in the cluster go down, the topology can continue executing with the same number of tasks as before.

3.4.3 Zookeeper Nodes

The Storm cluster contains a number of Zookeeper nodes which coordinate the communication between the master and the workers. Storm does this by storing the state of the cluster on the Zookeper nodes where both Nimbus and worker nodes can access it.

The cluster state contains worker assignments, information about topologies, and heartbeats sent by the worker nodes back to Nimbus. Apart from the cluster state, Storm is completely stateless. Hence, if the master node or a worker node fail the cluster continues executing. The only time the cluster stops executing is if all the Zookeper nodes die.

3.5 Serialisation

Since Storm topologies execute on a cluster all the components need to be serialisable. This is achieved with Apache Thrift. Components are defined as Thrift objects and Thrift generates all the Java serialisation code automatically.

Why is Thrift good?

Furthermore, since Nimbus is a Thrift service Thrift generates all the code required for remote procedure call (RPC) support. This allows defining topologies in any of the languages supported by Thrift and easy cross-language communication with the Nimbus service.

Chapter 4

Bringing Storm to Multi-core

The following chapter explains the design of Storm-MC. We describe how Apache Storm behaves on multi-core machines (4.1), how the Storm architecture was ported over to multi-core (4.2), how tuple processing was ported to multi-core (4.3), and finally, we list feature differences between Apache Storm and Storm-MC (4.4).

The design of Storm-MC was ported over from Apache Storm. This enabled rapid progress while guaranteeing compatibility with the Apache Storm API. Clearly, however, some differences had to be made to take advantage of multi-core machine performance implications.

4.1 Apache Storm on Multi-core

To begin, we discuss why Apache Storm does not perform optimally on a single multi-core machine. Storm can be ran in local mode where it emulates execution on a cluster. This mode exists so that it is possible to debug and develop topologies without needing access to a cluster. However, there are several reasons why the local mode is not as performant as it could be.

4.1.1 Tuple processing

Figure 4.1 shows how tuple processing is implemented inside a Storm worker process. The tuple is read from a message buffer by the receiver thread of the worker and put on a receive queue of the corresponding executor. The tuple is then picked up by the component thread of the executor for task execution.

After the component thread has executed the task it puts the tuple on the executor sender queue. There, it is picked up by the executor sender thread which puts the tuple on the global sender queue of the worker. Lastly, the global sender thread of the worker serialises the tuple and sends it downstream.

Alternatively, if the tuple is forwarded to an executor in the same worker process it is put on the receiver queue of the corresponding executor directly after processing.

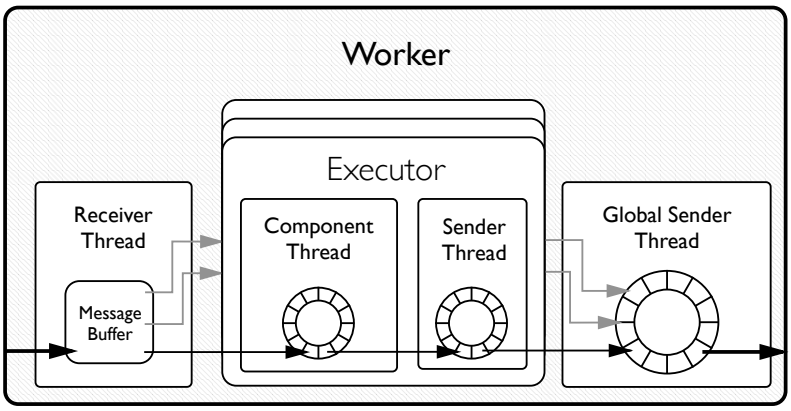


Figure 4.1: Tuple processing in Apache Storm.

Storm uses the term "queue" freely but in fact the queues are implemented as ring buffers using the LMAX Disruptor library [5]. Detailed background on how Disruptor works and its performance benchmarks can be found in [19].

There is a lot of overhead necessary to simulate sending tuples to executors in other worker nodes. For one, there is significant overhead from the tuple passing through the three queues of a worker. In [6], the authors of LMAX Disruptor show that a three step pipeline has half the throughput of a single consumer-producer pipeline.

Furthermore, Storm simulates the cluster network by running a local Netty [10] server.

4.1.2 Heartbeats & Timers

Additionally, every worker has a heartbeat thread that simulates sending heartbeat messages to the Nimbus process. It does this by writing to a local cache which is persisted to a file by a blocking write on every heartbeat. While heartbeats are essential in cluster mode, there is no need for them in local mode.

4.1.3 Acker & System Bolts

The acker and system bolts are ran for every topology. The acker bolt can be disabled via the configuration file. In such a case it executes idly.

4.1.4 Zookeeper

Another overhead is running a local Zookeper server which emulates the Zookeeper nodes of a cluster. Running the Zookeeper server is a massive addition to the list

of overheads. Indeed, during profiling we found that a topology with one worker and three executors was being executed with 55 threads (not including system JVM threads and threads created by the profiler). 28 of these threads were started by Zookeeper. Table 4.1 shows what the individual threads were used for.

Spout Parallelism	# of Threads
Main Thread	1
Worker Sender & Receiver Threads	2
Acker & System Component Threads	2
Executor Component Threads	3
Executor Sender Threads	5
Various Timers & Event Loops	13
Zookeper Server	28

Table 4.1: Break-down of threads used by Apache Storm to execute a topology with 3 components.

To find out what state the threads were actually in at any given time we run the topology and recorded a thread dump every second for five minutes. The results can be observed in table 4.2.

Spout Parallelism	# of Threads
RUNNABLE	8
TIMED WAITING	22
WAITING	25

Table 4.2: Recorded thread states over five minute period.

As can be seen from the table, most of the threads were either in state `WAITING` or `TIMED WAITING`. According to the Java documentation on thread states [?] these two states are used for threads that are waiting for an action from a different thread and cannot be scheduled by the scheduler until that action is executed.

In the subsequent sections we will discuss how the number of running threads was significantly reduced. In fact, to execute the same topology on Storm-MC would only require 5 threads.

Plot threads vs components

Maybe include Thread Dump and a graph of thread counts

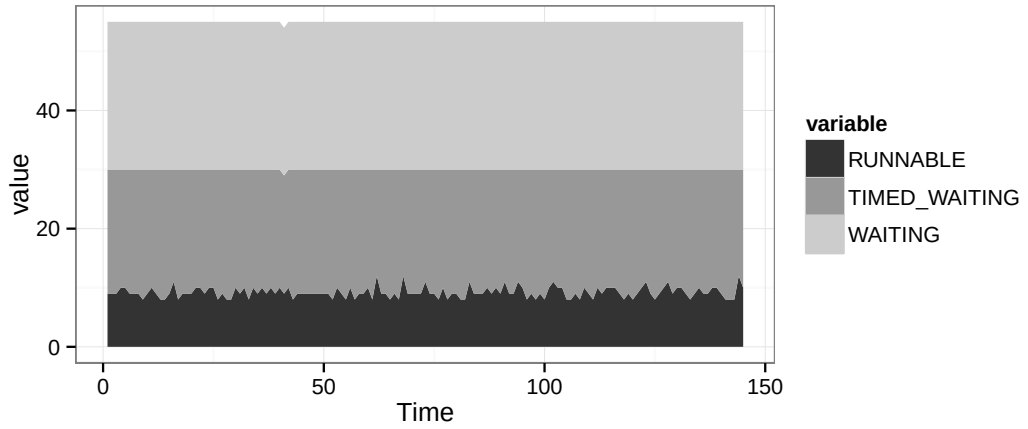


Figure 4.2: Threads over time.

4.2 Storm-MC Architecture

Probably don't call this Architecture

The design we adopted for porting worker nodes is to only have one worker process running all the executor threads of a topology.

Additionally, the code for the Nimbus daemon was merged with the worker. This was done because there is no need to run the Nimbus and worker specific code at the same time. Once Nimbus sets up the topology, all the work is done by the worker. Hence they can be executed serially.

4.2.1 Nimbus

Unlike Nimbus executing on a Storm cluster, Nimbus in Storm-MC does not support running multiple topologies at the same time. However, to do that one only needs to run the topology in a separate process. This is because unlike when executing on the cluster different topologies do not need to share any state and it is more natural to execute them as separate processes.

This has the added benefit of each process having its own part of main memory thus reducing cache conflicts as shown in [13] and providing higher security by not having different topologies share memory space. Additionally, if a single thread of one topology is blocking it does not block other topologies.

Nimbus on Storm-MC does not support scheduling topologies. Since within one process there is only one topology running at a time and the hardware configuration of the machine does not change, the parallelism is clearly defined by the number of executors per component specified in the topology configuration.

One way to implement scheduling could be to pin threads to specific cores. Unfortunately, Java does not provide support for CPU affinity, the assignments are handled automatically by the JVM. Potentially, this could be achieved by using C or C++, both of which support CPPU affinity, but this was not implemented in Storm-MC.

Mention porting over from Thrift.

The role of Nimbus in Storm-MC has effectively been reduced to validating the topology and passing it along to the worker part of the process which handles the topology execution.

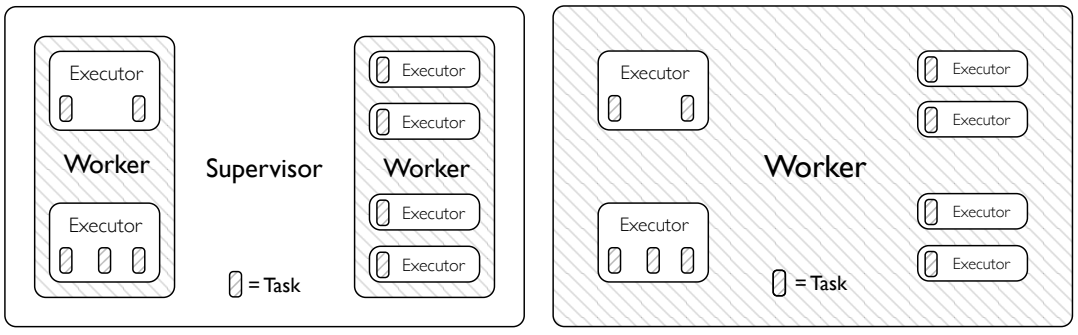
4.2.2 Worker

In Apache Storm, a worker node runs the supervisor daemon, which in turns launches worker processes which contain executor threads which contain tasks. In Storm-MC, however, there is only one worker process which contains all the executor threads and their tasks.

This design has several benefits:

- All the inter-thread communication is occurring within one Worker.
- Supervisor can be removed as there is no need to synchronise workers.
- There is no need to simulate over-the-network message passing.
- Message passing between executor threads within a worker stays the same as in Apache Storm.

A comparison of an Apache Storm worker node and its Storm-MC equivalent can be seen in figure 4.3.



(a) Worker in Apache Storm. (b) Worker in Storm-MC.

Figure 4.3: Comparison of a worker in Storm and Storm-MC

4.2.3 State

In light of previous subsections Storm-MC is completely stateless. The cluster state that was managed by Zookeeper in Apache Storm was completely stripped away. This state was only relevant when multiple topologies were sharing the cluster.

4.3 Tuple Processing

Mention overhead by running SystemBolt vs just timer in Storm-MC

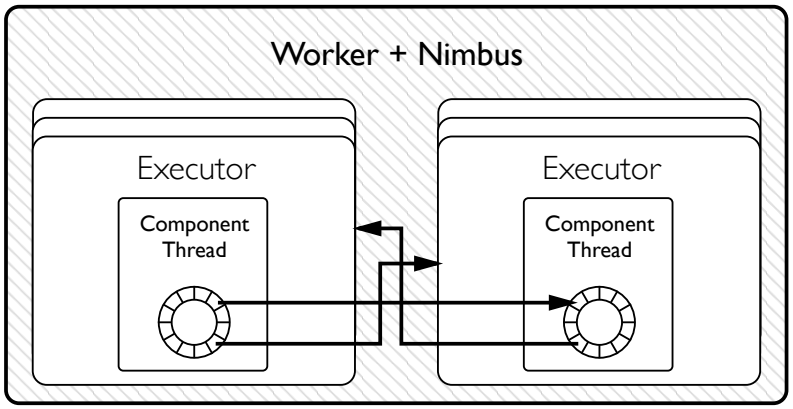


Figure 4.4: Tuple processing in Storm-MC.

The implementation of tuple processing in Storm-MC can be seen in figure 4.4. As can be seen from the figure, the queues used for remote message sending were stripped away and there is only one Disruptor queue for every executor. Once an executor is done processing a tuple it simply puts it on the Disruptor queue of the bolts downstream.

There were several other options we considered when implementing tuple processing. However, the Disruptor shows superior throughput and latency compared to alternative solutions in [?].

find the citation for above claim.

4.4 Differences between Apache Storm and Storm-MC

Maybe present this as a table of features.

Chapter 5

Storm-MC Implementation

The following chapter describes the implementation of Storm-MC.

5.1

Chapter 6

Evaluation

In this chapter we evaluate Storm-MC. We describe the metrics used to evaluate performance of Storm-MC (6.1), list the configuration used for benchmarking (6.2), compare Storm-MC to Apache Storm executing in local mode on a set of different topologies (6.3), and finally talk about challenges encountered while designing Storm-MC (6.4).

6.1 Evaluation Metrics

The system was evaluated on the following metrics:

Throughput
: The number of tuples processed by every component in the given time of the topology is recorded.

CPU utilisation
: Usage of CPU is recorded every **x** seconds throughout execution and is then averaged.

Memory utilisation
: Main memory usage is recorded every **x** seconds throught execution and is then averaged.

6.2 System Configuration

6.2.1 Software Setup

Change versions below as applicable. Link to GitHub for source?

All performance benchmarks were ran using the following software packages:

- Apache Storm version 0.9.2
- Storm-MC version 0.1.5
- A fork of IBM Storm Email Benchmarks version 0.1.4
- Storm-benchmark version 0.1.0

The Apache Storm source code had to be adapted to include a workaround for a deadlock bug present in version 0.9.2. This bug caused a topology to exit with threads left in Zombie state under certain conditions. This prevented Storm from logging the benchmark metrics after execution.

Version 0.1.5 is the latest version of Storm-MC as of this moment. The first release was version 0.1.0 which was production-ready but since then there were 5 minor versions fixing bugs as they were discovered during testing.

IBM open sourced a suite of benchmarks which they used to compare Apache Storm to their real-time stream system IBM Infosphere Streams. These benchmarks were adapted and used to benchmark Apache Storm and Storm-MC.

Lastly, a number of spout and bolt components were used from the storm-benchmark project which Apache Storm developers use to benchmark Storm.

Since Storm-MC reuses package names from Apache Storm, the same benchmark is directly executable by both libraries. This saved a lot of time and furthermore there is no need to maintain two benchmarks suites.

Go into more detail which components were re-used and where?

6.2.2 Hardware Setup

The benchmarks were executed on the following hardware:

This is currently student.compute, find out what it is.

6.3 Performance

Say how long they were executed for.

To assess performance of Storm-MC, 4 different benchmarks were executed, each with a different focus. The benchmarks were executed for a constant period of time after which the system was killed and metrics collected.

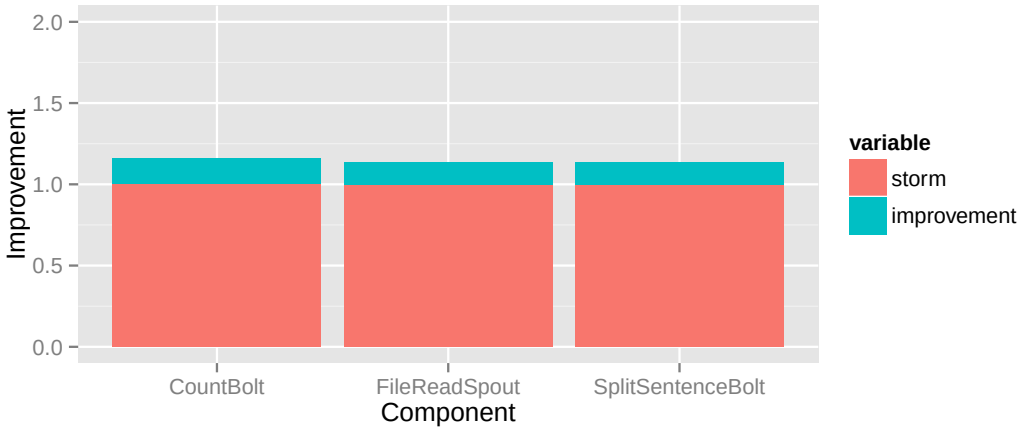


Figure 6.1: Improvement of Storm-MC over Apache Storm in number of tuples processed

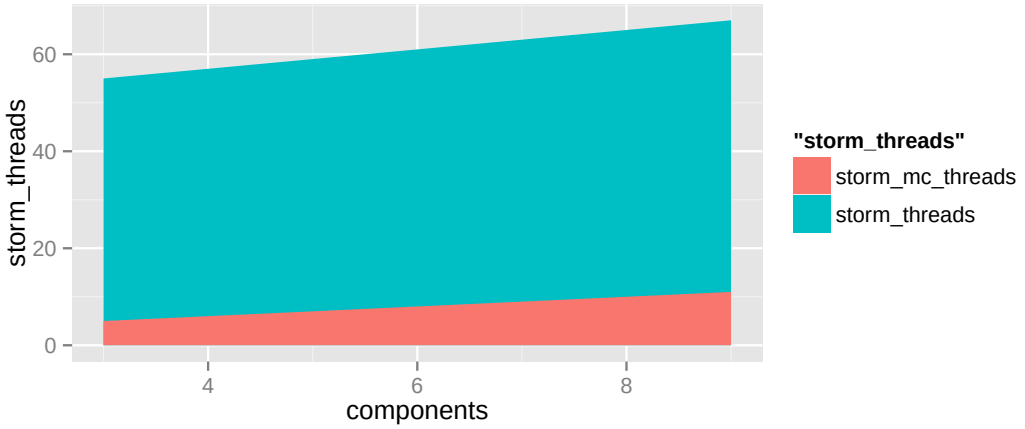


Figure 6.2: Number of threads used by Storm and Storm-MC

6.3.1 WordCount Topology

The first topology we tested for performance is a variant of the aforementioned WordCount topology. Recall, that this topology is shown graphically on figure 3.1. The topology was ran with 3 executors running for every component.

This topology is considered to be CPU-intensive.

Spout Parallelism	Split Bolt Parallelism	Count Bolt Parallelism	# of Threads
1	1	1	55
2	2	2	61
3	3	3	67

When the topology was run on Apache Storm in local mode, the process executed

with 55 threads. Compared to that, running it on Storm-MC required only 5 threads: the main thread (1), one thread for each component (3), and a user timer used for topology metrics and ticks (1).

6.3.2 Enron Topology

Next, we tested the Enron topology from the IBM benchmarks. In this topology, serialised emails from the Enron email database are read from a file by a spout. They are further deserialised by one bolt, filtered by another bolt, modified by yet another bolt and then finally metrics are recorded by another bolt.

Similarly, to the WordCount topology this topology is serial in nature. However, whereas the WordCount topology keeps the random sentences in memory, the Enron topology reads from a file. Thus, this benchmark is mostly I/O intensive.

6.3.3 RollingSort Topology

The RollingSort topology is ported over from the aforementioned storm-benchmark project. This topology includes one spout and one bolt. The spout produces hundred character-long strings of random digits from zero to eight. The bolt stores a rolling window of hundred of these messages and sorts them every **x** seconds.

This benchmark is included because it is considered memory-intensive.

change x depending on the actual benchmark.

6.4 Challenges

In this section we are going to discuss challenges we encountered while porting Apache Storm to multi-core machines.

Unfamiliarity with Clojure

Lack of Documentation

Chapter 7

Conclusion

7.1 Future Work

Make Storm-MC a server

Appendices

Appendix A

Listings

Listing 2 shows the definition of a spout that emits a randomly chosen sentence from a predefined collection of sentences.

Listings 3 and 4 show how a bolt defined in Python can be part of this Java-defined topology.

Finally, listing 5 shows how a bolt that counts the number of word occurrences can be implemented.

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    public void open(Map conf, TopologyContext context,
            SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }

    public void nextTuple() {
        Utils.sleep(100);
        String[] sentences = new String[]{
            "the cow jumped over the moon",
            "an apple a day keeps the doctor away",
            "four score and seven years ago",
            "snow white and the seven dwarfs",
            "i am at two with nature" };
        String sentence = sentences[_rand.nextInt(sentences.length)];
        _collector.emit(new Values(sentence));
    }

    public void ack(Object id) {}

    public void fail(Object id) {}

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }

}
```

Listing 2: RandomSentenceSpout.java

```
public static class SplitSentence extends ShellBolt implements IRichBolt {

    public SplitSentence() {
        super("python", "splitsentence.py");
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }

    public Map<String, Object> getComponentConfiguration() {
        return null;
    }
}
```

Listing 3: SplitSentence.java

```
import storm


class SplitSentenceBolt(storm.BasicBolt):

    def process(self, tup):
        words = tup.values[0].split(" ")
        for word in words:
            storm.emit([word])


SplitSentenceBolt().run()
```

Listing 4: splitsentence.py

```
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if (count == null)
            count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

Listing 5: WordCount.java

Bibliography

[1]

[2]

[3]

[4]

[5]

[6]

[7] Apache Kafka. http://kafka.apache.org. [Online; accessed 15-March-2015].

[8] Apache Thrift. https://thrift.apache.org. [Online; accessed 15-March-2015].

[9] Apache Zookeeper. http://zookeeper.apache.org. [Online; accessed 15-March-2015].

[10] Netty. http://netty.io. [Online; accessed 15-March-2015].

[11] Rabbit MQ. http://www.rabbitmq.com. [Online; accessed 15-March-2015].

[12] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *J. Mach. Learn. Res.*, 11:1601–1604, August 2010.

[13] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: A review. *SIGMOD Rec.*, 34(2):18–26, June 2005.

[15] Lukasz Golab and M Tamer Özsu. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003.

[16] G. Holmes, A. Donkin, and I.H. Witten. Weka: A machine learning workbench. In *Proc Second Australia and New Zealand Conference on Intelligent Information Systems*, Brisbane, Australia, 1994.

[17] Nathan Marz. History of apache storm and lessons learned - thoughts from the red planet - thoughts from the red planet, Oct 2014. [Online; accessed 15-March-2015].

[18] Ed Solovey. Handling five billion sessions a day – in real time.

[19] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. May 2011.