Scheduling for Real-Time Mobile MapReduce Systems

Adam J. Dou
UC Riverside
Riverside, CA
jdou@cs.ucr.edu

Vana Kalogeraki
AUEB
Athens, Greece
vana@aueb.gr

Dimitrios Gunopulos
Univ. of Athens
Athens, Greece
dg@di.uoa.gr

Taneli Mielikäinen
Nokia Research Center
Palo Alto, CA
{taneli.mielikainen,ville.h.tuulos}@nokia.com

Ville Tuulos

ABSTRACT

The popularity of portable electronics such as smartphones, PDAs and mobile devices and their increasing processing capabilities has enabled the development of several real-time mobile applications that require low-latency, high-throughput response and scalability. Supporting real-time applications in mobile settings is especially challenging due to limited resources, mobile device failures and the significant quality fluctuations of the wireless medium. In this paper we address the problem of supporting distributed real-time applications in a mobile MapReduce framework under the presence of failures. We present Real-Time Mobile MapReduce (MiscoRT), our system aimed at supporting the execution of distributed applications with real-time response requirements. We propose a two level scheduling scheme, designed for the MapReduce programming model, that effectively predicts application execution times and dynamically schedules application tasks. We have performed extensive experiments on a testbed of Nokia N95 8GB smartphones. We demonstrate that our scheduling system is efficient, has low overhead and performs up to 32% faster than its competitors.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Design, Experimentation, Reliability

Keywords

Distributed Systems, Mobile Systems, Real-Time, MapReduce

1. INTRODUCTION

In the last few years we have witnessed a significant growth of location-based systems where the physical location of the users is utilized to deliver information of interest, to encourage sharing of location-based information or to adapt the services in order to improve the level of service provided to the users. A number of location-based systems have emerged. Examples include traffic monitoring systems for real-time delay estimation and congestion detection [42], personalized weather information, location-based games or receiving spatial alarms upon the arrival to a reference time point [9], as well as social networking applications for sharing photos and personal data with family and friends [35]. In the search for economic power, companies are building these services over portable electronics such as smartphones, PDAs and mobile devices. These devices have significant processing and networking capabilities and are outfitted with a wide array of sensing capabilities such as GPS, WiFi, microphones, cameras and accelerometers, which have introduced new and more efficient ways of communication. However, this new application development brings new challenges to the design of system software. More specifically, providing real-time, low-latency and scalable execution for these applications presents three important challenges that need to be addressed:

1. we need to understand how the mobile setting affects the development of distributed applications over networks of smartphones and address the implications;
2. we need to provide a flexible and efficient way to program, develop and deploy the applications on the phones that is portability across multiple devices.
3. we need to be able to simplify programmability by hiding the difficult issues of distribution, real-time scheduling, device failures and connectivity issues from the application developer.

Application development over networks of smartphones. Several applications have been and are being developed for mobile platforms. Some applications focus on making the users daily lives easier by providing GPS guidance such as traffic congestion detection and delay estimates [42] and spacial alarm applications [9] which alert a user when they are a close by another user or a pre-designated area. Other applications help users stay connected. As people are expanding their use of social networking sites such as Facebook and Myspace, mobile applications are allowing them to view and modify their information in real-time from anywhere. The popularity of these real-time social sharing applications is evidenced by recent application compaigns [1] emphasizing these features. The usability of these systems is

DEBS'11, July 11–15, 2011, New York, New York, USA.

highly dependent on them running in a timely fashion when providing notifications and feedback. One challenge today is that there are still many problems which hinder the development of distributed applications on them. Mobile devices are not easy to develop for. There are many specialized languages and proprietary systems which have steep learning curves. With relatively limited resources compared to desktop and servers, memory management and application flow is different from traditional programming forcing new software paradigms, leading to many software defects. The main problem however is that distributed applications involving mobile devices exacerbate the problem by introducing several concurrency issues to consider. For example, because of user mobility and spotty connection coverage, network connectivity is often degraded or even lost.

Application Programmability. The second challenge is that several location-based applications, require the involvement of multiple users in the process of sensing and data processing, in order to achieve a task. These are deemed *participatory sensing* applications. For example, the MetroTrack system [3] is a mobile event tracking system capable of tracking mobile targets through collaboration among local sensing devices that track and predict the future location of a target. How to efficiently schedule work across a set of collaborating devices to accomplish a task is not an easy process. Recently, the MapReduce framework [14] was introduced to provide a highly scalable, distributed computing environment for conducting data processing computations. MapReduce has been successfully deployed in traditional server based environments [13] [43], specialized environments [25] [40], as well as in a variety of applications by many prominent companies such as Google, Yahoo, Facebook and IBM. The MapReduce framework's support for the weak connectivity model of computations across open networks, makes it very appropriate for a mobile network setting. It is important to note though, that, although we chose the MapReduce framework for our system, we do not target the same types of applications as traditional data warehouse based MapReduce systems, as the limited resources available on the mobile systems makes them unsuitable for any multi-petabyte data processing. Instead, our goal is to explore the use of the framework for participatory sensing applications such as monitoring and social networking applications and gain further insights into the possibilities of using the MapReduce framework for the next-generation of location-based applications over networks of smartphones.

Achieving Real-Time Response. The third challenge is how to support the execution of distributed applications with real-time response requirements on a network of smartphones. While the MapReduce framework provides the programming interfaces for developing mobile applications, all the low-level details of the real-time execution, fault tolerance and wireless network communication are not handled by the framework. Thus, the key question of how to provide support for applications with real-time response requirements, still remains. To date, most of the work on providing real-time application execution are either based on the deployment of static sensor networks (such as [33] and [26]), or are integrated within specific MAC or network layers [27] [18] to encapsulate application-specific tradeoffs in terms of resource constraints, shared wireless medium, lossy communication, and highly dynamic traffic or are restricted to a single node movement [30]. Supporting real-time mobile applications such as distributed mobile sensing is an important step for the wider adoption of the devices and to create opportunities for building new kinds of mobile application services. However, *timely execution* is a challenging problem in these settings due to highly dynamic topology, device unavailability and fluctuations in network quality and channel capacity. This makes it extremely difficult to estimate execution times and provide end-to-end real-time support to distributed applications. Unlike traditional cluster environments, mobile systems cannot rely on a static infrastructure and do not have control over the individual nodes. The problem is further exacerbated by *failures* of mobile devices. Permanent and transient failures such as battery depletion and user mobility can greatly affect the timeliness of distributed applications by reducing the processing power of the system, causing large delays and energy wastage.

In this paper we present *MiscoRT*, our system aimed at supporting the execution of real-time application tasks on mobile MapReduce systems. Our proposal follows a two-level approach to scheduling distributed real-time mobile applications. We first develop an analytical model to estimate the execution times of the applications in the presence of failures. Using this model, we determine the application urgencies based on our estimates of their execution times under failures and their timing constraints. Our goal is to maximize the probability that the end-to-end deadlines of the applications are met. By incorporating the expected failure model in the scheduling policy, we can adjust the fault-tolerance characteristics of the overall system. We see this as a major motivation for using the MapReduce model in mobile environments which are typically inherently unstable. Furthermore, we envision that methods developed for heterogeneous and unstable mobile environments of today can be useful in extremely loosely coupled computing environments of tomorrow which will not be confined to a single, well-controlled data center. We have built our approach on Misco [15], our MapReduce system that runs on mobile phones. We have implemented and tested our scheduling scheme on a testbed of Nokia's third generation NSeries phones [39]. We have built a mobile tourist application to evaluate our system. Our extensive experimental results demonstrate that our approach is efficient, has low overhead and completes applications up to 32% faster than its competitors.

2. OBJECTIVES

In this section we summarize the main objectives of our approach.

Supporting real-time applications: Our first objective is to support applications with real-time response requirements in the form of deadlines. In several applications such as traffic monitoring for real-time route suggestions and congestion detection, location-based notification of events or proximity notification for friends, users rely on the collaborative sensing of data from multiple phones and the timely collection and processing of the data, to generate outputs of interest and detect important events. However, mobile devices have limited computation and communication resources. Limited capacity, queuing and channel access delays can greatly affect the timeliness of the applications. Our approach is to provide real-time response to the applications, schedule the applications based on their relative

urgencies and respond to changing conditions by dynamically scheduling the execution of application tasks to meet their deadlines. We have to note that the applications we develop on our system do not have hard real-time response requirements; rather, we target soft real-time applications where our goal is to maximize the number of deadlines met, and missing a deadline is not catastrophic for the system.

Accounting for device failures: We consider mobile node failures an integral part of mobile environments, thus our approach is to account for these failures when deriving an estimate of the execution times of the applications. Device failures have a major impact on the timeliness of our system. These failures are the results of hardware and software issues, user actions and user mobility. These errors can be classified as *permanent* and *transient* failures.

Permanent failures are failures where the device becomes unavailable for an extended period of time during which it is unable to participate in processing results for the system. Permanent failures can occur from software errors where critical software has crashed due to software or hardware faults or when users are shutting off their devices. A permanent device failure is equivalent to having less processing units or power in our system. This results in an overall slowing of the processing rate for our applications and can cause more deadline misses.

Transient failures are the more interesting failures we deal with as the device recovers from the failures and continues to operate normally after a period of downtime. In software errors, phone self shutdowns [12] and user actions such as removing the battery from a visually unresponsive device or manually resetting a device contribute a bit to the rate of transient errors, the largest factor for mobile devices would be user mobility [31]. Mobile devices make use of wireless access points when they are able to, and these APs have a typical range of less than 100 meters. When a user is outside of coverage range he/she is unable to communicate with the server, and during these times, if the device needs to send results, they are considered to have failed. When the device regains communication coverage, it resumes processing. We use these mobility and failure statistics to extract a failure distribution for our worker devices. [31] reports that the time spent at any AP or location follows a log-normal distribution, and that the movement speeds of the users also follow a log-normal distribution. Based on this result, in our experiments, we have set the failure rates for our devices to follow such a distribution.

Devices which display transient failures are still very useful in our system. In our work, we predict the failure rate of these devices due to their self-similar nature, a device which fails often is expected to fail often in the future. We can produce an expected time for different devices to perform tasks which helps us develop a better schedule for the applications to meet their deadlines. By selectively replicating tasks to less failure prone workers and profiling the execution times of the tasks, we can effectively meet the real-time constraints, even under failures. Our focus in this paper is on device failures; server failures can be addressed with the use of backup servers and checkpointing techniques that record the state of the server and transfer it to a backup server in the event of a failure[19]. In our previous work, we have studied the problem of fault tolerance for distributed object systems[34] [38]. However, the issue of server failures is a research area by itself, and is not considered in this paper.

3. BACKGROUND

In this section we first provide an introduction to the MapReduce framework. For the baseline MapReduce implementation we have used Misco [15], a distributed mobile MapReduce platform targeted at any device that supports Python and network connectivity. This is a porting of the MapReduce system to run on a network of smartphones. For completeness, in this section we give a brief description of the specific implementation of Mapreduce we have developed to run on the network of smartphones. In the next sections we describe the current work which is the real-time model and the scheduling strategy.

3.1 The MapReduce Framework

The MapReduce framework [14] is a flexible, distributed data processing framework designed as an abstract machine 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 MapReduce programming model provides a simple way to split a large computation 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. The main insight of MapReduce is that the programming model is clean and simple to use, yet powerful enough to do sophisticated computations 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.

3.2 The Misco System

The Misco system is a MapReduce implementation that runs on mobile phones. Misco comprises a *Master Server* and a number of *Worker Nodes*. The Master Server keeps track of user applications, while the Worker Nodes are responsible for performing the map and reduce operations. The Master 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. [15] provides in-depth design and implementation details of the system. The Misco system is publicly available at http://www.cs.ucr.edu/∼ jdou/misco/.

The main responsibility of the *worker* is to process map and reduce tasks and return the results to the server. When a worker is free, it will contact the server and request a task. 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 worker stores locally: its input data, module and any results it has generated for each task. Finally, a logger component is used to maintain local statistics regarding the processing times of the tasks and progress.

The *Server* is in charge of keeping track of applications submitted by the user and assigning tasks to workers. The Server is multi-threaded, spawning a new thread to handle incoming worker connections. Applications are created and managed by the user using a browser interface. The server consists of 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 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 Server also gathers failure and computation time statistics for the workers in the *Client Information* component. The success rate of a worker is calculated as the number of successful results it has returned divided by the total number of tasks it has been assigned. In cases of transient failures, if the worker is unable to contact the server when it returns the results, it will abort the task and enter into idle mode and poll the server for new tasks.

We have extended the system with a *Scheduler* component that implements our scheduling approach (described in the next sections). We have developed the system in Python and run it on the Nokia N95 8GB smart-phones [39]. The reason we use Python is because the Nokia N95 smart-phones support a beta implementation of Python 2.5.4 called PyS60. Our design does not use any proprietary or any Python-specific features, and therefore can be run on any Python enabled phone or can be ported to different operating or programming environments.

4. MICRORT SYSTEM MODEL

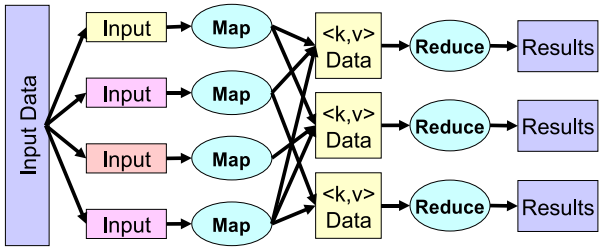

Figure 1: A simple visual representation of the map reduce model.

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 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.

Distributed applications are triggered by the user, they are aperiodic and their arrival times are not known a priori. Each application A_j is associated with a number of parameters: We call *ready time* r_j the time the application becomes available in the system. $Deadline_j$ is the time interval, starting at the ready time of the application, within which the application A_j should be completed. The *execution time*, $exec\_time_j$ of the application is the estimated amount of time required for the application to complete. This is estimated based on previous executions of the applications, by measuring the difference from the *ready time* of the application until all its map and reduce tasks complete. Thus, the $exec\_time_j$ of an application depends on (1) the number of T^j_{map} and T^j_{reduce} tasks, (2) the size of the application input data, and (3) the number of worker nodes available to run the tasks. This information is recorded and stored by the Server for each application run in the system. We associated with each application a laxity value, which represents a measure of urgency for the application and is used to order the execution of the application tasks on the worker nodes. $Laxity_j$ is computed as the difference between the Deadline of the application and its estimated execution time. The laxity value is adjusted dynamically based on failures and queuing delays experienced by the tasks. The advantage of using the laxity value is that it gives us an indication of whether the execution of the tasks has delayed and how close the task is to missing its deadline; the task with the smallest laxity value has the higher priority, while, when the laxity value becomes negative the task is estimated to miss its deadline and thus it can be dropped. Our work targets soft real-time systems, where missing a deadline is not catastrophic for the system. Thus, our goal is to maximize the number of applications that meet their deadlines.

For each task t of an application A_j we compute: the *processing time* $\tau^j_{t,k}$, the time required for the task to execute locally on worker W_k. This includes the time required to process input data, upload the results to the server and clean up any temporary files it generates. These times can be either provided by the user or obtained easily through profiling mechanisms with low overhead [29].

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.

We are not concerned about the network topology of the system, similar to other cell phone based systems [36], we assume that if we have connection to the server, it is over a one-hop HTTP connection. Any networking overhead is implicitly accounted for by monitoring worker timings and gathering statistics.

5. OUR SCHEDULING SCHEME

The main responsibility for our MiscoRT scheduler is to assign tasks to workers when they make requests. We have developed a two-level scheduling scheme, as follows (the architecture of our system is shown in figure 2):

- The first-level scheduler, the *Application Scheduler*, determines the order of execution of the applications based on their urgencies and timing constraints. It es-

timates the execution times of the applications using an analytical model that also considers mobile node failures.

- The second-level scheduler, the *Task Scheduler*, dynamically schedules application tasks and uses the measured laxity values of the tasks to adjust their scheduling order to compensate for queuing delays and worker node failures.

5.1 Failure Model

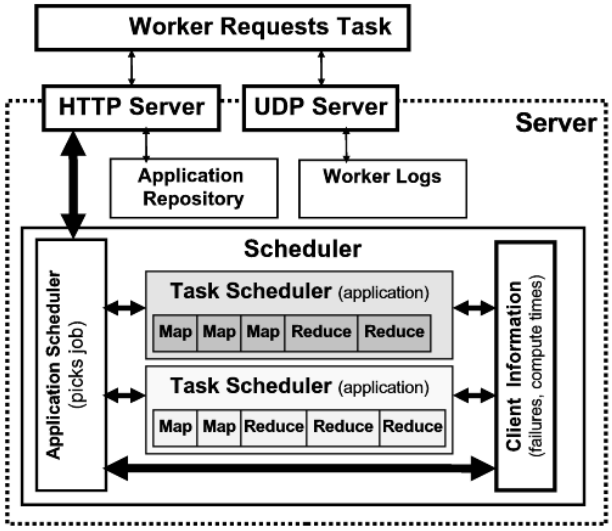

Figure 2: Our system architecture.

In this section we present our model to estimate the execution times of the applications under failures. We assume that the failures of the worker devices follow a Poisson distribution and that failures are transient. In cases where permanent failures occur, the total number of workers would be reduced as the failed workers cannot make any further requests for tasks and are, therefore, not assigned any additional tasks. It has been shown that the time to failure for systems can be accurately represented using a Poisson distribution [16]. Assume that λ_i is the failure arrival rate for a single worker W_i. When a worker fails, all progress on the task it was executing is lost, and the worker experiences a failure downtime following a general distribution with mean time for recovery μ_i.

For application A_j and worker W_i, we summarize these parameters as follows:

- λ_i - failure arrival rate for worker W_i
- τ_i^j - local processing time for task of application A_j on worker W_i
- μ_i - mean recovery time from a failure for worker W_i
- w_i^j - expected task processing time including failures

5.1.1 Single Task, Single Worker

Our basic unit of work is a single task executing on a single worker W_i with processing time, including failures, C (we omit superscripts and subscript where there is no ambiguity). The probability of failure during a task processing is $\tau\lambda$, the corresponding probability of success is $1-\tau\lambda$. Let F be the number of failures before the first success, this is a geometric series, from probability theory [32], we can compute the expected number of failures using:

$$E[F] = \frac{\tau\lambda}{1-\tau\lambda} \tag{1}$$

We now calculate the amount of time to successfully complete a task. This is comprised of 3 parts: (1) a successful run, requiring τ time, (2) the sum of all the times wasted (W) processing a task before failures occur and (3) the sum of all the downtime (D) in order for the worker to recover from failures:

$$C = \tau + \sum W + \sum D \tag{2}$$

Since failure arrival follows a Poisson distribution, failures occur at the workers with an exponential distribution and tasks are expected to fail halfway through processing, so the expected wasted processing time is given by:

$$E[W] = \frac{\tau}{2} \tag{3}$$

Let $E[D] = \mu$ be the mean recovery time from a failure. Finally, we can compute the expected processing time for a task on a node, including failures, as:

$$w = E[C] = \tau + \frac{\tau}{2} * \frac{\tau\lambda}{1-\tau\lambda} + \mu * \frac{\tau\lambda}{1-\tau\lambda} \tag{4}$$

5.1.2 Multiple Tasks, Multiple Nodes

We consider T tasks belonging to a single application A_j and M workers. Each worker W_i can be characterized by a different failure arrival rate parameter λ_i and mean failure downtime μ_i. The total execution time C^j for all T tasks of application A_j is the maximum of the individual processing times for each worker executing tasks for this application.

Since all workers are either processing a task or in a failure state, we can model this by considering a equal-time workload for each worker. The rate of work for worker W_i is the inverse of its expected processing time: $1/w_i$. For the workers to finish their tasks at the same time, the number of tasks ρ_i assigned to worker W_i ($1 \le i \le M$) is:

$$\rho_i = \lceil \frac{1/w_i}{\sum_{k\in M} 1/w_k} * T \rceil \tag{5}$$

Then, we can compute the expected execution time for the application as:

$$exec\_time_j = E[C^j] = max_{i\in M}(\rho_i * w_i) \tag{6}$$

5.2 MiscoRT Application Scheduler

Our application scheduler is based on the least-laxity scheduler and is used by the server to determine the order of execution for the applications in the system. The *Least Laxity First* (LLF) scheduling algorithm has been shown to be effective in distributed real-time systems [29]. In LLF scheduling, each application is associated with a laxity value which represents its urgency. We compute the laxity value $Laxity_j$ of an application A_j as the difference between its deadline

Pseudo Code 1 The MiscoRT Schedulers

MiscoRT Application Scheduler

Input: Set of applications A in system
for all Application A_j in A **do**
  calculate $Laxity_j$ of A_j
Order A by $Laxity_j$
Task ← TaskScheduler(A_j with smallest $Laxity_j$)
return Task

MiscoRT Task Scheduler

Input: worker W_k requests a task, job A_j
step 1. **if** unassigned task $T_i^j \in A_j$ **then return** T_i^j
step 2. **if** failed task $T_i^j \in A_j$ **then return** T_i^j
step 3. $T_i^j \leftarrow$ slowest task in A_j
  if T_i^j will complete after $deadline_j$
    and T_i^j will complete on W_k before $deadline_j$ **then**
      return T_i^j

and our estimate of the execution time of the application under failures:

$$Laxity_j = Deadline_j - current\_time - exec\_time_j \quad (7)$$

where the estimate of the application's execution time is computed using formula 6 to include worker node failures.

The laxity value for a distributed application is computed when the application first enters the system, this is denoted as the initial laxity. As an application executes, its laxity value is adjusted to compensate for variations in the processing speeds of the workers, worker node failures and queuing delays. As workers start to fail and their failure rates change, we use our analytical model to recalculate the expected execution time and laxity. To minimize computational overheads, the laxity value for each application is computed only when a worker makes a request. Applications with negative laxities are estimated to miss their deadlines and their tasks should not be scheduled ahead of applications which have positive laxities. Note that no applications are dropped and that all applications will complete eventually if at least one worker remains available.

The advantage of this scheduling scheme is that the schedule is driven by both the timing requirements of the applications and node failures, while it allows us to dynamically adapt to changes of resource availability or queuing delays. If the workers processing the tasks for a certain application is slower, or exhibit failures, the laxity value of the application will decrease and thus its priority will increase. For applications with the same laxity value, a simple tie breaking mechanism is used to decide which to schedule first; these applications are treated in a first-come-first-served order.

5.3 MiscoRT Task Scheduler

The goal of the task scheduler is two-fold: First, to ensure that all tasks of the application are scheduled for execution. Second, the task scheduler may dynamically change the number of workers allocated to the application to compensate for failures or queuing delays. If, however, the application completes more quickly than projected, the excess workers can be used by other applications. When a worker becomes available, the MiscoRT task scheduler decides which task to run next, following these steps:

Step 1: The application scheduler determines the application with the smallest laxity value (*i.e.,* this is the application with the highest *priority* in the system). The task scheduler first checks whether any of the tasks of the application have not been assigned yet. The primary insight of this is that an application completes when all of its tasks complete, thus we need to ensure that all of the application tasks are scheduled for execution. During the execution of an application, mobile devices may fail or become unavailable due to spotty connectivity; this can affect the capability of an application to finish within its deadline. In these cases we need to reinstantiate the tasks (these are called *task replicas*) to other mobile devices. To minimize the number of task replicas, we reassign tasks only if we have speculated that the workers executing those tasks have failed or if the task is not progressing as fast as it was estimated, causing the application to miss its deadline.

Step 2: The next step is to check whether the task failed to complete because of worker failures. We term that a task has failed when all of the workers which it was assigned to, are estimated to have failed. When the task scheduler assigns a task t to a worker W_i, it records the $start\_time_{t,i}$ of the task and the *worker_id* processing the task, so that multiple workers can be tracked in the case that the same task is later assigned to other workers. When the task completes, the task scheduler records the $completion\_time_{t,i}$ of the task. It then computes the task processing time, $\tau_{t,i}^j$ and averages it over multiple runs of the task. Note that this time represents the time required for one successful execution of the task. Thus, the task scheduler can estimate that a worker has failed if the time to process the task is significantly higher than the *average processing time* τ_t^j computed from previous runs of the task. This information can be obtained at runtime with low overhead (as we show in our experimental section), or, in the absence of information about previous task executions, we can use a user-defined threshold that can be updated from the current runs.

Step 3: The last step is to check the progress of the assigned tasks. Using formula 4 we can estimate the amount of time required for each task t to complete, as $\epsilon_{t,i}^j = w_i^j - elapsed\_time_{t,i}$. (In the case that a task has been assigned to multiple workers, the minimum $\epsilon_{t,i}^j$ is used). We then consider the task t with the largest remaining execution time, $\epsilon_{t,i}^j$, this is considered as the slowest task and we check whether this task can complete before its deadline. The idea is, that, if the completion time of the slowest task is later than the application's deadline, then with high probability the application is estimated to miss its deadline. This is achieved by performing an execution time projection to examine whether the progress of the task will cause the application to miss its deadline. The task scheduler will also evaluate whether allocating the task to the worker will enable the task to complete within the application's deadline. If this is not possible, then the task scheduler can use the worker for executing other applications in the system.

Note, that, it is possible for the task scheduler to not assign any tasks to the worker even if some tasks are not yet complete. This occurs when tasks have already been assigned to workers and their estimated remaining completion times $\epsilon_{t,i}^j$ is smaller than the time it would take the new worker to complete the task. Assigning an already assigned task to the new worker will most likely lead to duplicated work with no benefit to the application's performance. In our experiments we show that our scheme manages to minimize the duplicate work. Furthermore, note that, under

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

high failure rates where multiple nodes have failed and applications have strict timing constraints, applications can still miss their deadline. In such unstable environments, it might not be possible to find enough resources to run all the applications.

6. EXPERIMENTAL RESULTS

6.1 Experimental Setup

We have conducted an extensive set of experiments to evaluate the efficiency and performance of our scheduling scheme using applications of different sizes and various worker node failure rates.

Our experimental platforms consists of a testbed of 30 Nokia N95 8GB smart-phones [39]. 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 11 applications (mobile tourist application, described below), 8 with 100KB input data and 3 with 1MB input data. We set the deadlines for the applications such that 5 applications have tight deadlines, 2 have medium and 3 have loose deadlines. We derived these deadlines empirically by running the applications and observing their runtimes, the deadlines used were between 100 and 550s. The failure of worker nodes follows a Poisson distribution, as explained in section 5.1; to vary the failure rates of our workers, we adjust the failure arrival rate, λ. Each experiment is repeated 5 times.

To provide a fair comparison, we chose to compare our scheduling scheme with the ***Earliest Deadline First (EDF)*** scheduling policy, which is the most well known and effective real-time scheduler for single processor environments. In EDF, applications are ordered based on their deadlines, the application with the earliest deadline has the highest priority. We paired the EDF application scheduler with a sequential task scheduler. The same experimental parameters such as deadline and data input were used for the comparisons.

We measure the performance of our MiscoRT scheduling scheme using the following metrics: (1) *Miss Ratio* represents the fraction of applications that miss their deadlines and (2) *End-to-end time* measures the time the execution of the entire application set completes.

6.2 Mobile Tourist Application

We have built a mobile tourist application[4] [20], a location-based social networking application, to evaluate the performance of our approach. In the mobile tourist application, tourists seek pictures of other tourists and the places where these were taken in real-time, to identify popular locations in a given geographical area that they visit. Popular locations are identified through the number of pictures taken at these places.

To run the application we have compiled a dataset from the Flickr photo sharing system. Flickr is an example of a social application where users can keep photos of places they have visited. In Flickr each picture is tagged with the location (Latitude and Longitudes) where it was taken along with the corresponding dates and times. Our dataset consists of 50,000 image metadata (8.75 MB) taken from publicly available Flickr photos. We queried the initial seed photos from Santa Barbara and downloaded pictures from these users, the resulting image locations span the globe. This application operates on the user tags found in photo metadata. The application counts the occurrences of each tag and compiles a list of common tags to identify popular picture types.

The map function creates key-value pairs where the key is a tag and the value counts the instances of the key, the value is initially 1. When there are multiple duplicate keys from one map task input, they are grouped together and the values summed before being sent to the server. The reduce tasks add up all the values from the same key and arrive at the most popular tags for photos.

This mobile tourist application is an example of a mobile location-based social application that have recently seen wide adoption and fully exercises all aspects of the system and demonstrates the timeliness of our scheduler. More complex applications mainly differ in the functions performed in the map and reduce tasks and not in the system's execution sequence.

6.3 Performance of MiscoRT

For our first set of experiments, we have evaluated the performance of our MiscoRT by measuring the deadline misses and the end-to-end times of the applications. For this set of experiments, we have set all workers to the same failure rate and vary this failure rate. The reason we chose these metrics is because these are the standard performance metrics in real-time systems. To further establish our choice, we note here, that the reason for employing MapReduce is to satisfy our desiderata, specifically, have a system that offers ease of programmability and ease of adding and managing new resources. Our goal is not to use the additional resources (extra smartphones) to do the work faster and increase throughput, because it is difficult to send data around in a wireless environment, and there are also privacy constraints. Rather, the larger system can, easily and transparently to the programmer, be expanded to handle the additional data that come from the additional resources (smartphones).

Figure 4 shows the application **Miss ratio**. As the fig-

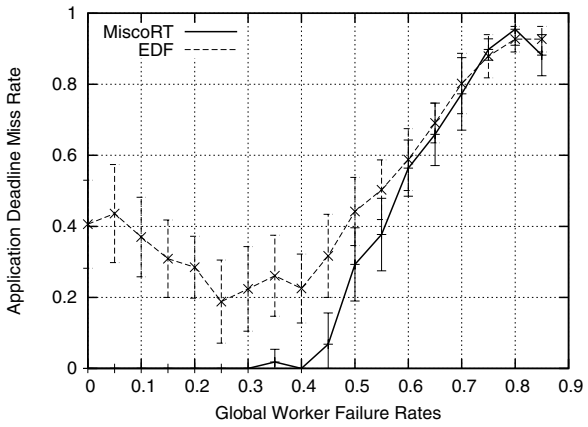

Figure 4: Application miss rate for MiscoRT compared to EDF with uniform distribution of worker failures.

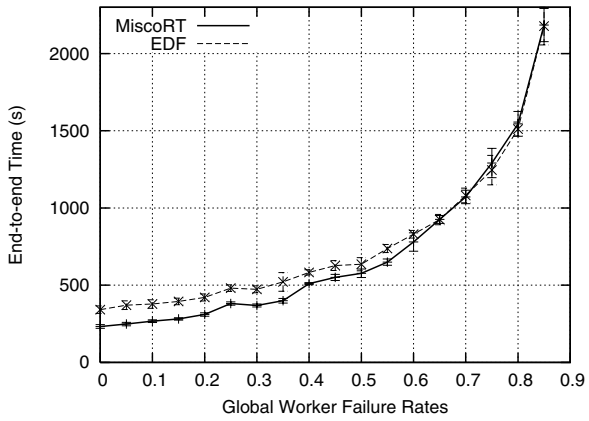

Figure 5: End-to-end times for MiscoRT compared to EDF with uniform distribution of worker failures.

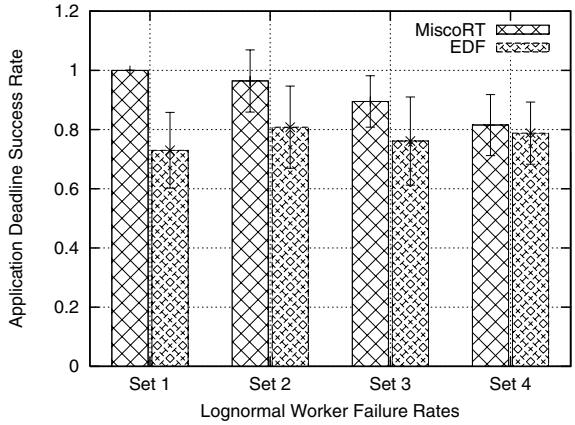

Figure 6: Application success rates for MiscoRT compared to EDF with lognormal distribution of worker failures.

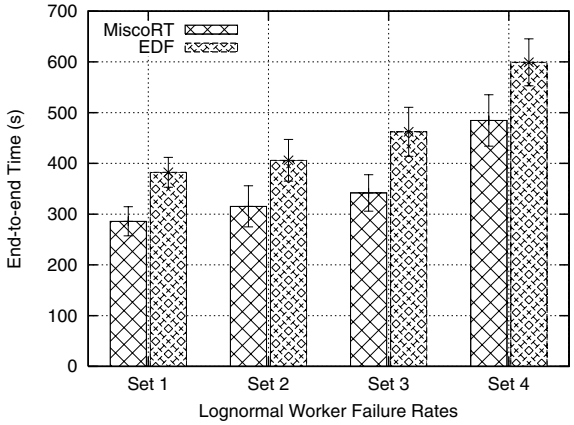

Figure 7: End-to-end times for MiscoRT compared to EDF with lognormal distribution of worker failures.

ure illustrates, for low worker failures, below 40% in this case, there is sufficient resources for MiscoRT to schedule all applications with only a few rare deadline misses. As the worker failures increase further, both schedulers perform very poorly, however, this is expected as very few devices are available to do useful work. The figure shows that at low failure rates the EDF algorithm causes some applications to miss their deadlines. The reason this happens even with small numbers of failures is that with EDF each task takes approximately 50our algorithm results in a much better performance.

The **End-to-end times** of the applications are shown in Figure 5. Our scheduling scheme consistently performs better than the EDF scheduler. At low failure rates, EDF applications complete 47% slower than MiscoRT and as failure rates increase, EDF continues to perform worse than MiscoRT, being 10% slower at 50% failure rates. The reason is that our task scheduler has the ability to adapt to failures by selectively providing redundancy even as failure rates become higher. As the failures increase, the end-to-end time for both schedulers increase exponentially.

In a second set of experiments, we use a log-normal failure distribution among the workers to simulate the failure characteristics from user mobility [31]. The 4 sets of failure rates are shown in table 1. From figures 6 and 7, we see that MiscoRT has a 30% higher success rate than EDF for lower levels of failures and maintains a higher success rate as more failures are introduced. MiscoRT also completes applications faster than EDF by approximately 20% for all our failure sets.

Comparison of MiscoRT with different Task Schedulers

To further illustrate the benefit of our MiscoRT task scheduler, we performed experiments to show our MiscoRT task scheduler's performance compared to alternative task schedulers. We use our MiscoRT application scheduler as the first level scheduler and use different task schedulers as the second level scheduler. We perform the same set of experiments as in section 6.2, but due to lack of space, we only show the results for log-normal worker failure rates. In particular, we used the following task schedulers for comparison:

The random task scheduler is a naive scheduler which

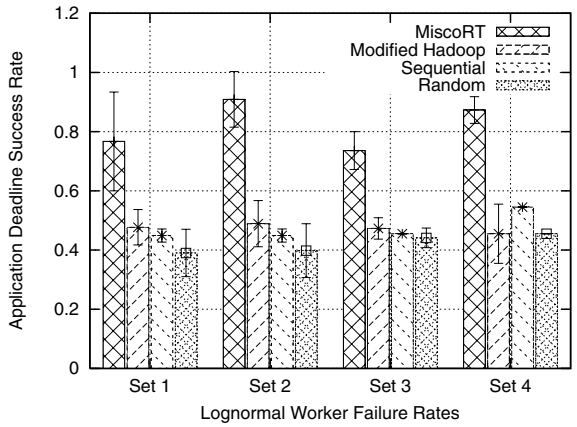

Figure 8: Application success rates for MiscoRT compared to other task schedulers. Each task scheduler was paired with the MiscoRT application scheduler

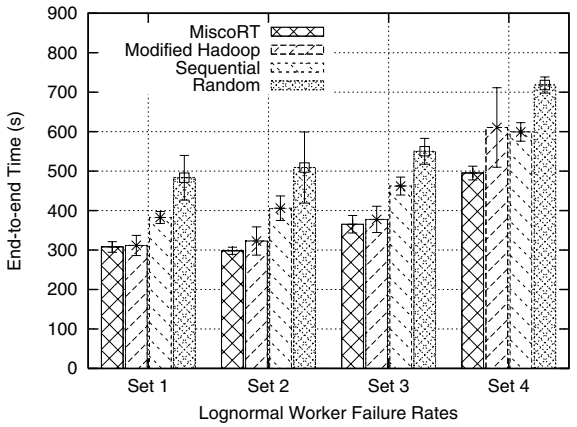

Figure 9: End-to-end times for MiscoRT compared to other task schedulers. Each task scheduler was paired with the MiscoRT application scheduler

Table 1: Log-normal Failure Rates for Workers

Worker	1	2	3	4	5	6	7	8	9	10
Set 1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.5
Set 2	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.8
Set 3	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.8	0.9
Set 4	0.0	0.0	0.0	0.2	0.2	0.4	0.6	0.8	0.9	0.9

picks a random task from the application to execute. The advantage of the random task scheduler is that it has very low overhead, it does not have to store any information about workers, but it wastes computational resources.

The sequential task scheduler is a baseline scheduler which improves on the random task scheduler by reducing the number of duplicate task assignments. It picks tasks in sequential order until they successfully execute. This scheduler has low overhead as it does not keep track of statistics, however, it does not consider worker failures.

The Modified Hadoop task scheduler is based on the FIFO-based task scheduler used by the popular Hadoop MapReduce framework [13]. Hadoop is designed for cluster-based environments where there is constant feedback from the workers informing the scheduler of their progress on tasks. Our system, on the other hand, has limited resources and no fixed infrastructure, thus, implementing such progress tracking would be infeasible. To provide a fair comparison, we follow the spirit of the Hadoop scheduler, however, we attempt to speed up the execution time by re-assigning tasks only when the previous worker is taking more than the average amount of time to complete that task.

The **Miss Ratio** in figure 8 and **End-to-end times** in figure 9 demonstrates that our task scheduler consistently outperforms its competitors. MiscoRT has a 25% to 40% higher success rate than the other task schedulers. MiscoRT has comparable end-to-end application times with the modified Hadoop scheduler when the failure rates are lower, but performs better when failure rates are higher.

6.3.1 Model Validation

In our next experiment, we wanted to validate our model by comparing the predicted execution time for an application (using the model described in section 5.1) with actual measured execution times of the application. For this experiment, we used a single application consisting of 73 tasks. We also assume that all worker nodes fail with the same rate, and we varied the failure rate of the nodes. As figure 10 shows, our model is very accurate, as it's predicted times are very close to the measured times observed from running the application in our system, even at high failure rates.

6.3.2 Scalability

We also wanted to measure the scalability on our system. For this set of experiments, we varied the number of applications in the system, while the failure rates of the phones were set to 0. Figure 11 shows the **End-to-end times**, it increases linearly with the number of applications, as we expected, this is due to us having a fixed processing power and linearly increase the amount of work.

6.3.3 Deadline Sensitivity

In this experiment, our goal was to test the sensitivity of deadline value on the application miss rate. We have varied the tightness of the deadlines by a constant factor, ranging from the original deadlines to 20% of their values, while leaving the other parameters the same. A tighter deadline means that the applications have less time to execute. We report the results when the global worker failure rate was set to 20%, but we have obtained similar results for other failure rates. Figure 12 shows that both schedulers perform worse as the deadlines tighten, but MiscoRT outperforms EDF consistently. When the deadlines are set very tight, at 0.2 of their original values, all applications miss their deadlines when using EDF while only 70% miss their deadline using MiscoRT.

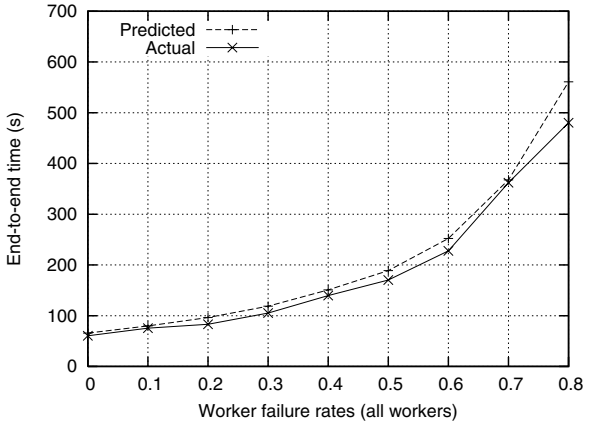

Figure 10: Model validation over various worker failure rates.

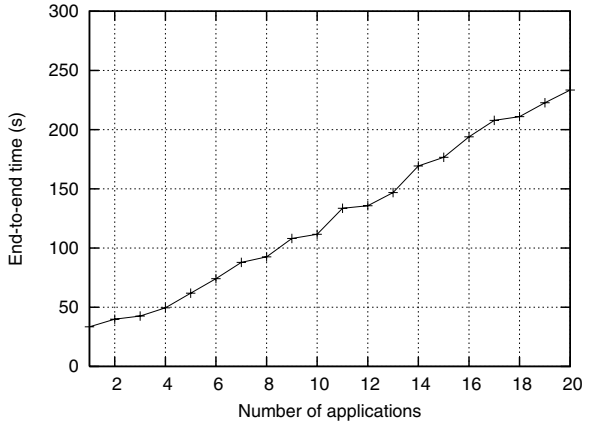

Figure 11: End-to-end time as number of applications are varied.

6.3.4 Overhead and Resource Usage

In this set of experiments, we measure the overhead and resource usage of our system. We monitor the CPU, memory and power consumption using the Nokia Energy Profiler [2]. The MiscoRT system only requires 800KB of memory and our scheduler does not introduce any additional overhead as it is very small, requiring only 150 lines of code. This memory usage is very small compared the 90MB of free RAM available.

As the application runs, the memory usage is dependent on the user's modules and how much of the data is stored into memory. For our applications, during the map tasks, we stored mapping key,value pairs in memory and required slightly more memory than our input data pieces, up to an additional 300KB. We use larger reduce partitions and our additional memory requirements there went up to 700KB.

We also monitored the power usage on the phones. We found that processing data requires 0.7 watts while network access requires more than twice that amount of power at 1.6 watts. It is much more energy efficient to process data locally than to send data over the network. The CPU utilization for tasks is dependent on the application and also on any other programs or program schedulers running on the phone. MiscoRT will gladly use any processing power available to it.

In the last experiment (figure 13) we also measured the overhead of the Master Server. Note, that, the Master Server is responsible to keep track of the user applications and maintain the input, intermediate and result data from each application submitted in the system. In this experiment, the workload we used was an application with 1MB input data; this was split to workers with 100 line pieces for each worker. The figure shows that the Server manages to keep its load below 6% at all times. We also measured the memory usage of the Server. Our measurements indicate that the MiscoRT Server uses 1,572,369 bytes on an Intel(R) Core(TM)2 Duo CPU E8200 (running at 2.66GHz) processor.

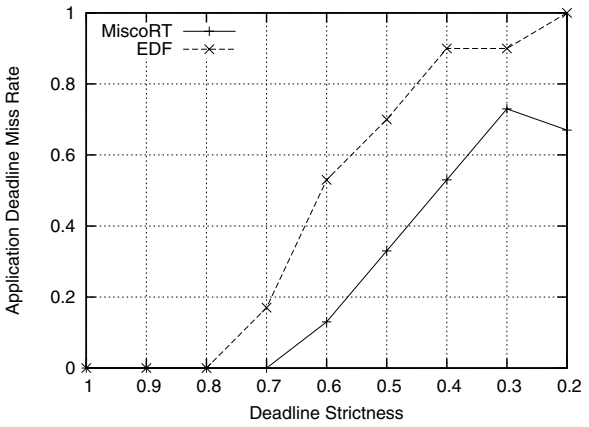

Figure 12: Application miss rates as a function of deadline strictness.

7. RELATED WORK

Recently, many MapReduce systems have been introduced [13] [43] [25] [40]. [25] targets graphics processors and [40] provides MapReduce for multi-core and multiprocessor systems while are concerned with mobile devices. Hadoop [13] is currently one of the most popular MapReduce frameworks, 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 [43] is another framework, targeted at the same environment as Hadoop. Disco provides a single first-come-first-serve scheduler, any excess capacity in the system is then allocated to subsequent applications on the queue. Unlike these works, we consider the problem of meeting the end-to-end real-time requirements of the applications in resource restrictive environments (mobile systems) and under unstable conditions. [37] has implemented a MapReduce system on cell phones and showed that cell phones already provide a significant source of computational power. However, they do not consider any timing concerns.

In our previous work, we have introduced Misco [15], a MapReduce framework aimed at mobile phones, in this work, we have extended the Misco system with a scheduling system to provide support for applications with timing constraints in the form of deadlines.

The problem of scheduling in the presence of failures has

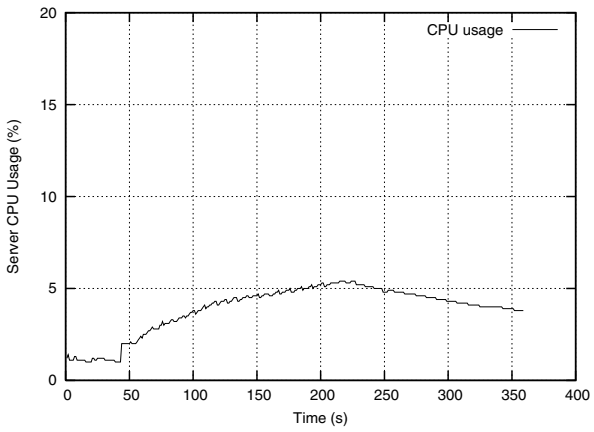

Figure 13: Master Server workload.

been studied in prior works, primarily in cluster-based and distributed system settings [7] [44]. The main methods of dealing with failures are *spatial redundancy*, *temporal redundancy* and *checkpointing*. Spatial redundancy replicates tasks on multiple nodes so that if any of the nodes fail, the execution of the task is not interrupted; however, this scheme requires extra nodes to run the replicas. Temporal redundancy re-executes tasks after they have failed, this has the disadvantage of requiring more time but does not require extra nodes. Checkpointing [19] is used to limit the amount of work lost when failures occur, but the frequency of checkpointing is an important consideration. Scheduling distributed applications in mobile settings with restrictive resources, frequent failures and network fluctuations, which is the problem we are dealing with in this work, is not trivial.

Fault tolerant real-time scheduling is explored in [21] [11] [6] [24] where the goal is to schedule applications to meet hard real-time deadlines. [11] replicates a task K times on homogeneous processors, while [24] considers heterogeneous processors. [21] tolerates one fault per time interval by reserving enough slack in the schedule and [6] uses an AI planner to generate feasible schedules for different possible faults. In these works, schedules are first constructed offline and are limited by the number of failures they can tolerate. Unlike their work, we place no restriction on the number of failures in our system and we are not considering hard real-time requirements. [44] mathematically determines the optimal replication factor for tasks. [17] explores the problem of placing replicas in a way so that the utility of the system degrades gracefully when failures do occur. Probabilistic reliability under failures is explored in [10] to provide the best reliability given a number of resources or find the minimum number of resources for a desired level of reliability. The imprecise computation model, proposed in [8] is based on tasks where precision can be exchanged for timeliness. Our work, on the other hand, provides results based on user supplied modules.

Middleware architectures for mobile users have also been proposed [41]. The primary focus is on concerns such as interoperability, context awareness, network connectivity and server/client APIs. A first come first server scheduling strategy for mobile devices has been proposed in [5]. However, these systems do not consider real-time or fault-tolerant issues. Fault-tolerant and real-time CORBA is explored in [28] [23]; unlike our work they propose light weight replications for task groups on different nodes. [22] proposes a method to split an application across both servers and phones to improve its performance, but only deals with single phone, server pairings. Our work is to improve the performance of a distributed systems of phones.

Social applications [42] [9] [35] for phones have emerged recently, however these are built over specialized systems. Our objective is to make distributed mobile applications easier to develop in order to become widely accessible to users and application developers.

8. CONCLUSION

In this paper, we presented MiscoRT, a system for supporting the execution of real-time applications on networks of smartphones. We propose a scheduling system which considers both the timing requirements of the applications and the failure rates of the worker nodes when scheduling the tasks. This is the first system, that we know of, that proposes a real-time MapReduce scheduler for mobile environments. Through an extensive evaluation on our testbed of Nokia N95 smart-phones, we demonstrate that our system (1) performs effectively, even under failures, (2) has low overhead, and (3) consistently outperforms its competitors. For our future work, we plan to further explore data locality aspects and consider device heterogeneity issues.

Acknowledgment

This research has been supported by the European Union through the Marie-Curie RTD (IRG-231038) project, the SemsorGrid4Env project and the MODAP project, and by AUEB through a PEVE2 project.

9. REFERENCES

[1] Kin: Its nice to meet you. http://kin.com.

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

[3] G.-S. Ahn, M. Musolesi, H. Lu, R. Olfati-Saber, and A. T. Campbell. Metrotrack: Predictive tracking of mobile events using mobile phones. In *IEEE DCOSS*, June 2010.

[4] G. Andrienko, N. Andrienko, P. Bak, S. Kisilevich, and D. Keim. Analysis of community-contributed space- and time-referenced data (example of flickr and panoramio photos). In *IEEE Symposium on Visual Analytics Science and Technology, Atlantic City, NJ, Oct*, 2009.

[5] 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.

[6] E. M. Atkins, T. F. Abdelzaher, K. G. Shin, and E. H. Durfee. Planning and resource allocation for hard real-time, fault-tolerant plan execution. *Autonomous Agents and Multi-Agent Systems*, 4(1-2):57–78, 2001.

[7] H. Aydin. On fault-sensitive feasibility analysis of real-time task sets. In *RTSS, Lisbon, Portugal*, pages 426–434, Dec 2004.

[8] H. Aydin, R. Melhem, and D. Mosse. Optimal scheduling of imprecise computation tasks in the presence of multiple faults. In *RTCSA, South Korea*, Dec 2000.

[9] B. Bamba, L. Liu, A. Iyengar, and P. S. Yu. Distributed processing of spatial alarms: A safe region-based approach. In *ICDCS*, pages 207–214, Washington, DC, USA, 2009.

[10] V. Berten, J. Goossens, and E. Jeannot. A probabilistic approach for fault tolerant multiprocessor real-time scheduling. *IPDPS, Greece*, 0:152, 2006.

[11] J.-J. Chen, C.-Y. Yang, T.-W. Kuo, and S.-Y. Tseng. Real-time task replication for fault tolerance in identical multiprocessor systems. In *RTAS, WA*, Apr 2007.

[12] M. Cinque, D. Cotroneo, Z. Kalbarczyk, and R. K. Iyer. How do mobile phones fail? a failure data analysis of symbian os smart phones. In *DSN*, pages 585–594, Washington, DC, USA, 2007.

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

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

[15] A. J. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, and V. H. Tuulos. Misco: A mapreduce framework for mobile systems. In *PETRA 2010*, Samos, Greece, June 2010.

[16] C. Ebeling. *An Introduction to Reliability and Maintainability Engineering*. McGraw-Hill, 1997.

[17] P. Emberson and I. Bate. Extending a task allocation algorithm for graceful degradation of real-time distributed embedded systems. In *RTSS, Barcelona, Spain*, Dec 2008.

[18] T. Facchinetti, L. Almeida, G. Buttazzo, and C. Marchini. Real-time resource reservation protocol for wireless mobile ad hoc networks. In *RTSS, Portugal*, Dec 2004.

[19] T. H. Feng and E. A. Lee. Real-time distributed discrete-event execution with fault tolerance. In *RTAS, St. Louis, MO*, Apr 2008.

[20] J. Freyne, A. Brennan, B. Smyth, D. Byrne, A. Smeaton, and G. Jones. Automated murmurns: The social mobile tourist application. In *SMW'09 - Social Mobile Web 2009*, 2009.

[21] S. Ghosh, R. Melhem, and D. Mosse. Enhancing real-time schedules to tolerate transient faults. In *RTSS, Pisa, Italy*, pages 120–129, Dec 1995.

[22] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso. Calling the cloud: Enabling mobile phones as interfaces to cloud applications. In *Middleware*, November 30 - December 4 2009.

[23] 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.

[24] S. Gopalakrishnan and M. Caccamo. Task partitioning with replication upon heterogeneous multiprocessor systems. *RTAS, San Jose, CA, USA*, 06.

[25] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *PACT, ON, Canada*, Oct 2008.

[26] T. He, J. A. Stankovic, C. Lu, and T. F. Abdelzaher. SPEED: A stateless protocol for real-time communication in sensor networks. In *ICDCS*, Tokyo , Japan, May May 2003.

[27] P. S. Huan Li and K. Ramamritham. Scheduling messages with deadlines in multi-hop real-time sensor networks. In *RTAS*, pages 415–425, 2005.

[28] H.-M. Huang and C. Gill. Design and performance of a fault-tolerant real-time corba event service. In *ECRTS, Dresden, Germany*, pages 33–42, Aug 2006.

[29] V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser. Dynamic scheduling of distributed method invocations. *RTSS, Orlando, Florida, USA*, Nov 2000.

[30] K. Karenos and V. Kalogeraki. Traffic management in sensor networks with a mobile sink. *IEEE TPDS*, 21(10):1515 – 1530, 2010.

[31] M. Kim and D. Kotz. Extracting a mobility model from real user traces. In *INFOCOM*, 2006.

[32] A. Leon-Garcia. *Probability and Random Processes for Electrical Engineering (2nd Edition)*. Prentice Hall, July 1993.

[33] C. Lu, B. M. Blum, T. F. Abdelzaher, J. A. Stankovic, and T. He. RAP: A real-time communication architecture for large-scale wireless sensor networks. In *IEEE RTAS*, pages 55–66, San Jose, CA, Sep. 2002.

[34] P. Melliar-Smith, L. Moser, V. Kalogeraki, and P. Narasimhan. The realize middleware for replication and resource management. In *Middleware'98*, The Lake District, England, September 1998.

[35] E. Miluzzo, C. Cornelius, A. Ramaswamy, T. Choudhury, Z. liu, and A. Campbell. Darwin phones: the evolution of sensing and inference on mobile phones. In *Mobisys 2010, June 15-18, 2010, San Fransisco, CA*, 2010.

[36] E. Miluzzo, N. D. Lane, K. Fodor, R. A. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *SenSys*, 2008.

[37] S. Mishra, P. Elespuru, and S. Shakya. Mapreduce system over heterogeneous mobile devices. In *SEUS, Newport Beach, CA, USA*, 2009.

[38] L. Moser, P. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki. Eternal: Fault tolerance and live upgrades for distributed object systems. In *Proceedings of the IEEE Information Survivability Conference*, Hilton Head, SC, Jan 2000.

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

[40] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. *HPCA*, 2007.

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

[42] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Madden, H. Balakrishnan, S. Toledo, and J. Eriksson. Vtrack: accurate, energy-aware road traffic delay estimation using mobile phones. In *SenSys*, USA, 2009. ACM.

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

[44] F. Wang, K. Ramamritham, and J. A. Stankovic. Determining redundancy levels for fault tolerant real-time systems. *IEEE Trans. Comput.*, 44(2):292–301, 1995.