Storm on Multi-core

Mark Nemec

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

2015

Abstract

This is an example of `infthesis` style. The file `skeleton.tex` generates this document and can be used to get a "skeleton" for your thesis. The abstract should summarise your report and fit in the space on the first page. You may, of course, use any other software to write your report, as long as you follow the same style. That means: producing a title page as given here, and including a table of contents and bibliography.

Acknowledgements

Acknowledgements go here.

Table of Contents

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.
- Running a distributed computation on commodity hardware usually requires a data centre or renting 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 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

Background

In this chapter we give background information necessary to understand the design of Storm-MC. To ensure API compatibility with Apache Storm, Storm-MC was developed in Java and Clojure.

2.1 Core Concepts

There are several core concepts used by Storm and hence by extension Storm-MC as well. The 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 or Kafka 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.

2.2 Example Topology

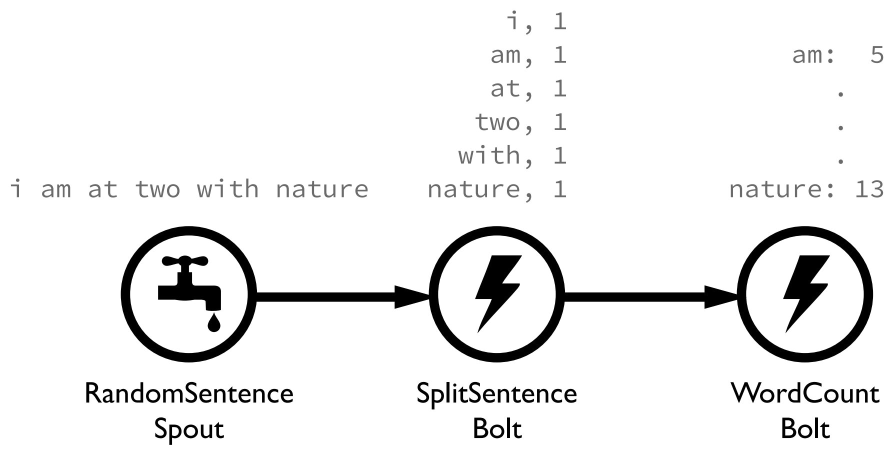


Figure 2.1: WordCount topology.

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 2.1 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.

```java
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 the pseudo cluster. The last argument to the setBolt/setSpout method is the number of parallel tasks we want Storm to execute for the respective component.

```
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

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

```
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

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

```java
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

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

2.3 Additional Concepts

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

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 consumer tasks.

Fields Hashes on a subset of the tuple attributes/fields.

All Replicates the entire stream to all the consumer tasks.

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

Global Sends the entire stream to a single bolt.

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

2.4 Storm Architecture

Maybe highlight similarities to Hadoop

Link to Master-Worker paper here?

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.

Link to Apache Zookeper here?

2.4.1 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:

Validating 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.

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

Scheduling 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 Round-robin strategy.

Confirm that above is actually Round-robin.

Activating the topology. Nimbus transitions the topology to active state which means the worker nodes can start executing it.

Monitoring the topology. Nimbus continues to monitor the topology by reading heartbeats sent by the worker nodes to ensure that it is executing as expected.

Is listing the steps involved the right way to explain it?

Nimbus is an Apache Thrift service (more on Thrift in section 2.5) that listens to commands submitted by clients and modifies the state of a cluster accordingly. Following are commands supported by Nimbus:

Link to Apache Thrift here?

Submit a topology
Clients can submit a topology defined in a 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.

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.

Is listing this really necessary?

2.4.2 Worker Nodes

The worker nodes run a daemon called Supervisor. There are 4 abstractions 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. 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 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 (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.

2.4.3 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.

2.5 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.

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

Chapter 3

Bringing Storm to Multi-core

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

3.1 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.

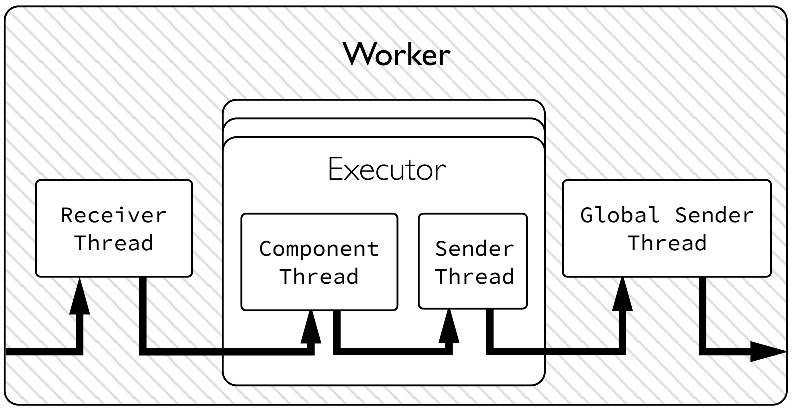


Figure 3.1: Inside the Worker process.

add arrows to other executors and show queues.

Figure 3.1 shows how tuple processing is implemented inside a Storm worker process. The tuple is read by the receiver thread of the worker and put on worker's receiver queue. The tuple is then picked up by the component thread of one, or possibly more, executors. After the executor has executed its task it puts the tuple on its sender queue. There, it is picked up by its sender thread which puts the tuple on the global sender queue of the worker. Lastly, the sender thread of the worker serialises it and sends it down-stream.

Clearly, there is a lot of overhead involved and there is not much use for the processing to be so complex on a single machine.

Storm runs many threads which are only useful in distributed context. Indeed, during our experiments we found that a topology with 8 executors was being executed with 64 threads.

Storm runs Zookeeper on localhost and sends executor heartbeats to it.

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. This is done by running a local Zookeper server which replaces the Zookeeper nodes of a cluster.

the only overhead in Storm is that the threads exist, they don't have to be used

3.2 Storm-MC Architecture

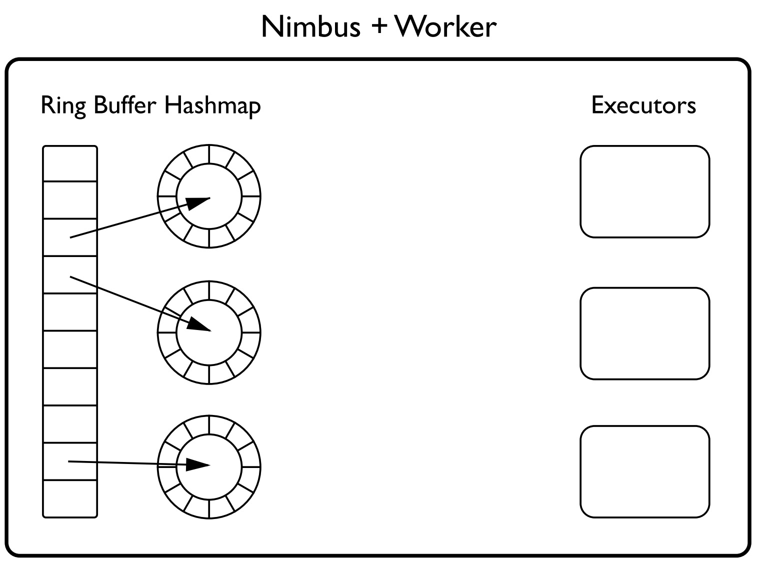


Figure 3.2: Storm-MC Architecture.

The overall architecture of Storm-MC can be seen in figure 3.2. 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.

3.2.1 Nimbus

Rewrite paragraph below, too informal.

In Apache Storm, Nimbus runs as a server. While appropriate for a system running on a cluster this is unnecessary on a single machine. Thus to run a topology on Storm-MC one only has to import the library into their project and use it as a standard Java library. To kill the topology, one only needs to send the kill signal as with a standard Java program. Furthermore, there is no need for activating and deactivating as the user can just kill and resubmit the topology. Lastly, rebalancing does not really transfer to a multi-core system and was thusly omitted.

As can be understood from the previous paragraph, Storm-MC does not support running multiple topologies at the same time. However, to do that one only needs to run a separate process. This is because unlike on the cluster different topologies do not need to share any memory and it is more natural to execute them as separate processes.

Mention things like cache lines in support of above argument.

Additionally, Storm-MC does not support any scheduling. 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 a C or C++ library but this was not implemented in Storm-MC.

next paragraph doesn't connect to anything.

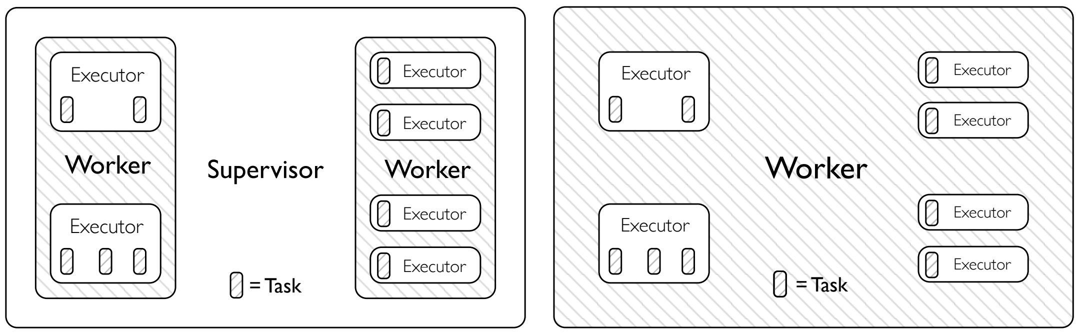
In light of this Storm-MC is completely stateless. The cluster state that was managed by Zookeeper in Apache Storm was completely stripped away.

3.2.2 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.



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

Figure 3.3: Comparison of a worker in Storm and Storm-MC

- There is no need to simulate over-the-network message passing.
- Message passing between executor threads within a worker stays the same.

A comparison of an Apache Storm worker node and its Storm-MC equivalent can be seen in figure 3.3.

3.2.3 Message Exchange

LMAX Disruptor is used for tuple passing between components of a stream. Detailed background of how Disruptor works and its performance benchmarks can be found in [1].

There were two types of tuple transfers in Apache Storm:

Inter-worker (remote) transfers
When the tuple is sent to a different worker, it is put on a disruptor buffer by an executor thread and picked up by a separate sender thread which sends it across the network to the executors subscribed to the stream. The worker of these executors runs a receiver thread which listens to messages on its port and puts the tuples on disruptor buffers of the corresponding executors.

Intra-worker (local) transfers
When the tuple is sent within the same worker, it is put directly on the disruptor buffer of the executor thus avoiding sending a network message to the same machine.

Since Storm-MC only has one worker process, all tuple transfers are local. Thus, a map of executor identifier to disruptor buffers is maintained. Hence, an executor thread picks up a tuple from the ring buffer corresponding to its identifier and after processing the tuple inserts it into a buffer of executors subscribed to its output stream.

3.3 Things Not Included in Storm-MC

This section name should be rephrased.

Maybe present this as a table of features.

Chapter 4

Evaluation

In this chapter we evaluate Storm-MC. We do this by comparing its performance against the local mode of Apache Storm.

4.1 Performance

Chapter 5

Conclusion

5.1 Future Work

Bibliography

[1] 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.