Job2P: Job Scheduling APIs for Wi-Fi Peer-to-peer Mobile Network

Le Dinh Minh
Utah State University
minh.le@aggiemail.usu.edu

Young-Woo Kwon
Utah State University
young.kwon@usu.edu

***Abstract*—These days, along with the rapid revolution of mobile device industry, software is also being built heavier and consumes more CPU performance and energy than they were in the past few years. Since the modern mobile devices support multiple connection methods like Wi-Fi, Bluetooth or NFC, the requirements of sharing the workloads and resources among the devices within a network to reduce full workload on an arbitrary device are being considered, especially in the areas Internet is not available, which can be found anywhere.**

To address those limitations on mobile devices, we proposed Job2P as the Job Scheduling APIs for Android development that leverages Wi-Fi Direct to support sharing workloads and resources over peer-to-peer mobile device network, which doesn't require any Internet connections. Job2P provides a simple and straightforward API interface to get rid of sophistication of network implementation, letting developers easily create their distributed mobile applications with capability of forming closed range network. In term of workload distribution, Job2P splits task and resource into parts and packs them into the smaller units called jobs and dispatch to the peers. To distributed jobs equitably among the peers, a Decision Maker is added to decide the amount of resource the peer has to handle bases on its percentage of availability. Moreover, our APIs can handle fault tolerant for network malfunctions.

Through our experiments with image processing, text processing and GPS location, we realized that Job2P improves significant productivity in term of performance and energy consumption.

I. Introduction

These days, mobile phones have been playing a very important role in human society. People need their cellphone everyday for surfing Internet, searching for information, online shopping, communicating with friends, taking and sharing photos etc. As a result of highly intensive completions between the global mobile phone manufactures like Samsung and Apple, the cellphones are more and more equipped themselves with high specifications (CPU, RAM and built-in storage) and top functionality so that they are powerful enough in compare with those in the past or even with a desktop.

With the hardware rapid development, software is also being built more and more sophisticated in many aspects. They may occupy more space on local storage, use more memory, and thus consume more energy. Even though the software is designed to run on mobile device, its size may vary up to 1GB (like image processing or image recognizing) and memory consumption can be up to 1GB as well. Since there are possibly many heavy workloads running on a device at some points, the needs of sharing workloads among the mobile devices is really necessary with respect to performance and energy efficiency.

Distributed computing are so far well-developed in PC or embedded networks, bringing the synthesized power of computation from multiple computers to solve a problem. Up to now, there are thousands of solutions for the developers to create a distributed computing system. For instance, the emergency of a number of message-oriented middleware like DataTurbine [1], RabbitMQ [2] or NaradaBrokering [3] lessened the effort of development for distributed applications to the minimum but still maintained all equipped functionality like data mirroring, dynamic network topology or fault tolerant. However, despite of stunning equipments, it is revealed that the most disadvantage of this kind of applications is the dependence of network connection or wireless access point. Without a network established, nothing happens.

Unlike the other PCs or wired devices, the mobile devices have their own advantage of multiple non-equivalent network capability. One of the remarkable network capability is installed on modern mobile devices are Wi-Fi Direct, which allows them to discover the others in any short distance less than 200 meters without utilizing Internet and wireless access point. No internet connections are required, and it will help the owner to connect to the devices within a closed distance. By establishing connection between the two devices to form a pair, Wi-Fi Direct can provide the simple way to dynamically initiate a peer-to-peer network. Available on Android devices from version 4.0 (which more than 96% of devices are using these days), as well as a number of Intel-featured laptops and game consoles, there is the high possibility of discovering the other mobile devices at anywhere.

To address those limitations with heavy software on mobile devices, we proposed Job2P as the Job Scheduling APIs for Android development that leverages Wi-Fi Direct to support sharing workloads and resources over peer-to-peer mobile device network, which doesn't require any Internet connections. Job2P provides a simple and straightforward API interface to get rid of sophistication of network implementation, letting developers easily create their distributed mobile applications with capability of forming closed range network. In term of workload distribution, Job2P splits task and resource into the smaller units called jobs, and dispatch to the peers. To distributed jobs equitably among the peers, a decision making module is added to decide the amount of resource the peer has

to handle bases on its percentage of availability. Moreover, our APIs can handle fault tolerant for network malfunctions.

Based on our experiments and results, our contributions are as follow

- **Simple programming model** a library for developers facilitates establishing mobile peer-to-peer network.
- **Job Scheduler** for scheduling job execution and resource heterogeneity adaptation
- **Runtime system** determines the number of devices with according appropriate jobs for the remote execution bases on applied algorithm on level of availability.
- **Systematic evaluation** our test case experiments will give the proof of APIs productivity

II. Motivation and Technical Background

A. Motivation

B. Technical Background

1) Machine-to-Machine Communication:

2) Remote Execution:

III. Proposed Approach: *Job2P*

A. Approach Overview

To retrieve the goals, as well as making our APIs widely adaptive to different context and usages, our consistent idea of design is hiding the complexity of system implementation and open to the developers the capability of customization. This below figure describes the internal architecture of a typical application utilizing our APIs to form a distributed mobile peer-to-peer system (Figure 1). Basically, our system comprises of two main components:

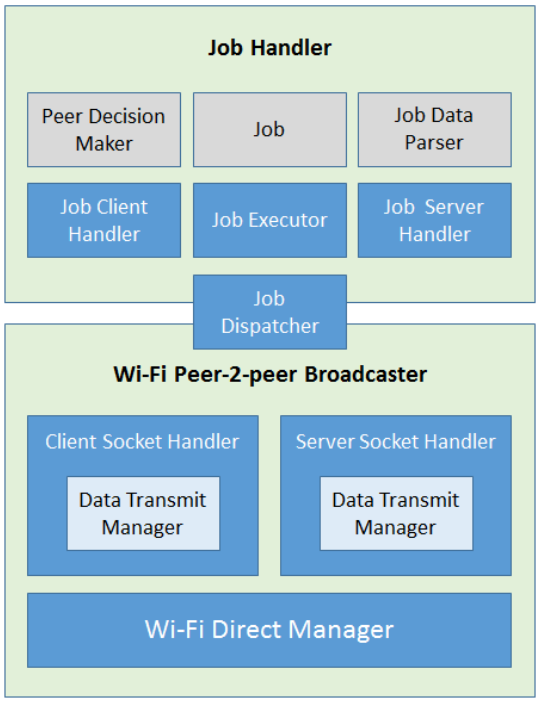

Figure 1: Job2P Architecture

- **Wi-Fi Peer-to-peer Broadcaster** (`WPB`) wraps up Wi-Fi Direct implementation, socket management and connection establishment. Moreover the `WiFiBroadcaster` provides a simplified API interface to form peer-to-peer network.
- **Job Handler** in charge of making decision on selecting available peers, determining and splitting the appropriate amount of jobs from the assigned task. The `JobDispatcher` sub module dispatches job and `JobExecutor` perform job execution at a single peer. Besides, `JobHandler` merges the scattered incoming results into the final placeholder and relay information messages to the main application.

1) Wi-Fi Peer-to-peer Broadcaster: To easily form up multiple pair connections between devices in the network, we utilized Wi-Fi Direct, the new feature available from Android APIs 4.0. We developed the `WiFiDirectManager` sub module inside the `WPB` to wrap up the complexity of Wi-Fi Direct implementation. Holding an instance of `WifiP2pManager`, `WiFiDirectManager` maintains connectivity over peer-to-peer network, discover available peers and setup the connections.

When the app starts up, `WPB` from a peer will spread acknowledgments by calling `discoverPeers()` to inform its availability to the other peers. If a peer receives such message, its `WPB` module will collect information from the new comer and update the list of devices. When all the peers are updated with their new device list, the network will be reformed. See figure 2

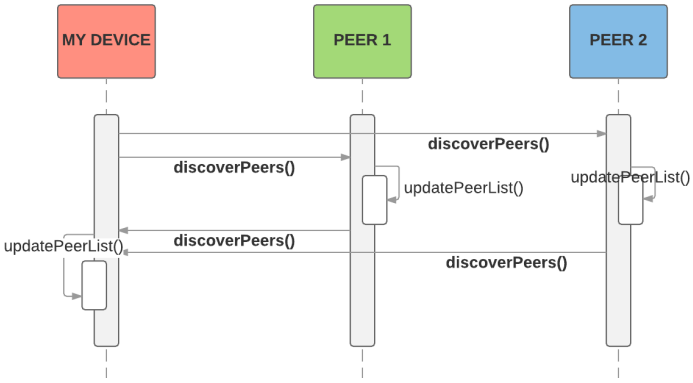

Figure 2: Forming peer-to-peer network

When peers have information of the others, they are ready to connect. We provide function `connectToADevice()` holding `WifiP2pDevice` as one input parameter to invoke the handshake with another device. Once connection established, the device actively invoked will be assigned as client, which utilize the `ClientSocketHandler` to listen to a client socket. The device passively received the invocation will be promoted as server and use the `ServerSocketHandler` to initiate server socket with an random port. If the invoked device is already a server, it will maintain its current state and continuously accept the new client. Moreover, since a device

can either be a server to some peers, or serve as client to the others, this method is able to build up a close range peer-to-peer network.

Both `ClientSocketHandler` and `ServerSocketHandler` are constituted by a sub module named `DataTransmitManager`, which periodically observes input and output streams of parent socket (Can be either server or client socket) to verify checksum, check data consistency, send/receive data from the socket. Inheriting the `TransmitManager` abstract class, developer can instantiate their own version of `DataTransmitManager` to customize the data consistency checker or improve security.

2) Job Handler: `Job` is defined BLAH BLAH BLAH `DataParser` is a loose coupling interface of the Job component facilitating adaptation with data heterogeneity. It helps determining mechanisms for: (1) serialization/deserialization object into binary format which is compatible for TCP transmission, (2) task partitioning, and (3) placeholder initialization and concatenation. To make the data type comprehensive to system in terms of reading, parsing or memory allocating, developer needs to supply instructions by instantiating an implementation of Data Parser for that type.

B. Programming Model of Job2P

1) Job Definition: To avoid any malfunctions and misconfiguration at remote execution, a conventional Job interface is defined like the code snipe below (1)

Listing 1: Job Definition

```
1  public class Job {
2    public Object exec(Object partialObject) {
3      // job implementation
4    }
5  }
```

Where the body of `exec()` will contain concrete implementation of the Job.

Since Dalvik VM didn't support loading class the way Java VM does. Instead it will load Dalvik execution ("dex") files from an alternative locations such as internal storage or network. We provide `DexCreator` tool, the windows commandline application supports compiling the `Job` java file into the dex job package (a jar file). Dex package and sliced resources in binary format will be added into one `JobData` object by `JobDispatcher`, signed with checksum for consistency and dispatched to the other peers.

When client peer receives an object of `JobData` sent from server, it will firstly check checksum to confirm the consistency, then deserialize it into `Job` and resources and execute the job with that data.

2) Data Parser: The code snipe 2 shows the main functionality of the `DataParser` interface that user needs to comply. Under the first released version, we also support 3 basic Data Parsers for Image, Text and GPS.

Listing 2: `DataParser` interface

```
1  public interface DataParser {
2    ...
3
4    public Class getDataClass();
5
6    public byte[] parseObjectToBytes(dataObject);
7
8    public Object parseBytesToObject(byteArray);
9
10   public Object getSinglePart(..., numOfParts,
11                firstOffset, lastOffset);
12
13   public Object createPlaceholder(jsonMetadata);
14
15   public Object copyPartToPlaceholder(...
16               partDataObject, partIndex);
17
18   public void destroy(dataObject);
19
20 }
```

To clarify usage of `DataParser`, we will go through several important abstract methods.

- `getDataClass()` return data type.
- `parseObjectToBytes(object)` instructs system to serialize an object to binary array. In Android, not every object can be serialized, for instance Bitmap.
- `parseBytesToObject(byte[])` deserialize a binary array back to object.
- `getSinglePart()` return a data slide bases on number of parts (`numOfParts`) and its `index`. Code snipe 5 shows an example getting one slice from a bitmap by `index`. When app runs, `JobDispatcher` will use this method to partition the task into slices.
- `copyPartToPlaceholder()` Once partial result is retrieved from a distant peer, this function will instruct application to merge it to the placeholder.

C. Job Scheduling and Decision Maker

In peer-to-peer network supported by our APIs, the server will distribute jobs to peers bases on investigating their availability by `DecisionMaker` (Figure 3). At the early stage before the transmission, the `DecisionMaker` of the calling device sends inquiring request for status (`IRS`) to some of selected peers in the network. If one receives this IRS request, its `DecisionMaker` will estimate its capability of response using the measurements of CPU, memory and battery usage at that time.

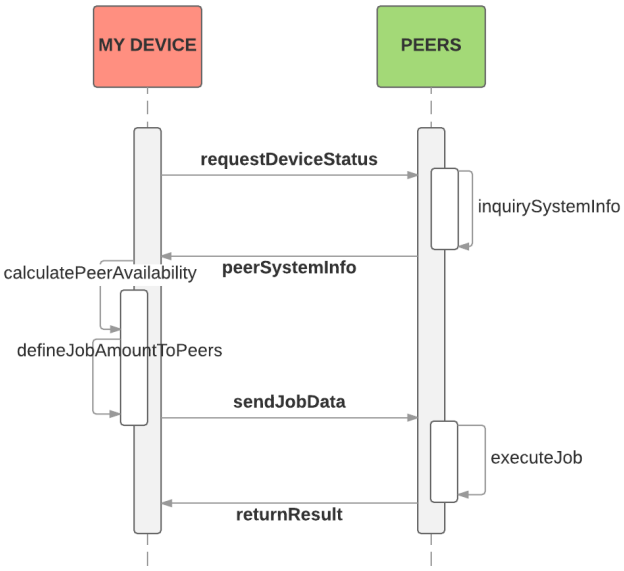

Figure 3: Decision making workflow

1) Detect feasible peers: A mobile device at any time maintains a certain number of connection with the others. Before delivering jobs, it dispatches IRS messages to all the peers for acknowledgment of their availability (RL - Level of Responsibility). If a peer receives an IRS, its `DecisionMaker` will capture its configuration and corresponding usage states to generate a response in JSON format, The code snipe 3 shows such a typical response.

Listing 3: JSON response from peer

```
{
  "device" : "LG-Volt",
  "RL" : 24.83,
  "availability" : "low",
  "network" : "low",
  "gps" : "on",
  "cpu" : {
    "usage" : "0.3",
    "speed" : "1.3",
    "cores" : 4
  },
  "memory" : {
    "usage" : "0.5",
    "total" : 2
  },
  "battery" : {
    "usage" : 0.85,
    "total" : 2800
  }
}
```

Where RL is peer's Level of Responsibility (for estimation of RL as well as the other usage parameters, see III-C2), `availability` indicates whether the peer is feasible for handling job. `network` gives the network status, which can be either `high`, `low` or `off` (no connections), `gps` is information of GPS `on` or `off`. Finally `cpu`, `memory` and `battery` are statuses of the essential resources at the time of response.

In any IRS response, parameter `availability` can be assigned with value of `low`, `medium` or `high`. The `low` availability will inform the calling peer that it is not available due to temporary resource limitation, therefore eliminate the caller from sending jobs to. Only the peers with `availability` higher than `medium` will accept jobs for execution. The decision is given bases on the thresholds of battery usage which are set by default at value higher than 0.8 (or 80% overall usage) for the `low`, 0.5 to 0.8 for the `medium` and lower than 0.5 for the `high`. User can override those thresholds by updating the `availability-thresholds` in the configuration file before app initialization.

In the above example (code snipe 3), the battery usage was 0.85 or 15% remained, so the `DecisionMaker` on it stated that its `availability` is `low`, in other words it is not able to handle any jobs and should be excluded from the dispatched list.

Algorithm 1 Select Available Peers Algorithm

```
1:  function SELECTPEERS()
2:      Send IRS requests to all peers
3:      P_AV ← HashMap(DeviceId, P)
4:      for all Resp_IRS in {Incoming IRS Responses} do
5:          if Resp_IRS[availability] = low then
6:              continue;
7:          else
8:              P_Info ← P2pDevice[DeviceId] + Resp_IRS[RL]
9:              P_AV[DeviceId] ← P_Info
10:         end if
11:     end for
12:     return P_AV
13: end function
```

The algorithm 1 explains how the caller select the suitable peers from its connection list. Where P_{AV} stands for map of available peers and $P2pDevice$ is the device information list.

2) Job quantitation for peers: The calling peer quantitates the jobs from a task to distribute to the other available devices in the network suitably. To achieve this, the RL parameter will be used as the main coefficient for the split.

To calculate RL as well as the other parameters in IRS, retrieved from `/proc/stat` system file [5], the percentage of CPU usage is expressed by this following expression in two short consecutive times

$$Usage_{CPU} = \frac{(\sum T_{CPU2} - T_{Idle2}) - (\sum T_{CPU1} - T_{Idle1})}{(\sum T_{CPU2} - \sum T_{CPU1})}$$

When $\sum T_{CPU}$ is total time of running CPU and T_{Idle} is idle time correspondingly in hertz. In term of memory usage, the $Usage_{Mem}$ can be determined by using `MemoryInfo` from Android API to retrieve Mem_{Avail} and Mem_{Total}, so the $Usage_{Battery}$.

Since in the mobile device, the lower resource usage state states the higher availability, and the higher specifications represents the better responsibility. Then the level of responsibility of device can be simply summarized by the below expression

$$RL = \frac{CPU_{Spec}}{Usage_{CPU}} + \frac{Mem_{Spec}}{Usage_{Mem}} + \frac{Battery_{Spec}}{Usage_{Battery \times 1000}}$$

Where CPU_{Spec} has GHz unit, Mem_{Spec} has GB unit and $Battery_{Spec}$ has uAh unit. Especially CPU_{Spec} is determined by number of its core. For example, a quad-core CPU at speed of 1.3GHz can be counted as 1.3×4 or 5.2. At a certain time of that device, if $Usage_{CPU}$ is 0.3, $Usage_{Mem}$ is 0.5 (half of 1GB memory consumed), $Usage_{Battery}$ is 0.7 or 70% used over a 2800uAh capacity battery, the value of RL will be

$$RL = \frac{5.2}{0.3} + \frac{1}{0.5} + \frac{2800}{0.7 \times 1000} = 23.33$$

To reduce the latency and to avoid miscalculation at the calling peer, RL is prematurely calculated by each available called peer and wrapped up in the IRS response sending back to the caller.

In a peer-to-peer network comprising of n devices, where i-device has responsibility level RL_i, the `DecisionMaker` will assign the job with carrying amount of data (M_i) which is equivalent to

$$M_i = M \frac{RL_i}{\sum_{j=\overline{1,n}} RL_j}$$

Where M is total size of data in bytes. This below algorithm 2 will represent the procedure that `DecisionMaker` judge the peer capability to assign the appropriate job. Where M_i is quantity of job in binary length, $firstOffset$ and $lastOffset$ indicate the location of the data trunk in the whole.

Algorithm 2 Assign Job Algorithm

```
1:  function ASSIGNJOBS()
2:      M ← getDataSize()
3:      RL_Total ← 0
4:      for P in {P_AV} do
5:          RL_Total ← RL_Total + P[RL]
6:      end for
7:
8:      firstOffset, lastOffset ← 0
9:      jobData ← Null
10:     job ← readJobFile()
11:     M_C ← 0
12:
13:     for i = 1 to P_AV.length do
14:
15:         M_i ← M (RL_i / RL_Total)
16:
17:         firstOffset ← M_C
18:         lastOffset ← M_C + M_i
19:         jobData ← DataParser.getSinglePart(
20:                                 firstOffset, lastOffset)
21:         dispatchJob({job, jobData})
22:
23:         M_C ← lastOffset
24:     end for
25: end function
```

D. Estimating Energy Consumption in WiFi Environment

Assume that we have a Wi-Fi peer-to-peer network available with n devices, each device at a certain time has level of responsibility $RL_i(i = \overline{1,n})$. According to the section III-C, if E is the energy consumed by the application for only completing the task regardless of other ambiances, the total energy E_0 will be

$$E_0 = E + E_w$$

Where E_w is energy the app requires for waiting. Also,

$$E_{p2p} = E(\frac{RL_0}{\sum_{i=1}^{n} RL_i}) + E_{WiFi} + E_w$$

Where E_{WiFi} is energy consumed by Wi-Fi for sending jobs to other peers.

In this estimation we skipped considering E_w since it will depend on appearance of applications. If application have no GUI like system background services, E_w will cause very little effect. From the two above equations, we can get the difference energy consumption between the two job processing mechanisms E_{Diff}

$$E_{Diff} = E_0 - E_{p2p}$$

or

$$E_{Diff} = E(1 - \frac{RL_0}{\sum_{i=1}^{n} RL_i}) - E_{WiFi}$$

According to [6], Wi-Fi caused battery drained linearly by time during the transmission, particularly the drain can be represented by $y = 17.01x - 0.93$ for downloading and $y = 17.31x - 2.28$ for uploading. Therefore, it is inferred that in a mobile system with a certain number of devices in different levels of responsibility, if E is big enough, in other words, if the task to perform is big enough, then $E_{Diff} > 0$ will happen, thus deploying a peer-to-peer cluster will give the great benefit in term of energy efficiency. The bigger value of E_{Diff}, the more benefit we will archive.

E. API Usage Scenario

The library should be simple, so that developer can integrate within just a few steps. This section will describe step-by-step the way to utilize our API to enable a typical peer-to-peer network.

1) Defining a Job: Job implementation is the prerequisite work to determine what to execute on the remote device and how to cast and manipulate data from the abstract object. According to section **??**, developer needs to implement the Job class file separately by overriding the `exec()` method and cast the input abstract parameter to the concrete. Then run the `DexCreator` tool on the Job class to compile and compact it into the DEX jar package. The final outcome is the `job.jar` file, it should be saved in the internal storage of the caller device. To simplify, we can store it in the `Download` folder.

2) Implementing an Application Using APIs: This sub section will describe 3 basic steps to implement an Android application to utilize our APIs

a) Message Handler: The Message Handler is required to instantiate at the beginning. In particular, an instance of `UIHandler` is initiated to receive messages from system when it goes into runtime. While system is in progress, the log and information messages will periodically be returned with label `MAIN_INFO` (Snipe code 4). When each job result comes, `JobHandler` will collect and handle by to partially put into a placeholder. When all results arrived, the placeholder with label `MAIN_JOB_DONE` will be returned to inform the completion of job collaboration, as well as bring the final result back to the main app UI.

Listing 4: UI handler

```
Handler mainUiHandler = new Handler() {
  @Override
  public void handleMessage(Message msg) {
    switch (msg.what) {
      case Utils.MAIN_JOB_DONE: {
        // when job is completely finished
      }
      case Utils.MAIN_INFO: {
        // to receive messages from the processor
      }
    }
  }
};
```

b) Data Parser: Secondly, developer needs to declare `DataParser` to determine data-type and parser to equip for manipulating data at run-time (see sub section **??**). `JobHandler` is the main component which wraps up the complexity , and exposes only the necessary functions like `discoverPeers()` and `dispatchJob()`. To send ACK messages to other peers for exchanging acknowledgments and reforming network, we need to call `discoverPeers()` function on the program, this work should be done as soon as application starts.

Listing 5: Example of `getSinglePart()` for Bitmap

```
@Override
public Object getSinglePart(data, partNum, idx){
  Bitmap bmpData = (Bitmap) data;
  int pWidth = bmpData.getWidth() / partNum;
  return Bitmap.createBitmap(bmpData,
              (pWidth * idx), 0, pWidth,
              bmpData.getHeight());
}
```

c) Discovery Peer and Dispatching Jobs: When network is formed and connections are held from some of the peers, `dispatchJob()` will be call to locate the resources and job which predefined in local storage, it then invokes `DecisionMaker` (sub section III-C) for job splitting and binary serialization. Finally jobs will be dispatched over the socket

Listing 6: Declare DataParser and JobHandler

```
// data parser: to determine datatype how to split the data
dataParser = new BitmapJobDataParser();
...
// handlers registration
jobHandler = new JobHandler(this, dataParser);
jobHandler.setSocketListener(
 new JobHandler.JobSocketListener() {
   @Override
   public void socketUpdated(... isConnected){
       // when socket is updated
   }
});
...
// update the device list
deviceList.setAdapter(
  jobHandler.getDeviceListAdapter());
...
// send ACK to other members to reconstruct network
jobHandler.discoverPeers();
...
// address resources and job to execute
String dataPath = downloadPath + "/mars.jpg";
String jobPath = downloadPath + "/Job.jar";
jobHandler.dispatchJob(dataPath, jobPath);
```

IV. Evaluation

A. Micro Benchmark

B. Case Study

To measure the performance of the system equipped with our APIs, we decorated a small testbed with collaboration of 5 different Android devices to perform our 3 test cases:

- **Image Processing** we will initiate the peer-to-peer network test to perform blurring a large scale image which is unable to process at any of our devices. Particularly, to process an image with size 4000×4000 and 4 bytes to express each pixel color, application must spare the amount of memory equivalent to 64MB which is too expensive for the system, occasionally this kind of image would be refused to load.
- **Text analysis** another proof of performance on text processing. Our APIs will yield the data parser to developer for data heterogeneous adaptation.
- **GPS** establishing a GPS connection is proof of high energy consuming. It would be impossible for a device with low battery to keep update with GPS frequently. We will base on Job2P to build a simple system so that one device can benefit GPS locations from the healthier devices.

C. Results

1) Performances: Diagrams go here

D. Discussion

V. Related Work

VI. Conclusions

The conclusion goes here.

ACKNOWLEDGMENT

The authors would like to thank...

REFERENCES

[1] Sameer Tilak, Paul Hubbard, Matt Miller, and Tony Fountain, *The Ring Buffer Network Bus (RBNB) DataTurbine Streaming Data Middleware for Environmental Observing Systems*, p125-133, e-Science and Grid Computing, Bangalore 2007.

[2] Maciej Rostanski, Krzysztof Grochla, Aleksander Seman, *Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ*, p879-884, FedCSIS 2014.

[3] Gadgil, H.; Fox, G.; Pallickara, S.; Pierce, M. *Managing grid messaging middleware*, Challenges of Large Applications in Distributed Environments, p83-91, 2006 IEEE

[4] Fred Chung, *Custom Class Loading in Dalvik*, http://android-developers.blogspot.com/2011/07/custom-class-loading-in-dalvik.html, July 2011

[5] */proc/stat Explained*, http://www.linuxhowtos.org/System/procstat.htm

[6] Kalic, G., Bojic, I., Kusek, M., *Energy consumption in android phones when using wireless communication technologies*, p754-759, MIPRO May 2012

[7] Marco de Sa, David A. Shamma, Elizabeth F. Churchill, *Live mobile collaboration for video production: design, guidelines, and requirements*, p693-707, Journal of Personal and Ubiquitous Computing, Volume 18 Issue 3, March 2014

[8] Cong Shi, Kaustubh Joshi, Rajesh K. Panta, Mostafa H. Ammar, Ellen W. Zegura, *CoAST: collaborative application-aware scheduling of last-mile cellular traffic*, p245-258, MobiSys 2014

[9] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong, *Rio: a system solution for sharing i/o between mobile systems*, p259-272, MobiSys 2014