Real Time Processing in Mobile Clouds

Zeng Lin¹, Shurong Li², Pu Xu³

School of traffic and transportation, Beijing Jiaotong University, Beijing100044, China

{16120833, 16120837, 16120755}@bjtu.edu.cn

Abstract. With rapid advances in mobile device and cloud computing technologies, a new computing paradigm in which large amounts of data are stored and processed on mobile devices is emerging. Despite the powerful hardware available, mobile devices have limited capacities as they are powered by battery and connected by unstable, low bandwidth, wireless networks. Apache Storm is a scalable platform that provides distributed real-time stream processing paradigm and fault tolerant capability. This paper studies the existing problems of applying Storm to mobile environment, and then proposes a new framework to address these problems with the goal that it would outperform Storm in performance in mobile environment. More specifically, we hope that our framework would reduce processing latency, energy consumption and provide guarantee that processing latency is under certain predefined threshold. Concretely, we formulate the resource allocation and task scheduling optimization problem and propose a heuristic solution to approximate the optimal solution. In our heuristic solution, we generate task scheduling and resource (worker node) allocation strategies according to collected inter-task traffics and latency information of running topologies. Extensive evaluations are performed through proof-of-concept real hardware implementation. Results show that our proposed framework effectively reduces processing latency by up to 50% compared with Storm, also it controls processing latency under certain predefined threshold in abnormal situation.

1. Introduction

With the help of rapid development of technology, people's lives are moving at such a fast pace that several seconds of latency in respond time will lead to impatience, which leads to the emergence of real-time data processing framework, such as Apache Storm, Apache Spark, at Big Data ecosystem in recent years.

As a result of the revolution of Mobile Internet in recent years, people are spending much more time on mobile devices, e.g., smartphones, tablets, than before, which makes it meaningful and urgent to introduce real-time stream processing framework into mobile environment. However, traditional big data solutions mainly focus on offloading massive data from terminals into powerful data center for computation and then fetching corresponding results [1]. For example, Apache Storm, a famous representative of these big data solutions, is designed to distribute the processing of unbounded streams of data to a cluster of connected computers, doing real-time processing for what Hadoop did for batch processing [2]. We observe that these traditional big data processing frameworks would not work well in mobile environment, as they do not take into account unstable or limited external network connection, limited resource, etc.

Directly applying Storm to mobile environment incurs many problems. At first, it does not take energy efficiency into account. Besides, Storm assumes that all the links are equal. However, we observe that the communication overhead is different from inter-node communication and intra-node
communication. At last, Storm only provides some default schedulers which only uses a round robin strategy to evenly distribute tasks over the cluster [3].

Based on the above observations, in this paper, we introduce two mechanisms into our proposed Storm framework to solve the aforesaid problems. First, we utilize a resource allocator to help to automatically tune the number of worker nodes allocated for each topology. Second, we design a new scheduler to reduce inter-node traffic, processing latency and energy consumption.

We summarize our contributions as follows:

- We introduce real-time stream processing framework into mobile environment.
- We realize automatically resource allocation with the purpose that all the Qos requirements would be met.
- We propose a heuristic task scheduling algorithm to reduce latency and traffic.
- We present design and implementation of our framework, as well as the evaluation and comparison of our framework with Storm via some benchmark experiments.

![Figure 1. The topology of storm.](image)

### 2. Background & State of Art

#### 2.1. Storm Overview and Background Introduction

Storm is a distributed, reliable, real-time stream processing framework developed by Twitter. The computation model of Storm is that each job (called topology in Storm) is delegated into different types of user implemented components, spout or bolt, that is each responsible for a user defined specific processing logic [2]. A topology could be viewed as a directed graph of spouts and bolts as depicted in Figure 1, where each vertex is a task and each edge is the connection between tasks. A task is a running instance of spout or bolt. Tasks are executed once for each incoming data piece called tuple. Unbounded tuples flowing through this pipeline of tasks is defined as a stream. In details, the input stream of a Storm cluster is handled by a component called spout. Spout passes the tuple to downstream bolt.

In a Storm cluster, as shown in Figure 2, there are two kinds of nodes: master node and worker node. Master node runs a daemon called Nimbus. Nimbus is responsible for accepting new topology from user, assigning tasks to each worker node, as well as monitoring the health of the topology for fault tolerant purpose [2]. Each worker node runs a daemon called Supervisor. Each worker node is configured with a limited number of Java processes, called worker slots, that is responsible for executing a portion of a topology. Worker slot runs a set of threads called executors, which runs one or more tasks of the same spout or bolt. Supervisor monitors and launches worker process in case of failure or new assignment from Nimbus node. The coordination between Nimbus and the Supervisor is through a third-party software call Zookeeper.
2.2. State of Art
There has been lots of research that attempt to bring big data framework into mobile environment. Marinelli [4] introduced the Hadoop based platform Hyrax into mobile cloud by directly migrating cloud computing framework, Hadoop, into Android operating system. Hadoop is designed for traditional data center rather than mobile environment, porting Hadoop directly onto Android devices dose not solve the problems faced in mobile environment. Johnu [5]etc. proposed to combine Mobile Distributed File System (MDFS) with Hadoop to solve problems they faced when porting Hadoop into mobile environment. They utilize the reliability and security of MDFS to enable reliable storage and processing of massive data on mobile environment. However, these works are mainly focus on Hadoop.

3. System Architecture
Now let us use mathematical model to formulate our problem. Suppose that there are a set of user submitted topologies $T = \{t_i\} (i = 1, 2...k)$, where $k$ is the total number of topologies. For each topology $t_i$, it contains a set of executors $E_i = \{e_{ij}\} (j = 1, 2...p_i)$, where $p_i$ is the total number of executors of topology $t_i$. Our Storm cluster contains a set of $\{n_l\} (l = 1, 2...N)$ nodes, where $N$ is the total number of nodes of our Storm cluster. Each node $n_l$ contains a set of worker slots $\{w_{lj}\} (j = 1, 2...W_l)$, where $W_l$ is the number of worker slots of node $n_l$. $y_i$ denotes the observed QoS performance (processing latency in our paper) of topology $t_i$. $r_i$ denotes the desired processing latency, i.e. reference input, for each topology $t_i$. $R_{ijmn}$ denotes the traffic of two executors. We define the cost function as:

$$J = \alpha \sum_{i=1}^{k} (y_i - r_i)^2 + \beta \sum_{\forall e_{ij}, e_{mn}} (R_{ijmn})$$

In Equation 1, $\alpha$ and $\beta$ are constants. The goal of our framework is to assign each executor $e_{ij}$ to worker slot $w_{lj}$, such that the cost function $J$ is minimized.

As the problem formulated above is known to be NP-complete. We propose a heuristic algorithm to figure out an assignment strategy in real-time manner. Our solution is that, we break the above executors assignment problem into two smaller subproblems: 1) Resource Allocation Subproblem, For each topology $t_i$, our framework would allocate specific resource (nodes) to it dynamically according to feedback processing latency information. 2) Task Scheduling Subproblem, After resource allocation procedure, the scheduler we designed would assign executors to allocated nodes, with the purpose that it will do its best to reduce latency, inter-node and inter-slot traffic amount and save energy consumption.
We design and implement our framework based on Storm by modifying and adding several components, including: a scheduler, processing latency monitors, traffic and load monitors, an external database, a transducer and a controller. System architecture is shown in Figure 3 (Note that in the figure, we highlight new components we add to Storm). The latency monitor would collect processing latency information for each worker slots and then upload this data to external database. Once finishing reading data from database, Transducer feeds transformed data to controller component. The controller component will dynamically figure out a shared resource allocation strategy for each topology such that the QoS requirement is met, i.e., processing latency is smaller than some desired value. Then scheduler will generate scheduling strategies for topologies.

4. Resource Allocation Design

For simplicity, we assume that the smallest resource granularity that resource allocator could manipulate is individual node. We observe that the system is running continuous, for ease of analysis, we first slice the continuous time line into time slots, namely discrete control intervals. Our resource allocation system is supposed to collect necessary information within a time slot and then feed this information into controller, which would then produce a resource allocation plan for next control interval.

Each topology specifies its desired processing latency requirement: \( r_i(k) \), denoting the reference input of topology \( i \) at control interval \( k \).

We use aggregate mean latency of all tuples within a time slot to represent the latency of the topology. We use \( \text{latency}_{ij}(k - 1) \) to denote the latency observed for tuple \( i \) of topology \( j \) at time interval \( k - 1 \). Then use \( y_j(k - 1) \) to denote the latency of topology \( j \) for time slot \( k - 1 \). Then we have the following equation:

\[
y_j(k - 1) = \frac{\sum_{i=1}^{n_j(k-1)} \text{latency}_{ij}(k - 1)}{n_j(k - 1)} \tag{2}
\]
In Equation 2, $n_j(k-1)$ is the total number of tuples that flow through topology $j$ at time slot $k-1$.

For each control interval $k$, topology $t_i$ has specific latency requirement $r_i(k)$, and will be allocated a certain number of nodes for use $u_i(k)$, the measured latency is $y_i(k)$. The goal of this controller is to determine the appropriate value for each $u_i(k)$ under certain restraint, such that each $y_i(k)$ could be smaller than reference input: $r_i(k)$. Current allocation strategy is that: once $y_i(k)$ is bigger than $r_i(k)$, controller will allocate more resource to topology $t_i$.

5. Scheduler Design for Energy Efficiency

Once the resource allocation process finishes, our system will use a scheduler to figure out tasks scheduling plans for each topology.

The goal of this scheduler is to assign executors (for ease of analysis, we will only assign one task to run on one executor, thus executor number is the same as task number) to available worker slots in order to minimize the processing latency for tuples and energy consumption while not violating external constraints, such as load constraints for each node. The key idea behind this scheduler is that: not all the communication links among executors are equal, e.g. intra-node communication is much cheaper and energy efficient than inter-node communication via wireless link.

The input is a cluster of nodes, each has some fixed number of available slots, and a topology consisting of several components, namely spouts and bolts. The scalability and parallelization of topology is realized by two steps: 1) user can assign task number for each component, e.g. spout or bolt, when defining the topology. 2) User can set the worker slot number for the topology.

The real-time nature requires an efficient and low complexity faster algorithm to solve the above task scheduling problem. Thus, we use a heuristic algorithm, in polynomial time complexity, to approximate optimal solution.

As the infrastructure consists of only mobile devices, limited link capacity, poses a major bottle neck compared with traditional data center. Thus, a feasible strategy to reduce latency is to reduce inter node traffic. We also observe that wireless communication consumes considerable battery power, meaning that reducing inter-node traffic would help to increase battery’s life. Another important consideration of our scheduling algorithm is to ensure that the load of any node is under control, namely, we could not assign too many tasks to a single node. In this way, we can ensure that each worker node is stable, that is, the queue of the worker node will not overflow. We observe that the load of any executor as well as the traffic amount between any two executers would stay the same during a short time period, thus it is reasonable to use the history statistics data to predict future behaviour.

Algorithm 1 illustrates the key part of our scheduler. It can be summarized by the following two steps.

step 1: We sort all pair of executors $(e_i, e_j)$ according to traffic value $R_{ij}$ in descending order and get eList. We also sort all nodes by the load in ascending order and get nList.

step 2: We iterate through eList in top-down order, and try to assign each pair of executors we encounter. If both executors in the executor pair have not been assigned yet, we will assign them to the first node having the property that the total workload after assignment would not exceed predefined workload threshold at nList. After assignment, we will recalculate the load of this node, and then insert it to the sorted node list. Note that if the accumulate workload of one node has exceed some predefined threshold value, this node will be removed from nList.

If one of the node of the pair has been assigned already, we will assign unassigned executor to the same node where the assigned one resides, as long as, after this assignment, the total workload would not exceed the threshold value. Otherwise, we will assign this executor to the first node at the nList with the property that the total workload after assignment would not exceed predefined workload.
threshold. If all the executors have been assigned already, we will do nothing to them and continue to the next pair of executors.

Algorithm 1 Energy Efficient Scheduling Algorithm

1: for $l = 1 \ldots N$ do
2: \hspace{1em} load \leftarrow 0
3: end for
4: Sort executors pairs $e_j, e_k$ in descending order according to $R_j$ and get $eList$
5: Sort $n_i$ in descending order according to $load_i$ and get $nList$
6: While $eList \neq$ empty do
7: \hspace{1em} HeadPair \leftarrow first element of $eList$
8: \hspace{1em} if both executors are not assigned then
9: \hspace{2em} \triangleright assign them to first node $n_i$ in $nList$
10: \hspace{2em} $n_i \leftarrow$ FINDNODE (executors pair, $nList$ )
11: \hspace{2em} assign both executors to $n_i$
12: \hspace{2em} load$_i$ \leftarrow COMPUTE LOAD($load_i$, $DeltLoad$
13: \hspace{2em} ReSort $nList$ according to $load_i$
14: \hspace{1em} else if either executor is not assigned then
15: \hspace{2em} $n_i \leftarrow$ FINDNODE (executors , $nList$ )
16: \hspace{2em} assign this executor to $n_i$
17: \hspace{2em} load$_i$ \leftarrow COMPUTE LOAD($load_i$, $DeltLoad$)
18: \hspace{2em} ReSort $nList$ according to $load_i$
19: end if
20: end while
21: Procedure COMPUTE LOAD($load_i$, $DeltLoad$)
22: \hspace{1em} load$_i$ \leftarrow load$_i$ + $DeltLoad$
23: \hspace{1em} if $load_i \geq load_i + DeltLoad$ then
24: \hspace{2em} Remove $n_i$ from $nList$
25: end if
26: end procedure
27: procedure FINDNODE($e_j$, $nList$)
28: \hspace{1em} for node $n_i$ in $nList$ do
29: \hspace{2em} if $L_j + load_j \leq$ Threshold then
30: \hspace{3em} \triangleright $L_j$ is load of $e_j$
31: \hspace{2em} break
32: end if
33: end for
34: return $n_i$
35: end procedure

6. Performance Evaluation
We use Apache Storm stable release 0.9.2-incubating [6] for our implementation. We add a scheduler component, controller component, as well as load monitors, latency and traffic data collection components into Storm.

![Diagram of Word Count Topology](image)

**Figure 4. Word Count Topology.**

We evaluate the performance of Mobile Storm using the Word Count topology [7], as shown in Figure 4. We consider the following performance metrics: 1) Average processing latency; 2) Average inter slot traffic and inter node traffic. To measure average processing latency, we implemented our own latency monitoring and collection logics on top of Storm taskHook interface. In our implementation and experiments, we sample and collect latency as well as traffic statistics every 10 seconds. We are interested in the following parameters: 1) The number of worker slots assigned for Word Count topology. 2) Nodes are overloaded or not.

| Name                                    | Value     |
|-----------------------------------------|-----------|
| Latency and Traffic Collecting Interval | 10 s      |
| Number of Nodes                        | 3         |
| Worker Slots Per Node                  | 4         |
| Running Time of Each Experiment        | 700 s     |
| Scheduling Interval                    | 300 s     |
| Topology                                | Word Count Topology |

Table 1 summarizes the main experiment settings. We compare the performance of Word Count topology using our Mobile Storm and Apache Storm respectively on the same experiment settings, and then compare their performance. We set the number of tasks for Sentence spout, SplitSentence bolt, and WordCount bolt to be 3, 25, 25 respectively. As ackers could easily become bottleneck for performance, we set the number of ackers for this topology to be 60, roughly one acker for one task.

| Slots Num | Node1 | Node2 | Node2 | Node1 | Node2 | Node3 |
|-----------|-------|-------|-------|-------|-------|-------|
| 2         | 1     | 1     | 0     | 2     | 0     | 0     |
| 5         | 2     | 2     | 1     | 4     | 1     | 0     |
| 10        | 4     | 3     | 3     | 4     | 4     | 2     |

Table 2. Worker Slots Distribution.

6.1. Effect of number of worker slots
We set the workers (worker slots rather than worker nodes) number of Word Count topology to be 2, 5 and 10 respectively and run experiment. Table 2 shows how the workers are distributed across three nodes by Storm and Mobile Storm respectively. We can find that Storm will evenly distributed workers among nodes, whereas, Mobile Storm will try to assign workers to nodes as compact as possible. The average processing latency is depicted in Figure 5. We observe that the latency curves drop sharply, the reason is that when Storm deploys a new scheduling assignment, it will shut down all the related worker processes and then initialize new worker processes. During this period, the affected workers are not working, whereas, workers that are not affected are still working, as a result, tuples get time out or
dropped as internal queue are full. Thus, processing latency is very high during this period, e.g., 20 seconds in some cases. After all the workers have been launched, processing latency starts to drop sharply. We do not draw some very large values, that is why there are some gaps and sharp slops on those figures.

![Figure 5. Average Processing Latency.](image)

We can find from Figure 5a that the average latency for Storm is around 20 ms, 50 ms, 200 ms for 2 workerslots, 5 workerslots, and 10 workerslots respectively. The reason why average processing latency increases with more workerslots is that: 1) The inter slot and inter node traffic amount increases (see Table 3) when tasks are distributed to more slots. 2) network bandwidth is the bottleneck of Mobile Environment. We can also observe similar result for Mobile Storm from Figure 5b.

![Table 3. Inter Node & Inter Slot Traffic.](image)

| WorkerSlots Num | Storm | Mobile Storm |
|-----------------|-------|--------------|
| InterNode TFC (t/s) | 3511 | 5586 | 6397 | 4548 | 5808 |
| InterSlot TFC (t/s) | 3511 | 4677 | 4729 | 1725 | 4111 |

6.2. Effect of different schedulers
Figure 6 depicts the performance metrics of running Word Count topology using Storm and Mobile Storm, with increasing number of worker slots. We can see that Mobile Storm achieves a 57%, 79%, and 81% speed up over Storm in term of processing latency. From Table 3 we can find that Mobile Storm reduces inter slot and inter node traffic significantly compared with Storm. This explains why the performance of Mobile Storm is better than Storm in terms of processing latency. Table 3 also indicates that Mobile Storm is more energy efficient than Storm, as transmission via Wifi link consumes much more energy than via inter node transmission channels. However, we can see from Figure 6 that it takes average 100 seconds for Storm or Mobile Strom to stabilize after a new assignment is applied, the overhead of applying new assignment is so high that new assignment should not be deployed too frequently. In order to achieve best performance, users of Mobile Storm need to find a trade off between the overhead of applying new scheduling assignment and the benefit of applying new scheduling assignment. Besides, from this figure, we can observe that the average processing latency curve of Mobile Storm is much more smoother than Storm. The reason is that tries to schedule task pairs with higher inter-task traffic on the same node while storm may have these high inter-traffic tasks scheduled on different nodes. As a result, Mobile Storm minimizes the effects from wireless communication and thus attains better stability.
Figure 6. Performance on word count topology.

7. Conclusions
In this paper, we propose a traffic and processing latency aware resource allocation controller and scheduler for Apache storm in mobile network. It solves the processing latency and inefficient scheduler issues of traditional storm system in mobile environment. The system implementation experiment shows that results show that our system could achieve more than 50% reduction in processing latency compared with Storm and it could also provide quality of service guarantee in term of processing latency. Continuing the work, we plan to implement the optimal controller, especially we are interested in the situation when we need to achieve some optimal goals, such as minimizing overall average latency of all users, or priority service guarantee for multiple user environment.

References
[1] K. Kumar and Y.-H. Lu, “Cloud computing for mobile users: Can offloading computation save energy?” Computer, vol. 43, no. 4, pp. 51–56, 2010.
[2] “Apache storm.” [Online]. Available: https://storm.incubator.apache.org/
[3] M. Rychly, P. Koda, and P. Smrz, “Scheduling decisions in stream processing on heterogeneous clusters,” in Complex, Intelligent and Software Intensive Systems (CISIS), 2014 Eighth International Conference on, July 2014, pp. 614–619.
[4] E. E. Marinelli, “Hyrax: cloud computing on mobile devices using mapreduce,” DTIC Document, Tech. Rep., 2009.
[5] J. George, C.-A. Chen, R. Stoleru, and G. Xie, “Hadoop mapreduce for tactical clouds,” in Proc. of CloudNet, 2014.
[6] “Apache storm 0.9.2-incubating release.” [Online]. Available: http://www.apache.org/dyn/closer.cgi/storm/apache-storm-0.9.2-incubating/apache-storm-0.9.2-incubating.tar.gz
[7] “Word count topology,” https://github.com/apache/storm/blob/master/examples/stormstarter/src/jvm/storm/starter/WordCountTopology.java.