ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/221410563

Misco: A MapReduce Framework for Mobile Systems

CONFERENCE PAPER · DECEMBER 2010

DOI: 10.1145/1839294.1839332 · Source: DBLP

CITATIONS
59

READS
220

5 AUTHORS, INCLUDING:

Vana Kalogeraki
Athens University of Economics and Business
154 PUBLICATIONS **2,582** CITATIONS
SEE PROFILE

Dimitrios Gunopulos
National and Kapodistrian University of Athens
225 PUBLICATIONS **10,269** CITATIONS
SEE PROFILE

Taneli Mielikäinen
Nokia Siemens Networks
62 PUBLICATIONS **1,149** CITATIONS
SEE PROFILE

All in-text references underlined in blue are linked to publications on ResearchGate, letting you access and read them immediately.

Available from: Vana Kalogeraki
Retrieved on: 28 March 2016

Misco: A MapReduce Framework for Mobile Systems*

Adam Dou
Dept. of Computer Science
University of California, Riverside
jdou@cs.ucr.edu

Vana Kalogeraki
Dept. of Informatics
Athens University of Economics and Business
vana@aueb.gr

Dimitrios Gunopulos
Dept. of Informatics
University of Athens
dg@di.uoa.gr

Taneli Mielikainen and Ville H. Tuulos
Nokia Research Center
Palo Alto, CA
{taneli.mielikainen, ville.h.tuulos}@nokia.com

ABSTRACT

The proliferation of increasingly powerful and ubiquitous mobile devices has provided a new and powerful sensing and computational framework. Software development and application deployment in such distributed mobile settings is especially challenging due to issues of failures, concurrency and lack of easy programming models. In this paper, we address this problem by presenting a framework which provides a powerful software abstraction that hides many of the complexities from the application developer. We design and implement a mobile MapReduce framework targeted at any device which supports Python and network connectivity. We have implemented our system on a testbed of Nokia N95 8GB smartphones and demonstrate the feasibility and performance of our approach.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Real-time systems and embedded systems; D.4.7 [**Operating Systems**]: Distributed systems

Keywords

mobile systems, map reduce, distributed real-time systems

*This research has been supported by the European Union through the Marie-Curie RTD (IRG-231038) Project, the SemsorGrid4Env Project and the MODAP Project, the NSF CNS-0627191 Award and a gift from Nokia.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. PETRA'10, June 23 - 25, 2010, Samos, Greece. Copyright Âl' 2010 ACM ISBN 978-1-4503-0071-1/10/06... $10.00

1. INTRODUCTION

In recent years, there is growing demand for applications that execute on powerful, programmable mobile devices and need to operate under timing constraints. Two are the major trends for this demand: First, mobile phones, smart-phones and PDAs have become increasingly powerful and their computation power, local storage, network capacity and other resources are continuing to evolve. Second, these technologies are becoming increasingly popular as they enable users to stay connected with each other at increasing levels of quality. With over 4 billion active cell phone subscriptions worldwide, they are becoming a significant platform for distributed applications.

The expanded capabilities of the recent mobile devices have strengthen the need for and the development of programming constructs that simplify the programmability and deployment of the applications on the mobile devices. Current practices include the development of platform-specific languages and run-time systems such as nesC [5] and TinyOS [9] for sensor systems or open programming platforms such as Android [7]. In all such developments, the main goal is to provide a simple way to facilitate the programming of distributed applications on the embedded devices.

Recently, the MapReduce framework[4] has been introduced, to provide a highly scalable, and distributed computation programming environment. The MapReduce framework has been applied successfully by many large and prominent companies such as Google, Yahoo, IBM, Amazon and Facebook for a large variety of problems including document clustering, machine learning and machine translation. Many different implementations have emerged for both traditional cluster environments [3] [17] as well as for specialized environments [8] [15]. The MapReduce programming model supports the weak connectivity model of computations across open networks, such as mobile networks, which makes it very appropriate to use in a mobile setting.

Application development is a challenging problem in mobile environments due to the complex nature of the distributed systems. Issues like concurrency, resource allocation, software development platform and device failures must all be taken into consideration.

In this paper, we address the problem of developing applications in distributed and mobile settings. We propose `Misco`, a MapReduce framework targeted at cell phones, mobile devices and personal commodity machines. Our goal is to provide a powerful programming abstract to allow software development without the need to deal with the underlying problems of distributed computing. We have implemented our system on a testbed of Nokia's third generation NSeries phones [13]. Our experimental results demonstrate the feasibility and performance of our approach.

The rest of the paper is organized as follows: We highlight some applications that can be implemented over our system in section 2. We discuss some related and background work in section 3. Then we discuss the design of our system in section 4, and our implementation details in section 5. We then show our experimental results in section 6 and conclude in section 7.

2. APPLICATIONS

Here, we highlight some applications which can be easily developed using the programming abstracts provided by our system.

An assisted living application for monitoring the activity of elderly people at home. Adult caregivers of elderly with cognitive decline (e.g., due to Alzheimer's disease) must be able to closely monitor the elder's physical and social activity in order to detect potential problems which require a followup (phone call, visit to a hospital) or to detect trends over many months. Remote monitoring would afford appropriate privacy to the caregiver and elder while enabling elders to maintain an active life. A mobile phone provides several important sensing capabilities such as location, images, motion (accelerometer) and acoustics and can be used to detect potential problems such as excessive time spent in bed in the morning, excessive walking around the house in the dark, motion in the backyard, vibration in slippers, voice conversations, etc.

Using our system, we can use the mapping phases (of the MapReduce framework) to detect activities and environmental conditions such as noise and movement. Abnormal conditions such as prolonged movement in the dark or excess and unusual noises can then be mapped to special monitoring nodes where further processing can be done and alerts can be sent out where applicable. Such an application can be easily modified and distributed using our system. Modifying the types of events detected is accomplished by appropriately setting the map function and alert conditions can be implemented in the reduce function.

An evacuation application for emergency situations in hospitals and other public structures. People are generally aware of emergency exit routes and exit locations, but it is much more difficult to communicate dynamically changing conditions such as congestion or obstacles. People generally carry phones with them. These phones can be used to collect relevant information such as locations within a building and acoustic information. Using this information, an application can be built on MapReduce to detect locations of congestion and infer obstacles at exits. The application would then use this information to intelligently provide users with the best routes to take.

3. RELATED WORK

The MapReduce framework [4] is a flexible, distributed data processing framework designed as an abstract to automatically parallelize the processing of long running applications on petabyte sized data in clustered environments where nodes have high and stable connectivity, relatively low failure rates and a shared file system.

The main insight of the MapReduce programming model is that a large computation is split into a number of smaller tasks; these tasks are independent of each other and can be assigned on different worker nodes to process different pieces of the input data in parallel. Due to the independent nature of the tasks, replication of the tasks due to worker failure is simply a matter of reassigning the task at another worker if the server does not receive a response from the first worker.

MapReduce was inspired by two functional language primitives: *map* and *reduce*. The map function is applied on a set of input data and produces intermediary $< key, value >$ pairs, these pairs are then grouped into R partitions by applying some partitioning function (e.g. $hash(key)$ MOD R). All the pairs in the same partition are passed into a reduce function which produces the final results.

The popularity of the MapReduce framework is attributed to its simplicity, portability and powerful functional abstractions. Application development is greatly simplified as the user is only responsible for implementing the map and reduce functions and the system handles the scheduling, data flow, failures and parallel execution of the applications.

Hadoop [3] is currently one of the most popular open source, java-based, MapReduce frameworks. Hadoop is targeted at a cluster environment with long running applications. It offers a *Capacity Scheduler* which is a fair scheduler that guarantees a predefined fraction of the computing capacity to each queue. Disco [17] is another open source MapReduce framework, targeted at the same environment as Hadoop. [12] has implemented a MapReduce system on cell phones and showed that cell phones already provide a significant source of computational power.

Middleware for mobile devices such as Olympus [14] and the work by Ranger *et al* [16] share the same objectives with our work: they aim at easing application development across devices with different capabilities by focusing on concerns such as interoperability, context awareness and server/client APIs. However they do not provide the distributed computation features or the simplicity that we need. A first come first server scheduling strategy for mobile devices has been proposed in [2]. However, these systems do not consider real-time or fault-tolerant issues. Fault-tolerant and real-time CORBA is explored in [10] [6]; unlike our work they propose light weight replications for task groups on different nodes.

4. THE MISCO SYSTEM

4.1 System Model

We consider a system that consists of a set of N distributed applications $A = A_1, A_2, ..., A_N$ running on a set of M worker

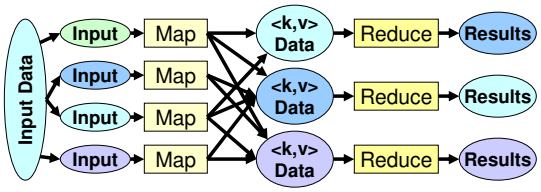

Figure 1: A simple visual representation of the map reduce model. Map and reduce operations operate on nodes (rectangles), data is represented as ovals.

nodes (mobile phones) $W = W_1, W_2, ..., W_M$. Each distributed application A_j is represented as a flow graph (shown in Figure 1) that consists of a number of *map tasks* (T^j_{map}) and a number of *reduce tasks* (T^j_{reduce}) executing in parallel on multiple worker nodes. Additionally, each application A_j has a deadline $Deadline_j$, which is the time interval, starting at the time the application is submitted to the system, within which the application should complete.

Distributed applications can either be triggered by the user or by an automated system. Applications can also be triggered aperiodically or periodically.

Our system schedules map and reduce tasks to execute in parallel on the worker nodes. Each worker node is able to run either a map or reduce task at any one time. Tasks cannot be preempted once they have been assigned to a worker, however, the execution of tasks from different applications can interleave. The worker is only responsible for executing the current task it is assigned, it does not keep track of the tasks (and from which applications) it has completed as the server maintains this information. This is possible because all tasks are independent of each other and the system is responsible for providing the proper input data for each task.

4.2 Misco Overview

Although Misco follows the general design of map reduce, it does vary in several major ways. Table 2 goes into further details related to the tasks in the system.

Misco comprises a *Master Server* and a number of *Worker Nodes* (Figure 2). The Master Server keeps track of user applications, while the Worker Nodes are responsible for performing the map and reduce operations. The Misco server also maintains the input, intermediary and result data associated with the applications, keeps track of their progress and determines how application tasks should be assigned to workers. In this section we discuss the design and implementation challenges we faced when developing Misco.

4.2.1 Polling-based Approach

One of our first design challenges was how to implement the server-worker communication when retrieving task assignments from the server and transferring data. There are two well known communication mechanisms: interrupt based and polling based [11].

Our design choice was to use polling. Its main advantages are its ease of implementation and low-overhead. The interrupt based approach, on the other hand, is much more costly and impractical in a mobile setting because it requires

Table 1: Misco Framework Definitions

Job	A Misco job consists of map/reduce operations and the input data to be processed. The server breaks a Job into a series of map and reduce tasks in order to parallelize the processing. **Start:** map and reduce functions (along with any supplemental functions), input data **Finish:** Results data is generated
Task	A task is a single map or reduce operation along with a piece of the input data **Start:** A single map or reduce operation (along with any supplemental functions), a piece of input data **Finish:** The operation (map or reduce) is performed by the client on the input data and its results are sent back to the server.
Master Server	The master server node is responsible for assigning tasks to workers when the worker node requests work. The server is also responsible for handling intermediary data generated by the mapping tasks and prepares it for input into the reduce tasks.
Worker Node	A worker node performs map and reduce operations. This node can be a smart-phone, mobile device or regular computer running the Misco client application.

that the phone be able to listen for incoming messages from the server when a task becomes available. It also requires the inclusion of large libraries, which reduces the amount of memory available on the phones. This is generally not a problem in controlled environments, usually the worker can use a server daemon to listen for incoming messages from the server. However, in our target mobile environment, interrupts are difficult to implement because it requires the inclusion of large libraries. Furthermore, the cellular networks which they operate on may not properly forward packets directed at the phones. Other approaches, such as using a SMS (short message service) sent to the worker do not work well as it is costly in terms of sending and receiving SMS messages. Also, SMS messages are only viable for phones which have a SIM card and an active account.

This polling approach has several advantages: the polling frequency can be adjusted to the load of the worker, the worker polls the server each time it becomes available. If there are no tasks available, the worker will idle for a period of time before requesting a task again. This, however, has the disadvantage that it is difficult to match the polling rate to the task arrival rate. If the polling frequency is too short, energy will be wasted due to the frequent checking for tasks, if the polling frequency is too long, the increased response time will adversely effect the application performance. If a task becomes available immediately after the worker polls the server, the worker will waste resources while tasks are

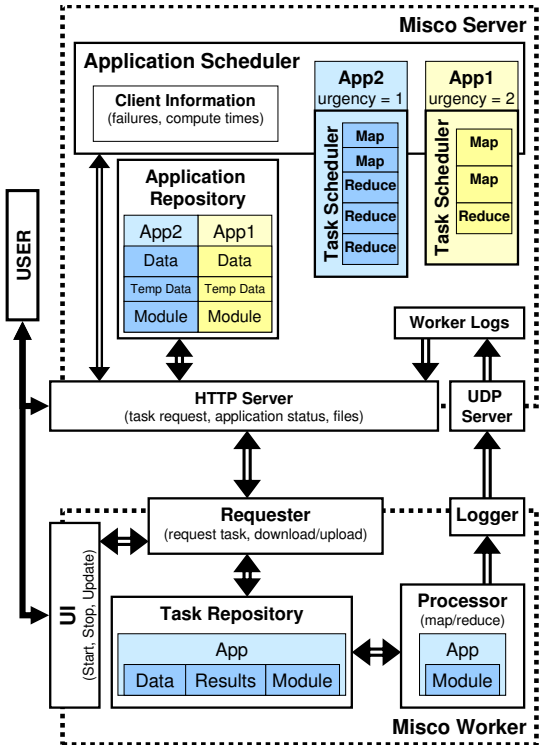

Figure 2: Architecture of the Misco System.

waiting. Thus, determining the polling frequency is not trivial. In our experiments, we set the polling frequency to $10sec$, however, we have implemented this polling frequency as a tunable parameter, and thus, can be adjusted to suit the application needs.

Our second design consideration is the method of data transfer. Traditional MapReduce systems assume a shared file system; this is not possible in a mobile setting. We decided to use HTTP to communicate requests, task information and transfer data. HTTP is a standard built on top of a reliable transport layer and most devices already support it via common libraries.

To perform the map and reduce operations, the user must provide a module with the required and optional functions (see Table 3). Although the overall idea for processing map and reduce operation are similar - the worker retrieves data and processes the data using the map or reduce operation - there are several significant differences.

For the mapping operation, the input data is a map input data piece with no particular formatting: the user provides a function for reading and formatting the data for input to the map function. This input reader function is included with the module file defining the main map and reduce operations. Additionally, the server also provides the worker with the number of reduce partitions in the job so the worker can partition its key-value pairs properly. The user provides a function for partitioning the keys into different reduce partitions, generally, it is advisable for the partitioning function to partition the keys uniformly between the partitions - this allows reduce tasks later to have roughly the same computation times. Finally, as some jobs generate a lot of duplicate

Table 2: Misco Task Part Definitions

Module	A python module, supplied by the user, which contains the map and reduce implementations along with any optional functions the user provides. This does not include any input data to be processed.
Operation	A single map or reduce algorithm which is implemented in a Python module the user provides. This does not include the data to be processed.
Input Data	This is the data file the user inputs into the system.
Map Input Data Piece	Input data is submitted by the user, it is split into many pieces. Pieces are assigned to worker nodes along with a map operation. Only one piece is assigned along with the map operation, these two together form a map Task (see above)
Map Result	When a worker finishes a map task, it will produce a series of result files containing $< key, value >$ pairs. These data pairs will be divided into partitions where each partition will be used as a part of the input for a reduce task.
Reduce Input Data	All the files belonging to the same partition but produced by different map tasks are merged together. The merged data file containing the map result from all the map tasks in a job are used as the input for a reduce partition. It is up to the user to decide how many partitions the map data will be split into and thus, how many reduce tasks will be created. A reduce input data along with a reduce operation form a reduce task.

key-value pairs or otherwise redundant data, the user can optionally specify a combiner function in the module file used to mitigate this redundancy.

The reduce task is much simpler than the map tasks, the input for the reduce operation is a list of key-value pairs generated by the map tasks. The reduce operation is performed on this list of key-value pairs and the final results are generated. In both cases, the results are saved as files and uploaded to the server.

4.2.2 *Misco Worker*

The main responsibility for the Misco worker is to process the individual map and reduce tasks and return the results to the server.

The Misco worker consists of a *Requester* component, a *Task Repository* component and a *Logger* component. The *Requester* component is used for interactions with the Misco server to request tasks and download and upload data, trig-

Table 3: Misco Job Module functions the user provides in the python module when creating a job

Map	The main map function, it takes the output of the map_reader function below as input and produces a list of key-value pairs
Reduce	The main reduce function, it is given an iterator which will iterate over all the key-value pairs generated by the map tasks which belong to the reduce task's partition
Map reader (optional)	This function takes the input file and processes it into pieces which is sent to the map function. If it is not specified, the default behavior is to simply pass the map function one line from the file at a time.
Partition (optional)	This function determines which partition a key-value pair belongs to when given the key and the total number of partitions. If it is not specified, the default behavior is to use Python's built in hash function on the key, where the key is cast into a string.
Combiner (optional)	This function is applied to key-value pairs which the map function generates. It can maintain a buffer between runs and thus maintains state. The default behavior is to do nothing.

ger the local execution of the tasks, and handle the communication with the Misco system during upgrades. When a worker is free, it will contact the server and request a task. The communication between the server and worker is accomplished through the *HTTP server* using a special URL where the server listens for requests. Each task is characterized by the name of the application which the task belongs to, the location of the module containing the map or reduce operations and the location of the input file. The Misco worker stores locally in its *Task Repository* component: the input data, module and any results it has generated for each task. Finally, the *Logger* component is used to maintain local statistics regarding the processing times of the tasks and progress. When a task completes, these statistics are sent to the Misco server, along with the task results, to be used for future executions of the tasks.

4.2.3 Misco Server

The Misco server is in charge of keeping track of applications submitted by the user and assigning tasks to workers. It comprises a *Scheduler* component that implements our two-level scheduling scheme, an *Application Repository* component that keeps track of application input and output data, and an *HTTP Server* that serves as the main communication between the workers and the Misco server. It is responsible for receiving requests, displaying application statuses to the user via the web UI and handling the downloading and uploading of data files. The *UDP Server* is used to listen for incoming worker logs, which it stores in *Worker Logs*. The UDP Server uses the UDP transport protocol.

Applications are created and managed by the user using a browser interface. The Misco Server will initialize the state of the application by creating the folders to hold the input, intermediate and final result data files and the application will be inserted into the scheduler's queue. When a worker is available to run a task, the Misco Scheduler is responsible to assign a task to the worker. Once the worker completes the task, the results are uploaded to the server and stored in the application repository.

Scheduler: The scheduler component is responsible for determining which map or reduce task to assign to a worker. We have developed a two-level scheduling scheme. A first-level scheduler, the *Application Scheduler* determines the order of execution of the applications. The second-level scheduler, the *Task Scheduler*, chooses the actual Map and Reduce tasks from the application determined by the *Application Scheduler*.

Our work focuses on aperiodic, soft real-time tasks, where our objective is to maximize the number of tasks meeting their deadlines. In such cases, missing a deadline is not catastrophic for our system.

We chose to use the well-known Earliest Deadline First (EDF) scheduler for our application scheduler. Our goal is to minimize the number of deadline misses. EDF gives higher priority to the task with the smallest deadline. For this, we simply choose the application with the earliest deadline to select the next task from. For the task scheduler, we use a *Sequential Scheduler*. This scheduler assigns tasks sequentially to workers. When all tasks have been assigned, the scheduler loops back to the first uncompleted task and sequentially assigns any other unfinished tasks. Although this scheduler is fairly simplistic, it performs much better than a more naive random scheduler which simply picks a random uncompleted task to assign workers. This is due to the number of duplicate tasks assigned by the random task scheduler.

5. IMPLEMENTATION

We designed Misco to operate on the Nokia N95 8GB smartphones. These phones are S60 phones and support an simplified, beta implementation of Python. Although we started out targeting this specific phone, we developed the main part of the Misco worker code to run on any Python system.

We used Python for the implementation of both the server and worker components due to its advantages of expressiveness, ease-of-development and portability across many devices. In particular, the worker uses Python's *urllib* module to download files on the phones, and the *httplib* module to create multi-part POST requests for uploading results to the server.

5.1 Misco Worker

As a purely Python implementation, our Misco worker can theoretically run on any platform that supports Python and its networking libraries. While the main component of the Misco worker are shared between all clients, the setup scripts are device specific and we implemented two client setup scripts: one for PyS60 2.5.4 enabled, 3rd edition smart-

phones (e.g. N80 and N95), and a script for use on commodity machines running Python.

The Misco worker provides the ability to install and upgrade the worker code. It also had a utilities for cleaning any residual data files if a client crashes and is manually restarted. The worker retrieves all information from the server and uploads results via HTTP. When the worker requests a task, it requests a special URL which returns the task information as a plain text representation of a Python dict object. The worker will perform an *eval*() operation on the returned data and load it into its environment. If the server has no tasks to assign, it will return $'None'$ and the worker will idle for a period before requesting a task again.

The Misco worker downloads all files using Python's *urllib* module and uploads results to the server using Python's *httplib* module to create a multipart POST request. For storing data and modules on the client, the worker creates a folder for each. The client avoids downloading any modules it had previously downloaded - the server uses a unique filename for each module, the module is named after job names, which cannot be duplicated. The map tasks generate one file for each reduce partition which it maps a key into while the reduce task generates only one result file. Initially, we did not remove any data files, but found that if we let them accumulated, there some problems removing them later using Python on the smart-phones. Although it adds a bit of time to task time, the data is removed after the final result is uploaded to the server.

After the data and module is downloaded to the client, the worker will use Python's dynamic import ability to extract the functions related to the task. If an optional function is not defined by the user, a default function will be imported in its place.

During a task, log messages can be sent to the server's UDP log listener. Currently, these log messages carry basic timing information - a breakdown of how long each portion of a task takes. The server does not currently take this into consideration and only uses it to display status to the user.

5.2 Misco Server

The user interacts with the Misco server using a HTML interface which can be accessed through a web browser (our server is currently running on a local LAN and is accessed at http://192.168.1.101:16000). For creating a job, the user provides the job deadline, the input data, the module, the size of the map input pieces, the number of reduce partitions and optional parameters which are to be passed into the user's functions defined in the module.

Once the job is submitted, the server first creates a directory structure for it consisting of a folder for the module, map input pieces, reduce input pieces and final results. Then, the server splits the input data into pieces, creates an object to keep track of the progress and other information regarding the job and places this job object into the job scheduler. The original input data and module are stored in a separate folder independent of the other job data so that it can be used for subsequent jobs without having to re-upload.

After the initial creation, the Misco server splits the input file into a set of M input files labeled 1 through M. These labels are used to keep track of which pieces are complete later when the worker returns results. At this point, there are no reduce tasks yet as the reduce input data has not been generated. However, the number of reduce tasks will equal the number of reduce partitions, R, the user specified when creating the job, these reduce tasks are numbered 1 through R.

The HTTP server component of the Misco server handles both worker requests for tasks and also requests for HTML pages and workers downloading data. When a user interacts with our system, they simply go to the proper URL and the server acts like a normal web server with handlers tied into the Misco server system. When a Misco worker makes a request, the server returns a plain text page containing a Python dict object described earlier. When a worker requests input data, it simply downloads it from the URL contained in the earlier Python dict object returned by the server containing information about the task.

The HTTP server also handles file uploads from the workers when they are submitting the results. The worker upload files using a multi-part POST request and specifies whether the results are for a map or reduce task along with which piece of data was processed. When a map result is uploaded, the server checks if all the map input pieces have been processed. If they have, then the server creates the reduce inputs by concatenating the map results which belong to the same partition. The result data received from reduce operations are simply stored as final results.

The server is multi-threaded, spawning a new thread to handle each incoming connection (e.g. users, workers) and uses locking of critical sections for dealing with concurrency. The server also contains a UDP log server for receiving worker log messages as described in the Misco worker section.

6. EXPERIMENTAL RESULTS

6.1 Experimental Setup

We have conducted a set of experiments to evaluate the efficiency and performance of our system. Our experimental platform is a testbed of 30 Nokia N95 8GB smart-phones [13]. The Nokia N95 has ARM 11 dual CPUs at 332 Mhz, supports wireless 802.11b/g networks, bluetooth and cellular 3g networks, 90 MB of main memory and 8 GB of local storage. Our server is a commodity computer with a Pentium-4 2Ghz CPU and 640 MB of main memory. The server has a wired 100 MBit connection to a Linksys WRT54G2 802.11g router. All of our phones are connected via 802.11g to this router.

For the experiments we used a set of 10 applications; each consisting of between 16 to 73 tasks. We implemented a WordSearch application where each worker searches through different sections of a large text file for keywords in the map phase and the results are aggregated and ordered in the reduce phase. Our motivation is to provide an interactive method to analyze user web logs to find places of interest within a city. This application is representative of general MapReduce applications and fully exercises all aspects of the system and demonstrates the timeliness of our sched-

Figure 3: Our Misco testbed of Nokia N95 8GB phones and a Linksys Router.

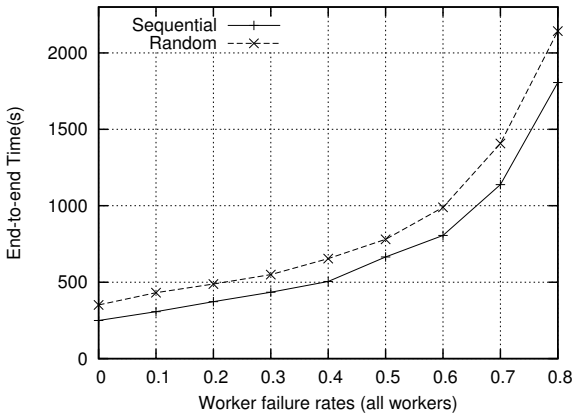

Figure 4: End-to-end times for our sequential task scheduler with an EDF application scheduler when compared to a random task scheduler. All Workers have the same failure rate which is varied.

uler. More complex applications mainly differ in the functions performed in the map and reduce tasks and not in the system's execution sequence. We varied the data input sizes between 100KB to 1MB and set different deadlines for the applications, varying between 100s to 550s to simulate both tight and loose timing constraints. The failure of worker nodes follows a Poisson distribution; to vary the failure rates of our workers, we adjust the failure arrival rate, λ. Each experiment was run 3 times and results were averaged.

6.2 Experimental Results

6.2.1 *End-to-end Execution Times.*

For our first set of experiments, we wanted to test the performance of our system with different failure rates for the worker nodes. We set all workers to the same failure rate and vary this failure rate.

For the first set of the experiments we measured the **end-to-end execution times** of the applications running on our system. The end-to-end times capture the time it takes to run the entire computation and also the communication times. The **End-to-end times** of the applications are shown in Figure 4 for different failure rates for the worker nodes. As the figure shows, our system handles failures by providing redundancy even as failure rates become higher. As expected, asthe failures increase, the end-to-end times increase exponentially.

6.2.2 *Overhead and Local Task Performance Measurements.*

In this set of experiments, we wanted to measure the overhead and feasibility of our system on our phones. We used the Nokia Energy Profiler [1] to take CPU, memory and power measurements.

We first measured the memory overhead of Misco by measuring the RAM used when loading the Misco software on the mobile phones. This includes data structures implemented by our system and also Python libraries. Misco requires only 800KB of memory, this is less than 1% of the 90MB of free RAM available on our phones. Figure 5 shows the memory usage of our application, the increased memory usage at 125, 165, 205 and 235 are from data structures used by the map task while the gradual increase of memory usage at 240, 280 and 315 are due to data structures built by the reduce tasks. This extra memory usage is application dependent.

Figure 6 shows the power usage, processing tasks uses 0.7 watts while network access draws more than twice the power at 1.6 watts, these are the spikes seen in the figure for when the worker is requesting and receiving data from the server.

CPU utilization for executing the tasks is dependent on the application and also on other programs running on the phone. The Symbian operating system will share the CPU resources among running programs.

We also present, in Table 4, the local task performance measurements using different data sizes for map tasks (similar results for reduce tasks). The *download* time reflects the time it takes to download locally the input data, *compute* time is the time spent performing the task, while *upload* time denotes the time to upload results to the server. *Cleanup* time is for deleting the input, results and any temporary files generated. Clearly, the measurements show that the majority of processing is spent performing computations, and only a small fraction of the time is required for downloading, uploading and cleanup, which is quite efficient for resource constrained devices such as the phones we are considering.

7. CONCLUSION

In this paper, we presented `Misco`, a MapReduce framework for supporting applications on mobile systems. This is the first system, that we know of, that proposes a MapReduce framework for mobile environments. We have also demonstrated the feasibility of our system by deploying applications on our N95 testbed.

Misco is the first MapReduce framework implementation for

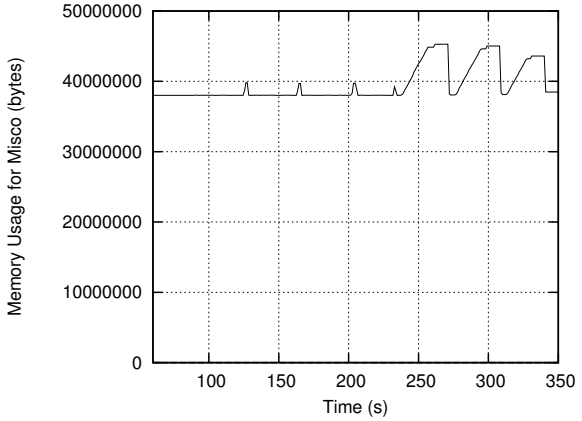

Figure 5: Memory usage for Misco showing four map tasks followed by 3 reduce tasks.

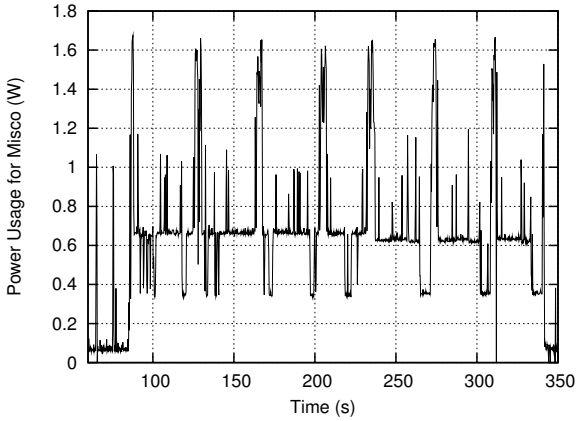

Figure 6: Power use for Misco showing four map tasks followed by 3 reduce tasks.

Table 4: Local Task Performance Measurements

input (KB)	down (s)	up (s)	clean (s)	compute (s)
Map Tasks				
62	0.5	1.2	0.5	8
94	0.6	1.2	0.6	10
125	0.7	1.4	0.7	11
Reduce Tasks				
96	2	0.42	0.05	3.8
150	3	0.5	0.04	5
192	4	0.7	0.04	7

mobile settings that we know of, and there are many research directions to proceed in. We plan on extending our system to consider data locality, decentralization and device heterogeneity. Further, the new mobile setting introduces the opportunity to explore completely different types of applications, applications where the worker nodes not only process data provided to them, but are also active participants in the production of data.

8. REFERENCES

[1] Nokia energy profiler. http://www.forum.nokia.com/main/ resources/ user experience/ powermanagement/nokia energy profiler/.

[2] H. K. Anna and J. Gerda. A robust decentralized job scheduling approach for mobile peers in ad-hoc grids. In *CCGrid. Rio de Janeiro, Brazil*, 5 May 2007.

[3] D. Cutting. Hadoop core. http://hadoop.apache.org/core/.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI, San Francisco, CA, USA*, pages 137–150, Dec 2004.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI'03*, San Diego, CA, June 9-11 2003.

[6] A. S. Gokhale, B. Natarajan, D. C. Schmidt, and J. K. Cross. Towards real-time fault-tolerant corba middleware. *Cluster Computing*, 7(4):331–346, 2004.

[7] Google. Android. http://www.android.com.

[8] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *The Seventeenth International Conference on Parallel Architectures and Compilation Techniques (PACT), Toronto, Canada*, pages 260–269, Oct 25-29, 2008.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and operating Systems (ASPLOS'00)*, pages 93–104, Pittsburgh, Pennsylvania, March 13-17, 2010.

[10] H.-M. Huang and C. Gill. Design and performance of a fault-tolerant real-time corba event service. In *18th Euromicro Conf. on Real-Time Systems (ECRTS'06), Dresden, Germany*, pages 33–42, Aug'06.

[11] K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal. Integrating polling, interrupts, and thread management. In *Frontiers of Massively Parallel Computing*, Oct 1996.

[12] S. Mishra, P. Elespuru, and S. Shakya. Mapreduce system over heterogeneous mobile devices. In *Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009), Newport Beach, CA, USA*, Nov 2009.

[13] Nokia. N95 8gb device details. http://www.forum.nokia.com/devices/N95_8GB.

[14] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. M. s. Olympus: A high-level programming model for pervasive computing environments. *PerCom, Kauai, Hawaii*, 0:7–16, 2005.

[15] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. *HPCA, Phoenix, AZ*, Feb'07.

[16] T. Salminen and J. Riekki. Lightweight middleware architecture for mobile phones. In *PSC, Las Vegas, NV*, Jun 2005.

[17] V. Tuulos. Disco. http://discoproject.org/.