Modeling Performance of a Parallel Streaming Engine: Bridging Theory and Costs

Ivan Bedini Sherif Sakr Bart Theeten Alessandra Sala Peter Cogan
Alcatel-Lucent Bell Labs, Ireland and Belgium
{firstname.lastname}@alcatel-lucent.com

ABSTRACT

While data are growing at a speed never seen before, parallel computing is becoming more and more essential to process this massive volume of data in a timely manner. Therefore, recently, concurrent computations have been receiving increasing attention due to the widespread adoption of multi-core processors and the emerging advancements of cloud computing technology. The ubiquity of mobile devices, location services, and sensor pervasiveness are examples of new scenarios that have created the crucial need for building scalable computing platforms and parallel architectures to process vast amounts of generated streaming data. In practice, efficiently operating these systems is hard due to the intrinsic complexity of these architectures and the lack of a formal and in-depth knowledge of the performance models and the consequent system costs. The *Actor Model* theory has been presented as a mathematical model of concurrent computation that had enormous success in practice and inspired a number of contemporary work in this area. Recently, the *Storm* system has been presented as a realization of the principles of the Actor Model theory in the context of the large scale processing of streaming data. In this paper, we present, to the best of our knowledge, the first set of models that formalize the performance characteristics of a practical distributed, parallel and fault-tolerant stream processing system that follows the Actor Model theory. In particular, we model the characteristics of the data flow, the data processing and the system management costs at a fine granularity within the different steps of executing a distributed stream processing job. Finally, we present an experimental validation of the described performance models using the Storm system.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Multiple Data Stream Architectures (Multiprocessors)— *Parallel processors*; C.1.4 [**Computer Systems Organization**]: Parallel Architectures—*Distributed Architecture*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICPE'13, April 21–24, 2013, Prague, Czech Republic.
Copyright 2013 ACM 978-1-4503-1636-1/13/04 ...$15.00.

Keywords

Distributed Stream Processing Engine, Performance Model, Measurement

1. INTRODUCTION

Today's era of *Big Data* is witnessing a continuous increase of user and machine connectivity that produces an overwhelming flow of data which demands a paradigm shift in the computing architecture requirements and large-scale data processing mechanisms. Therefore, concurrent computations have been receiving increased attention due to widespread adoption of multi-core processors and the emerging advancements of cloud computing technology. For example, the MapReduce framework [**?**] has been introduced as a scalable and fault-tolerant data processing framework that enables the processing of a massive volume of data in parallel on clusters of horizontally scalable commodity machines. By virtue of its simplicity, scalability, and fault-tolerance, MapReduce is becoming ubiquitous, gaining significant momentum within both industry and academia. However, the MapReduce framework, open sourced by the *Hadoop*[1] Implementation, and its related large-scale data processing technologies (e.g. *Pig*[2] and *Hive*[3]) have been mainly designed for supporting *batch* processing tasks but they are not adequate for supporting *real-time stream* processing tasks [**?**]. The ubiquity of mobile devices, location services, sensor pervasiveness and real-time network monitoring have created the crucial need for building scalable and parallel architectures to process vast amounts of *streamed* data.

In general, distributed stream processing systems support a large class of applications in which data are generated from multiple sources and are pushed asynchronously to servers which are responsible for processing. Therefore, stream processing applications are usually deployed as continuous jobs that run from the time of their submission until their cancellation. The *Actor Model* [**?**] is a mathematical model of concurrent computation which has enjoyed enormous success and has inspired a number of systems in this area. Today, this field is mature and provides the proper foundation to operate a horizontally scalable system for massive parallel computations of big data [**?**, **?**]. Recently, Twitter has open-sourced ***Storm***[4] as a parallel, distributed and fault-tolerant system which is designed to fill the gap of providing a platform that supports real-time processing of large scale

[1] http://hadoop.apache.org/
[2] http://pig.apache.org/
[3] http://hive.apache.org/
[4] https://github.com/nathanmarz/storm/wiki

streaming data on clusters of horizontally scalable commodity machines. In principle, the Storm system represents a realization of all the principles of the actor model in the context of concurrent processing on streaming data. In particular, it relies on the idea of creating and interconnecting interactive universal entities, called *Actors*, which are responsible for executing tasks concurrently and asynchronously while communicating with each other through a message-passing protocol [**?**, **?**].

Contributions. In practice, many researchers and engineers are currently increasingly shifting to parallel and distributed systems for large-scale data precessing. However, the lack of expertise and in-depth understanding of the theoretical foundation of this area interferes with the efficient analytic design or with the proper resource allocation strategies to achieve expected performance [**?**, **?**]. In order to tackle this challenge, we introduce, to the best of our knowledge, the first set of performance models that formalize the performance characteristics of an Actor Model based distributed stream processing engine, i.e. Storm. This performance model can play a fundamental role in guiding the user of these systems with the understanding of the system costs which are independent of the executed jobs; the level of parallelization necessary to efficiently execute a particular job; and to better predict the latency, throughput and physical resource costs. In particular, we formalize three main performance cost components as follows:

- *Data Flow Costs*: This component captures information about the amount of data flowing (transmitted) through the different processing steps of a distributed stream processing job.
- *Data Processing Costs*: This component captures information about the execution time of the different processing steps within a distributed stream processing job.
- *System Management Costs*: This component captures information about the system communication overhead between the different components of a distributed stream processing job for ensuring the reliability and fault tolerance during the execution steps.

We use these components to model the general performance characteristics of an Actor Model based implementation of a distributed stream processing system. Finally, we leverage these models to drive our experimental evaluations within the Storm framework.

RoadMap. The remainder of this paper is organized as follows. Section **??** defines the preliminary concepts used in this paper about the *Actor Model* and the *Storm* system. We present the modelling of the data flow costs in Section **??**. The modelling of the data processing costs is presented in Section **??** while the modelling of the system management costs is presented in Section **??**. The results of our experimental evaluation are presented in Section **??**. We discuss the related work in Section **??** before we finally conclude the paper in Section **??**.

2. BACKGROUND AND MOTIVATIONS

This section introduces the fundamental concepts of the Actor Model which are used in this paper and provides an overview of the Storm system which is used as our case study for implementing distributed stream processing jobs using the concurrent computations principles of the Actor Model.

2.1 Actor Model Theory

The Actor Model is one of the first computational models that proposes substantial differences from the Turing Machine Model [**?**]. In particular, the fundamental difference with the Turing Model is the absence of global states in the Actor Model. Indeed, the Turing Model makes use of global states that have also been described by McCarthy and Hayes in 1969, as "*the complete state of the universe at an instant of time*". The Actor Model is defined under a fundamentally different concept of "*reacting to input events*". These events are messages that flow through Actors and trigger responses/computations when computed in some Actor, therefore Actors are based on the concept of Partial ordering [**?**]. Since the arrival orderings are *indeterminate*, they cannot be deduced from prior information based on mathematical logic alone. This has been referred to as indeterminacy in concurrent computation by Hewitt [**?**]. Specifically, this logic evolved into the concept of unbounded nondeterminism in which the possible delay in serving a request can become unbounded given the arbitration of contention for shared resources. However, an important property is still guaranteed that "*the request will eventually be serviced*". In principle, the fundamental concepts of the Actor Model focuses on interpreting everything as an Actor entity. The concurrency model introduced with the Actor Model is based on three fundamental concepts:

1. The *Actors* are the computational units whereby they execute a computation on some input and generate an output.
2. *Actors' input* and *output* are realized through the concept of messages that are defined as the communication unit for exchanging data among Actors.
3. *Actors' communication channels* are realized through buffers or queues from where Actors read and send the data to process. In particular, Actors communicate with each others via simple primitives such as: receipt of messages from other Actors, execution of some computation on incoming messages, generation of output messages, and finally dispatch of output messages to other Actors.

In practice, jobs usually involve multiple Actors. These Actors communicate among each other only through the concept of *addresses* and can be visualized as a *graph topology* where each node is an Actor and a direct edge from Actor A to Actor B means that there is a flow of messages from A to B. However this graph may be highly dynamic where Actors have the possibility to reconfigure their topology on the fly based on the input messages, i.e. Actors can be dynamically added and removed from the system. Therefore, any concurrent system implemented through the Actor concurrency model is theoretically perfectly parallel and scalable.

2.2 From Theory to a Real Streaming System

The Storm system has been presented as a distributed and fault-tolerant stream processing system that instantiates the fundamental principles of Actor theory. The key design principles of Storm are:

- *Horizontally scalable*: Computations and data processing are performed in parallel using multiple threads, processes and machines.
- *Guaranteed message processing*: The system guarantees that each message will be fully processed at least

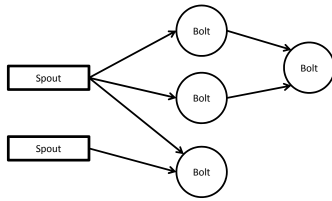


Figure 1: Sample Storm Topology

once. The system takes care of replaying messages from the source when a task fails.

- *Fault-tolerant*: If there are faults during execution of the computation, the system will reassign tasks as necessary.
- *Programming language agnostic*: Storm tasks and processing components can be defined in any language, making Storm accessible to nearly anyone. Clojure, Java, Ruby, Python are supported by default. Support for other languages can be added by implementing a simple Storm communication protocol.

2.2.1 Design Specifics in Storm

The core abstraction in Storm is the *stream*. A stream is an unbounded sequence of *tuples*. Storm provides the primitives for transforming a stream into a new stream in a distributed and reliable way.

The basic primitives Storm provides for performing stream transformations are *spouts* and *bolts*. A *spout* is a source of streams. A *bolt* consumes any number of input streams, carries out some processing, and possibly emits new streams. Complex stream transformations, such as the computation of a stream of trending topics from a stream of tweets, require multiple steps and thus multiple bolts.

A *topology* is a graph of stream transformations where each node is a spout or bolt. Edges in the graph indicate which bolts are subscribing to which streams. When a spout or bolt emits a tuple to a stream, it sends the tuple to every bolt that subscribed to that stream. Links between nodes in a topology indicate how tuples should be passed around.

Each node in a Storm topology executes in parallel. In any topology, we can specify how much parallelism is required for each node, and then Storm will spawn that number of *threads* across the cluster to perform the execution. Figure **??** depicts a sample Storm topology.

Communication Semantics. The Storm system relies on the notion of *stream grouping* to specify how tuples are sent between processing components. In other words, it defines how that stream should be partitioned among the bolt's tasks. In particular, Storm supports different types of stream groupings such as:

1. *Shuffle grouping* where stream tuples are randomly distributed such that each bolt is guaranteed to get an equal number of tuples.
2. *Fields grouping* where the tuples are partitioned by the fields specified in the grouping.
3. *All grouping* where the stream tuples are replicated across all the bolts.
4. *Global grouping* where the entire stream goes to a single bolt.

In addition to the supported built-in stream grouping mechanisms, the Storm system allows its users to define their own custom grouping mechanisms.

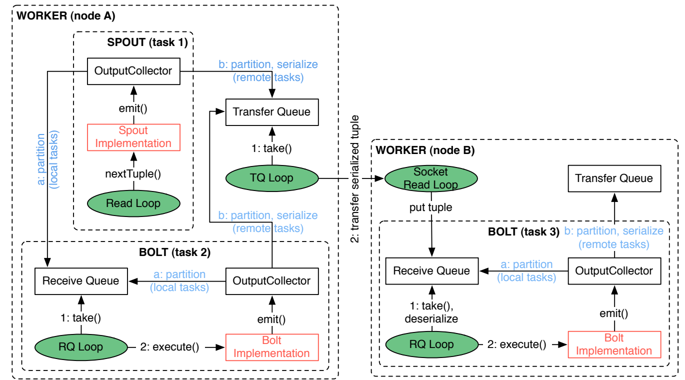


Figure 2: Execution of a Storm Topology

Storm Cluster. In general, a Storm cluster is superficially similar to a Hadoop cluster. One key difference is that a MapReduce job eventually finishes while a Storm job processes messages forever (or until the user kills it). In principle, there are two kinds of nodes on a Storm cluster:

- *The Master node* runs a daemon called ***Nimbus*** (similar to Hadoop's JobTracker) which is responsible for distributing code around the cluster, assigning tasks to machines, and handling failures.
- *The worker nodes* run a daemon called the ***Supervisor***. The supervisor listens for work assigned to its machine and starts and stops worker processes as necessary based on what Nimbus has assigned to it.

In a Storm cluster all the interactions between Nimbus and the Supervisors are done through a ***ZooKeeper***[5] cluster, an open source configuration and synchronization service for large distributed systems. Both the Nimbus daemon and Supervisor daemons are fail-fast and stateless, where all state is kept in ZooKeeper or on local disk.

Communication between workers living on the same host or on different machines is based on ZeroMQ sockets over which serialized java objects (representing tuples) are being passed.

2.2.2 The Execution Steps of Storm Topologies

In practice, a storm job is called a *topology* which continuously processes stream(s) of incoming tuples (events) within a Java Virtual Machine (JVM). Figure **??** illustrates the various phases in processing a tuple in a sample topology consisting of a single *spout* (task 1) and two *bolts* (task 2 and task 3) distributed over a 2 node cluster (Worker Node A and Worker Node B). The figure should be read starting at the *Read Loop* part of the spout. Here, tuples are continuously being read from an unspecified tuple source, which could for example be a text file or a queue. The spout's `nextTuple()` method is called continuously. The user-defined implementation of this method can for example read the next line of text or read the next tuple of the queue and transform the read data into a new tuple (i.e. a sequence of key-value pairs). Then, the implementation calls the `emit()` function on the spout's `OutputCollector` object to actually inject the tuple into the storm cluster. The `OutputCollector` applies the stream groupings onto the tuple which associates one or more *task ids* for each tuple. A task id uniquely refers to a particular task process/thread in the cluster where the tuple

[5]http://zookeeper.apache.org/

should be sent for processing. If it turns out that the task id refers to a local task, it will simply add the tuple onto the receive queue of the local task (task in the same worker). Otherwise, it needs to send the tuple to another worker (i.e. JVM), either on the same node or a different node in the cluster and therefore it is first serialized and then added to the transfer queue. The continuous loop, (*TQ Loop*), reads the serialized tuples off the transfer queue and sends them to the destination worker(s).

As shown in Figure **??**, the tuples emitted by the spout are sent directly to the subsequent bolt which is executing task 2. As this is a local task which is running in the same worker as the spout, the tuple is directly placed onto the receive queue of the local bolt. The *RQ Loop* continuously reads tuples off the receive queue and calls the user-defined `execute()` method on the bolt referred to by the task id associated with that tuple. When the implementation decides that a new tuple should be triggered, it should construct a new tuple and emit it to the bolt's `OutputCollector`. In the case of task 3, which is a remote task, the new tuple is first serialized and then placed on the worker's transfer queue. The *TQ Loop* will read the tuple and send it to node B where the tuple will be placed on the receive queue of task 3. The *RQ Loop* at node B will read the tuple, deserialize it and the same process as described above continues.

3. DATA FLOW COST MODEL

The data flow cost model is designed to capture information about the size of data flowing through an active storm topology and the cost of transferring these data between the processing actors.

3.1 Data Size

The size of data flowing in a Storm job is expressed per time unit (e.g. per second) and is composed of the following two components:

- **The input data size**: It represents the total amount of tuples which are injected into the storm cluster by the various spouts in the topology.
- **The output data size**: It represents the total amount of tuples which are generated and emitted by the various processing bolts in the topology and are consumed during further processing by other processing bolts.

Input Data Size In general, an input spout can generate input tuples of different types. The data size of an input spout per time unit can therefore be modeled as:

$$S.dataSize = \sum_{t=1}^{n} S.numTuples_t * tupleSize_t$$

where S refers to a particular input spout, n refers to the number of different types of input tuples produced by that spout, $S.numTuples_t$ refers to the average number of tuples (of type t) emitted by spout S per time unit and $tupleSize_t$ represents the average byte size of tuples of type t. The total size of data input per time unit of a particular storm topology is then defined as follows:

$$T.inputSize = \sum_{i=1}^{s} S_i.dataSize$$

where T refers to a particular topology and s refers to the total number of input spouts for that topology.

Output Data Size Each processing bolt of a topology consumes at least one stream of tuples and possibly generates and emits one or more new streams of tuples. The number of output tuples of a processing bolt for a specific input tuple type t per time unit is defined as follows:

$$B.numTuples_t = \sum_{d=1}^{m} E_d.numTuples_t * B.selectivity_t$$

where E_d refers to a source executor (i.e. spout or bolt) that emits tuples for which B is a destination, m is equivalent to the number of direct edges in the topology graph from an executor E to bolt B, and $B.selectivity_t$ represents the ratio between the number of input tuples of type t and the number of new output tuples being emitted as the result of processing those input tuples by a particular bolt B. In principle, each processing bolt can have different selectivity values for the different types of tuples it can process and it can produce tuples of multiple types as output. Therefore $B.selectivity_t$ can be expanded to:

$$B.selectivy_t = \sum_{t'=1}^{n'} B.selectivity_{t,t'}$$

where n' refers to the number of different types of output tuples produced by bolt B in response to receiving tuples of type t. As a consequence, the output data size of a processing bolt per time unit is defined as follows:

$$B.dataSize = \sum_{t=1}^{n} B.numTuples_t * \overline{tupleSize}$$

where $\overline{tupleSize}$ represents the average size of output tuples, which can be further expanded as:

$$\sum_{t=1}^{n} \sum_{d=1}^{m} \left(E_d.numTuples_t * \sum_{t'=1}^{n'} (B.selectivity_{t,t'} * tupleSize_{t'}) \right)$$

where n refers to the number of different input tuple types received by bolt B and n' refers to the number of different output tuple types produced and emitted by B.

Topology Data Flow Size. The total data flow size for a storm topology can then be represented as the summation of the total data input size injected by all spouts into the topology and the total data output size emitted by all the processing bolts in the topology. This can be expressed as:

$$T.dataSize = \sum_{i=1}^{s} S_i.dataSize + \sum_{j=1}^{b} B_j.dataSize$$

where s refers to the total number of spouts and b refers to the number of bolts which are defined in the executed topology.

3.2 Data Transfer Cost

The data transfer cost is the cost of actually delivering the tuple to a destination task (i.e. bolt instance). A distinction must be made between various allocations of communicating spouts and bolts across the cluster because running tasks remotely or locally produces substantially different communication costs in terms of bandwidth consumption and communication time, i.e. latency. Therefore, we distinguish between three categories of data transfer cost (α) as follows:

1. **Local Java virtual machine (JVM) allocation** (α_{local}): where communicating actors (i.e. spout→bolt or bolt→bolt) are allocated within the same worker and thus within the same JVM. In this case, the tuple serialization/ deserialization step via the transfer queue is by-passed and tuples are immediately placed on the consuming bolt's receive queue.

2. **Local node allocation** (α_{node}): where communicating actors are allocated on different workers that are hosted on the same cluster node. In this case, a serialization/ deserialization cost to send the tuple from one JVM to another is incurred.
3. **Remote allocation** (α_{remote}): where communicating actors are allocated on different workers hosted on different cluster nodes. This is the most expensive case as both a serialization/deserialization cost and a network transfer cost are incurred.

In practice, Storm's allocation strategy *sequentially* and *iteratively* allocates all instances of each component (spout or bolt) up to the configured parallelization factor, whereby an outer loop iterates across all workers and an inner loop iterates across all nodes in the cluster. In the following subsections, we describe how the fraction of tuples that can be processed according to each category can be calculated.

Calculating the Local JVM Allocation. Let $\mathcal{P}$ be the collection of worker processes running across the cluster and let $\mathcal{D}$ be the collection of communicating actors (spout→bolt or bolt→bolt) in the topology. For each process $p \in \mathcal{P}$ and for each communicating actor $d \in \mathcal{D}$ in the sender role (i.e. emitting tuples), let $Aemitter_{p,d}$ be the number of instances of that actor that are allocated in the process p. Similarly, for each process $p \in \mathcal{P}$ and for each communicating actor $d \in \mathcal{D}$ in the receiver role (i.e. receiving tuples), let $Areceiver_{p,d}$ be the number of instances of that actor that are allocated within the same process p. Finally, for each communicating actor $d \in \mathcal{D}$, let us assume that $Aemitter_d$ and $Areceiver_d$ represent respectively the total number of instances of the sending and receiving actors across the entire cluster. It follows that the fraction of tuples which are processed within a single JVM can be computed as:

$$\alpha_{local} = \frac{\sum_{d=1}^{|\mathcal{D}|} \sum_{p=1}^{|\mathcal{P}|} \left(Aemitter_{p,d} * Areceiver_{p,d} \right)}{\sum_{d=1}^{|\mathcal{D}|} Aemitter_d * Areceiver_d}$$

Calculating the Local Node Allocation. Let $n \in \mathcal{N}$ be a node in the cluster (where $|\mathcal{N}|$ is the cluster size), and $Aemitter_{n,d}$ and $Areceiver_{n,d}$ represent the number of instances of that actor d, respectively in the sender and receiver role, that are allocated on the same node n. The fraction of tuples that are processed among tasks within a single node α_{node} can then be derived by computing the distribution of the tasks among the nodes of the cluster, which we call α_{intra_node}, given by the following formula:

$$\alpha_{intra_node} = \frac{\sum_{d=1}^{|\mathcal{D}|} \sum_{n=1}^{|\mathcal{N}|} \left(Aemitter_{n,d} * Areceiver_{n,d} \right)}{\sum_{d=1}^{|\mathcal{D}|} Aemitter_d * Areceiver_d}$$

Thus, the fraction of tuples that are processed on the same node but on a different processes:

$$\alpha_{node} = \alpha_{intra_node} - \alpha_{local}$$

Calculating the Remote Allocation. Given the above formulas, the fraction of tuples that are transferred across the network to another node, contributing to the overall data transfer cost can be expressed as follows:

$$\alpha_{remote} = 1 - \alpha_{node}$$

Overall Data Transfer cost. Given the information of the fraction of tuples that are processed in each node and in each JVM, we can now compute the total IO cost per time unit as follows:

Let us assume that the stream comprises of a single tuple type and let Γ_{local}, Γ_{node} and Γ_{remote} be respectively the CPU cost of sending a tuple to the local queue (i.e. within the same JVM), to the same node (but on a different JVM), and to different nodes in the cluster, the total IO cost per time unit can be represented as follows:

$$IOCost = numTuples * IOTranfer$$

where

$$IOTransfer = \alpha_{local} * \Gamma_{local} + \alpha_{node} * \Gamma_{node} + \alpha_{remote} * \Gamma_{remote}$$

In the case where the stream comprises different types of tuples, the $IOCost$ would be computed for each tuple type t and summed together which leads to:

$$IOCost = \sum_{t=1}^{n} numTuples_t * IOTransfer_t.$$

In the above formulas, $numTuples$ and $numTuples_t$ refer respectively to the total amount of tuples and those tuples of type t, flowing through the system.

4. DATA PROCESSING COST MODEL

The data processing cost model describes the execution time of the different processing steps within an active storm topology as well as the IO cost of exchanging tuples between spouts and bolts. The model is again expressed per time unit and has the following two main components:

- *Spout processing cost*: This represents the cost of reading a raw event of an unspecified source and injecting a storm tuple into the topology.
- *Bolt processing cost*: This represents the cost of processing a tuple in a bolt and possibly emitting new tuples into the topology for further processing.

In the following, we provide an in-depth analysis of the CPU processing costs for spouts and bolts, by identifying all the logical operations that are a factor of these costs.

4.1 Spout Processing Cost

Figure **??** illustrates the data processing steps in the processing component (e.g. spout or bolt) which can be split in a number of sequentially executed phases:

1. *Read*: reads a raw data item from an event source (e.g. a queue or a file).
2. *Transform* (γ): formats the raw data item into a storm tuple sequence.
3. *Emit*: adds the storm tuple sequence to the spout's output collector. There will be one destination per direct edge from the spout to a bolt in the topology.
4. *Partition*: defines where (i.e. which task) to send the tuple sequence to for each destination defined in the *emit* step. Depending on the chosen stream grouping, this could be one or more tasks (i.e. bolt instances identified by a node and port pair).
5. *Serialize*: transforms the tuple sequence in a format that can be transmitted efficiently across the network or across JVMs. This phase will only be executed when there is at least one remote destination, i.e. a bolt task that is not running within the emitting spout's JVM.

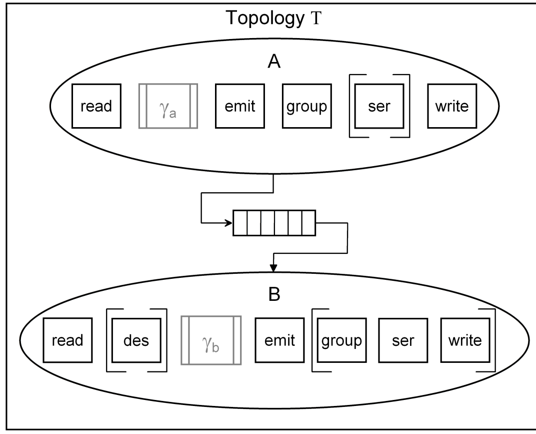


Figure 3: Data Processing Steps in a Processing Component

The *Read*, *Transform* and *Emit* phases are part of the user-defined implementation logic of the spout, while the remaining phases are part of the Storm platform.

The CPU processing cost for an input spout (S) per time unit can therefore be modeled as:

$$S.CPUCost = \sum_{t=1}^{n} S.numTuples_t * S.CPUCost_t$$

with

$$S.CPUCost_t = S.inputCost_t + S.outputCost_t$$

where

$$S.inputCost_t = S.CPURead_t + S.CPUTransform_t$$

and where

$$\begin{aligned} S.outputCost_t &= \sum_{i=1}^{S.numDest_t} B_i.groupingCost_t \\ &+ S.hasExternalTask_t * CPUSerialize_t \end{aligned}$$

where

- n is the number of the different types of tuples which are emitted by the spout S.
- $S.numTuples_t$ is the number of tuples of type t, emitted by spout S per time unit.
- $CPURead_t$ is the CPU cost of reading a raw input tuple of type t.
- $CPUTransform_t$ is the CPU cost of transforming a raw input tuple of type t into a Storm tuple of type t.
- $S.numDest_t$ is the number of destination bolts that consume tuples of type t emitted by spout S.
- $B_i.groupingCost_t$ represent the cost of partitioning the input tuples of type t which varies according to the stream grouping algorithm (*Shuffle*, *Fields*, *All*, *Global*) chosen by the processing bolt B_i.
- $S.hasExternalTask_t$ is a boolean variable which stores the value 1 if the tuples of type t emitted by spout S are consumed by at least one external task. An external task is an instance of a bolt running on a different worker than the one which is hosting the instance of spout S that is emitting the tuple. Note that this could still be on the same host, but on a different JVM. Otherwise it stores the value 0.
- $CPUSerialize_t$ is the CPU cost of serializing a tuple of type t.

4.2 Bolt Processing Cost

In practice, bolt processing involves running through the same sequence of phases as spout processing with the only difference being that of instead of having a *transform* phase, there is a more general *execute* phase in which the actual bolt's processing logic is executed. It is also in this execute phase that the decision will be made as to which and how many new tuples (of type t') will be emitted by the bolt. Therefore, the CPU processing cost for a processing bolt (B) per time unit is modelled as the sum of the cost to process all input tuples of type t received per time unit and the cost of processing all resulting output tuples (of type t'), for all possible tuple types this bolt can receive. This formula is shown as:

$$B.CPUCost = \sum_{t=1}^{n} B.numInputTuples_t * B.inputCost_t + \sum_{t'=1}^{n'} (B.numOutputTuples_{t,t'} * B.outputCost_{t'})$$

with

$$B.inputCost_t = B.CPURead_t + B.CPUExecute_t + B.isExternalTask * CPUDeserialize_t$$

and

$$\begin{aligned} B.outputCost_{t'} &= \sum_{i=1}^{B.numDest_{t'}} B_i.groupingCost_{t'} \\ &+ B.hasExternalTask_{t'} * CPUSerialize_{t'} \end{aligned}$$

where

- $B.numInputTuples_t$ is the number of tuples of type t received by the bolt B, which, as was shown before, can be represented as:

$$B.numInputTuples_t = \sum_{d=1}^{m} E_d.numTuples_t$$

  where E_d refers to the source executor (spout or bolt) that emits tuples for which B is a destination, m is the amount of executors that emit to this bolt and $E_d.numTuples_t$ refers to the number of tuples of type t emitted by executor E_d
- $B.numOutputTuples_{t,t'}$ is the number of new tuples of type t' generated and emitted by the bolt B in response to receiving input tuples of type t, which can be expressed as:
  $B.numOutputTuples_{t,t'} =$
  $B.numInputTuples_t * B.selectivity_{t,t'}$
- $B.CPURead_t$ is the cost of reading an input tuple of type t by bolt B, excluding potential deserialization cost.
- $B.CPUExecute_t$ is the cost of the algorithm implemented by bolt B.
- $B.isExternalTask$ is a boolean variable that stores the value 1 if the tuple being read was sent by an external task and is therefore in serialized form. Otherwise, it stores the value 0.
- $CPUDeserialize_t$ is the CPU cost of deserializing a tuple of type t.

4.3 Topology cost

The total processing cost for a storm topology represents the summation of the total CPU processing cost for all of its spouts and bolts. Therefore, the topology cost, i.e.

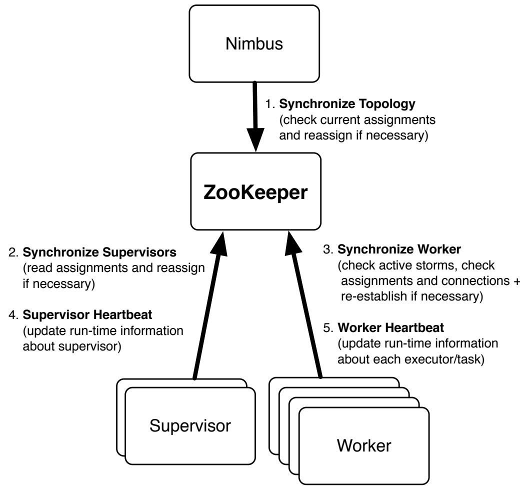


Figure 4: ZooKeeper interactions

$T.CPUCost$, for a storm topology is defined as:

$$T.CPUCost = \sum_{i=1}^{s} S_i.CPUCost + \sum_{j=1}^{b} B_j.CPUCost$$

where s represents the total number of spouts in the topology and b represents the total number of bolts in the topology.

5. SYSTEM MANAGEMENT COST MODEL

The *System Management* costs model the impact of the set of tasks, abstracted away from the user, which are required to run a Storm cluster. These tasks include provision of support for node failure/addition/removal, JVM failures, network issues etc. Storm's system management tasks are coordinated through ZooKeeper and are mainly related to the interaction between ZooKeeper and Nimbus, Supervisor and Worker as shown in Figure **??**. There are 5 recurring System Management tasks that interact with ZooKeeper, and the frequency of these interactions can be configured manually. These parameters are described in the following list:

1. **Synchronize Topology**: Every configurable $\mathcal{S}_t$ seconds, Nimbus checks the active assignments and compares them to the required assignments according to the topology specification. If a difference is detected, e.g. because of node failure, Nimbus will reassign the unassigned tasks over the available worker processes in the cluster. *(period: $\mathcal{S}_t$, number of requests: $\mathcal{R}_t$, bandwidth: $\mathcal{B}_t$)*
2. **Synchronize Supervisors**: Every configurable $\mathcal{S}_s$ seconds, each supervisor reads its assignments from ZooKeeper and re-assigns them if it detects a difference between what it has currently assigned across its workers. Re-assignment takes the form of updates to ZooKeeper's assignments for the workers to query during their next poll cycle.
(period: $\mathcal{S}_s$, number of requests: $\mathcal{R}_s$, bandwidth: $\mathcal{B}_s$)
3. **Synchronize Workers**: Every configurable $\mathcal{S}_w$ seconds, each worker reads its assignments from ZooKeeper. If there is a mismatch, the missing connections are established. In addition to this, each worker also checks the active storm topologies. If the storm topology for which it is running tasks would no longer be active (because it was explicitly killed), the worker would need to stop processing.
(period: $\mathcal{S}_w$, number of requests: $\mathcal{R}_w$, bandwidth: $\mathcal{B}_w$)
4. **Supervisor Heartbeat**: Every configurable $\mathcal{S}'_s$ seconds, each supervisor will send a heartbeat to ZooKeeper. A heartbeat takes the form of storing some run-time information about the supervisor in ZooKeeper.
(period: $\mathcal{S}'_s$, number of requests: $\mathcal{R}'_s$, bandwidth: $\mathcal{B}'_s$)
5. **Worker Heartbeats**: Similarly, every configurable $\mathcal{S}'_w$ seconds, each worker sends heartbeats to ZooKeeper. A worker heartbeat includes statistics information about each task running in that worker.
(period: $\mathcal{S}'_w$, number of requests: $\mathcal{R}'_w$, bandwidth: $\mathcal{B}'_w$)

In principle, the System Management Cost has two components: a component that reflects the load on ZooKeeper, expressed in number of requests per time unit ($\mathcal{R}$) and a network load component which represents the network bandwidth consumption ($\mathcal{B}$).

5.1 Number of requests to ZooKeeper

The number of ZooKeeper requests per time unit is a function of the amount of nodes (i.e. supervisors) and the amount of workers in the cluster as reflected by the following equation:

$$\mathcal{R} = \frac{\mathcal{R}_t}{\mathcal{S}_t} + numNodes * \left(\frac{\mathcal{R}_s}{\mathcal{S}_s} + \frac{\mathcal{R}'_s}{\mathcal{S}'_s}\right)$$

$$+ numWorkers * \left(\frac{\mathcal{R}_w}{\mathcal{S}_w} + \frac{\mathcal{R}'_w}{\mathcal{S}'_w}\right)$$

where $\mathcal{R}_t$, $\mathcal{R}_s$, $\mathcal{R}_w$, $\mathcal{R}'_s$ and $\mathcal{R}'_w$ are all fixed (implementation-related) constants and $\mathcal{S}_s$, $\mathcal{S}_w$, $\mathcal{S}'_s$ and $\mathcal{S}'_w$ are configurable constants. It should be noted that the number of requests per time unit is independent of the amount of tasks/components instantiated around the cluster[6] and is linear with the size of the cluster.

5.2 Network Bandwidth Consumption

The network bandwidth consumption depends both on the number of supervisors and workers in the cluster, and also on the number of tasks running on those workers. Specifically, an increase in the number of workers leads to more worker heartbeat messages per time unit as each worker sends the heartbeats separately; an increase in number of supervisors leads to more assignment info messages being exchanged with ZooKeeper as each supervisor is in charge of managing its own assignments and finally, an increase in the number of tasks increases the size of both the worker heartbeat and assignment info messages as they contain information that are specified to each task.

In order to present a formula to calculate the bandwidth consumption of system management tasks, we will first look at two of the most important data structures used to communicate with ZooKeeper, i.e. Worker Heartbeat (WHB) and Assignment Info (AI).

Worker Heartbeats A Worker Heartbeat data structure contains information about the status of the various

[6]with the caveat as explained above that $numWorkers$ is a dynamic thing in case $numTasks$ is smaller than the sum of the configured maximum number of workers on each node.

tasks assigned to a worker. The byte size of one round of worker heartbeats can be defined as:

$$\sum_{i=1}^{numWorkers} |WHB_i| = a * numTasks + b * numWorkers$$

where a and b are constants related to the specific implementation of this data structure in Storm, which will not be further detailed to improve readability.

Assignment Information Assignment information contains a mapping from task ID to host and port and the timestamp of when the component was started.

The byte size of the assignment information data structure is given by

$$|AI| = c * numNodes + d * numTasks + e$$

where again c, d and e are constants related to the specific design of the AI data structure, which will not be further detailed. It should be noted that the overhead of the AI is independent of the amount of workers per node.

Bandwidth As was mentioned previously, an approximation of system management bandwidth consumption can be written as the sum of the bandwidth consumed by each of the recurring tasks as follows:

$$\begin{aligned}\mathcal{B} \quad = \quad & \frac{|\mathcal{R}_t|}{\mathcal{S}_t} + numNodes * \left(\frac{|\mathcal{R}_s|}{\mathcal{S}_s} + \frac{|\mathcal{R}'_s|}{\mathcal{S}'_s}\right) \\ & + numWorkers * \left(\frac{|\mathcal{R}_w|}{\mathcal{S}_w} + \frac{|\mathcal{R}'_w|}{\mathcal{S}'_w}\right)\end{aligned}$$

where $|\mathcal{X}|$ is used to denote the size (in bytes) of $\mathcal{X}$; $numNodes$ is the number of nodes in the Storm cluster running a supervisor; $numWorkers$ is the sum of all active workers on all nodes in the Storm cluster. A worker becomes *active* if it has at least one task assigned to it.

By leveraging an in depth analysis that correlates $\mathcal{R}_t$, $\mathcal{R}_s$, $\mathcal{S}_w$, $\mathcal{S}'_s$ and $\mathcal{S}'_w$ with $numNodes$, $numWorkers$, $|AI|$ and $|WHB|$ (which we do not report here for brevity), we can express the System Management Bandwidth as:

$$\mathcal{B} = k_1 + k_2 * |AI| + numNodes * (k_3 + k_4 * |AI|)$$

$$+ \; numWorkers * (k_5 + k_6 * |AI|) + k_7 * \sum_i |WHB_i|$$

where all of k_1 through k_7 are constants.

In conclusion, with this final formulation we can see that ZooKeeper traffic bandwidth increases proportionally with the number of tasks and nodes running in the cluster, which is also validated by the experiment reported in Figure **??**. In the figure we plot the measured BW (in KB) attributed to ZooKeeper communication as a function of the number of tasks running around the cluster, for various cluster sizes. Note also that ZooKeeper communication is independent of the specific algorithms implemented in spouts or bolts.

6. EXPERIMENTAL EVALUATION

This section provides a detailed experimental evaluation of the performance characteristics of the Storm system using real industrial datasets. This section aims to experimentally quantify the performance variation in running different

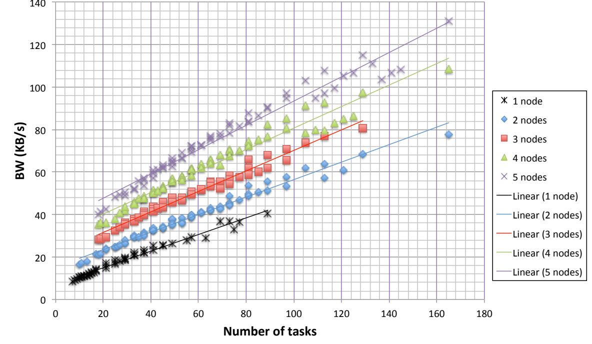


Figure 5: System management traffic as a function of the number of components

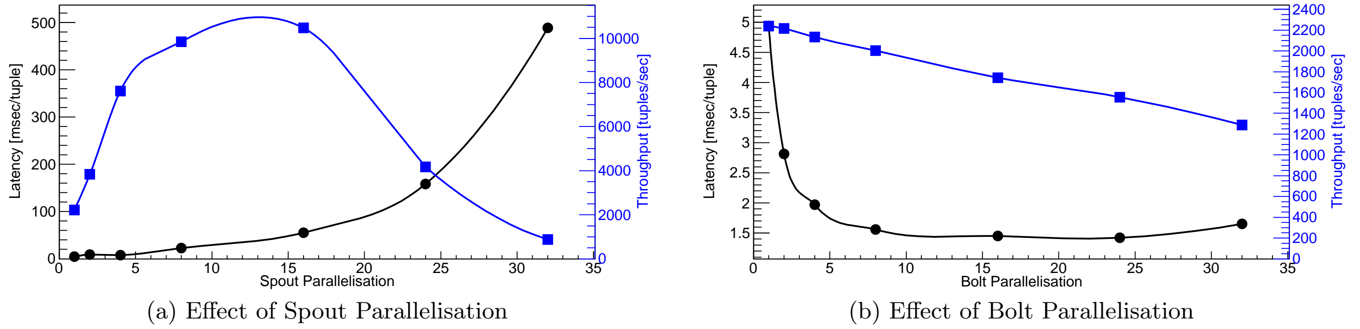
Storm configurations by leveraging insights from the performance model. Furthermore, the intent of this experimental analysis is to detect the causes of this variability in the system performance and shed light on the precautions required in order to run the system under optimal configurations.

6.1 Dataset and Environment Setup

The experiments have been run with a real Telco dataset consisting of Performance Management (PM) observations of a large Femtocell[7] network. Femtocells were designed for use in a home or small business to improve localized cellular service and offload bandwidth usage from macrocells (i.e., traditional cell towers). During operation, Femtocells produce many low-level performance logs (e.g. number of successful handovers, number of call initiation attempts) across a range of performance categories (e.g. packet data performance, handover performance). The considered dataset is composed of hourly PM logs collected over 15 days for a network of 70k Femtocells, totalling approximately 11 million XML files. These files contain a list of 128 key-value pairs of operational and statistical counters with a mixture of integer and floating-point values. These data are usually analyzed to monitor network performance characteristics and predict traffic peaks or failures. The experiments presented in this paper deploy a simple topology composed of two actors (one spout and one bolt) whose task it is to identify the Femtocells which require the most bandwidth. The spout reads input messages from an external queue (this corresponds to the XML data) and produces a stream of tuples. The bolt receives the stream of tuples with a shuffle grouping and emits the Femtocell list.

Cluster Configuration. Each test runs on a cluster comprising 5-nodes of identical configuration. Additional machines were used to generate load into this cluster and to host the external message queue(s). Each machine is a dual 4-core Intel Xeon 3Ghz 32bit 16 GB Memory, 1 Gb/s network interface. Nodes are interconnected through a 10Gb OmniSwitch 6850 Ethernet Switch. Each node runs Linux version 2.6.32-220.4.1.el6.i686. The following software components were used: Java 1.6 OpenJDK Runtime Environment (IcedTea6 1.10.4), ZeroMQ 2.1.7, Zookeeper 3.4.2 and Kestrel 2.3.4. We used a modified version of Storm, Storm-0.8.0-SNAPSHOT (version of July 2012), to include timers

[7] Alcatel-Lucent 9360 Small Cell. http://www.alcatel-lucent.com/small-cells/



(a) Effect of Spout Parallelisation

(b) Effect of Bolt Parallelisation

Figure 6: Measuring latency and throughput on the different system configurations.

and related statistical tools for the purpose of the experiments.

6.2 Experiment Configurations

A comprehensive test automation suite was developed which is composed of various shell scripts and Java programs to automatically execute the Storm topologies and collect runtime statistics on the system performance. Storm configurations were tested while varying the number of nodes, spouts, bolts and workers. Each specific configuration was tested 10 times in order to accumulate statistical information. Careful consideration was taken to ensure that each configuration was tested under the same operational conditions.

Each individual test case is executed on a clean cluster, meaning that all processes from a previous run were killed on all cluster nodes before starting the new ones. In addition, all data generated by the previous run is erased from the filesystem. Each test consists of an initial warm-up phase of 1 minute (in which we do not collect statistics because the system may not be in its computational steady state yet) followed by a 5 minute measuring phase during which statistics are being gathered, including:

1. Throughput (μ) which represents the total number of events processed per second.
2. Latency ($\mathcal{L}$) which represents time to process a tuple both within a single actor and within the entire system.
3. External and internal queue sizes and their growth rate.
4. Network bandwidth in terms of the number of messages exchanged among workers on different nodes.
5. Various statistics on system management overhead (e.g. communication with ZooKeeper).
6. Approximate memory usage and CPU usage per thread.

Each test is run with a different Storm configuration which is determined by the following parameters:

- *Parallelization factor*: represents the number of tasks, from $\{1, 2, 4, 8, 16, 24, 32\}$, instantiated per actor.
- *Cluster size*: represents the number of nodes participating in the Storm cluster, i.e. from 1 to 4.
- *Worker pool size*: represents the number of workers (JVMs) per node, which host the tasks. In our experiment this number is selected from $\{1, 2, 4, 8, 16, 24, 32\}$.
- *Event injection rate*: represents the number of events injected per second into the queue which provides data to the spouts. In our experiments the event injection rate varies among three different rates, 5K, 10K and 50K tuples per second, to observe how Storm adjusts to different conditions.

6.3 Experimental Analysis

In order to optimize the system performance while running Storm jobs there are several variables and parameters which need manual configuration. The configuration is thus a complex task that requires a precise knowledge of the most relevant parameters and how they impact the system performance. In particular, we focus on the following metrics: the throughput (μ), the latency ($\mathcal{L}$) and the system resource utilization (CPU, memory, network bandwidth).

In the following subsections we present a set of experiments that aim to shed light on how to configure this system based on the theoretical understanding derived from the performance model presented in the previous sections.

6.3.1 Performance Variability for Different System Configurations

The experiments presented in this section comprise observations of the impact on latency and throughput produced by different configurations in terms of parallelization, workers and cluster size.
A configuration is expressed as the specification of the system parameters, i.e. number of nodes, spouts, bolts, and workers (hereafter simply $< n, s, b, w >$). For brevity, we only report experiments run on a single node cluster with external queue fed with 50k messages per second. However, the same experiments run on a 1 through 4 node cluster show similar results, with only a few small deviations attributed to the increased system management costs to run more nodes which will be discussed in detail in the next sections.

Spout Parallelisation In order to study the impact of the number of spouts on the overall system performance, we fixed all other parameters at 1 while varying the number of spouts in $\{1, 2, 4, 8, 16, 24, 32\}$. Figure **??** illustrates the effect upon throughput (in terms of tuples per second) and the latency (in terms of milliseconds to process a tuple) as the number of spouts is adjusted. The system throughput can be increased by increasing the number of spouts - however as the number of spouts continues to increase beyond some threshold, the throughput declines. This can be understood by observing the latency which exhibits exponential growth. Beyond some threshold (determined by the system hardware), the system is overstressed with many processes and the context switch among them kills the system performance.

Bolt Parallelisation In order to study the impact of the number of bolts on the overall system performance, we

fix all other parameters at 1 while varying the number of bolts in $\{1, 2, 4, 8, 16, 24, 32\}$. Figure **??** illustrates the effect upon throughput (in terms of tuples per second) and the latency (in terms of milliseconds to process a tuple) as the number of bolts is adjusted. An increase in bolt parallelisation reduces throughput due to the extra CPU load associated with scheduling. However, a reduction in latency towards a lower limit is also observed when the number of bolt is within $[8, 24]$ range. This limit represents the fastest possible bolt execution time, which is the cost of the system from the emission of the tuple by the spout up to the completion of the algorithm implemented by bolt B, defined in the performance model in Section **??**.

6.3.2 *Costs of Distributing Computation*

This section aims to quantify the variation in performance when varying the number of workers, and the way communicating tasks are distributed across these workers. Section **??** showed that the dataflow cost depends on whether two communicating tasks are hosted (a) within the same JVM, (b) within the same node but on different JVMs, or (c) on different nodes. Specifically, cases (b) and (c) incur a serialization/deserialization cost, while case (c) additionally incurs network transfer cost. Here we demonstrate its practical impact.

Cost of Communicating JVMs Figure **??** displays the system latency $\mathcal{L}$ when some tasks are assigned to the same physical node, but to different JVMs running on that node. Each point in the plot represents a different system configuration. On the x-axis we report the specific configuration setting, i.e. the actual instantiation of $< n, s, b, w >$ and on the y-axes we plot two different results: the left-hand side axis displays the latency, in milliseconds, of the system for each configuration while the right-hand side y-axis estimates α_{node} defined in Section **??**, i.e. the percentage of tuple exchanges across different JVMs on the same physical node. In this experiment, an increment of the latency corresponds to the cost of transferring the associated tuples from one JVM to another, which mainly comes from the serialization/ deserialization costs. We show that when the theoretical approximation of α_{node} increases the corresponding latency for that configuration goes up, which means that α_{node} can be used to predict the system latency of a particular configuration. Within the yellow box, starting with configuration $< 1, 8, 8, 1 >$, we observe that the system falls in an unstable configuration in the sense that the serialization/deserialization costs are not the dominant costs anymore and the latency degrades from 5ms to 28ms in the last configuration. We have increased the number of tasks and we can observe that the system is affected by extra costs such as: management overhead, Java garbage collection, scheduling and context switching among JVMs.

Impact of Increasing Workers. One of the benefits of a distributed system is its capacity to increase the amount of parallel threads of execution and reliably distribute them over the cluster nodes so as to improve processing efficiency in time and capacity. However, the *number* of allocable JVMs per single node has a direct impact on performance. This number is highly related to the number of CPUs/cores per node, therefore, the number of allocable threads per worker per CPU needs to be carefully calibrated to achieve optimal system utilization before undesirable ef-

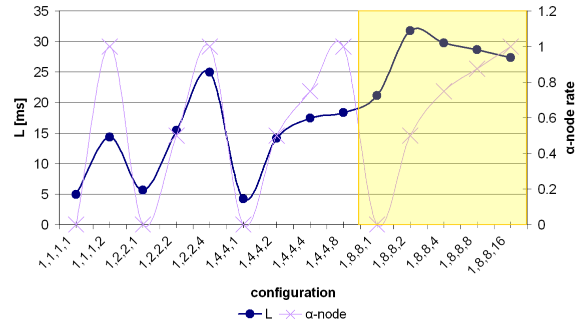


Figure 7: The latency for different system configurations.

fects on the performance show up. Figure **??** and Figure **??** show the throughput and the latency for different system configurations. In this experiment, we highlight the effect of instantiating from 1 worker with 16 threads to 16 workers each with 1 thread. The direct proportional effect of the tuple serialization/deserialization cost on the latency, which is clearly visible in Figure **??**, is no longer apparent for higher thread counts. In fact, for higher thread counts, Figures **??** and **??** show that we can improve throughput and latency by instantiating multiple workers, each with a lower thread count to achieve the same overall level of parallelism.

This experiment shows that when there are more threads than cores the system performance can have unexpected behavior. For instance, if the number of threads per JVM is too high, as in configuration $< 1, 8, 8, 1 >$ with 16 threads per JVM, the throughput is lower than when there is only a single thread per JVM as in configuration $< 1, 8, 8, 16 >$. One of the factors explaining this observed behavior is the increased JVM garbage collection activity for higher thread counts. Figure **??** and **??** confirm that this trend is the same for all cluster sizes.

Bandwidth Consumption. As we have previously shown, the more cluster nodes we leverage, the better throughput we may expect to achieve. However, the participating nodes need to communicate by transferring tuples among each other, which comes at a cost that is constrained by the available network bandwidth.

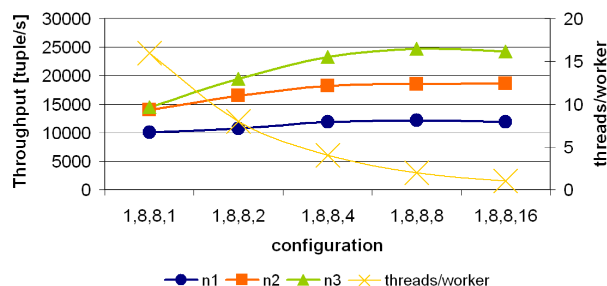
In the simple topology we have used for our experiments, we can estimate total bandwidth consumption as follows:

$BW_{total} = BW_{external} + BW_{system} + BW_{dataflow}$

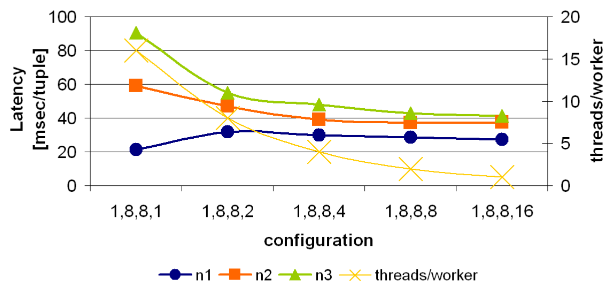
where

- $BW_{external}$ is the bandwidth required by the spout to read raw events from the event source (i.e. a Kestrel queue in our experiments).
- BW_{system} is the system management bandwidth we calculated in Section **??**. For simplicity, we can overestimate this portion of the overall bandwidth by taking an upper bound for at least 100 active tasks as reported in Figure **??**, i.e. $56K$ bytes per second for the 2-node configuration.
- $BW_{dataflow}$ is the bandwidth related to exchanging tuples between spout and bolt instances. For this portion, we can use the formulas presented in Section **??**. More specifically, $BW_{dataflow} = numTuples * \alpha_{remote} * tupleSize$ where $numTuples$ represents the amount of tuples emitted by the spout per time unit.

From measurements performed on our dataset, we have determined that the size of an external event is 360 bytes



(a) Effects of the parallelism on the Throughput (μ)



(b) Effects of the parallelism on the Latency ($\mathcal{L}$)

Figure 8: Measuring latency and throughput with 1 to 16 JVMs respectively with 1 to 16 threads per node.

and the size of a transformed event, i.e. a Storm tuple, is 44 bytes. Therefore, in this experiment, the bandwidth consumption can be computed using the following formula:

$$BW_{total} = numTuples * (360 + 44 * \alpha_{remote}) + 56000$$

Figure **??** shows the calculated bandwidth consumption as a function of system configuration for clusters consisting of 2,3 and 4 nodes. These bandwidth calculations were verified using the Unix *dstat* tool.

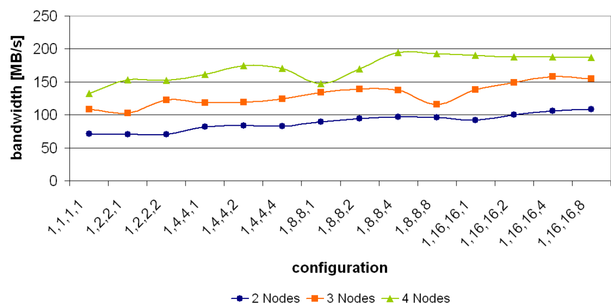


Figure 9: System Communication Cost

7. RELATED WORK

The growing demand for large-scale data processing and data analysis applications has spurred the development of novel solutions from both the industry (e.g., web-data analysis, click-stream analysis, network-monitoring log analysis) and the sciences (e.g., analysis of data produced by massive-scale simulations, sensor deployments, high-throughput lab equipment). MapReduce [?] is a framework which was introduced by Google for programming commodity computer clusters to perform large-scale data processing. The framework is designed such that a MapReduce cluster can scale to thousands of nodes in a fault-tolerant manner. However, the basic architecture of the MapReduce framework requires that the entire output of each map and reduce task be *materialized* into a local file before it can be consumed by the next stage. Therefore, it is not adequate for supporting real time processing of streaming data. Several approaches have been proposed to tackle this challenge. For example, the *MapReduce Online* approach [?] has been proposed as a modified architecture of the MapReduce framework in which intermediate data is *pipelined* between operators while preserving the programming interfaces and fault tolerance models of previous MapReduce frameworks. The *Incoop* system [?] has been introduced as a MapReduce implementation that has been adapted for incremental computations which detect the changes on the input datasets and enables the automatic update of the outputs of the MapReduce jobs by employing a fine-grained result reuse mechanism. In particular, it allows MapReduce programs which have not been designed for incremental processing to be executed transparently in an incremental manner. The M^3 system [?] has been proposed to support the answering of continuous queries over streams of data bypassing the HDFS so that data are processed only through a main-memory-only data-path and totally avoids disk access. In this approach, Mappers and Reducers never terminate where there is only one MapReduce job per query operator that is continuously executing. While these approaches could improve the performance of the MapReduce framework for incremental and streamed data processing, they are still unable to fully support the distributed *real time* processing requirements of data intensive applications.

Several distributed stream processing systems have been presented in the literature such as *Aurora* [?], *Borealis* [?], *SPADE* [?], *Stormy* [?] and *Apache S4*[8]. For example, in Borealis, the collection of continuously running queries are treated as one giant network of operators. The processing of these operators is distributed to multiple sites where each site runs an instance of the Borealis server. The query processing is controlled by an Admin component which takes care of moving query diagram fragments to and from remote Borealis nodes when instructed to do so by other components. SPADE is a declarative stream processing engine which supports a set of basic stream-relational operators with powerful windowing and punctuation semantics. The system is designed to execute a large number of long-running jobs that take the form of data flow graphs where each graph consists of a set of processing elements connected by streams and each stream carries a series of Stream Data Objects. The processing elements can communicate with each other via their input and output ports. The concepts and ideas of our proposed models can be easily adopted to be used within the context of these systems. Some studies have been presented for analyzing and modeling the service latency of event-based systems. They have been mainly focusing on different classes of implementations such as the Publish/Subscribe systems [?, ?] and message-oriented middlewares [?, ?].

When designing parallel and distributed systems, a critical problem is the one of how to tune the system configuration (e.g., in terms of number of nodes), so as to achieve certain performance goals (e.g., maximize throughput, minimize latency or meet a prefixed latency constraint). To this

[8]http://incubator.apache.org/s4/

end, various authors tried to build abstract models for the performance of such systems, starting from the well-known Amdahl law [?] and its various enhancements, some of which are summarised in [?].

Giurgiu [?] has presented an approach for estimation of performance for mobile-cloud applications. The approach tried to identify the factors which impact interaction response times, such as the application distribution schemes, workload sizes and intensities, or the resource variations of the mobile-cloud application setup. It also attempted to find correlations between these factors in order to better understand how to build a unified and generic performance estimation model. The *Starfish* system [?, ?] is the most relevant system for our work. It represents a cost-based optimizer for MapReduce programs which focuses on the optimization of configuration parameters for executing these programs on the Hadoop platform. It relies on a *Profiler* component that collects detailed statistical information from executing the programs and a *What-if Engine* for fine-grained cost estimation processing. For a given MapReduce program, the role of the cost-based optimizer component is to enumerate and search efficiently through the high dimensional space of configuration parameter settings, making appropriate calls to the What-if Engine, in order to find the optimal configuration setting. It clusters parameters into lower-dimensional subspaces such that the globally-optimal parameter setting in the high-dimensional space can be generated by composing the optimal settings found for the subspaces. Our current study represents the first step in the implementation of a similar what-if analyzer component in the more complex environment of real time distributed stream-processing engines.

8. CONCLUSIONS

The lack of expertise and in-depth understanding of the theoretical foundation for the performance modeling of large scale distributed processing system is a crucial problem. Many researchers, engineers and organizations are currently shifting their focus to exploit the advantages of the new data processing paradigm shift. In this paper, we presented a detailed set of models that formalize the performance characteristics of the Storm system as a representative of a practical distributed, parallel and fault-tolerant stream processing engine that follows the Actor Model theory. Our approach is applicable to estimate the performance characteristics of distributed stream processing jobs which follow the same principles in addition to optimizing the configuration settings of the underlying system and hardware cluster. Several directions represent good candidates for future work to further understand and optimise the complex environment of real time distributed streaming processing engines. For example, we are planning to build a What-if Engine that can help users to find the right configuration setting and parameters for their jobs. In addition, we plan to implement a more intelligent allocation strategy for the processing components across the running workers which can optimize the data transfer cost.

9. REFERENCES

[1] Abadi, D. J., et al. Aurora: A Data Stream Management System. In *SIGMOD Conference* (2003), p. 666.

[2] Abadi, D. J., et al. The Design of the Borealis Stream Processing Engine. In *CIDR* (2005), pp. 277–289.

[3] Agha, G. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[4] Aly, A. M., et al. M^3: Stream Processing on Main-Memory MapReduce. In *ICDE* (2012).

[5] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS '67, Spring Joint Computer Conference* (1967), pp. 483–485.

[6] Bhatotia, P., et al. Incoop: MapReduce for Incremental Computations. In *SOCC* (2011).

[7] Cherkasova, L. Performance modeling in mapreduce environments: challenges and opportunities. In *ICPE* (2011), pp. 5–6.

[8] Clinger, W. D. Foundations of Actor Semantics. Tech. rep., Cambridge, MA, USA, 1981.

[9] Condie, T., et al. MapReduce Online. In *NSDI* (2010), pp. 313–328.

[10] Cucinotta, T. Optimum scalability point for parallelisable real-time components. In *Proceedings of SOMRES* (2011).

[11] de Gooijer, T., et al. An industrial case study of performance and cost design space exploration. In *ICPE* (2012), pp. 205–216.

[12] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI* (2004), pp. 137–150.

[13] Gedik, B., et al. Spade: the system s declarative stream processing engine. In *SIGMOD Conference* (2008), pp. 1123–1134.

[14] Giurgiu, I. Understanding performance modeling for modular mobile-cloud applications. In *ICPE* (2012), pp. 259–262.

[15] Herodotou, H., et al. MapReduce Programming and Cost-based Optimization? Crossing this Chasm with Starfish. *PVLDB 4*, 12 (2011), 1446–1449.

[16] Herodotou, H., et al. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR* (2011), pp. 261–272.

[17] Hewitt, C. ORGs for Scalable, Robust, Privacy-Friendly Client Cloud Computing. *IEEE Internet Computing 12*, 5 (2008), 96–99.

[18] Hewitt, C. ActorScript(TM): Industrial strength integration of local and nonlocal concurrency for Client-cloud Computing. *CoRR abs/0907.3330* (2009).

[19] Hewitt, C. Actor Model for Discretionary, Adaptive Concurrency. *CoRR abs/1008.1459* (2010).

[20] Hewitt, C., et al. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI* (1973), pp. 235–245.

[21] Loesing, S., et al. Stormy: an elastic and highly available streaming service in the cloud. In *EDBT/ICDT Workshops* (2012), pp. 55–60.

[22] Meseguer, J., and Talcott, C. L. A partial order event model for concurrent objects. In *Proceedings of CONCUR* (1999).

[23] Mühl, G., et al. Stochastic Analysis of Hierarchical Publish/Subscribe Systems. In *Euro-Par* (2009).

[24] Sachs, K., et al. Benchmarking of message-oriented middleware. In *DEBS* (2009).

[25] Sachs, K., et al. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Perform. Eval. 66*, 8 (2009).

[26] Sakr, S., et al. A Survey of Large Scale Data Management Approaches in Cloud Environments. *IEEE Communications Surveys and Tutorials 13*, 3 (2011), 311–336.

[27] Schröter, A., et al. Stochastic performance analysis and capacity planning of publish/subscribe systems. In *DEBS* (2010).