MOBILE STORM: DISTRIBUTED REAL-TIME STREAM PROCESSING FOR MOBILE CLOUDS

A Thesis

by

QIAN NING

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Radu Stoleru
Committee Members,	I-Hong Hou
	Jyh-Charn (Steve) Liu
Head of Department,	Dilma Da Silva

December 2015

Major Subject: Computer Engineering

ABSTRACT

Recent advances in mobile technologies have enabled a plethora of new applications. The hardware capabilities of mobile devices, however, are still insufficient for real-time stream data processing (e.g., real-time video stream). In order to process real-time streaming data, most existing applications offload the data and computation to a remote cloud service, such as Apache Storm or Apache Spark Streaming. Offloading streaming data, however, has high costs for users, e.g., significant service fees and battery consumption. To address these challenges, we design, implement and evaluate Mobile Storm, the first stream processing platform for mobile clouds, leveraging clusters of local mobile devices to process real-time stream data. In Mobile Storm, we model the workflow of a real-time stream processing job and decompose it into several tasks so that the job can be executed concurrently and in a distributed manner on multiple mobile devices. Mobile Storm was implemented on Android phones and evaluated extensively through a real-time HD video processing application. The result shows that Mobile Storm effectively processes HD Video Stream in a mobile cloud, which would be impossible on a single mobile device.

DEDICATION

To my parents, sister and girl friend.

ACKNOWLEDGEMENTS

This thesis was made possible due to the masterly guidance of Prof. Radu Stoleru. I would like to express my sincere gratitude to him for giving me invaluable guidance and sharing his knowledge and experience in my study. I would also like to thank Jay Chen for all he has done to help me with my thesis. Finally, I am very much grateful to my family for their constant support.

NOMENCLATURE

GOP	Group of Picture
JVM	Java Virtual Machine
DVM	Dalvik Virtual Machine

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

1.1 Motivation

Mobile devices are generating multimedia data more and faster than ever. Mobile users today not only share images or videos stored on their phones, but they also stream real-time video from one mobile device to another, e.g., Skype, FaceTime, HangOut. As a result of faster cellular networks, e.g., 3G and LTE, the stream data can be transmitted seamlessly. However, due to the limited computational power of mobile devices, processing the stream data in real-time is still impractical.

Offloading real-time stream data to a remote cloud is a widely used technique to process stream data generated by mobile devices. Major companies like Twitter use Apache Storm [26] - a real-time stream processing platform, to process large amount of stream data produced by its users. However, offloading real-time stream data to a remote cloud has several limitations: i) streaming applications require high bandwidth communication links. Although the current 3G/4G technology is capable of handling such traffic, users have to pay for data sent to or received from the cellular network; ii) the available bandwidth of cellular network depends on the number of users connected to the cellular tower, so it can be highly unstable. For example, during the 2009 U.S. Presidential Inauguration, many wireless data services failed due to millions of people attending this event [23]; iii) as shown in table 1.1, 3G or 4G technologies consume much higher power compared to WiFi [20]; iv) to meet the increasing demand of mobile users, the bandwidth and processing power of remote cloud platform also need to be improved regularly. As a result, harvesting computational resources from local mobile devices, i.e., mobile cloud, becomes an attractive solution. Instead of pushing streaming data to a remote cloud, one can

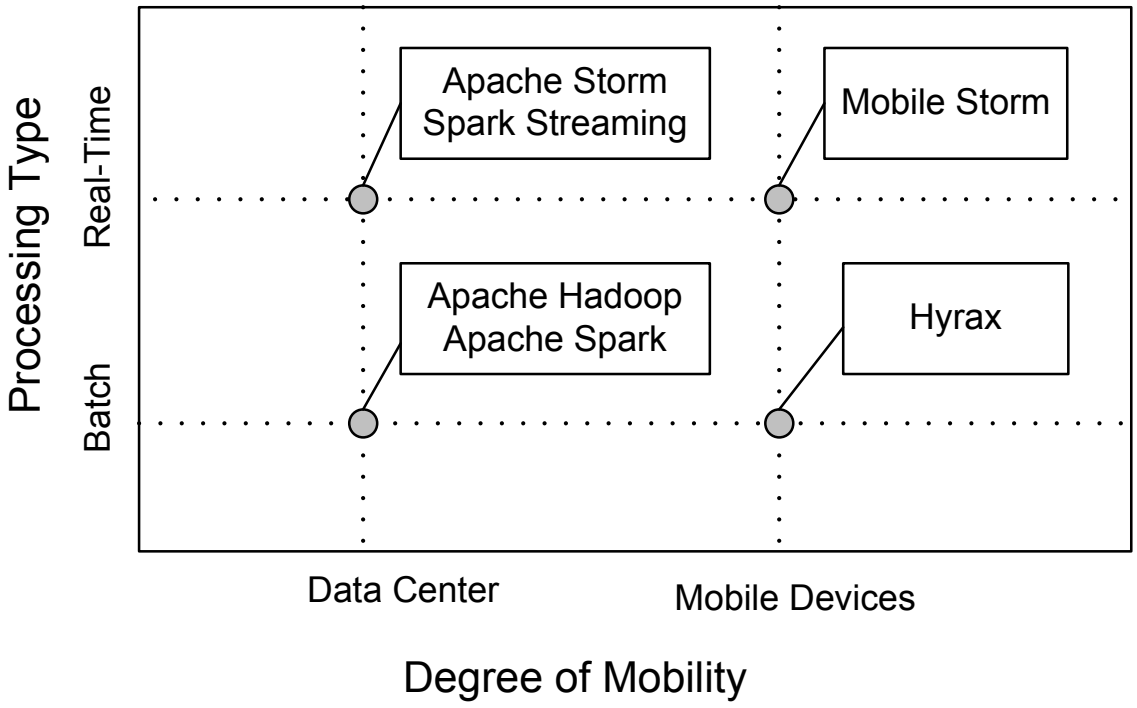

Figure 1.1: Overview of current distributed computing technologies

	Up (mW/Mbps)	Down (mW/Mbps)	Base (mW/Mbps)
LTE	438.39	51.97	1288.04
3G	868.98	122.12	817.88
WiFi	283.17	137.01	132.86

Table 1.1: LTE, 3G, and WiFi data transfer power

distribute stream data to mobile devices in the vicinity and utilize them to process stream data in real time. This way, the cost of transmission energy, service fees, and cloud maintenance are all drastically reduced.

To the best of our knowledge, no distributed computing technologies have been developed for distributed real-time stream processing on mobile devices. As shown in Figure 1.1, the current distributed computing solutions focus on either batch/stream

processing in data centers or only batch processing on mobile devices. Apache Hadoop and Spark [28] are used in large data centers for batch processing. Apache Storm and Spark Streaming are developed for real-time stream processing, and are also used for large data centers. Hyrax [23] is the mobile version of Apache Hadoop that is used for batch processing on mobile devices.

1.2 Research Challenges

This section describes the challenges involved in bring real-time distributed stream processing to mobile devices.

As shown in Figure 1.1, there are two intuitive options to bring distributed stream processing to mobile devices. The first option is to change the design of Hyrax [23] such that real-time processing of stream data is possible. However, Hyrax is implemented by porting Hadoop's Data Node to mobile devices, so it is difficult to change its design to process real-time stream data. The second option is to port the existing traditional server based real-time stream processing frameworks, e.g., Storm [26] and Spark Streaming [10], to mobile devices. However, several major challenges prohibit this:

1) both platforms have a very large code base and are written using different programming languages (e.g. Storm has 100,000 lines of codes written in Java, Clojure and Python, Spark Streaming is even more complex and is written using Scala, Java and Python.

2) many third-party libraries are used in these projects and most of them are designed and used for traditional servers.

3) they both use JVM (Java Virtual Machine) instances as worker processes. However, mobile devices, like Android, use their own VMs, which leads to significant incompatibilities.

4) their system design is too heavy for mobile devices with limited memory resource and computing power. These will reduce the processing speed, impacting the real-time aspects of the stream processing. Additionally, they use multiple JVM processes in each worker node to execute jobs, which is too demanding for mobile devices;

5) they are designed for and used in large data centers, where devices are connected using wired networks, which are much more reliable and have higher capacities than wireless networks employed by mobile devices.

6) their coordination architecture only works for a single cluster. However, in a mobile environment, multiple clusters may exist. Since it is hard to port existing distributed real-time stream processing platforms to mobile environment, we designed Mobile Storm, a new solution from scratch, referencing design ideas and architectures of existing real-time stream processing frameworks [26] [10]. We also use mobile platform's programming language and libraries to implement and evaluate it. Compared with Spark Streaming, Storm's design is more suitable for mobile devices due to its simplicity. Consequently, we based our design decisions on Storm.

1.3 Contributions

The contributions of this thesis are as follows:

1) it presents the design of the first real-time stream processing system for mobile clouds.

2) it argues for the decisions we made during our design of Mobile Storm, which were based on what could be or not be inherited from the design of Storm.

3) it presents an API that allows mobile application developers to build real-time stream processing applications easily.

4) it demonstrates the feasibility and performance of our system design through

a real system implementation on Android devices.

1.4 Related Work & Background

In this section, we present the state of the art from two perspectives: distributed real-time stream processing on traditional server clusters, and distributed computing on mobile devices.

1.4.1 Distributed Batch Processing on Server Clusters

Many distributed batch processing frameworks on traditional server clusters have been developed and widely used today. One of the most well-known one is Apache Hadoop [3], which is developed based on MapReduce framework. Hadoop mainly contains two components, MapReduce framework and HDFS (Hadoop Distributed File System). The distributed processing is done by two distinct tasks- the *Map Task* and the *Reduce Task*. User implements mapper and reducer interfaces in their applications. The mapper is responsible for taking the input data set and producing a set of intermediate <key,value> pairs which are sorted and partitioned per reducer. These pairs are then sent to reducer which is responsible for producing the final output. The HDFS is a distributed file system used to provide high-throughput access to application data.

The high-level architecture of Hadoop is shown in Figure 1.2. It consists of two different nodes, NameNode and DataNode. The NameNode is the master node of an HDFS file system. It keeps the directory tree of all files in the file system, and tracks where file data is kept. The DataNode is the slave node which is responsible for storing data in HDFS file system. In MapReduce layer, there are two important modules, JobTracker and TaskTracker. JobTracker is responsible for receiving user jobs and splitting them into tasks which are assigned to the TaskTrackers in slave node. TaskTrackers act as mappers and reducers. During the job execution, Job-

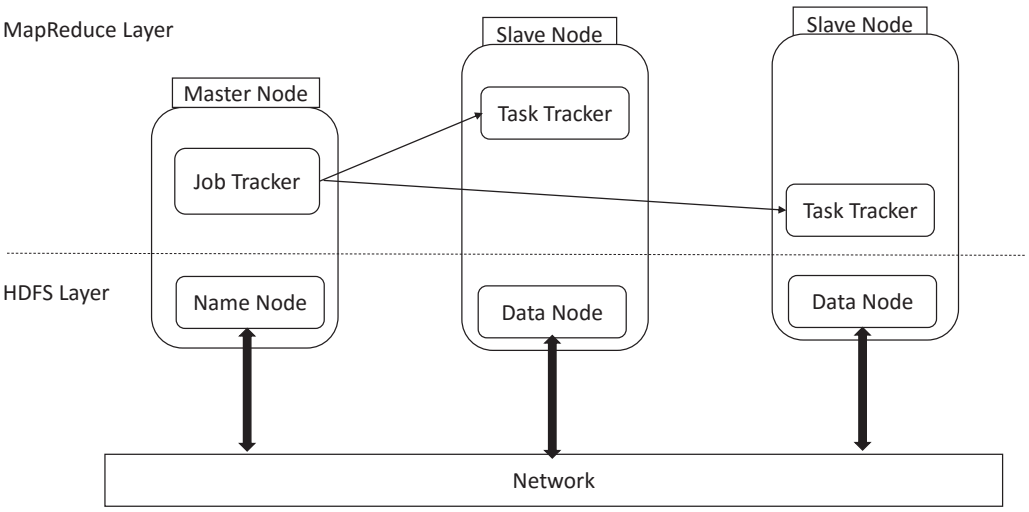

Figure 1.2: Hadoop architecture

Tracker monitors the health status of the execution and re-execute the failed tasks. In HDFS layer, the data is transferred between DataNodes through network.

Apache Spark [28] is an another distributed batch processing framework. Spark can run in Hadoop clusters, and can process data in HDFS. It enables programs run up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk. Spark is based on two concepts: Resilient Distributed Datasets (RDD) [27] and directed acyclic graph (DAG) execution engine. RDDs support two different kinds of operations: transformations and actions. Transformations is responsible for creating new data sets from the input, which is like what mapper does in Hadoop. And then actions producing output from the data sets, which is like what reducer does in Hadoop. The DAG engine can eliminate the MapReduce multi-stage execution model and improve the performance significantly.

The high level architecture of Spark is shown in Figure 1.3. Spark applications runs as independent sets of processes, and they are coordinated by Spark Driver. The Spark Driver is connected to Cluster Manager which is responsible for allocating

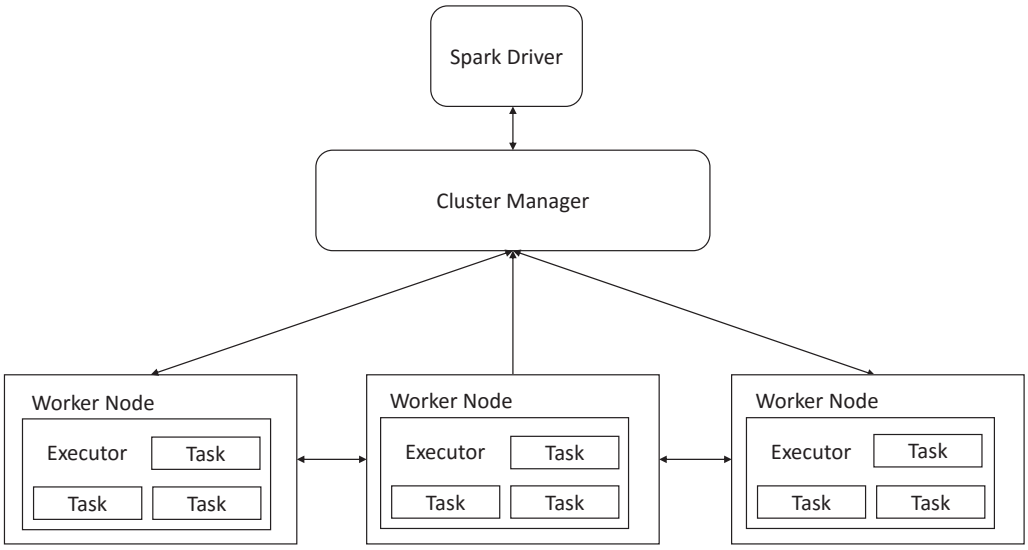

Figure 1.3: Spark architecture

resources across applications. Once connected to Cluster Manager, the Spark Driver sends tasks to the executors in the worker node to run.

1.4.2 Distributed Real-time Stream Processing on Server Clusters

There have been many previous solutions for distributed real-time stream processing on traditional server cluster. Yahoo developed its own distributed real-time stream processing platform S4 [24]. In S4, a stream, which is a sequence of events, is processed using PEs (Processing Elements). Events are emitted and consumed by PEs. S4's framework also provides the capability to route events to appropriate PEs. Another well-known system is Streaming API project of Apache Spar named Spark Streaming [10], which can process real-time stream date with low latency. Spark Streaming runs stream processing as as series of very small, deterministic batch jobs. Spark Streaming partitions the stream into micro batches of data which are then used as input for Spark. Spark takes these micro batches of data as RDDs and processes them using RDD operations.

1.4.3 Distributed Computing on Mobile Devices

Distributed computing on mobile devices has been previously attempted either by offloading data and computation to remote cloud servers, or by constructing a distributed computing cluster with local mobile devices.

Cuervo [17] proposed MAUI, an architecture that improves the energy efficiency by offloading some code to remote servers. MAUI offloads code to a remote server only if the remote execution results in higher energy efficiency. Zhang [29] proposed an elastic application model that enables the seamless and transparent use of cloud resources to improve computational capability of mobile devices. He designed a cost model to decide the execution configuration of application during runtime, in order to optimize power consumption, monetary cost, performance, security and privacy. Chen [15] proposed an aspect-oriented programming architecture that allows mobile application developers to easily offload part of the computation to servers in the cloud. This architecture inserts offloading code into the application's source code, based on static and dynamic profiling. They also implemented a proof of concept system on Android. Many similar works have been described in [19, 22, 25, 16, 18, 14]. They all involve offloading local data to remote servers to improve the mobile application's performance. Though they designed their algorithms to minimize the energy cost of transferring the data to remote servers, they cannot avoid offloading data to remote servers and are not suitable for real-time stream data. Besides, both proposed architectures need large investments in remote servers.

Marinelli [23] developed a mobile phone-based cloud computing platform by porting Hadoop's Data Node to Android phones. Processing data in Hyrax does not require data transfers to remote servers. In Hyrax, NameNode is running on traditional servers. The user's job is submitted to NameNode, and then NameNode send

tasks to Data Node running on mobile phones. A similar work also has been done in [21]. However, these previous works can only do distributed batch processing, since they are based on Hadoop are not suitable for real-time stream data processing as Hadoop is only suitable for batch processing.

1.4.4 Overview of Apache Storm

Storm [4] is a scalable and fault-tolerant distributed real-time stream processing platform that is used by many companies such as Yahoo and Twitter. The high level architecture of Storm cluster is shown in Figure 1.4, it consists of three parts, Nimbus, ZooKeeper and Worker nodes. Nimbus, is the master node of the system, responsible for coordinating the execution of tasks, such as scheduling and distributing tasks to worker nodes. ZooKeeper [13] stores the information needed for coordination between Nimbus and Worker nodes, such as the tasks assignment and heartbeat information of worker nodes. Worker node processes the actual stream data. Each worker runs a Supervisor daemon that listens to tasks assigned to this node from Nimbus. Workers are separate Java Virtual Machines, and each one contains multiple executors (threads) that execute multiple tasks.

In Storm, the unbounded stream data is described as a sequence of tuples. Tuple is the smallest data entity in Storm that can be serialized and transferred through networks. As shown in Figure 1.4, Storm uses a topology consisting of *Spouts* and *Bolts* to describe the workflow of a real-time stream processing job [4]. Spouts read tuples from the stream data source and emits them to bolt. Bolts consume the received tuples by user-defined processing procedures. Bolts may also generate new tuples and send them to other bolts. In Storm, a single spout or bolt can have multiple instances, and each instance corresponds to a single task that to be executed. Storm provides the following stream grouping policies to decide how to

partition and distribute stream data to bolts' tasks [11]:

- *Shuffle grouping*: Tuples are randomly distributed to bolt's tasks to guarantee each task receives equal number of tuples.
- *Fields grouping*: The stream is partitioned by the specified fields such that tuples with the same specified field will be sent to the same task.
- *All grouping*: Each tuple is sent to all the bolt's tasks.
- *Global grouping*: All tuples are sent to a single bolt's task with the lowest task ID.
- *Direct grouping*: The producer of the tuples decides which task that the tuples should be sent to.
- *Local or shuffle grouping*: If there are bolt's tasks existing in the same node, then send tuples to these tasks randomly. Otherwise, use shuffle grouping.

Storm's parallelism is expressed in two levels, abstract model level (topology) and system level (Storm cluster). At abstract level,the parallelism is expressed by allowing topology's spouts and bolts to have multiple tasks. At system level, the parallelism is expressed by having multiple executors to execute bolt's or spout's tasks.

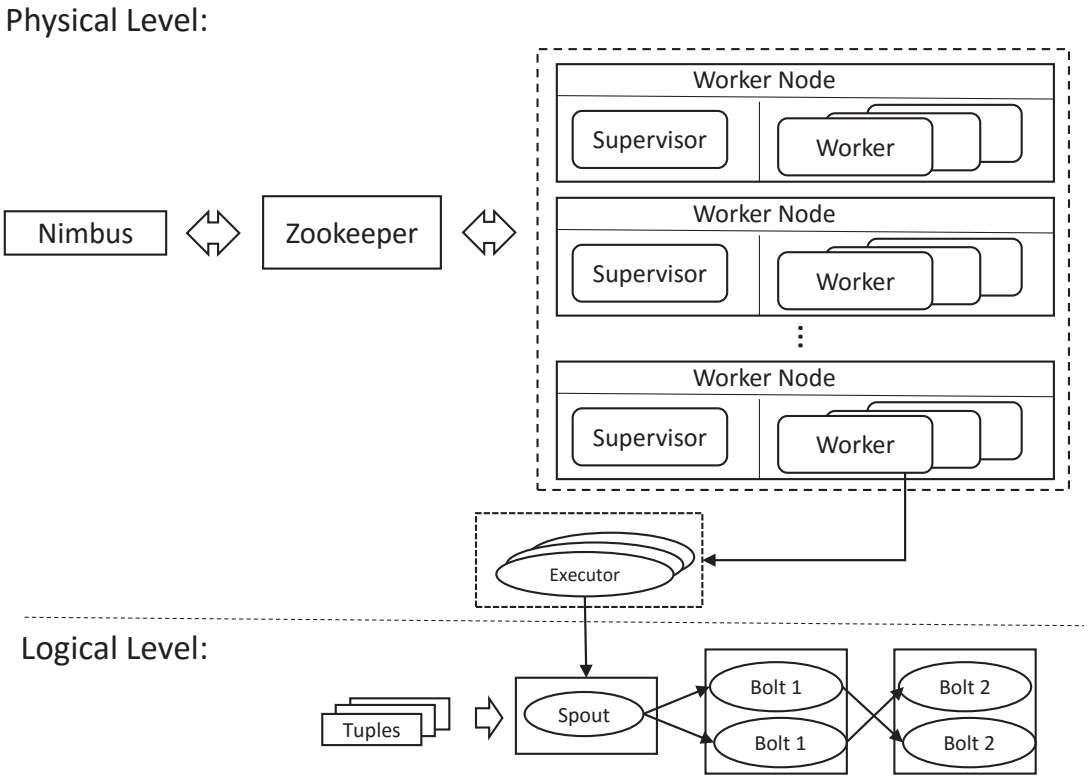

Figure 1.4: High level overview of Storm Cluster and Storm Topology

2. SYSTEM DESIGN

In this section we present the design of Mobile Storm and the decisions we made when investigating what functions/internals of Storm needed to be supported by Mobile Storm. We present our design from two perspectives: a *Logical Level* and a *Physical Level*. In the Logical Level, we explain how the processing of a real-time stream job takes place in Mobile Storm; in the Physical Level, we present the architecture of Mobile Storm.

2.1 Logical Level

Our design for Mobile Storm from a Logical Level perspective, as shown in Figure 2.1, employs *Spouts*, *Bolts* and *Topologies*, similar with Storm. A Topology is a graph that describes the workflow of a user's real-time processing of a job. A Topology contains two types of nodes: a) a *Spout* is used to partition the stream data from a source into *tuples*, which are then serialized and distributed to *Bolts*. How the tuples are generated is defined by the application executing on the Spout; b) a *Bolt* is responsible for processing tuples received from the Spout. Users define how incoming tuples should be processed in the Bolt. The method for distributing tuples to Bolts, either from a Spout or from a Bolt, is called *Stream Grouping*. Take the Topology in Figure 2.1 as an example. The Spout receives the stream data from the data source, partitions it into tuples, and then distributes them to Bolt 1. After Bolt 1 finishes processing the data, it generates and sends new tuples to Bolt 2. Bolt 2 again processes the incoming tuples by the application defined processing functions.

The tuples generated by Spouts in Mobile Storm are different from those in Storm. In Storm, the tuple is a list of Java objects which must be serialized before they are distributed to Bolts. In Mobile Storm, a tuple is format-free which means the user

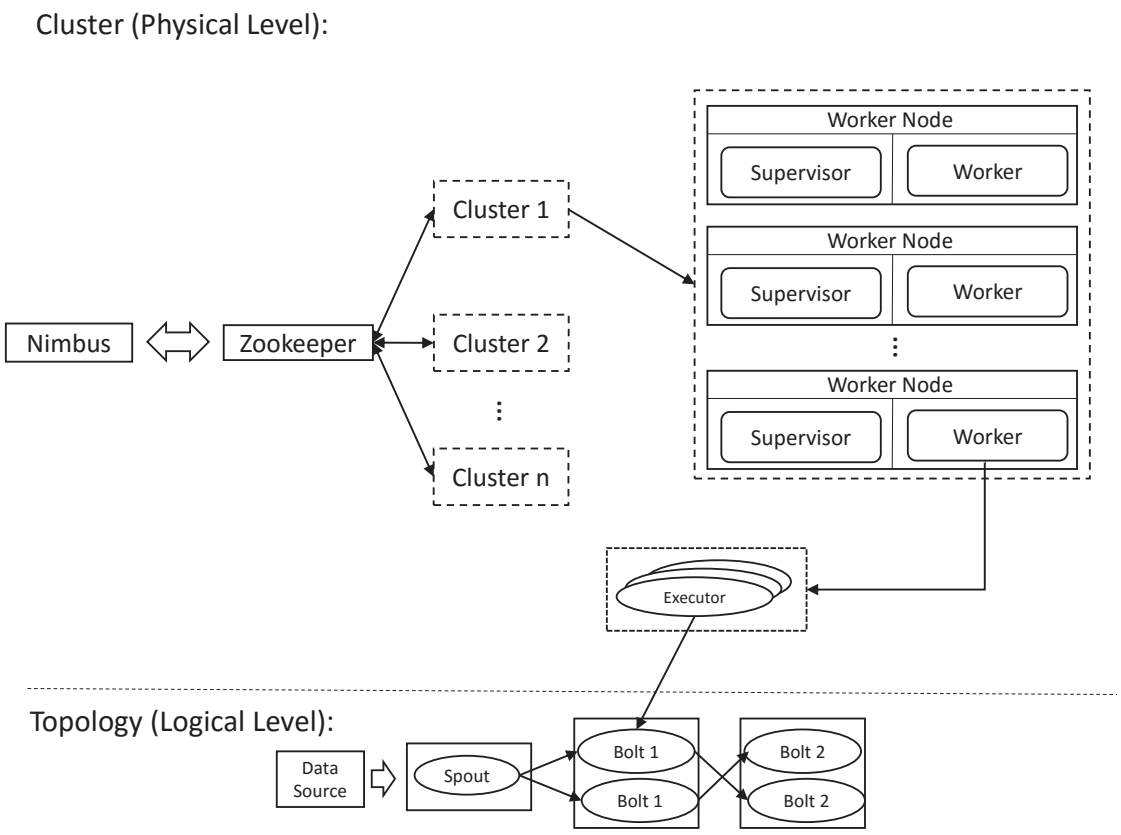

Figure 2.1: The logical and physical design levels for Mobile Storm

can define his/her own format for tuples, and the tuples do not need to be serialized. Users implement the abstract method `decode()` in Bolt to decode the format-free raw data that they define. We made this decision based on the following: a) some data generated by mobile devices do not need to be serialized, e.g. output data of video/audio codec can be sent directly through the network; b) DVM on Android is not optimized for serializing or deserializing Java objects, so it can be very inefficient when a stream of objects need to be serialized or deserialized in real-time.

We designed Mobile Storm to provides two types of Stream Grouping methods: *Shuffle Grouping* and *Local or Shuffle Grouping*. In Shuffle Grouping, tuples are randomly distributed to Bolt's tasks to guarantee each task receives an equal number

of tuples. In Local or Shuffle Grouping, if there are Bolt's tasks on the same node, then send tuples to these tasks randomly; otherwise, use Shuffle Grouping. Local or Shuffle Grouping is very useful in Mobile Storm, as it can reduce the inter-node wireless communications to conserve energy on mobile devices. We leave for future work the implementation of other Stream Grouping methods, which are not as popular as the aforementioned two. Other grouping methods include: *Fields Grouping*, in which the stream is partitioned by the specified fields such that tuples with the same specified field will be sent to the same task; *All Grouping*, in which each tuple is sent to all the Bolt's tasks; *Global Grouping*, where all tuples are sent to a single bolt's task with the lowest task ID; and *Direct Grouping*, in which the producer of the tuples decides which task that the tuples should be sent to.

2.2 Physical Level

In this section we present the architecture of Mobile Storm, as shown in the bottom of Figure 2.1, and its components: Clusters, Worker Nodes, Nimbus and ZooKeeper.

2.2.1 Cluster

Mobile Storm is designed for environments where mobile devices, organized in Clusters, are connected to each other through local wireless networks. E.g., mobile users in Starbucks who connect their devices to the same router can form a cluster. Multiple clusters may exist in mobile environments as shown in Figure 2.1. We use a Cluster to organize a collection of Worker Nodes (i.e., mobile devices). A user's job can only be executed on the cluster to which his mobile device belongs. To manage clusters, each cluster is assigned a unique cluster ID. Only with this ID, a mobile device can join the cluster. Clusters must have access to Nimbus and ZooKeeper services, which we decided to place in a remote cloud. In our design of

Mobile Storm, Nimbus and ZooKeeper are deployed on remote Cloud servers as the memory, computation, and link capacity requirements of ZooKeeper are significantly more demanding than what is available on mobile devices. ZooKeeper and Nimbus coordinate Worker Nodes of Mobile Storm and their performance directly impacts the performance of the entire system.

Although running Nimbus and ZooKeeper on remote servers requires all mobile devices to have a connection to the servers, this is no longer a problem for today's mobile phones, as most of them have internet connections through 3G/4G wireless networks. Additionally, the communication with Nimbus and ZooKeeper does not require a high-bandwidth network connection, as both of them only transmit/receive configuration data which only happens at the beginning of a job execution, and not the stream data to be processed during the job execution.

2.2.2 Worker Node

A Storm cluster consists of multiple Worker Nodes. Each Worker Node mainly has two components: a) a *Supervisor*, which is responsible for receiving tasks from Nimbus and assigns task to Workers; b) multiple *Workers*, which are independent JVM processes, with each containing multiple threads that execute tasks assigned by the supervisor.

In Mobile Storm, the worker nodes are Android devices whose application runs on a single Dalvik Virtual Machine (DVM) [5] instance. Storm needs to create multiple JVM processes for each Worker Node. Similar to Storm, Mobile Storm creates one Service Process [2] for each worker node where each process has separate memory address space and can communicate through Inter-Process Communication. Figure 2.2 illustrates an example of one worker process and one supervisor process running on a single worker node (one Android device). Each worker node has one

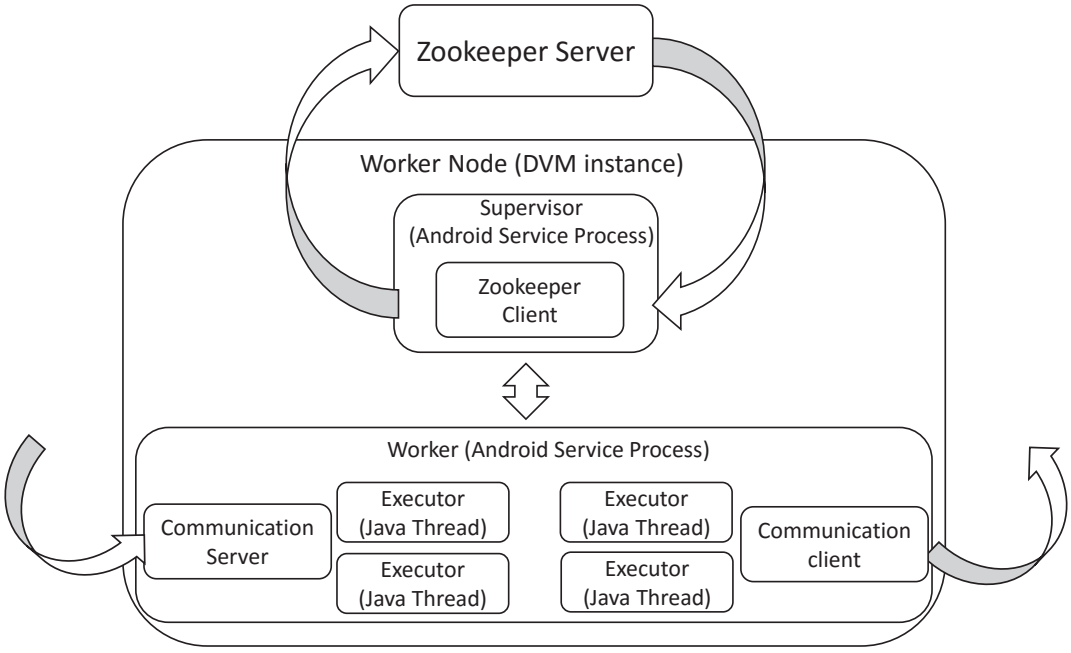

Figure 2.2: Processes running on worker nodes

worker process and each worker process has n Executors (threads) where n is the number CPU cores on the mobile device. These executors are managed by Android thread manager.

In each work node, a Zookeeper client is running on the background and listening to the assignment directory in Zookeeper server. Zookeeper keeps checking the communication channels with a work node, and once a node lose connection for a particular time, this node will be identified as being off-line. In worker node, there are communication server and client which are responsible for receiving and emitting tuples, respectively. The communication server and client is supported Netty [8] which is an asynchronous event-driven network application framework. Netty has very good performance and is also used in Storm for inter-node communications.

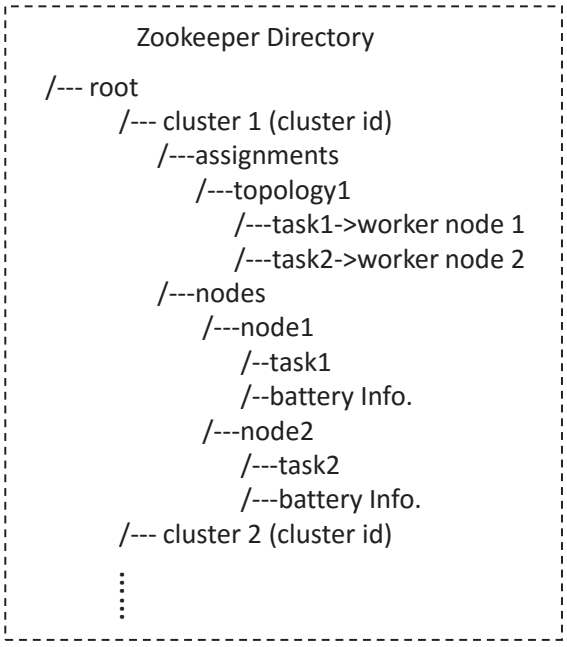

Figure 2.3: Zookeeper directory

2.2.3 Nimbus

Nimbus, the master node in Storm and Mobile Storm is responsible for scheduling and coordinating task executions. It has two functions: a) it schedules Topology's tasks to available Worker Nodes inside a cluster. The default scheduler of Storm distributed tasks to nodes in a round-robin manner; b) it monitors the execution of tasks and recovers the system from node failures.

In Mobile Storm, we modify Storm's default scheduler in order to reduce energy consumption caused by wireless communications among different mobile devices. Our strategy is to classify tasks into several groups. Tasks within a group communicate with tasks in the same group. We then try to assign a group of tasks to the same Worker Node. So in this way, we can reduce a lot of inter-node communication which is very expensive for mobile devices.

In Mobile Storm, the failure recovery process is also different from that in Storm. In Storm, when a Worker Node dies, Nimbus assigns all its tasks to free Workers. In Mobile Storm, we first assign the dead node's tasks to available Executors of the busy Workers in the same topology. Then we assign the remaining tasks to new Workers. In this way, we can utilize the limited number of Executors more efficiently.

2.2.4 ZooKeeper

In Storm, ZooKeeper is responsible for the coordination between Nimbus and the Worker Nodes in the cluster. It mainly uses two types of information for coordination: a) *Assignments information for this cluster*: ZooKeeper maintains an assignment directory that stores the [task, worker] assignment information for each topology; b) *Worker Node information*: the tasks currently executing on a node and the heartbeat information of a node. The directory name for each node contains its IP address and port number.

As aforementioned, multiple clusters exist in Mobile Storm. In order to manage these clusters, Mobile Storm gives each cluster a unique cluster ID and creates a corresponding directory using this ID, as shown in Figure 2.3. Under each cluster's directory, Mobile Storm stores the information about the assignment for this cluster and node's information in this cluster. It also stores worker node's battery information on ZooKeeper which will be used by the scheduler in Nimbus. All these are updated periodically.

2.2.5 Parallelism

High processing speed is achieved by processing data in parallel. In Storm, each Spout or Bolt can have multiple instances where each instance is considered a *task*. Tasks of a Spout/Bolt can be executed in parallel by Executors in the cluster. When specifying the Topology, a user sets the number of tasks (instances) and the number

of Executors for each component (Spout of Bolt). If the number of tasks is set higher than the number of Executors, some Executor will execute more than one task. In Mobile Storm, we set the number of tasks equal to the number of Executors for each component. We made this decision for simplicity and to avoid overloading the mobile nodes.

2.2.6 Programming Model

The use of Mobile Storm API is straightforward and flexible. Developer only needs to create Spouts and Bolts for his real-time stream processing job's workflow graph. When creating Spout, developer first needs to implement `execute()` method to obtain stream data and chop up it into tuples, and then call `emit(tuple, taskID)` to distribute these tuples to particular tasks. When creating bolts, developer have to implement `decode()` method to reconstruct tuples from the received raw data. The received tuples are processed by Bolt's `execute(tuple)` method. In Bolt's `execute(tuple)` method, developer defines his own way of processing data pieces.

Once the Spouts and Bolts are created, developer can use topologyBuilder's `setSpout()` and setBolt methods to map the relationships between distributors and processors to a directed workflow graph. Use And then call `submit()` to submit this topology to Nimbus.

We demonstrate this using a real-time video stream decoding job an example. This job's topology has one Spout and on Bolt. Spout obtains H.264 encoded stream video from somewhere and chops up it into GOPs (Group of Pictures) which can decoded independently. Bolt decodes GOPs into original frames. So in distributor's `execute()` method, developer reads video stream data from either local file or remote server and chops up it to GOPs according to the H.264 encoded standard, and each GOP is sent to Bolts as a tuple. Accordingly, in Bolt's `decode()`

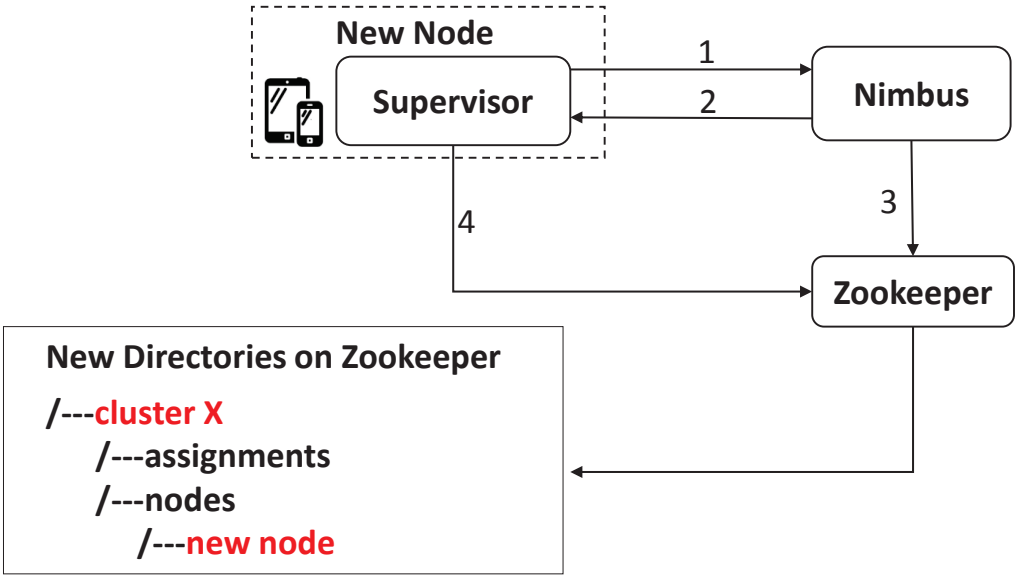

Figure 2.4: Steps for creating a new Mobile Storm Cluster

method, developer needs to extract GOPs from raw received stream data. In Bolt's `execute(data_piece)`method, developer can use some third-party libraries to decode received GOPs. Using topologyBuilder methods to create a new topology and submit this topology to Nimbus.

2.3 Mobile Storm System Operations

In this section we present four fundamental operations in Mobile Storm: the creation of a Cluster, a new node joining an existing Cluster, the execution of a job, and recovery from node failure.

2.3.1 Set up a New Cluster

Figure 2.4 depicts the steps needed for the creation of a new cluster by a node.

Step 1: The node sends a request to Nimbus, to create a new cluster.

Step 2: Nimbus replies back with a unique cluster ID to the node.

Step 3: Nimbus creates a new directory on ZooKeeper using the cluster ID as directory name.

Step 4: Once the node receives the cluster ID, its supervisor sends a request to Zookeeper server to create a new subdirectory under /cluster ID/nodes/ on ZooKeeper.

2.3.2 Join an Existing Cluster

To join an existing cluster, the user needs to know cluster's ID. With this information, the node's Supervisor will send a request to Zookeeper, which contains its IP address and port number. And then Zookeeper will create a new subdirectory under /cluster ID/nodes for this new node. At the same time, nimbus will receive an notification, since a new node comes. It record this new node's information such as battery life and available number of executors. And then Nimbus add this new node to the free node list.

2.3.3 Job Execution

Figure 2.5 illustrates an example of how a user's job is executed and the Topology of the job. In this cluster, there are three Worker Nodes and each node is one mobile device. Out of these three Worker Nodes, Node 1 executes Spout's tasks, Node 2 and Node 3 execute Bolts' tasks. Each node has two Executors. The step by step workflow is as follows:

Step 1: User submits the job's Topology and application code file (.apk file) to Nimbus, and waits for a response from Nimbus. The Topology is serialized to JSON [7] format, and will be deserialized on Nimbus side. The code file consists of two parts, class file (.dex file) and libraries. The class file contains the user-defined Spouts and Bolts and the native libraries they need to reference. User can use tools provided by Android to convert .jar file to .dex file very easily.

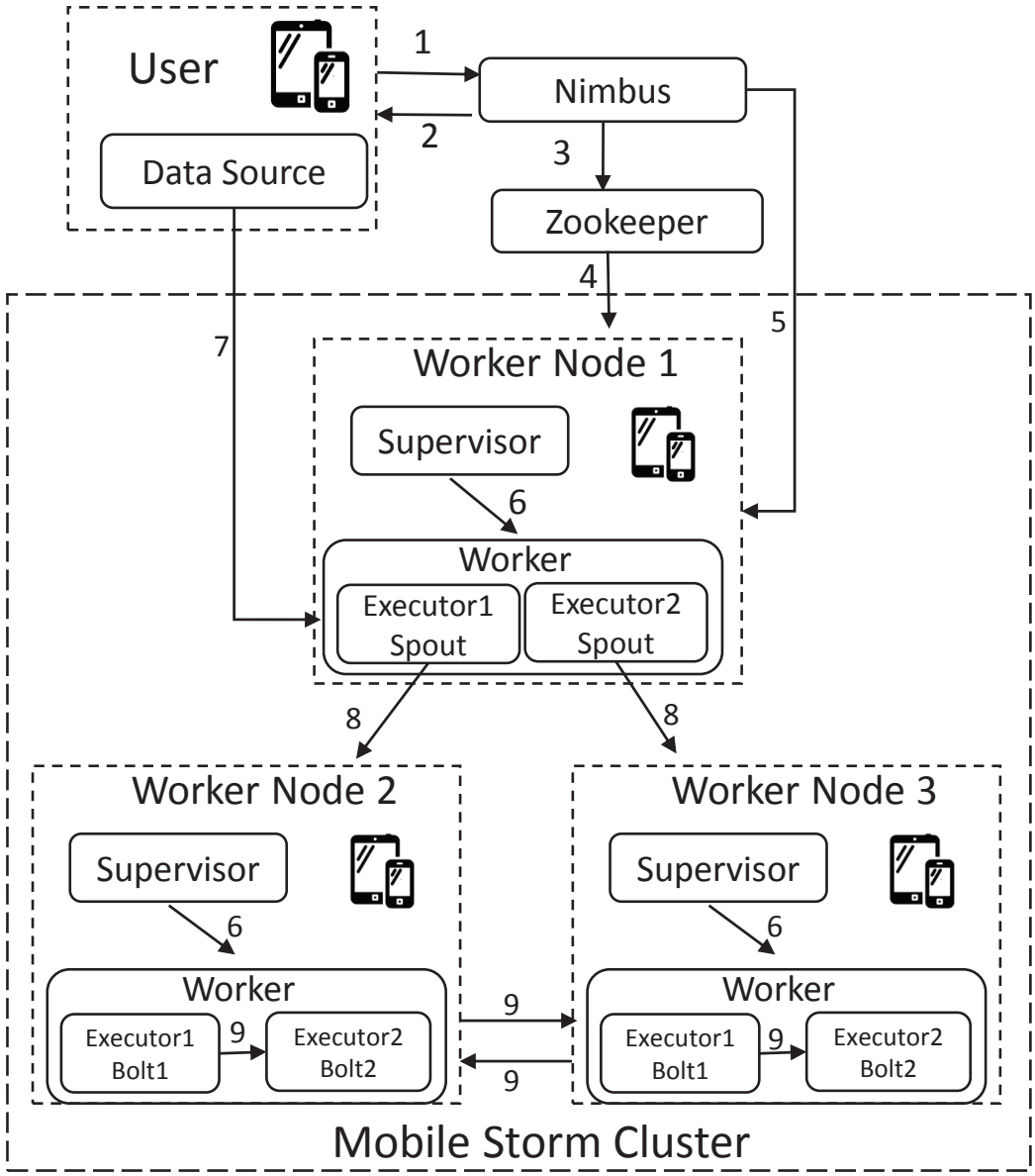

Figure 2.5: Steps for executing a job in a Mobile Storm Cluster

Step 2: If there are enough free Executors for the tasks requested by the Topology, Nimbus assigns one Executor for each task and all tasks can execute in parallel. However, if there are not enough free Executors for the requested tasks, the number of concurrent tasks that Nimbus can start is at most equal to the number of free Executors. Nimbus then notifies the user that the submitted Topology is ready to execute.

Step 3: Nimbus writes assignment information to the assignment directory of user's cluster on ZooKeeper. All the nodes that are assigned tasks to are added to the busy node list.

Step 4: Once the assignment of the new job is ready, ZooKeeper notifies the Supervisor on each Worker Node. Supervisors then download the assignment information from ZooKeeper.

Step 5: Supervisors also download application code file from Nimbus.

Step 6: Once a worker node has downloaded the necessary code files, the supervisor starts a worker process (Android service process). In worker process, the communication server and client first begin to work. We need to analysis the user's input and assignment from Nimbus, and figure out which nodes we need to set up connections. And then communication client begins to set up communication with other nodes according to assignment information and user's topology. A map is also constructed which is used to tell which task we need to send the received tuple to.

Step 7: After all these initializations are done, the worker process then starts multiple Java threads where each thread corresponds to an Executor. Each Executor loads the code files required by the task (Spout/Bolt) and starts executing the task.

Step 8: Once the Mobile Storm cluster starts to execute a Topology, the Spout's tasks continuously retrieve stream data from the data source.

Step 9: Spout's tasks generate user-defined tuples and distribute them to Bolt1's

tasks, and Bolt's tasks begin to process the received tuples according to user defined method.

Step 10: After Bolt1's tasks finish processing, they generate and distributed new tuples to Bolt2's tasks. And then Bolt2's tasks begin to process these new tuples and generate the final results.

2.3.4 Stop Job Execution

Step 1: User send a request to Nimbus to stop the execution for the previously submitted topology.

Step 2: Nimbus will check the record to find out all the nodes that are currently executing this topology. And then it delete the tasks information on Zookeeper and add these nodes to free node list.

Step 3: All the involved nodes receive a notification, since the tasks information is deleted. These node stop the all the executor threads, close any inter-node communications. and then delete the code file and any other related files.

2.3.5 Recovery from Node Failure

We describe the operations for recovering from Worker Node failures using the Cluster and Topology shown in Figure 2.5 as examples. We assume Worker Node 3 disconnects from the system, maybe because its battery discharges completely. Mobile Storm performs the following recovery operations:

Step 1: ZooKeeper detects node failure when Worker Node 3 becomes unresponsive for a specified timeout. ZooKeeper than notifies Nimbus that Worker Node 3 has failed.

Step 2: Nimbus tries to reassign the unfinished tasks of the failed node to other worker nodes that also execute the same topology/job as the failed node. E.g., if Worker Node 2 has two free Executors and is executing the tasks from the same

Topology as Worker Node 3, Nimbus reassigns two of the unfinished tasks from Node 3 to Node 2.

3. SYSTEM IMPLEMENTATION

We implemented Mobile Storm for a Cluster of consisting six Nexus 5 Android phones (partially shown in Figure 3.1). The Android phone has a Qualcomm Snapdragon 800 CPU, 2GB RAM, Adreno 330 GPU and runs Android 5.0.1 OS. All nodes are connected through a 5GHz 802.11n, 300Mbps Wi-Fi network.

In our implementation, the Nimbus and ZooKeeper are deployed on an Amazon EC2 instance. All Worker Nodes in our implementation of Mobile Storm cluster are executing on Android phones, and each phone acts as a single Worker Node. Two Android service processes run continuously on each Worker Node. One process is the Supervisor and the other process is the Worker, which contains multiple Executors (Java Threads). The Supervisor uses the Java ZooKeeper library [12] to communicate with ZooKeeper, and the Worker uses the Netty [8] library for inter-node communication.

When implementing ZooKeeper clients on Android phones, we tried different versions of ZooKeeper API on Android. We found version 3.4.6 works best. Nevertheless, we had to disable SASL (Simple Authentication and Security Layer) from environment variables, so that the ZooKeeper client can work, as it is not supported by Android SDK. This can be done through system properties setting.

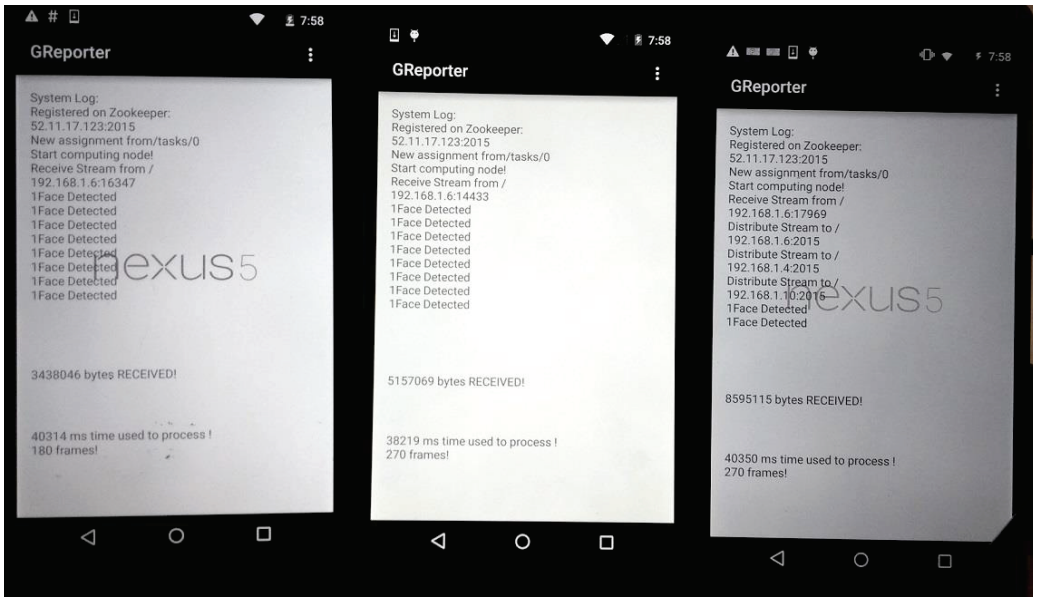

Figure 3.1: Cluster of Android phones used in our implementation

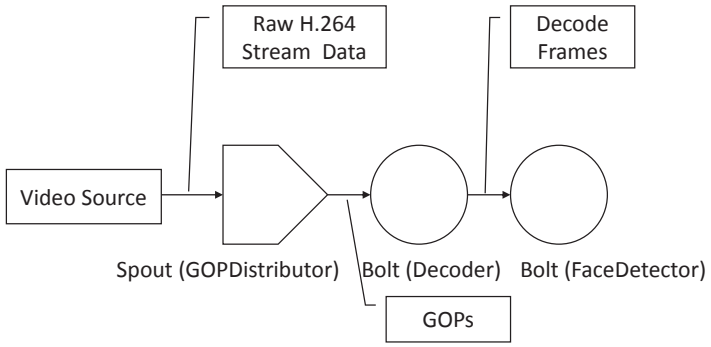

Figure 3.2: Mobile Storm Topology used in our evaluation

4. PERFORMANCE EVALUATION

In this section, we present the performance evaluation of Mobile Storm when processing a real-time video stream on the Cluster/hardware mentioned in the previous section.

We implement a real-time facial feature extraction application that utilizes our Mobile Storm framework. This type of real-time video processing application is too computationally intensive to run on a single mobile device. Figure 3.2 shows the topology of this video processing job. *GOPDistributor* (Spout) reads GOPs (groups of pictures) [6] from the video source and emits them to *Decoder* (Bolt); *Decoder* decodes GOPs into separate frames which then are consumed by *FaceDetector* (Bolt); *FaceDetector* extracts features of faces that appear in the decoded frames.

In this topology, FaceDetector is the most computationally intensive task and is the bottleneck for performance. We try to maximize the throughput of FaceDetector while ensuring that its upstream tasks (GOPDistributor and Decoder) can provide the input data fast enough. The number of tasks (instances) for FaceDetector is set to the number of nodes in the Cluster because each phone can run at most one FaceDetector task due to the limitation of image processing library. The number of tasks for Decoder is also set to the number of nodes in the cluster as each Decoder essentially connects to one FaceDetector. The number of GOPDistributor's task is set to one because there is only one video source. The FaceDectector can also do blink detection, gaze tracking, smile value and face orientation measurement.

We leverage the hardware acceleration for both video decoding and face detecting. The decoding is supported by Android codec library [1], and the face detecting is supported by Qualcomm SnapDragon SDK [9]. There are two different mode when

using this SDK to do face detection, static and video mode. In static mode, we extract facial properties for each input frame. In video mode, we only extract facial properties every 15 frames and keep tracking face between the first and fifteenth frame. We use video mode to improve the face detection speed, however, in the following evaluation result we can find the processing speed is still very slow when video's resolution become higher.

GOPDistributor uses *Shuffle Stream Grouping* to distribute GOPs to Decoder to ensure that each node receives equal number of GOPs. However, because the output of Decoder are large size raw image frames which can reach to several megabytes per frame, to reduce inter-node wireless communications, Decoder uses *Local or Shuffle Stream Grouping* to distribute raw frames to FaceDetector, so the raw frame will be only transferred inside the same node.

The input video video stream is encoded with the H.264 encoder and has 1 Mbps bit rate and 15fps frame rate. Three different resolutions are evaluated: 1) low resolution (800×600); 2) medium resolution (1280×720); and 3) high resolution (1920×1080).

We are interested in the processing speed of our Mobile Storm. In particular, we evaluate the performance of Mobile Storm under input data with different resolutions, frame rates, and degrees of parallelism (i.e., cluster size). To demonstrate the necessity of Mobile Storm, a stand alone video processing application that runs on a single mobile phone was also implemented. This stand alone application, which is referred as *Local Mode* in this section, serves as a performance baseline for Mobile Storm.

For a data stream application to meet the real-time requirement, it must consume the stream data at least as fast as the speed the stream data is generated (e.g., if a video source delivers 15 frames per second (15fps), the video processing application

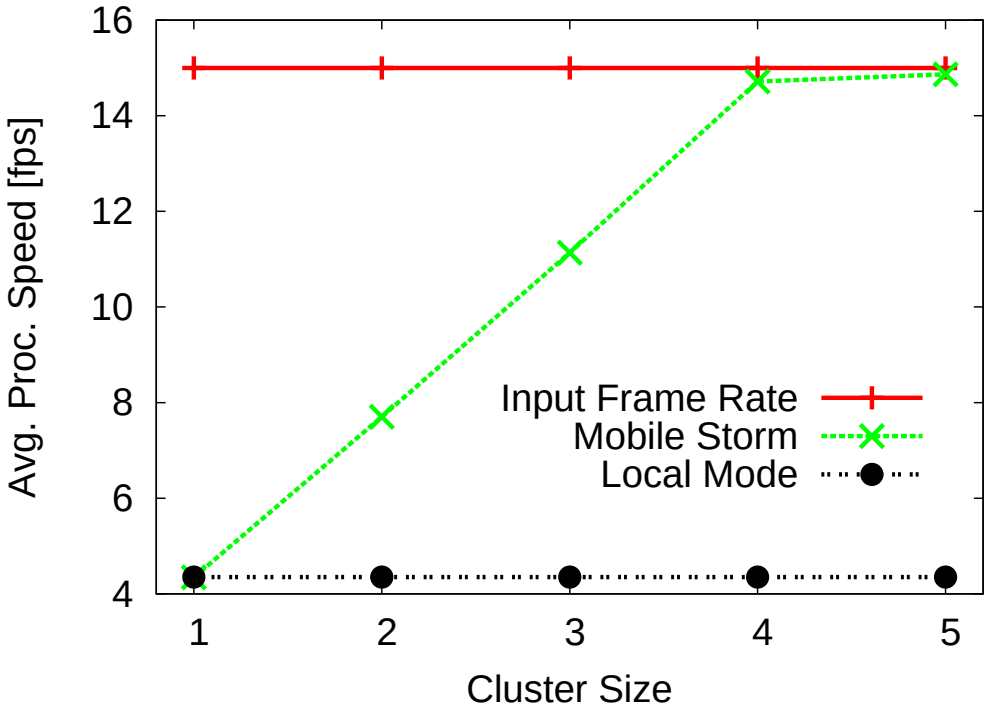

Figure 4.1: Processing speed of Mobile Storm and Local Mode for a video stream at 1920×1080 resolution

must be able to process 15 frames per second). If the processing speed lags behind the data generating speed, the application can not handle the video in real-time.

4.1 Effects of Video Stream Resolution on Processing Speed

As shown in Figures 4.1, 4.2 and 4.3, Mobile Storm is able to meet the input frame rate requirement in all experiments, while the Local Mode can only handle the low resolution video stream. It can be seen that the processing speed of Mobile Storm decreases as the video resolution increases. Similarly, the processing speed increases with the number of Worker Nodes (i.e., cluster size).

Figure 4.1, 4.2 and 4.3 also suggest the needed cluster size for a specific video type. For example, given a low resolution video with 15fps, using a single phone is the best choice (from Figure 4.3); if the video is high resolution, then a cluster of

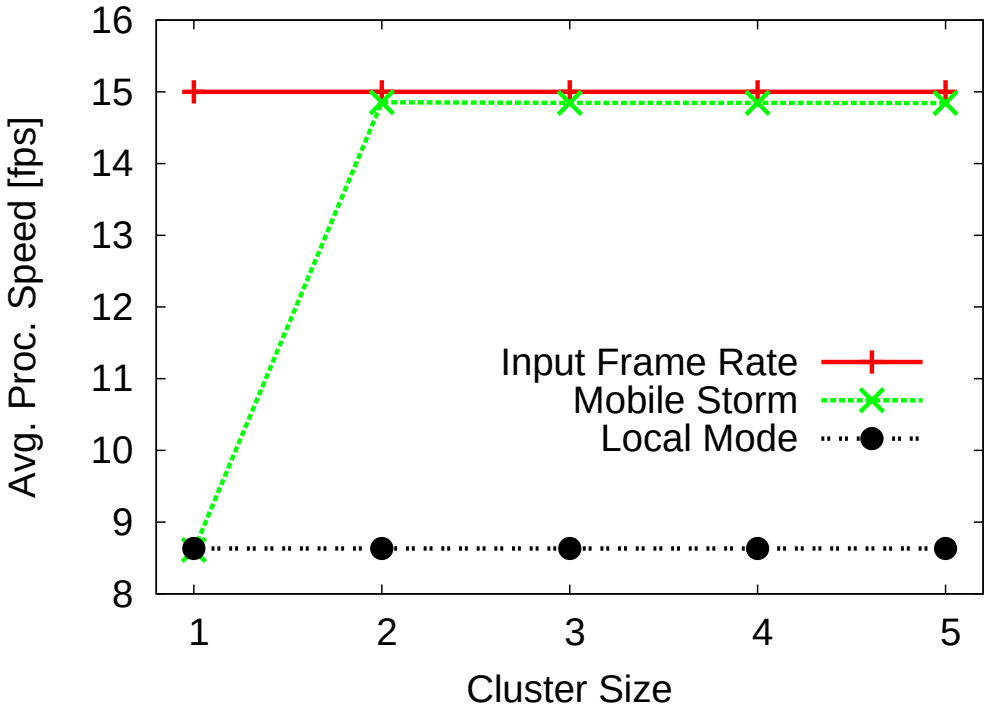

Figure 4.2: Processing speed of Mobile Storm and Local Mode for a video stream at 1280×720 resolution

4 nodes is sufficient (from Figure 4.1). The linear improvement of processing speed makes Mobile Storm highly scalable.

Figure 4.4 shows that the frame transfer speed is fast enough to transfer generated video frames in a real-time manner. We also find that the increase in the cluster's size slows down the transfer speed slightly, due to the overhead introduced by the added Worker Nodes.

4.2 Effects of Video Stream Frame Rate on Processing Speed

To understand the computation capability of Mobile Storm, we purposely increase the video frame rate to overload the system. The results are shown in Figure 4.6, 4.7, 4.8, 4.9 and 4.10. As shown, as the input frame rate increases, Mobile Storm eventually reaches a bottleneck (time when its processing speed reaches the

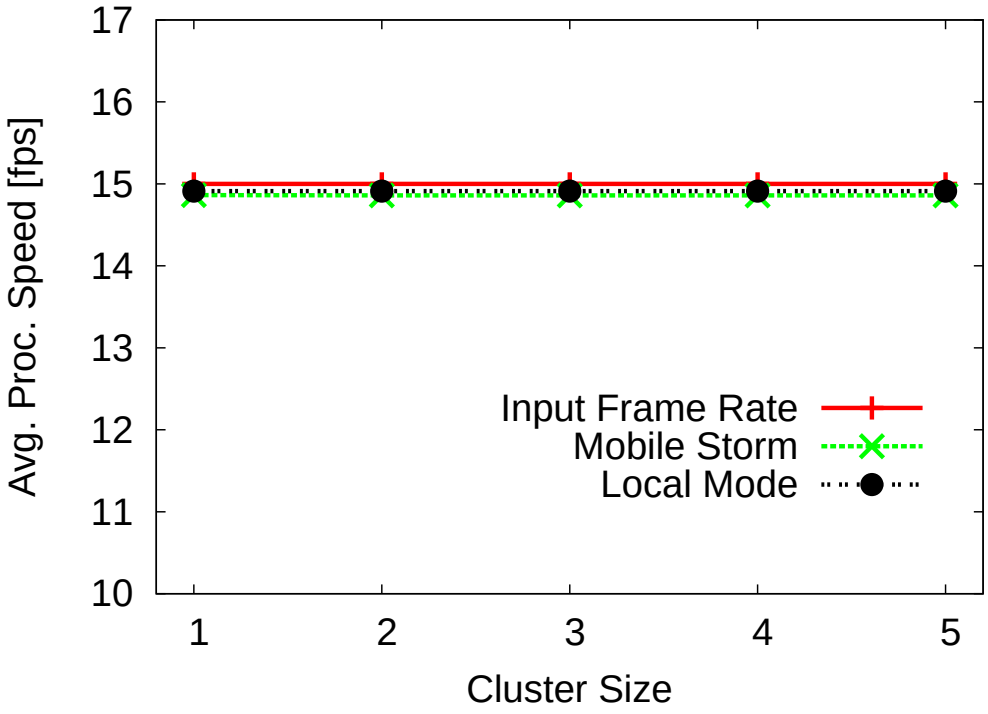

Figure 4.3: Processing speed of Mobile Storm and Local Mode for a video stream at 800×600 resolution

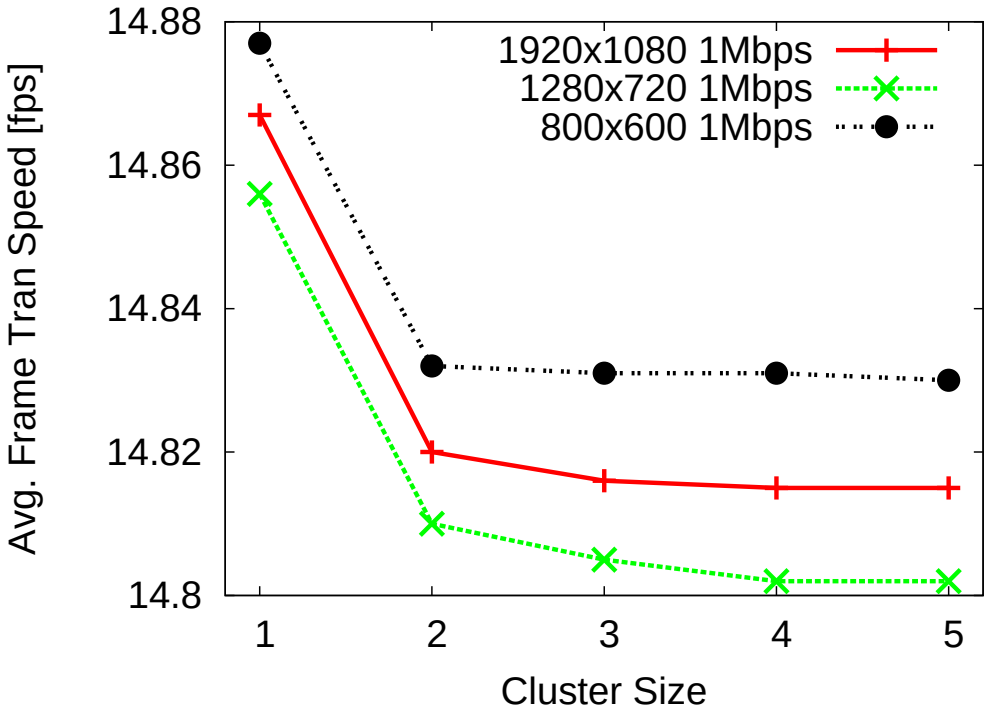

Figure 4.4: Frame transfer speed

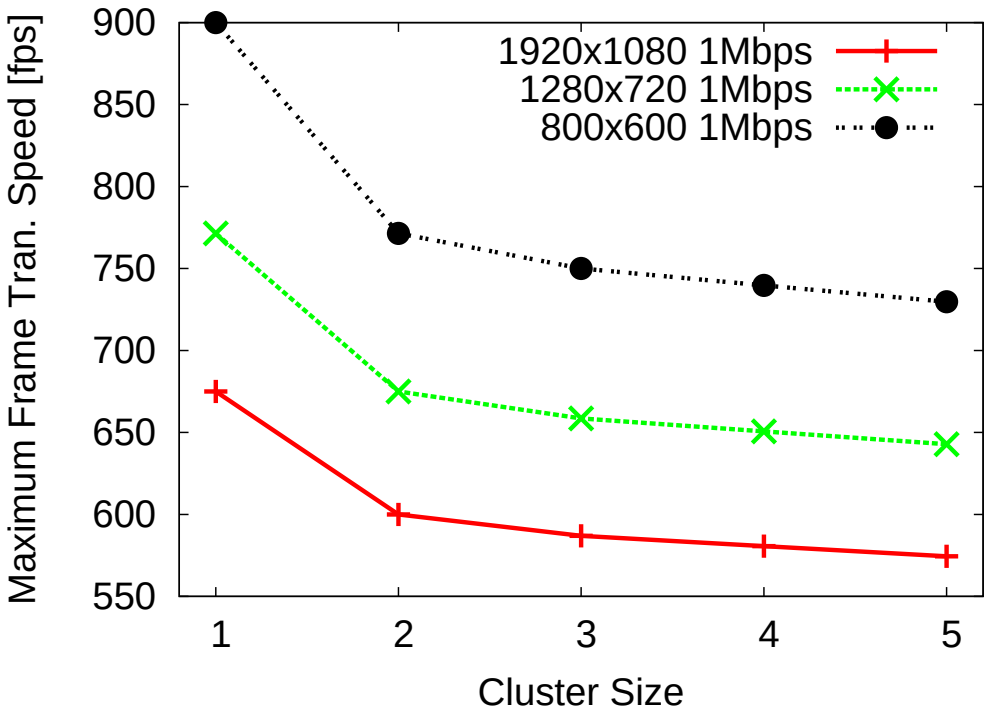

Figure 4.5: The maximum frame transfer speed

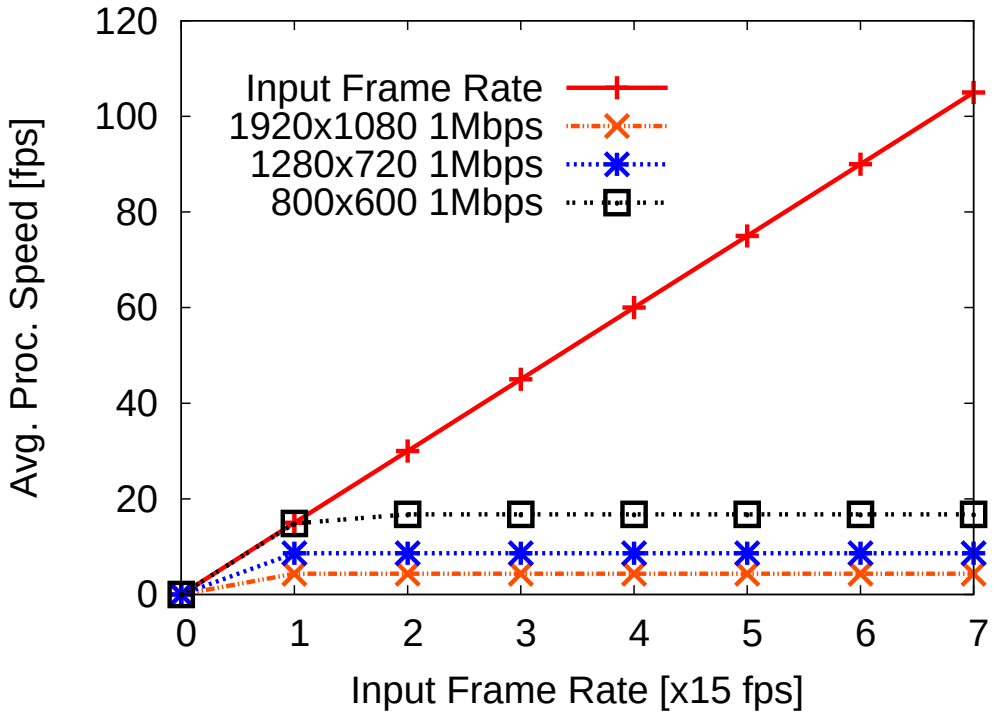

Figure 4.6: Processing speed change while increasing the input frame rate when cluster size=1

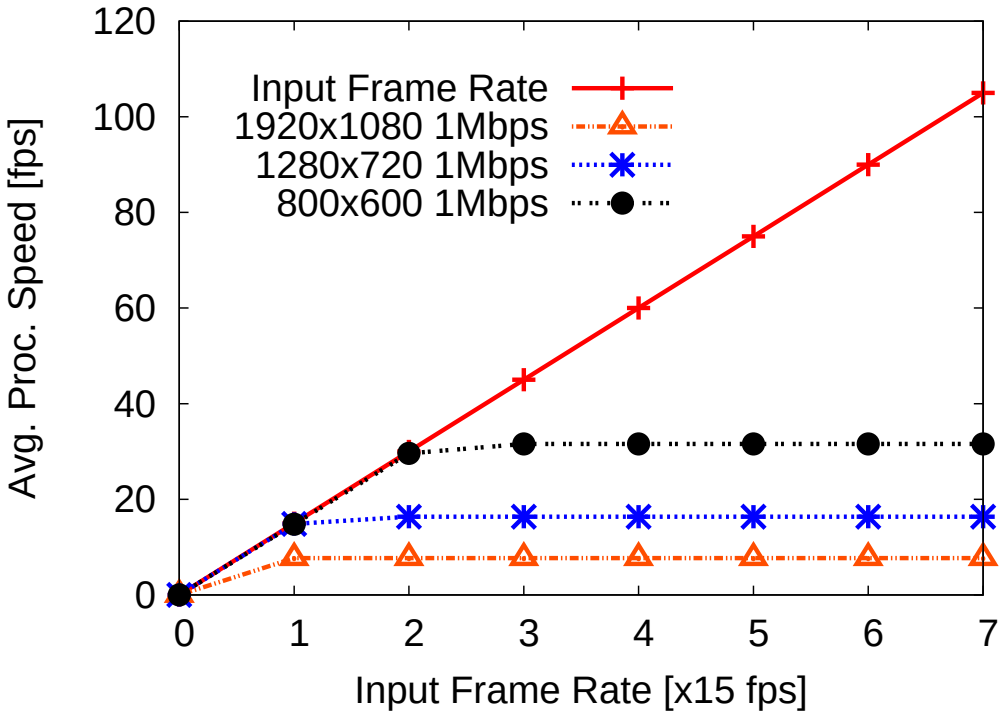

Figure 4.7: Processing speed change while increasing the input frame rate when cluster size=2

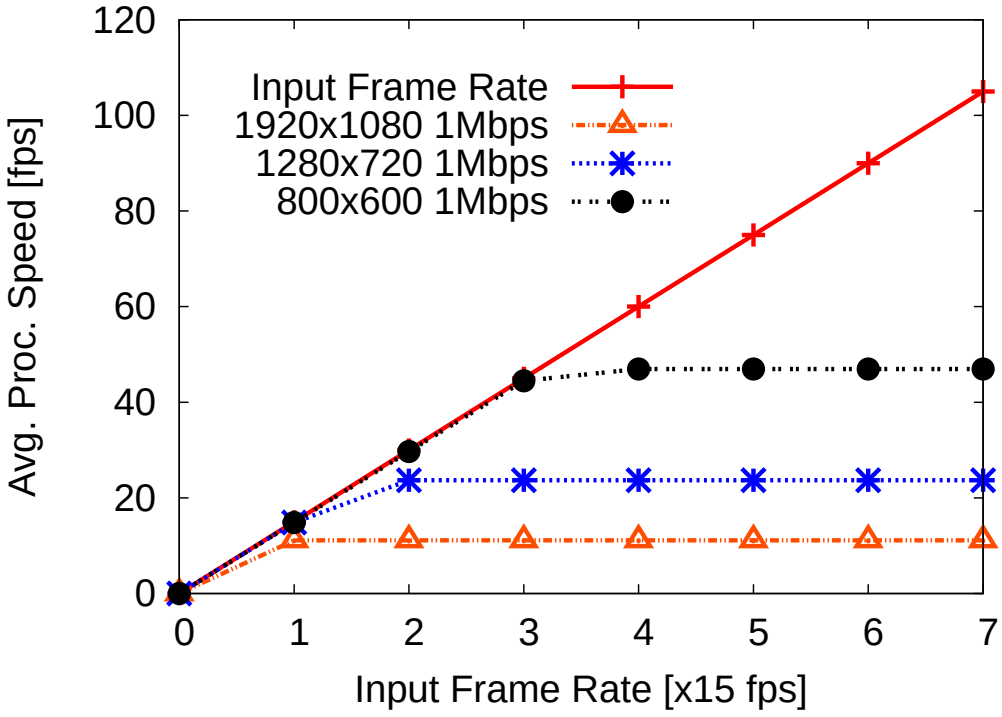

Figure 4.8: Processing speed change while increasing the input frame rate when cluster size=3

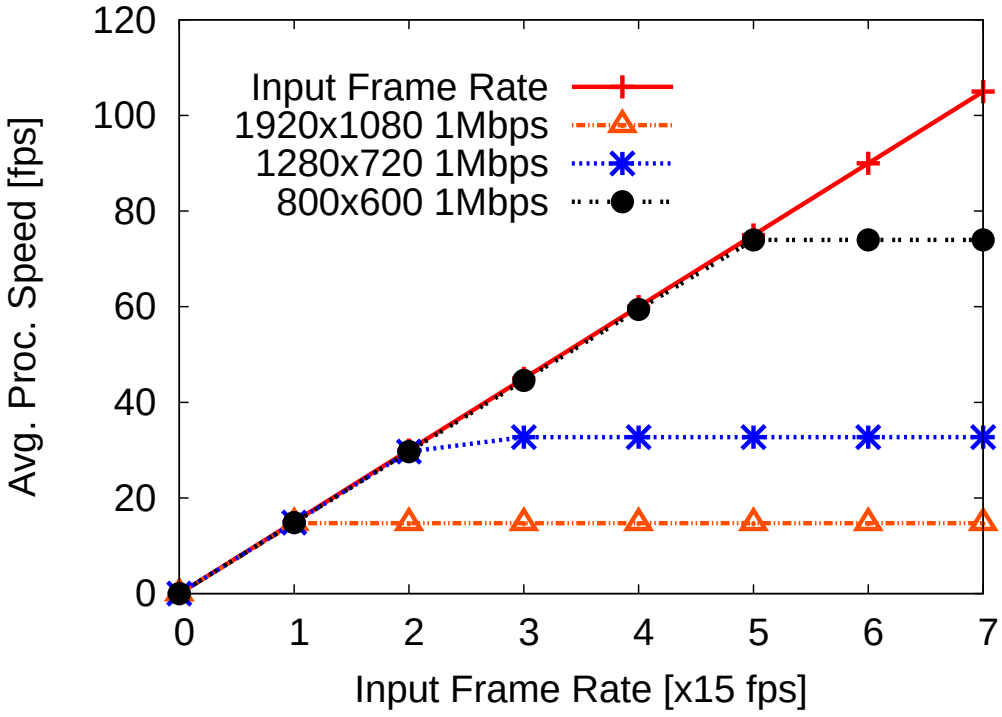

Figure 4.9: Processing speed change while increasing the input frame rate when cluster size=4

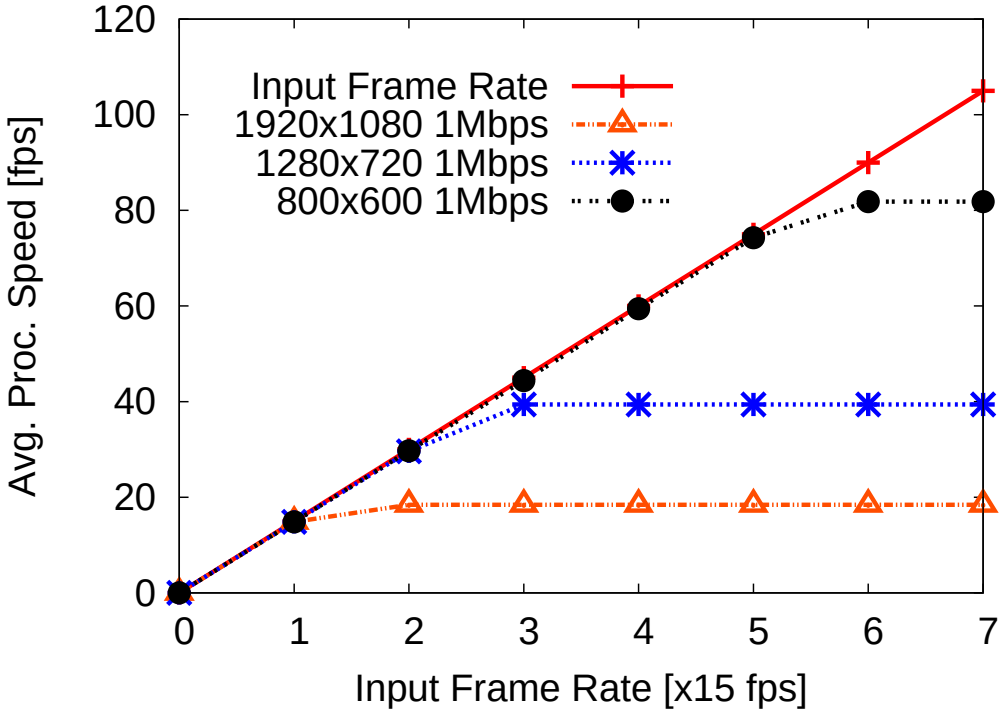

Figure 4.10: Processing speed change while increasing the input frame rate when cluster size=5

maximum). From our experiment, a Mobile Storm Cluster of 5 nodes is capable to handle low, medium, and high resolution video stream at 82fps, 39fps, and 19fps respectively. In contrast, a Mobile Storm Cluster of just 1 node can only handle low resolution video stream at 17fps. Neither high resolution, nor medium resolution video streams can be processed on this 1 node Cluster.

4.3 Network Throughput

We also measure the network throughput by overloading the system. Figure 4.5 indicates that the maximum frame transfer speed for low, medium and high resolution video streams are 900fps, 771fps and 675fps, respectively. The increase of cluster size leads to more communication overhead, which reduces the maximum frame transfer speed.

5. CONCLUSIONS

This paper presents the design, implementation and evaluation of Mobile Storm. It is the first distributed real-time stream processing system for mobile cloud. Without offloading computation to remote servers, Mobile Storm processes real-time streaming data using a cluster of mobile devices in a local network. We implemented Mobile Storm on Android phones and developed a video stream processing application to evaluate its performance. The evaluation results show that Mobile Storm is capable of handling video streams of various frame rates and resolutions in real-time.

The future development in our roadmap is to further optimize energy efficiency and design a dynamic scheduler that accounts for processing capability, communication capability, and battery level of each individual mobile device.

REFERENCES

[1] Android mediacodec. http://developer.android.com/reference/android/media.

[2] Android service. http://developer.android.com/guide/components/services.html.

[3] Apache hadoop. http://hadoop.apache.org/.

[4] Apache storm. http://storm.apache.org/.

[5] Dalvik virtual machine. http://en.wikipedia.org/ wiki/Dalvik_(software).

[6] Group of pictures. http://en.wikipedia.org/ wiki/Group_of_pictures.

[7] Json. http://www.json.org/.

[8] Netty project. https://github.com/netty/netty.

[9] Snapdragon sdk. https://developer.qualcomm.com/mobile-development/add-advanced-features/snapdragon-sdk-android.

[10] Spark streaming. https://spark.apache.org/streaming/.

[11] Storm grouping methods. https://storm.apache.org/ documentation/Concepts.html.

[12] Zookeeper 3.4.6 api. http://zookeeper.apache.org/ doc/r3.4.6/api/l.

[13] Zookeper project. http://zookeeper.apache.org/.

[14] Paramvir Bahl, Richard Y Han, Li Erran Li, and Mahadev Satyanarayanan. Advancing the state of mobile cloud computing. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, pages 21–28. ACM, 2012.

[15] Hsing-Yu Chen, Yue-Hsun Lin, and Chen-Mou Cheng. Coca: Computation offload to clouds using aop. In *CCGrid*, 2012.

[16] Byung-Gon Chun and Petros Maniatis. Dynamically partitioning applications between weak devices and clouds. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, page 7. ACM, 2010.

[17] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *MobiSys*, 2010.

[18] Heungsik Eom, Pierre St Juste, Renato Figueiredo, Omesh Tickoo, Ramesh Illikkal, and Ravishankar Iyer. Snarf: a social networking-inspired accelerator remoting framework. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 29–34. ACM, 2012.

[19] Ioana Giurgiu, Oriana Riva, Dejan Juric, Ivan Krivulev, and Gustavo Alonso. Calling the cloud: enabling mobile phones as interfaces to cloud applications. In *Middleware*. 2009.

[20] Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *MobiSys*, 2012.

[21] Gonzalo Huerta-Canepa and Dongman Lee. A virtual cloud computing provider for mobile devices. In *WMC&S: Social Networks and Beyond*, 2010.

[22] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4), 2010.

[23] Eugene E Marinelli. Hyrax: cloud computing on mobile devices using mapreduce. Technical report, DTIC Document, 2009.

[24] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *ICDMW*, 2010.

[25] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, 2009.

[26] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *MOD*, 2014.

[27] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[28] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *HotCloud*, 2010.

[29] Xinwen Zhang, Anugeetha Kunjithapatham, Sangoh Jeong, and Simon Gibbs. Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing. *Mobile Networks and Applications*, 16(3), 2011.