UNIVERSITY OF CALIFORNIA
RIVERSIDE

Scheduling in Multiprocess Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Ji Adam Dou

December 2011

Dissertation Committee:

Dr. Harsha V. Madhyastha , Chairperson
Dr. Srikanth V. Krishnamurthy
Dr. Walid Najjar

The Dissertation of Ji Adam Dou is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am grateful to many people who have helped me along the way. I have tried to list as many of them here as I can remember, but there are probably many more.

First, I would like to thank my advisor, Harsha V Madhyastha and the rest of my committee, Srikanth V. Krishnamurthy and Walid A. Najjar for their guidance and advice. I would also like to thank my former advisor and committee member Vana Kalogeraki and Dimitrios Gunopolus, who guided me through the early part of my stint in graduate school, helped direct my research and assisted in my publications.

I am thankful for all my friends who have kept me sane during my stay here, have given me guidance and let me bounce ideas off them: Kyriakos Karenos, Tomas Repentis, Xiaoqing Jin and Chong Ding. I would like to especially thank Song Lin coauthored my first paper, I learned a lot from that experience which helped me through my entire graduate studies.

The text of this dissertation, in part, is a reprint of the material as is appears in: "Data Clustering on a Network of Mobile Smartphones" (Adam Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen, Ville H. Tuulos, Sean Foley and Cutis Yu) 2011, "Scheduling for Real-time Mobile MapReduce Systems" (Adam Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen, Ville H. Tuulos) 2011, "Using MapReduce Framework for Mobile Applications" (Adam Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen, Ville H. Tuulos) 2011, "Misco: A MapReduce Framework for Mobile Systems" (Adam Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen, Ville H. Tuulos) 2010, "RG-EDF: An I/O Scheduling Policy for Flash Equipped Sensor De-

vices" (Adam Dou, Vana Kalogeraki), 2008. The co-authors, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen and Ville H. Tuulos, listed in the publications directed and supervised the research which forms the basis for this dissertation. The co-author Curtis Yu helped perform the energy and resource measurements used in Section 3.5. The co-author Sean Foley helped implement part of the system in Section 3.3.4.

Finally, I would like to thank the administrative and technical staff, who have helped me with all manners of computer system problems, guided me through many administrative minefields and, perhaps more importantly, got me my monthly paycheques: Amy Ricks, Victor Hill, Madie Heersink and Terri Phonharath.

To my parents, Xuefen Song and Rensheng Dou.

ABSTRACT OF THE DISSERTATION

Scheduling in Multiprocess Systems

by

Ji Adam Dou

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2011
Dr. Harsha V. Madhyastha , Chairperson

Applications and the systems that support them are becoming increasingly more complex, more powerful and more ubiquitous. Applications ranging from traffic sensing to online retailers to social networks are part of our every day lives. People have begun to rely on these systems and expect them to perform their operations in a timely fashion. Providing a quick response is critical in some applications and has a significant effect in others. The complexity of modern applications is not the only challenge of meeting user expectations, providing a personalized experience requires the processing of huge amounts of raw data. Although generalized schedulers have shown good results for speeding up many systems, they are not well suited and cannot provide the most efficient solution in many situations.

In this dissertation, I design, build and evaluate several different specialized scheduling system for speeding up and providing quality of service across a range of different systems. I present *RG-EDF*, *Misco* and *MiscoRT* and *Cacheflow*. Starting with low level systems, *RG-EDF* is an efficient storage scheduler for sensor devices equipped with

flash memories. *Misco* is a distributed data processing system and application framework for mobile smart-phones, *MiscoRT* is its failure aware, real-time, application scheduler. Finally, *Cacheflow* is a system for improving client response times in internet scale content delivery networks aimed at social networking sites. Through experimental evaluations, I show that my systems provide a significant performance improvement over generalized scheduling mechanisms and existing approaches.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer systems have radically influenced every aspect of human society by enabling a wide range of applications for business, commerce and entertainment. As these systems continue to progress and become more ubiquitous, smaller, faster, more powerful, their range of utilities will only continue to grow. This increase in capabilities leads to more complex applications and even greater demands on the underyling infrastructure. All types of systems, from low level sensor systems to internet scale networks, are affected.

Not only are systems becoming more powerful, they are also everywhere. Ubiquitious computing is quickly becoming a reality. Sensors and sensor network are used to huge variety of applications we rely on every day to keep us safe, informed and entertained. Applications include monitoring forest fires, landslides, pollution, providing intelligence for military engagements and detecting traffic congestion. Websites and social networks keep us connected to our friends and families. They help connect us to people who share common interests, plan and keep track of events and find recommendations on places to

go. With the increasing capabilities of smart-phones, we can take all this with us wherever we go.

As people are interacting more with these systems in more ways, they have come to depend on timely responses to their queries and requests. In many real-time systems, timeliness is a key requirement for a useful system. Such systems include a video surveillance systems, GPS mapping software, traffic congestion avoidance systems and location based social applications. In many other systems, although not critical, providing a quick response is still very important. For example, even small delays have been shown to cause significant reductions in user traffic and on online sales. With these recent types of complex, personalized systems, a magnitude more data needs to be processed and served. Effective scheduling systems play a big role in alleviating these issues.

Scheduling is a very well studied area and there are many generalized principles that can be applied to many situations. However, such scheduling methods are not efficient or applicable in many settings, especially as new technologies appear. These different application classes with unique system properties impose restrictions on the types of scheduling which will be effective. Specific scheduling techniques which take advantage of underlying system characters is required to achieve the maximum utilization of available resources.

1.1 Sensors

Sensors are low power, low cost devices which are equipped with a variety of sensing capabilities. A typical sensor features a low-frequency, low-power processor, limited memory, a wireless radio for communication, on-chip sensors and an energy source such as a battery. When deployed collaboratively in networks of hundreds to hundreds of thousands of nodes, they are able to support complex applications in many situations where traditional sensing and measurement methods impractical due to hostile environments, remote locations or extended periods of time. Applications include environmental monitoring, military surveillance, inventory tracking and traffic control.

Traditionally, wireless sensor networks (WSNs) senses data and forward the sensed data to a powerful sink node for further processing, only minimal processing and aggregation is performed in the sensors themselves. The use of wireless transmission is a very large drain on the limited power supply.

Flash memories is a relatively new storage technology that provide a low power method of storing large amounts of data. This has led to a new paradigm of storing sensed data, then only processing or retrieving relevant portions when necessary. Low cost, high capacity storage has also enabled much richer types of media such as pictures and videos to be stored. However, many applications are required to perform real-time tasks or respond to queries in a timely fashion. The actual reading and writing of this rich data to flash memory is such a real-time task. In systems with multiple processes all sensing data, the storage subsystem can become a bottleneck, especially in the resource constrained setting of sensors. Efficient schedulers must be developed to support such

intensive real-time input/output (I/O) traffic.

Although there are existing quality of service (QoS) methods for storing data, they cannot be applied directly in this setting. Techniques for hard disk drive (HDD) based systems exploit the physical characteristics of hard drives to provide improved performance: access time affected by the positioning of the data access arm. This particular limitation is not applicable for flash memories, however flash memories do have other constraints and characteristics which must be considered. File systems specific for flash memories work around the constraints by using log based structures, but these are designed for systems with much more resources and are more concerned with providing structures for storing files and efficient garbage collection for freeing space than QoS and I/O efficiency.

1.2 Phones

The rapid adoption of smartphones and their Moore's law abiding progressing in computating power makes them a very attractive platform for distributed applications. There are currently over four billion active cell phone subscribers, with smart-phone being the fastest growing segment in the mobile devices market. The latest generation of smart-phones feature GHz speed dual processors, gigabytes of main memory and tens of gigabytes of persistent storage. Network connection is also expanding and getting faster with IEEE 802.11n, WiFi and 4G networks.

Smart-phones also feature a wide variety of complex sensors such as global positioning systems (GPS), proximity sensors, radio receivers, cameras, microphones, mo-

tion sensors and digital compasses. Even more advanced sensors are being developed for phones, such as those that detect deceases and air pollution. These built in sensors enable phones to be used to provide environmental data such as sound, connectivity, movement, images and pollution. Phones can also collect social information from user input, user interaction, contact lists and user locations.

With the increased smart-phone use, a wide range of applications are being developed, covering personal life, travel, work, commerce and entertainment. Example applications include traffic monitoring for avoiding congestion, location-based services for personalized spatial alarms and social networking applications for sharing photos and personal data with family and friends. Providing real-time, low-latency and scalable execution for these distributed applications presents significant challenges.

Developing simple applications is not too difficult. Developing and deploying distributed applications requires considerably more expertise. There are many roadblocks that complicate distributed applications: concurrency, resource allocation, software distribution, and device and network failures. With relatively limited resources compared to desktop and servers, memory management and application flow is different from traditional applications. There are specialized languages and proprietary system for different types of smart-phones. These factors force new software paradigms, leading to more software defects. Distributed applications takes all these problems and add in multiple mobile devices, introducing coordination and concurrency issues.

Another challenge is the efficient scheduling of work across a set of collaborating devices. MapReduce is proposed as a distributed computation framework for mobile

devices. Since its introduction, MapReduce has grown immensely in popularity and supports a wide array of applications spanning machine learning, simulations and media processing. Although MapReduce started as a framework geared to run on systems in large data centers, it has been successfully implemented for other environments such as graphics processing units (GPUs), shared memory systems and JavaScript clients. It is important to note, that, although MapReduce is chosen as the framework, the types of applications targeted is not the same as traditional data warehouse based MapReduce systems. The limited resources available on the mobile systems makes them unsuitable for any multi-petabyte data processing. Instead, the goal is to explore the use of the framework for monitoring and social networking applications which take advantage of the mobility and personalization of the devices.

Supporting real-time mobile applications is an important step for the wider adoption of the devices and to create opportunities for building new kinds of mobile application services. To date, there are many methods used to support real-time mobile applications. Most of the work has been focused on developing wireless networking protocols or integrating solutions directly with specific network layers. Mobile systems present challenging problems for timely execution 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.

1.3 Internet

Internet adoption has transformed the way people interact through a wide range of applications for business, commerce, entertainment and social networking. Rich and personalized content found at many websites are more demanding on system resources and causing page load times to increase. Social networking sites, such as Facebook, have over 800 million active users, process 3.9 trillion feed actions per day and serves more than 200 billion web pages per month. Keeping page load times low and maintaining user satisfaction is a major concern.

Slow loading web pages has significant adverse effects. Amazon found that they lost 1% in sales for every 100ms of extra latency. Google found that an extra 500 ms load time dropped their search traffic by 20%. A web page's load time is the result of individual latencies along an end-to-end network path. For example, when requesting a website, delays from the workstation, a DNS lookup, a cache miss at a proxy server, the network latency to the web server and processing on the web server all contribute to increasing the client's request response time. Efforts have been made to speed up user experiences at each of these contributing components.

Two methods of reducing load times are caching and content delivery networks (CDNs). Caching is a popular and well studied method of improving network and system

performance on the Internet. By placing copies of objects closer to the user, network latencies are reduced. Caches are very effective at improving performance for static web content where each user requests results in a single item being loaded. CDNs rely on a similar principle of locating data closer to the user, however, they typically duplicate all the contents of the original servers to their distributed nodes and can perform more sophisticated actions such as providing dynamic content. Neither of these solutions are well suited to handling the type of workload imposed by social networking and personalized sites where a single page request can spawn hundreds of back-end item requests on the server.

In order to provide good response times, Facebook sets up large data centers with huge in memory caches to decrease server processing time. In order to store data at the scale that Facebook deals with requires a lot memory. This translates to a lot of servers. Servers are expensive. The energy required to run them is significant.

1.4 Thesis and Contributions

In multiprocess systems, specialized schedulers that take the system characteristics into account can significantly improve the performance of the system over generalized schedulers.

In this dissertation, my contribution spans across several different domains: low level single sensor system, multiple smart-phones system, and a high level distributed Internet based system:

First, Reordering Grouped Earliest Deadline First (RG-EDF) is a scheduling

policy developed to provide efficient quality of service for flash-equipped sensor devices. This system aims at improving storage quality by taking advantage of flash memory characteristics. Requests originating from multiple processes are combined and reordered in a way that provides much better performance than existing systems. RG-EDF is implemented on a CC1010 sensor node with a SD flash card attached. Experiments show the benefits of the system over other schedulers.

Second, MiscoRT is a real-time application and task scheduler for the Misco system. Misco is a MapReduce framework developed for smart-phones. MiscoRT uses a two-level approach to schedule tasks so that applications meet their deadlines. The scheduler incorporates an expected failure model to predict execution times in inherently unstable settings. Misco and MiscoRT are implemented and tested on a testbed of Nokia NSeries smart-phones. Extensive experiment results demonstrate that Misco is efficient, has low overhead and out performs its competitors.

Finally, Cacheflow is a system for reducing client response times and improving server memory utilization in content delivery systems for social networking and personalized services type sites. The system intelligently retrieves items from multiple servers simultaneously, reducing the total number of round trips required. Through experiments performed on PlanetLab, Cacheflow is shown to provide better client latencies using the same amount of memory resources and provides the same client latencies with less memory resources.

1.5 Organization

The rest of this dissertation is organized as follows: Chapter 2 describes the *RG-EDF* system for providing scheduling in flash equipped sensors. Chapter 3 presents *Misco* and *MiscoRT*, a MapReduce system for mobile phones and its real-time scheduler. Chapter 4 describes *Cacheflow*, a technique used to provide better client response times in a CDN-like system. Finally, Chapter 5 concludes the dissertation..

Each chapter follows a similar format. The first section offers a brief introduction and motivates the problem. The second section presents background material and previous related works. The systems settings chapter then presents the assumptions and notation used to describe the problem. System design and implementation details are presented. Following that, experiment results are presented and discussed. A final section summerizes the chapter.

Chapter 2

RG-EDF: Scheduling I/O on Sensors

Wireless Sensor Networks (WSNs), composed of small, low cost and low power sensor devices, have found useful applications in many situations, inclusing those which would otherwise be impractical due to hostile environments, remote locations or if extended periods of time is required. Popular applications include environmental monitoring, military surveillance, inventory tracking and seismic detection. Typical sensor devices feature a low-frequency, low-power processor, limited memory, a wireless radio for communication, on-chip sensors, and an energy source such as a set of AA batteries or solar panels.

The introduction of flash equipped sensors (RISE [9], PRESTO [79]) have significantly expanded the storage capacity of sensors, allowing for the storage of large amounts of data locally. Traditional WSNs collect data and immediately relays it to a centralized,

resource rich sink for later processing. Since the power consumed by transmitting data is a magnitude higher [117] than storing it locally, the sensors can now store and process data, sending only relevant information to the sink in response to preset conditions or queries. Such in-network data storage provides a significant reduction in energy usage and a corresponding increase in the lifetime of the WSN.

As flash-based devices become increasingly popular, they are required to perform real-time tasks, such as the storage and retrieval of multimedia data. Visual (Cyclops [95], CMUcam [23]) and audio (EnviroMic [82]) sensors enable the sensing of high bandwidth, rich data; providing much more information than simple scalar measurements of temperature, humidity, etc [76]. This rich data can supplement the simple measurements with more complete context information. Multimedia data requires large storage capacity and must support intensive real-time I/O traffic. It is common for data generated by the tasks to be sequentially positioned on the flash, while data generated concurrently by different tasks is multiplexed. However, such sequential storage can lead to significant latencies, limiting the ability to satisfy the real-time requirements of the tasks.

Although there are several *quality of service* (QoS) methods for traditional *hard disk drives* (HDD) [12, 69, 81], they do not readily lend themselves for implementation on sensors or for flash memories. Most of the traditional HDD based schedulers exploit the physical characteristics of hard drives to provide improved performance: access time affected by the positioning of the data access arm. This particular limitation is not applicable for flash memories, but flash memories do have other constraints and characteristics which must be considered. File systems specific for flash memories (ELF [29],

JFFS [113], YAFFS [115]) work around the constraints of flash memories using log based structures, but these are more concerned with providing structures for storing files and efficient garbage collection for freeing space than QoS and I/O efficiency, which is the focus of this work.

In this chapter, Reordering Grouped Earliest Deadline First (RG-EDF) is presented as a scheduling policy for flash-based sensor devices. Another version of this scheduling policy without reordering (G-EDF) is also shown to perform well. The policy aims to improve quality by combining multiple requests from the same task and selectively reordering the requests so that several requests can be written on the flash together. The method provides much better performance than current methods of data storage by taking advantage of the unique characteristics of flash memories. The system has been implemented on a CC1010 sensor node with a SD flash card attached. RG-EDF and G-EDF schedulers are experimentally compared with, and shown to outperform, FIFO and regular EDF schedulers.

2.1 Background and Related Works

WSNs [2] are composed of a multitude of low-cost, low-powered, multi-functional sensor devices which are small in size and communicate wirelessly over short distances. Through the collaborative efforts of a large number of such sensor devices, complex applications and improvements over traditionally higher powered, but sparser, sensors are realized. WSNs have found popular applications in many situations including environmental monitoring, military surveillance, inventory tracking and seismic detection.

Although the specific capabilities and resources available vary from sensor to sensor, there is always a need to try to keep their cost low. A typical sensor such as a Mica2 mote [27] consists of a low power Atmega microcontroller (4 or 7MHz, 4KB RAM, 4KB internal EEPROM, 128KB internal flash, 512 KB external flash). Such a sensor also has limited power which it draws from an attached battery.

Traditional WSN data collection and in-network aggregation schemes (TinyDB [85], TAG [84], Cougar [116]) take measurements of the environment and relay the readings immediately to a centralized, resource rich sink, where it is stored for later processing and analysis. This approach requires a large amount of energy for the transmission of data and leads to a lot of waste from transmitting data which is never used. Such an energy inefficient approach is unsuitable for long-term monitoring applications.

The recent trend of equipping sensors with flash memories allows for sensors to store large amounts of data locally. Sensors can now exploit the relatively low energy requirements of local data processing and data storage by only transmitting the processed data results in response to specific queries. Such in-network storage schemes yield significant energy savings since the communication costs are greatly reduced, prolonging the lifetime of the sensor network.

In this section, traditional hard drive based scheduling systems are first introduced, following that, the basics of flash memories and their major properties are discussed and finally some current flash file systems and special scheduling techniques for flash memories are examined.

2.1.1 Hard Disk Scheduling

There is a lot of background work on scheduling systems for mechanical disk based systems. One of the largest source of latency in hard drives is from the positioning and movement of the drive arm. Thus, many traditional schedulers [90] [81] [16] try to optimize the weakness of the storage medium by rearranging the order of operations in to minimize drive arm movement.

Although the majority of these system take advantage of the mechanical nature of HDDs and cannot be directly paralleled in the target environment, many of them do provide useful insights which helped influence the design decisions. In particular, data should be organized in a way to take advantages of the storage system characteristics.

An algorithm for predicting future disk requests and positioning the disk arm in anticipation is presented in [90]. This prediction is made based on the history of of disk requests. Through trace-driven simulations, the authors show an 11 to 23% seek time reduction.

A disk's potential media bandwidth utilization is improved in [81] by filling the rotational latency periods with useful data transfers. Their freeblock scheduling algorithm recovers 20-50% of a disk's potential media bandwidth for background requests without impacting foreground requests.

Two management techniques are proposed in [16] for the disk controller cache that can significantly increase disk throughput: File-Oriented Read-ahead and Host-guided Device Caching. The first technique adjusts the number of read-ahead blocks according to file system information and the second technique gives the host control over

part of the disk controller cache. Trace driven simulations show that FOR and HDC can increase disk throughput by up to 34% and 24% respectively and up to 47% when combined.

A competitive prefetching strategy [78] is proposed in systems where concurrent I/O workloads are interrupting each other. Their proposed algorithm achieves at least half the performance of the optimal offline policy without using any a-priori information. They show that their implementation improves performance by up to 53%.

DiskSeen [32] is a technique which overcomes the inherent limitations of prefetching at a logical file level. The technique performs prefetching directly at the level of disk layout in a portable manner. DiskSeen analysis the temporal and spacial relationships of disk accesses to improve the sequentiality of disk access and prefetching performance. Implementations in the Linux 2.6 kernel shows that it can reduce execution times by 20-53% for micro-benchmarks as well as real applications such as *grep, CVS,* and *TPC-H.*

NVCache [12] uses a small flash memory cache to extend disk spin-down times and reduces a disk's power consumption by up to 90%.

Anticipatory scheduling [69] is a proposed solution to combat deceptive idleness. Deceptive idleness is caused by a scheduler incorrectly assuming that the last request issuing process has no further requests and switches to a handling a request from another processes. This leads to inefficiencies because requests issued by a single process tends to have locality properties on the disk. Using anticipatory scheduling, an Apache web server delivers between 29 to 71% more throughput on a disk-intensive workload, the Andrew file system benchmark runs faster by 8% and variants of TPC-B database benchmark improve

by 2 to 60%. While flash storage does not have the problem of deceptive idleness, the concept of waiting for additional requests is used to increase the overall throughput for flash storage.

2.1.2 Flash Memories

Recent improvements in flash technologies (storage capacity, data access speed, energy efficiency, price) have made it the fastest growing memory market over the past several years. Flash memories are being increasingly used in mobile and wireless electronic devices (e.g. digital cameras, personal digital assistants, cell phones) due to their unique advantages over other storage methods: low energy consumption, large capacity, shock resistance, non-volatile storage, small physical size and light weight.

There are two types of flash memories: NOR and NAND. They differ in the type of logic gates used to store their data. NOR flash has a full address/data interface which allows access of data in byte granularity, but has a longer erase/write time and is more expensive. NAND flash, developed later, has a faster erase/write time, higher density, lower cost and 10 times the endurance. Current flash-equipped sensor devices [9, 26, 79, 31] predominately use NAND flash as an external storage medium because of its many advantages.

The structure of a flash memory consists of many sectors ($4KB \sim 16KB$) which are further divided into blocks ($128B \sim 512B$). The constraints of NAND flash are summarized as follows:

1. Wear: Each block slot in flash memory can only be written a limited number of

times ($\approx 1,000,000$).

2. Read: Data can be read from the flash memory in size ranging from a single byte to a sector ($4KB \sim 16KB$).

3. Write: Data can be written to the flash memory at a block ($128B \sim 512B$) granularity.

4. Delete: a Sector ($4KB \sim 16KB$) is the smallest unit that can be deleted in the flash memory.

From these constraints, several design principles arise: A even level of wear (uniform number of block writes) should be maintained across the entire memory to prevent unexpected bad sectors. These bad sectors can greatly increase a program's complexity in order to handle them correctly. It is important to avoid deleting blocks, as this will cause its entire sector to be delete, if any of the other blocks in the sector should not be removed, they have to be read into memory and then written back after the delete. Writing should be done in as close to block sized pieces as possible.

2.1.3 Flash Memory Systems

The desirable traits of flash memories make them a very good fit for use with sensor devices and there have been several different sensors which use flash memories. A flash-equipped sensor device is composed of five major components:

- **Micro Control Unit**: the core component of a sensor device, it performs all the data processing and computation

- **Sensor Unit**: a sensor device consists of one or more sensor units used to measure various environmental quantities
- **Communication Unit**: provides communication capabilities
- **Power Unit**: typically a battery that provides power for the other sensor components
- **External Flash Access Unit**: provides access to an external flash memory

It is the last unit which distinguishes the flash-equipped sensor devices from traditional sensor devices (such as MICA2 [25], iMote [68] and XYZ [83]); it provides access to an auxiliary data store typically with a size in the range of $32MB \sim 32GB$. All these components are connected together by the Micro Control Unit, which collects data from the Sensor Unit, conducts communication through the Communication Unit and stores the sensor data via the External Flash Access Unit.

PRESTO [79] is a two-tier WSN system which makes use of flash memories in its sensors to provide local archival storage and queries on historical data. The system introduces a prediction based model to reduce transmitted packets by only transmitting values which fall outside a threshold of the predicted values.

RISE-Co-S [9] is a CC1010 sensor device with an attached flash memory model. The addition of this extra storage capacity allows for sensing raw data in a much large quantity than more mundane temperature and humidity sensing. This ability leads to a new sens-and-store paradigm where raw data is processed and stored instead of trasmitted right away.

A device driver for flash memories [72] is presented which supports a conventional UNIX file system. The driver uses an underlying *Log-structured File System* (LFS) to avoid the flash endurance problem. Additionally the system provides a cleaner to reclaim invalid sectors. This system emulates a block device using its underlying flash memories: smaller sector sized write requests are simulated by reading a whole sector, modifying the appropriate parts of the buffer and then erasing and rewriting the entire sector. This method is used to provide a standard file system over the emulated device. This approach is inefficient and leads to insufficient wear levelling.

ELF [29] is an efficient log-structured file system for use in flash memories on sensor nodes. ELF provides memory efficiency, low power operation, data reliability, wear levelling and garbage collection. A key design principle of ELF is to achieve memory and energy efficiency with a small RAM footprint. Its log-structured file system is designed to allow wear levelling by aging all pages accross the flash memory evenly.

JFFS [113] is a journalling file system included in the Linux kernel since version 2.4 designed specifically to handle flash memories. Errors and corruption is avoided by keeping journalled metadata. JFFS provides efficient memory usage by not using any additional layers of indirection to the underying storage system such as a translation table. JFFS was designed for flash-based PDA systems such as the Hewlett Packard iPAQ. YAFFS [115] is another such file system that uses NAND flash instead of NOR like JFFS.

A garbage-collection mechanism is presented in [18] for flash memory system with hard real-time performance guarantees. Their proposed mechanism also supports

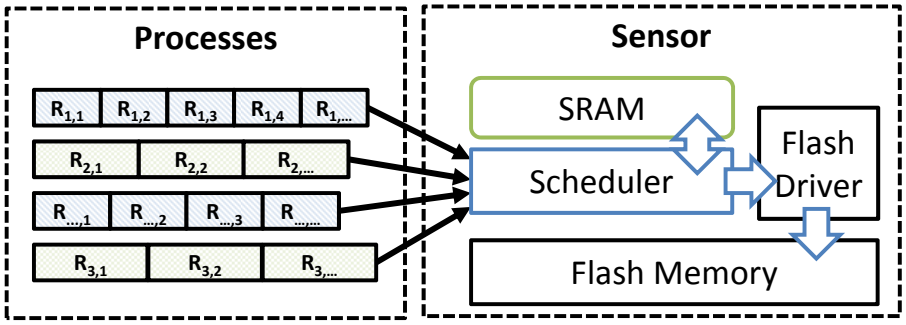


Figure 2.1: The sensor system: each process generates a series of requests, these requests are handled by the scheduler. The scheduler then interacts with the flash driver to complete the requests.

non-real-time so that the bandwidth of the storage system can be fully utilized. However, their techniques had too much overhead to be used in sensors and automated garbage-collection is not a huge concern in systems where data on flash is dealt with at a low level.

2.2 System Settings

The system focuses on the I/O process on a single sensor device. The sensor runs n processes $P_1 ... P_n$, each process P_i produces a series of requests $R_{i,1}, R_{i,2}, Rs_{i,3}, ...$ (refer to Figure 2.1). Each request $R_{i,k}$ can be represented by a tuple $< t_{i,k}, d_{i,k}, size_{i,k} >$ where $t_{i,k}$ is the time the request is issued, $d_{i,k}$ is a deadline (time that the request must complete), and $size_{i,k}$ is the size of the request. The requests are assumed to arrive sequentially. A request $R_{i,k}$ is considered to be written successfully if it is stored to nonvolatile memory (flash memory, in this case) before time $t_{i,k} + d_{i,k}$, otherwise, it is considered to be a miss.

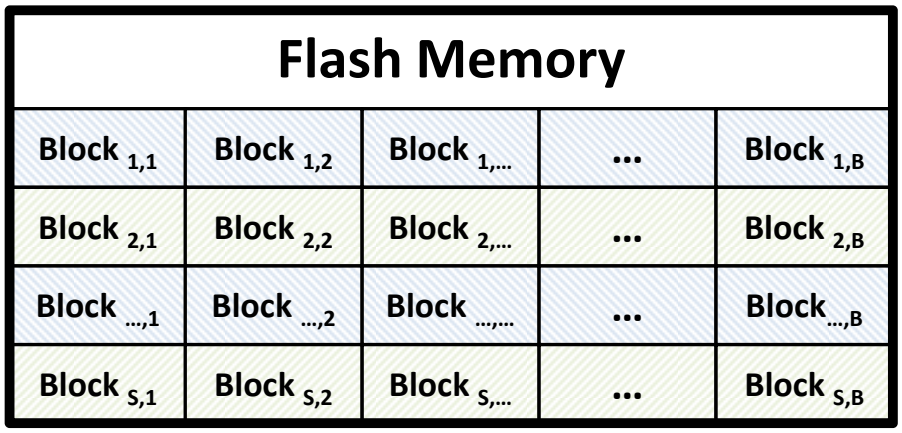


Figure 2.2: The structure of a flash memory with S sectors, each containing B blocks. Blocks are labeled as $Block_{sector,block}$.

The flash memory is divided into S sectors with each sector containing B blocks of size B_{size} (this is shown visually in Figure 2.2. As mentioned earlier, due to the constraints of flash memory: Each write to flash memory takes $write_{time}$, must be exactly B_{size} and must occupy an entire block. Similarly, each read from flash Memory takes $read_{time}$, can read at most B_{size} and cannot cross block boundaries. When a block $b_{i,k}$ in sector s_i is deleted or modified, all other blocks $b_{i,l}$ in s_i must also be deleted (and possibly re-written).

2.3 System Design

To control the order of execution and improve the quality of service, a scheduler is inserted to intercept I/O requests to group and reorder them as necessary. A system diagram for is shown in Figure 2.1.

The scheduling policy design is based on the main observation of the read and

write property of flash memories: all native operations are performed at a block level. When data is written or read from flash memory, requests for data smaller than a block still takes the same amount of time and energy as if an entire block is read or written. Consequently, the system should avoid wasted capacity and maximize utility by trying to read and write only full blocks of data.

This property is exploited in the scheduler by grouping requests and completing them together instead of separately. Grouping requests together improves performance in two ways: by combining multiple requests together, I/O utilization is increased and writing the extra requests is essential *free*. Additionally, by writing multiple requests, the items waiting in the scheduler to be written is cleared out faster and number of drops due to scheduler saturation is reduced. In situations where request injection is bursty, grouping allows many of the requests from the bursts to be written together, making it especially effective in these cases when compared to the simpler, non-grouping schedulers.

2.3.1 Requests

When a request is generated, space is allocated for data in the $SRAM$, and the request object is sent to the scheduler. The request object contains: process id, sequence number, the deadline of the request, its data size and a pointer to the actual data. A pointer to the data is used rather than the storing the data in the request object because this allows for a constant sized scheduler (in memory) even when requests differ in size.

When the I/O component becomes idle and the application is ready to perform the next I/O operation, it will retrieve a request object from the scheduler. Then, a check will be performed to determine if the deadline of the request can be met and the operation

is performed depending on the system's policies (e.g. drop request on miss).

2.3.2 Reordering Grouped EDF Scheduler

Reordering Grouped-EDF (RG-EDF) attempts to avoid performing partial block requests by grouping consecutive tasks together each time the request with the smallest deadline will be served. Performance is further improved by allowing for additional flexibility in scheduling the order of writes.

A Grouped EDF (G-EDF) scheduler was first designed. G-EDF and RG-EDF use the same priority heap structure as the regular EDF scheduler. When a request needs to be retrieved, instead of just returning the top item in the heap, the G-EDF scheduler will search through the heap and try to find sequential requests from the same process to combine.

The request $R_{i,k}$, is at the top of the heap. The scheduler will scan the heap searching for requests $R_{i,k+1}, R_{i,k+2}, ...$ and combine the requests until

$$\sum_{j=0}^{n} size_{i,k+j} > B_{size}$$

or all the items have been searched. The scheduler will then combine and return the requests $R_{i,k}...R_{i,k+n-1}$.

The RG-EDF scheduler improves upon the base G-EDF scheduler by selectively reordering the requests and waiting for new requests before returning a set of grouped requests. RG-EDF checks if better I/O utilization can be achieved by allowing another request with a later deadline to proceed before the current earliest deadline request in cases where the deadline of the requests permit.

Suppose $R_{i,k}$ is the earliest deadline request, a request $R_{j,l}$ is allowed to proceed ahead if the service time for $R_{j,l}$ can be accurately predicted:

$$t_{i,k} > current_{time} + write_{time} + buffer_{time}$$

$$\sum_{a=0}^{n} size_{i,k+a} < \sum_{b=0}^{m} size_{j,l+b}$$

where

$$\sum_{a=0}^{n} size_{i,k+a} \leq B_{size} and \sum_{b=0}^{m} size_{j,l+b} \leq B_{size}$$

This allows for the group of requests $R_{j,l}...R_{j,l+m}$ which occupies a larger portion of a block to proceed before $R_{i,k}...R_{i,k+n}$. An additional benefit is that this provides the opportunity for more requests from process i to arrive while the I/O operation for process j is underway.

2.4 Implementation

The systems have been implemented on a CC1010 sensor with a SD flash card attached through the *Serial Peripheral Interface* (SPI). This section discusses the main components of the CC1010 sensor, the interface and characteristics of the SD flash card and gives implementation details on the different schedulers. An implementation system diagram is shown in Figure 2.3.

2.4.1 CC1010 Sensor

The sensor used is a Chipcon CC1010 sensor with an 8051 Enhanced Microcontroller [21]. The main features which are relavent to the implementation are the CC1010's memory model and its interrupt timers.

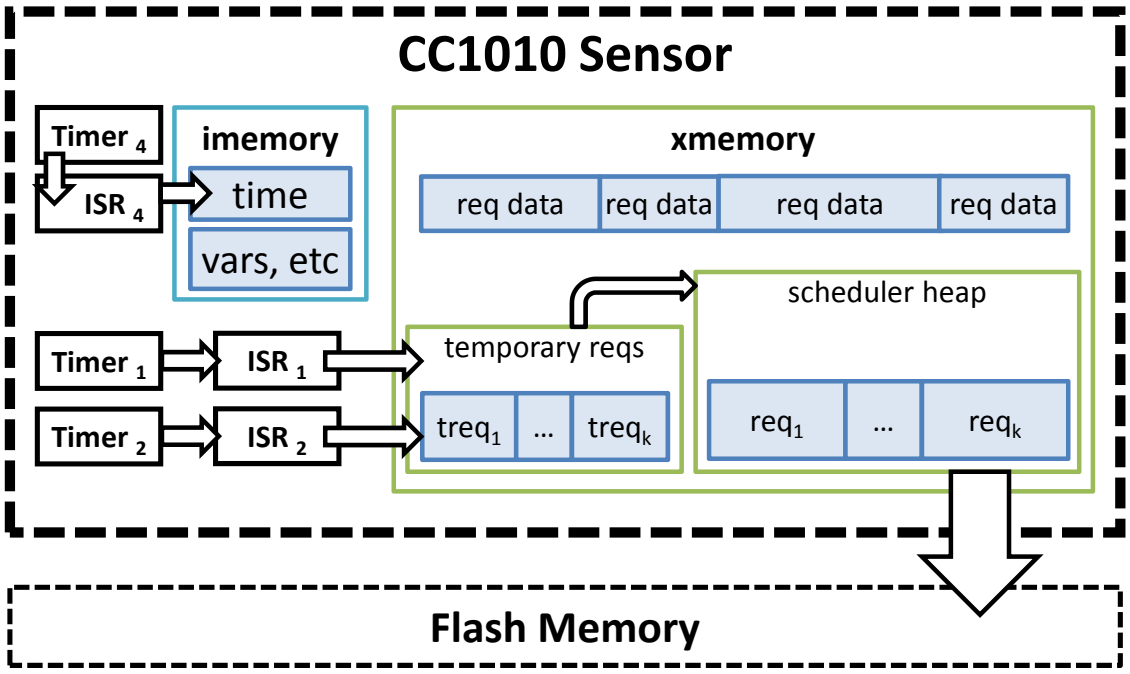


Figure 2.3: Architecture of the scheduler.

The memory is divided into two main sections: an internal memory (imemory) of 128 bytes and an external memory (xmemory) of 2024 bytes. Since the imemory is faster than the xmemory, the most frequently used and time sensitive data items, such as the loop and time counters, are placed here. Xmemory is used for data structures which require larger amounts of memory, such as the queue and scheduler heap.

2.4.2 SD Flash Card

The flash memory used was a 128 MB Sandisk SD card with 512 byte blocks arranged in 256 block sectors. There are two basic operations supported by the SD card:

reading a block and writing a block. Any erasing and recopying of information in a sector are handled internally by hardware on the flash card itself. The custom flash driver implements the two basic read and write operations.

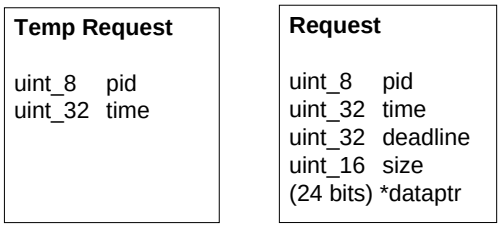
The flash card was connected to the sensor using a SPI bus. While this does limit the speed of reading and writing to the card, it also greatly simplifies the hardware interface. Both of the basic operations mentioned above must be completed in blocks of exactly 512 bytes. To perform an operation, the command is first placed onto the bus, followed by the address of the block. The serial interface is then polled to either read or write data to and from the card. In addition, after a write completes on the sensor, there is an additional wait time while the card finalizes the operation.

Write time for a block to flash requires 9 ms, but reading from xmemory and writing the data typically takes up to 13 ms for a full block of 512 bytes. For the first write into a new sector, an extended write time ranging from 60 to 140 ms is required, this is a characteristic of the internal hardware of the SD flash card. This extra time is used to reorganize the data internally and speeds up subsequent writes. Reading a block takes slightly more time than writing ranging from 11 ms to 15 ms.

2.4.3 Timing and Request Injection

The CC1010 sensor provides 4 interrupt timers ($timer_1$ - $timer_4$), 3 of which can be used. $timer_4$ is set as the highest priority interrupt for keeping a counter to measure elapsed time in 1 ms increments. Multiple processes issuing requests are simulated by using the *interrupt service routines* (ISRs) associated with $timer_1$ and $timer_3$.

To keep the interrupt handlers as compact as possible; a full request object is not



(a) Data Structure of Request in temporary request queue

(b) Data Structure of Request in schedulers

Figure 2.4: RG-EDF data structures

created at the time of an interrupt. Instead, a temporary request (treq) object is placed into the *temporary request queue* (TRQ), and a full request object is created between I/O operations. The treq structure is shown in Figure 2.4(a) and contains the timestamp at the time the request is injected. Between I/O requests, complete request objects (shown in Figure 2.4(b)) are created for any treqs in the TRQ and are inserted into the scheduler. Without using temporary requests, the actual process of generating full requests was found to have a significant detrimental effect on the overall system performance.

Request injection is handled by the ISRs invoked by $timer_1$ and $timer_3$. The period that each timer fires its interrupt can be adjusted and is used to vary the request injection frequency. When a ISR is invoked, a treq is created for each process and placed into the TRQ. $timer_1$ is always used to control process 1 and $timer_3$ is used to control processes 2...n where there are n total processes. $timer_1$ can be adjusted independently of $timer_3$ for experiments where the frequency of only one process is varied.

Bursty request injection is modeled by having the ISR inject multiple requests

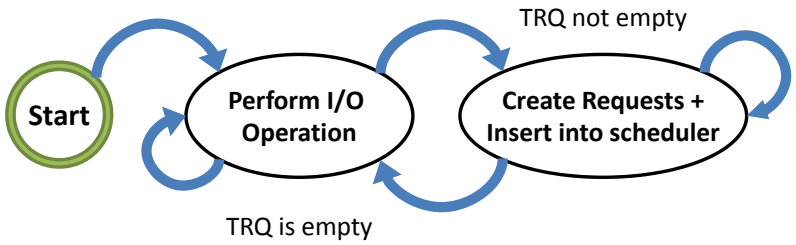


Figure 2.5: RG-EDF application states

instead of a single request. For example, on every 10th run of the ISR for $timer_1$, 5 requests are injected for process 1 instead of a single request.

2.4.4 Scheduler Implementation

The application has two main states (shown in Figure 2.4.4): one state where I/O operations are being performed and another when requests are generated and insert into the scheduler from the TRQ. When the TRQ is empty, any temporary requests are transformed into full requests, when the TRQ is not empty, storage operations are performed.

Two classes of schedulers are implemented: a FIFO queue and a EDF scheduler. The RG-EDF and G-EDF schedulers are built on top of the EDF scheduler base. The schedulers also have the ability to drop requests which are determined to miss their deadlines.

FIFO Queue

For a FIFO queue, the TRQ is used directly. The TRQ is implemented as a circular queue in an array of treq objects. When this queue becomes full, no further requests are accepted and these requests are counted as dropped requests. In the I/O phase, we

simply retrieve the first treq object from the queue, generate a full request object and perform the I/O operation.

EDF Schedulers

The base EDF scheduler is a priority queue implemented as a binary heap. The data structure of each request in the heap can be seen in figure 2.4(b). The time a request is generated is only tracked for statistical purposes and can be optimized out.

By keeping the size of each request object constant, a simple array-based heap structure can be used. The top item in the heap will always be at index 0 and the child nodes for an item at index i would be at $i * 2 + 1$ and $i * 2 + 2$. In addition to being able to remove items from the top of the heap, an item can be removed from the heap by specifying its index.

Building on top of this base EDF scheduler heap, functions are introduced to group and reorder requests. When a request is made, a list of indices from the heap is returned containing requests which can be grouped together. Once the application receives the list of indices, it then proceeds to retrieve, and remove from the heap, each request, starting with the last index (this ensures that the remaining heap indices remain valid). After retrieving all the requests in the group, a single I/O operation can be performed.

Knowing the typical time required to write a request to flash, the scheduler can allow requests which have later deadlines proceed ahead of requests which have an earlier deadlines, if the earlier deadline is not violated. The scheduler compares the grouped size of the top request with the grouped size of the next request from a different process. If the second group of requests has a larger total size, the second group of will be allowed to

proceed first. This improves the I/O utilization and also allows more requests from the first process to enter the scheduler while the I/O operation is underway.

2.5 Results

2.5.1 Experiment Settings

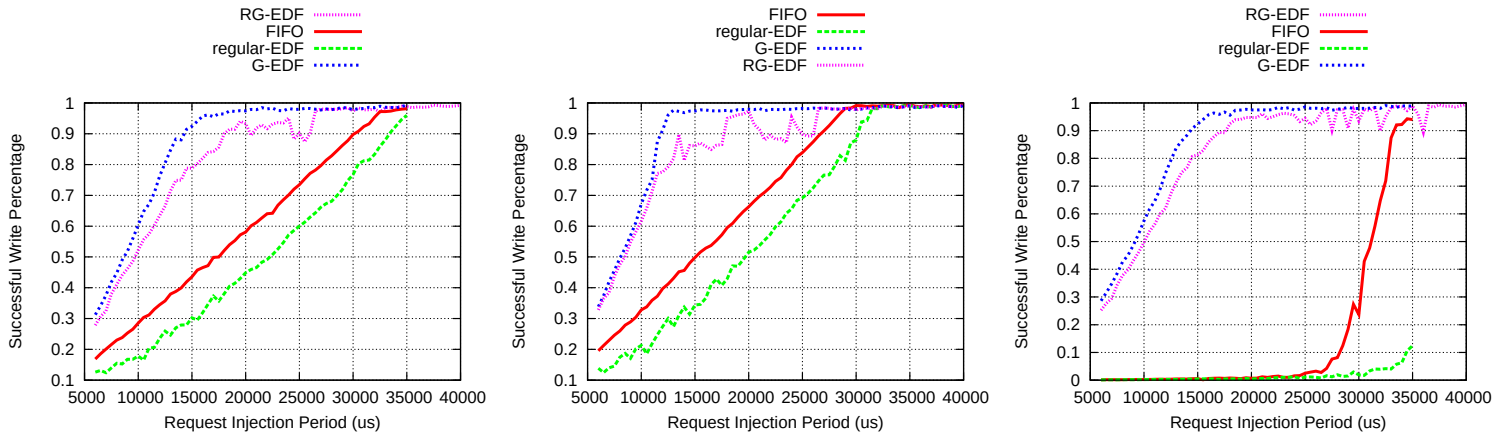
The performance of RG-EDF is compared with the traditional FIFO approach, a regular EDF implementation and G-EDF. Three data flows are used in the experiments. Each data flow, i, is defined by the period of the request injections, the request sizes ($size_{i,k}$) and the deadline delay ($d_{i,k}$) for each request.

The schedulers are evaluated by varying a single parameters over a range of values. Conservative default values were chosen so that each of the schedulers perform well at these values:

- injection period: 40 ms
- request size: 64 bytes
- deadline delay: 70 ms

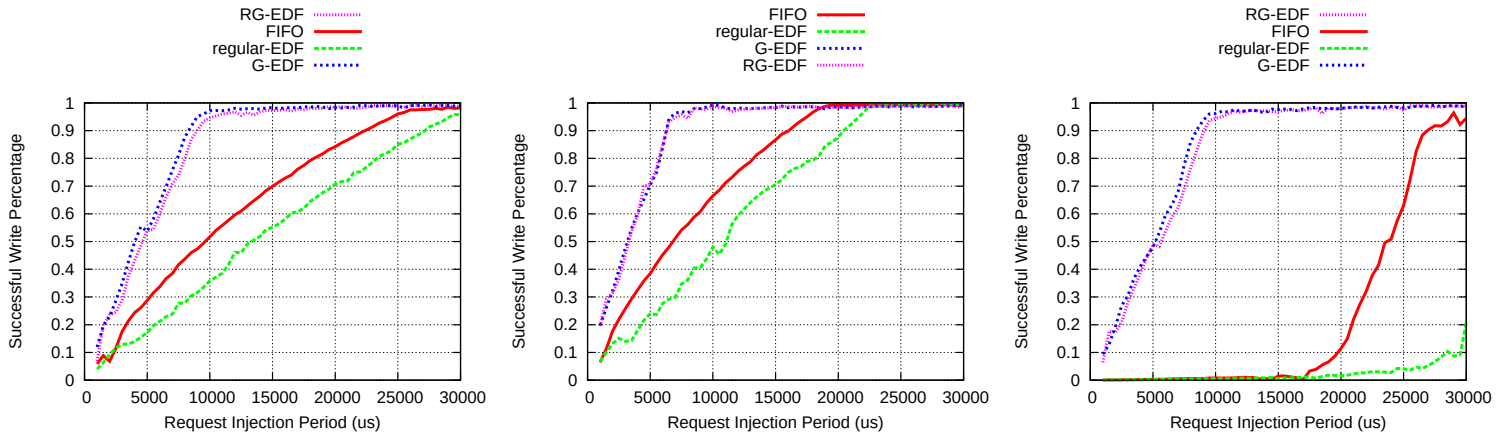
In addition to varying these parameters, the effects of allowing schedulers to drop requests is also measured. In these cases, requests can be dropped if it is determined that their deadlines cannot be met. The performance of these systems in the presence of bursty traffic is also examined. For bursty requests, instead of a single request, 5 requests are injected for process 1 every 10 injection cycles.

For each experiment, 10 sets of 10 second runs were completed and the results averaged. The results are also aggregated across all three processes running.



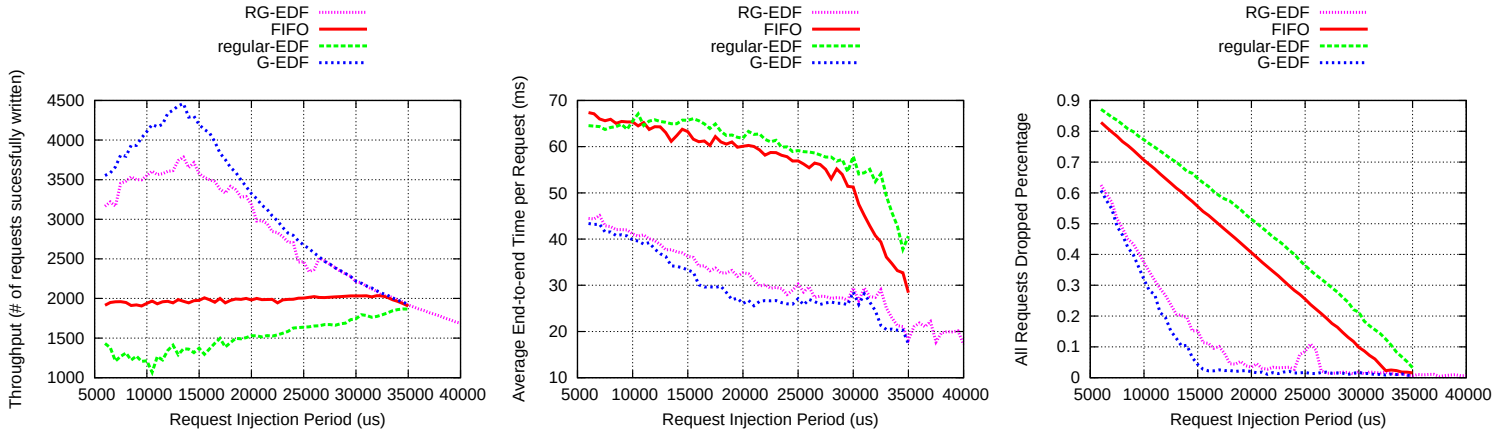
(a) Bursty requests with dropping misses (b) Non Bursty requests with dropping misses (c) Bursty requests without dropping misses

Figure 2.6: Varying period of request injection for all 3 processes with default deadline delay ($d_{i,k} = 70$ ms) and request size ($size_{i,k} = 64$ bytes).



(a) Bursty requests with dropping misses (b) Non Bursty requests with dropping misses (c) Bursty requests without dropping misses

Figure 2.7: Varying period of request injection for 1 of 3 processes with default deadline delay ($d_{i,k} = 70$ ms) and request size ($size_{i,k} = 64$ bytes).

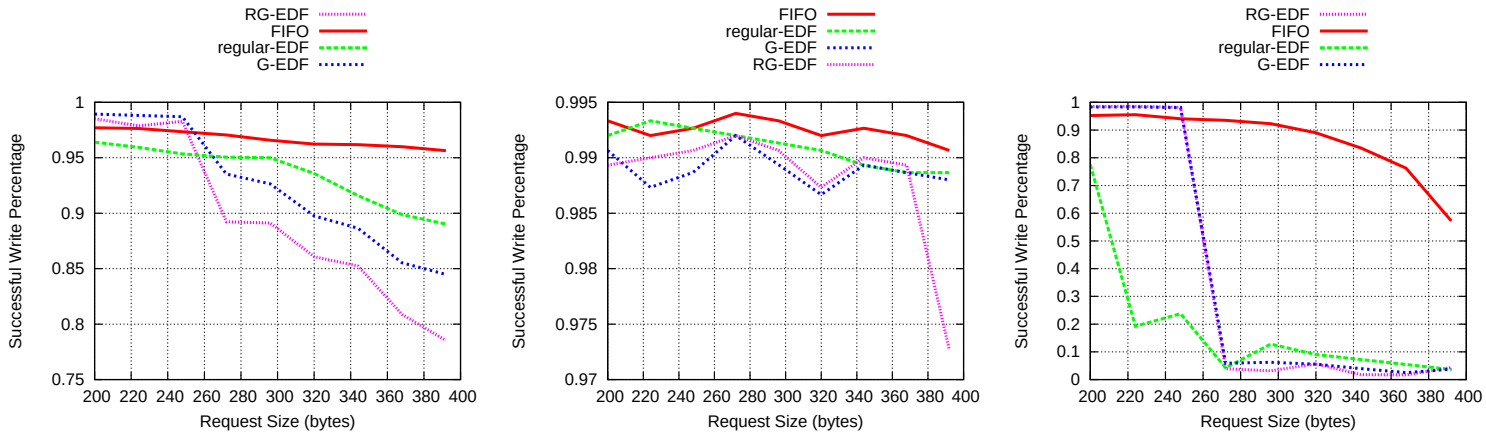


(a) Throughput in number of requests successfully written

(b) End-to-end time each request spends in the system

(c) Total # of dropped requests due to full scheduler and dropping misses

Figure 2.8: Varying period of request injection for all 3 processes with default deadline delay ($d_{i,k} = 70$ ms) and request size ($size_{i,k} = 64$ bytes).



(a) Bursty requests with dropping misses (b) Non Bursty requests with dropping misses (c) Bursty requests without dropping misses

Figure 2.9: Varying request size for all 3 processes with default injection period 40 ms and deadline delay ($d_{i,k} = 70$ ms).

2.5.2 Request Frequency

The frequency is varied for all three process at the same time. Since the deadlines for all the request are the same and the periodic requests are injected at the same time, the EDF scheduler is acting exactly like the FIFO scheduler, except the EDF scheduler has the additional overhead associated with the scheduler, leading it to perform worse. The G-EDF and RG-EDF schedulers share the same overhead as the EDF scheduler but is able to compensate by grouping requests together, leading to much better performance. Figure 2.6 shows that G-EDF performs better than RG-EDF because there are not many opportunities for reordering and the extra overhead of reordering logic is demonstrated.

Figure 2.6 (a) and (b) show the difference in performance between bursty and non bursty traffic, all schedulers perform better when the traffic is non bursty, due to there being less requests overall, but G-EDF and RG-EDF handle bursty traffic better due to their ability to group these requests together.

When dropping misses is disallowed (compare Figure 2.6 (a) and (c)) by allowing requests to complete even when they are going to miss their deadlines, FIFO and EDF performance drop off rapidly due to missed deadlines causing more misses. When drops are enabled, FIFO and EDF perform much better since they only have to perform on-time requests. Further, figure 2.8(c) shows that the majority of the drops for G-EDF and RG-EDF are due to scheduler saturation rather than being dropped due to misses.

Figure 2.7 shows the results when only varying injection period for a single process. FIFO and EDF perform better because there are less results in total to deal with: while the injection period of the single process is increased, the other two processes

are injecting in 40 ms periods. G-EDF shows a modest amount of performance gain while RG-EDF has many more reordering opportunities and shows improvement over Figure 2.6. As the injection rate for the single process increases, even more reordering opportunities present themselves. However, this reordering still does not fully compensate for the extra overhead from the computations required by the RG-EDF scheduler.

Figure 2.8(a) shows the throughput of each scheduler as the injection rates are increase. The FIFO scheduler has a constant throughput; it can write a constant number of request independent of how many are injected. EDF suffers from the scheduler over head and is unable to keep up. G-EDF and RG-EDF both perform well until scheduler saturation causes drops. Figure 2.8 (b) shows that requests spend much less time in the G-EDF and RG-EDF schedulers.

2.5.3 Request Size

In this set of experiments, the request sizes are varied. From the results (Fig 2.9), there is a sharp drop in performance from the grouping schedulers. This occurs when the request size increases above 256 bytes; beyond this point, the scheduler can no longer group multiple requests together because the block size is only 512 bytes.

G-EDF and RG-EDF rely on grouping multiple requests together to improve performance. These schedulers introduce extra sophistication and overhead; if requests are large and grouping not possible, they act like the EDF scheduler and the extra overhead will cause the grouping schedulers to perform poorly.

2.6 Discussion and Future Work

One of the key issues is the amount of processing power available. From the experiment results, it is clear that the processing overhead is responsible for many dropped and delayed requests. Comparing RG-EDF with G-EDF, the more sophisticated scheduler has more optimizations for increasing throughput and should outperform the simpler one if computation time was not a concern. However, this is not the case, the processing time for even relatively simple tasks of searching through a heap for requests from a process require significant computation time. With faster processors, this problem should be alleviated and even more complex schedulers may be supported.

A second issue when buffering requests to be written is the amount of system memory available. With only 3 times the block size available for buffering, there is little flexibility for multiple applications and complicated task keeping data structures. With more memory, it may be possible to perform more complicated scheduling. Another area of research which can benefit from more memory is in providing complicated data query capabilities for flash equipped sensors [36]. Scheduling in these systems with more complicated indexing structures for storing data in flash memories can lead to better performance when performing queries.

Techniques for lowering energy requirements of storing data on flash memories can be applied to the system. For example, [101] presents an interesting study on how to write data to flash memories using lower than regular voltages. However, the probability of the data being successfully written is lowered. This probability of failure would need to be factored into any scheduling system in order to provide QoS.

2.7 Summary

Sensor applications are become increasingly complex: requiring multiple streams of data storage and taking rich measurements such as visual and audio data which are frequently bursty. In this chapter, the RG-EDF and G-EDF scheduling policies are introduced and shown to provide excellent performance in many cases where simple schedulers are insufficient. RG-EDF achieves superior results by taking into consideration and taking advantage of the unique characteristics of flash memories by grouping requests together and re-ordering them.

The design and implementation of RG-EDF and G-EDF is presented and experiments show up to 225% improvement in throughput in some cases over a traditional FIFO scheduler. Experiments also show that even relatively small calculations require significant processing times on sensors and a less sophisticated scheduling system can perform as well or better than a more sophisticated one.

Chapter 3

Misco-RT: Scheduling Applications on MapReduce

Smart-phones are everywhere. There are currently over four billion active subscribers and that number of subscribers continues to grow. Smart-phones are also becoming more powerful. The latest generation of smart-phones boasts 1 GHz Snapdragon processors, 512 MB of main memory and 32 GB of persistent storage [64]. Network connectivity is also expanding and getting faster with IEEE 802.11n, WiFi and 4G cell networks. In addition to their increasingly powerful resources, smart-phones are being equipped with motion sensors, proximity sensors, FM radio receivers, Global Positioning System (GPS), digital compass, cameras and microphones.

All these factors combine to make smart-phones a very lucrative platform for developing distributed applications. Phone sensors provide environmental data such as sound, connectivity, movement, images and social information from user input, user inter-

action, contact lists and locations. Even more complex sensors are being developed and embedded into phones, such as sensors that detect deceases [15] and air pollution [63].

With increased smart-phones adoption, researchers are investigating new applications in a wide variety of domains covering personal life, travel and work. Examples include traffic monitoring for real-time delay estimation and congestion detection, location-based services such as personalized weather information, location-based games, spacial alarms upon the arrival to a predetermined location, and social networking applications for sharing photos and personal data with family and friends. Providing real-time, low-latency and scalable execution for these distributed applications presents significant challenges in mobile systems.

While developing simple applications on smart-phones is not too difficult, development and deployment of distributed applications requires much more expertise. There are many factors leading to the complicated nature of distributed applications: concurrency, resource allocation, software distribution, and device and network failures. 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 applications, forcing new software paradigms, leading to many software defects. Distributed applications involving multiple mobile devices exacerbate the problem by introducing additional concurrency issues.

Another challenge is the efficient scheduling of work across a set of collaborating devices to accomplish a task. MapReduce is proposed as a distributed computation framework for mobile devices. Since its introduction, MapReduce [30] has grown immensely in

popularity, supporting a wide array of applications spanning machine learning, simulations and media processing [20]. MapReduce is being heavily used by prominent companies such as Google, IBM, Yahoo and Facebook to tractably process their large amounts of information. Although MapReduce started as a framework geared to run on systems in large data centers, it has been successfully implemented for other environments such as Graphics Processing Units (GPUs) [60], shared memory systems [96] and even Javascript clients on browsers [59]. There have been some recent explorations with implementations of MapReduce on smart-phones [86, 40, 33, 34, 35].

It is important to note, that, although MapReduce is chosen as the framework, the types of applications targeted is not the same as traditional data warehouse based MapReduce systems. The the limited resources available on the mobile systems makes them unsuitable for any multi-petabyte data processing. Instead, the goal is to explore the use of the framework for monitoring and social networking applications which take advantage of the mobility and personalization of the devices. Furthermore, the MapReduce framework's support for the weak connectivity model of computations across open networks, makes it suitable as a framework for smart-phones and mobile devices.

While the MapReduce framework provides the building blocks for developing mobile applications another major challenge remains: how to provide support for applications that require performance guarantees. To date, most of the work on supporting real-time mobile applications has focused on developing wireless networking protocols [61] or integrating their solutions within specific MAC or network layers [65] [45], to encapsulate application-specific trade-offs in terms of resource constraints, shared wireless

medium, lossy communication, and highly dynamic traffic. 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.

Misco and *MiscoRT* are presented in this chapter. Misco is a MapReduce framework developed for smart-phones and *MiscoRT* is a scheduling system aimed at supporting the execution of real-time application tasks on mobile MapReduce systems. A two-level approach is used for scheduling distributed real-time mobile applications. An analytical model to estimate the execution times of the applications in the presence of failures is first developed. Using this model, the system determines the application urgencies based on estimates of their execution times under failures and their timing constraints. The 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, MiscoRT can adjust the fault-tolerance characteristics of the overall system. This is a major motivation for using

the MapReduce model in mobile environments which are typically inherently unstable. Furthermore, these methods developed for heterogeneous and unstable mobile environments of today may be useful in extremely loosely coupled computing environments of tomorrow, which will not be confined to a single, well-controlled data center. Misco and MiscoRT have implemented and tested on a testbed of Nokia's third generation NSeries phones [92]. Extensive experiment results demonstrate that the approach is efficient, has low overhead and completes applications up to 32% faster than its competitors.

3.1 Background and Related Works

3.1.1 Smart-phones

Smart-phones are becoming increasingly powerful and are the fastest growing segment in the mobile devices market [54]. A look at the evolution of successive generations of iPhones show that their memory and storage has been doubling every year and their processing power has doubled in two years. In terms of these resources, the progression in smart-phones appears to follow Moore's law. Network speeds have also been increasing with recent migrations from 3g networks which provide 0.2 14 Mbps to 4G networks which provide 0.1 1 Gbps [99].

Recently, there has been a lot of attention on using the phones as a low-cost alternative for more specialized machinery, for example, to measure pollution and detect decease. The N-SMARTS [63] project aims a building a distributed system of phones for monitoring the environment. One of their research areas is the development of Micro-electromechanical Systems(MEMS) based particulate mass sensors which can be embed-

ded in phones [15] have built a mobile phone mounted light microscope which can be used for light microscopy, a simple, cost-effective and vital method for diagnosing and screening hematologic and infectious diseases. These efforts in creating more powerful and low cost sensors for smart phones are only expanding the phone's capabilities as a mobile sensing platform.

Currently, developing applications for smart-phones is not an easy ordeal and there are major obstacles to develop for any of the big three smart-phone platforms: Symbian, iPhone and Android. First, each vendor provides platform specific tools that must be used to develop for and deploy on their devices. Each phone also has its own unique program life cycle and flows. This means that the way memory is managed, how a program starts and stops and how interrupts are handled are different for each phone and vastly different from developing on more traditional desktop and server environments. In order to write efficient code for Android devices, object instantiation and internal getters and setters must be avoided and *virtual* is preferred over *interface*. Android also had a slightly different implementation of the Java Virtual Machine (JVM) and shell tools which are incompatible with desktop or server JVMs, making it difficult to port existing Java applications. For Symbian and iPhone applications, expensive developer certificates and accounts are required in order to install application. Being a closed system, iPhone provides application installation and log viewing within the XCode Integrated Development Environment (IDE) but does not support execution of arbitrary shell commands or the creation of files. The SymbianOS is currently undergoing an open sourcing effort but as for yet has not made too much progress.

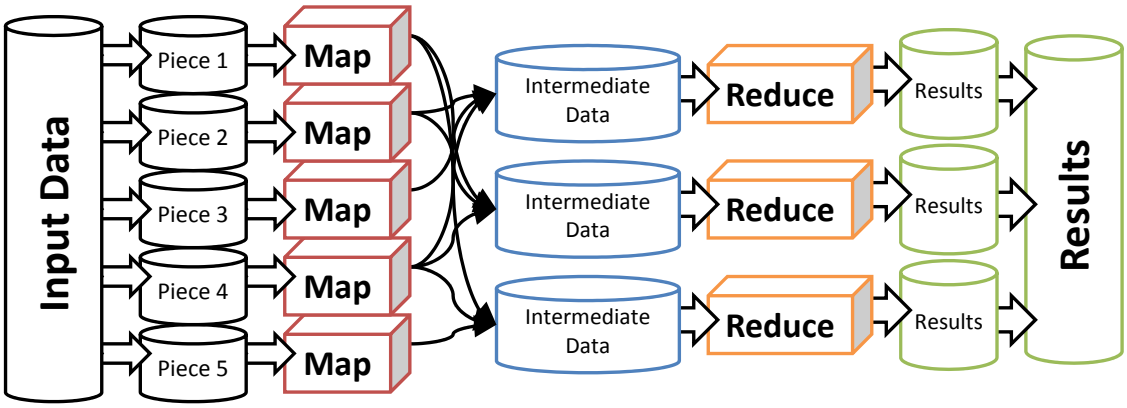


Figure 3.1: A simple visual representation of the map reduce model.

Even after a developer has gone through the effort of learning the new programming paradigm for smart-phones and acquired the tools and software certificates, writing distributed applications is still not an easy undertaking. There are many things to consider when creating a distributed application: concurrency issues, communications, and device and network failures are among a few of the many issues. One of the major motivations for developing a MapReduce framework that can be used on mobile devices is to address this difficulty in developing distributed applications on smart-phones. The MapReduce framework provides a flexible platform for developing applications without having to worry about the underlying distributed nature of the application.

3.1.2 MapReduce

The MapReduce [30] framework is a computational programming model originally introduced to support distributed computing on very large data sets on clusters of computers. The system is inspired by the map and reduce functions from functional

programming, although their specific usage is not the same as their original forms. To use the model (Figure 3.1), an application implements a map and a reduce function. The map function takes a piece of the input data and outputs intermediate $< key, value >$ pairs. These intermediate pairs are then grouped by key using a partitioning function and sorted if necessary. The reduce function then processes a group of $< key, value >$ pairs and generates the desired output data. The framework automatically parallelizes the execution of the functions, the distribution of the function code and data and handles the failures of faulty nodes. Since it's introduction, MapReduce has been used in a wide range of applications [20] including distributed grep, distributed sort, web link-graph reversal, data mining, machine learning and statistical machine translation.

Currently, there are several close and open source MapReduce systems. The most popular environment for MapReduce systems is still mainly confined to their originally designed for setting of server clusters. However, there have been efforts in introducing the MapReduce paradigm to other types of systems such as graphics processors [60], multi-core, multiprocessors systems [96], Javascript enabled browsers [59] and even other mobile phone based solutions [86] [40].

Currently, Hadoop [28] is the most popular open source implementation of MapReduce. It is developed for applications that work with thousands of nodes and petabytes of data. Hadoop is inspired by and follows the design of Google MapReduce [30] and Google File System [55]. The system is aimed at data center environments, with many rack-aware optimizations that take into account geographical clustering of servers. Hadoop also provides its own Hadoop Distributed File System (HDFS) [103] which it

uses to store input and output data, although it has been recently extended to support other file systems such as Hypertext Transfer Protocol (HTTP). Scheduling in Hadoop is performed by a Job Tracker which pushes tasks to idle Task Tracker nodes.Each Task Tracker has a number of available slots for tasks and each task takes up one of these slots. If the tracker node fails or times out, the task is reassigned to another Task Tracker.

Due to its popularity, there are many efforts to extend and optimize Hadoop. YAHOO has been a major contributor to the Hadoop code base and is actively researching methods to allow for real-time MapReduce. This advancement would be very useful for applications such as personalization, user feedback, malicious traffic detection and real-time search which all require very fast response and scalability. In search, if the output data sets are made available to the system before the user executes her next search, adaptations can be made to the search models based on the user's intent. One approach used to provide this type of rapid feedback possible is MapReduce Online [24], which modifies the MapReduce architecture so that data can be pipelined between operators. This change extends MapReduce beyond batch processing and allows for early partial results, continuous queries and stream processing.

Disco [106] is another MapReduce system which can be seen as a complementary project to Hadoop. Disco is a much smaller scale project but has been proven to scale to hundreds of CPUs, tens of thousands of simultaneous tasks and process datasets in the scale of tens of terabytes. A central difference between Disco and Hadoop is the compactness of Disco's core, it currently sits at under 6,000 lines of code due to its implementation language, Erlang. In addition to the Erlang core, Disco uses Python

extensively and due to the language's relative ease of development real-world needs can be responded to quickly.

Phoenix [96] was developed to evaluate the suitability of the MapReduce model for multi-core and multi-processor systems. Its implementation runs on a shared memory system and provides automatic thread creation, dynamic task scheduling, data partitioning and fault tolerances across processor nodes. They show that Phoenix leads to scalable performance and has similar performance to parallel code written directly in P-threads.

Mars [60] is a MapReduce framework designed for graphics processors. It takes advantage of the much higher computation power and memory bandwidth of GPUs compared to CPUs. Their system shows up to 16 times improvement in computation on a GPU than a comparable CPU-based counterpart.

3.1.3 Mobile Map Reduce Systems

There are currently a few existing MapReduce implementation targeted at smartphones: Hyrax [86], an implementation for a master's project, a research prototype implementation [40] and Misco [33]. All three systems were developed independently around the same time frame between 2008 and 2010.

Hyrax is a mobile MapReduce platform derived from Hadoop and supports cloud computing on Android smart-phones. The developers for Hyrax attempted to port directly from Hadoop, but had to make several modifications to the code to allow it to run on mobile phones. The first issues they overcame was the differences between Android's JVM and the JVMs found on desktops and servers. Secondly, Hadoop was not coded with efficiency in mind, as evidenced by its heavy use of interfaces and inheritance. While

interfaces and inheritance make code more maintainable, they incur heavy computational overheads and are not suitable for Android devices. Further, Hadoop makes liberal use of Extensible Markup Language (XML) for communications, which is flexible but requires very high processing overheads to parse. Due to these problems, Hyrax is largely a proof of concept system and due to its minimal changes to the Hadoop code base, suffers from poor computation performance results. For example, Android only allocates 16 MB of memory for an application, but with Hadoop and Hyrax's monitoring overhead, only 1 MB remains to be used for buffering MapReduce output. In comparison, Misco was designed from the beginning for use in systems with limited resources and the framework system uses very little overhead. However, because Misco is relatively new, it does not support many of the more advanced features of Hadoop, such as automatic data replicas, which may need to be significantly re-worked for mobile systems.

[40] deployed a research prototype used to study the feasibility of using smart-phones as a part of a MapReduce system. The prototype is developed for iPhones and uses Objective-C for the client and a server implemented in both PHP and Ruby. Their system uses a task requesting architecture similar to ours where the workers actively requests tasks from the server instead of the server assigning tasks to the workers. However, they only use the phones for performing map tasks and leaves the reduce tasks to be performed on the server. The researchers, using data throughput as a benchmark, showed that iPhones are able to consistently perform at only one order of magnitude lower than traditional desktop clients. This contribution to total processing power of smart-phones is considerable and a large portion of computation can be offloaded if there are enough devices.

Recently, there has been some interest in using Javascript [59] to drive MapReduce clients. This would allow any device with a Javascript enabled browser to participate in a MapReduce task. Due to Javascript's popularity and pre-installation in all major web browsers (from desktops to netbooks to smart-phones), this idea can potentially add millions of extra computing resources to any MapReduce project. Also, efforts into making Javascript engines more efficient [38] [43] will greatly benefit any such Javascript driven systems.

3.1.4 Target Applications

The goal of Misco is not to reproduce the typical massive data backed applications which are currently running on MapReduce systems in data centers. Traditional MapReduce implementations have been primarily focused on processing huge data sets in large data centers. These are not the types of applications Misco is aimed a reproducing. While the processing and network resource of smart-phones are growing at an incredible rate, current phones do not have the processing power, network stability and most importantly, battery power to make much impact on these types of applications. Instead, this work is focused on the applications which take advantages of the strength of smart-phones and present Misco as an abstraction framework for enabling developers to concentrate on the functionality of their applications rather than wrestle with the complexities of phone development environments and distributed applications. The types of applications developed on Misco will take advantage of the new types of capabilities which are unique to mobile devices and smart-phones. These new mobile MapReduce nodes can become the producers of data rather than just processors of pre-collected data.

The types of applications currently targeted are: monitoring applications and social applications. However, like the introduction of the MapReduce framework itself, there may be many more application domains which have not yet been considered.

Monitoring represents a large class of applications for which smart-phones are particularly well suited. The ubiquity and sensing capabilities of phones allow them to provide very good coverage over large areas, particularly where there is high population density. For these types of applications, the phones can be thought of as taking the place of traditional sensor network nodes.

- Traffic Monitoring applications can be used to detect areas of high density and congestion. These areas can then be used to help guide user and shorten travel times. Phones with GPS can be used to detect traffic conditions, and then used to avoid further congestion.

- Environmental Monitoring applications can be used to plot different levels of noise, temperature, seismic activity or pollution on a map. The resulting maps of these measurements can then be used to raise alerts for abnormal patterns or to study conditions throughout a city.

There are many existing traffic monitoring projects underway which use smart-phones. Projects include Mobile Millennium [10], VTrack [105] and CarTel [67]. The Berkeley project is a traffic monitoring system which uses the GPS in smart-phone to gather traffic information, processes it and then distributes it back in real-time in an effort to mitigate traffic delays. VTrack tackles the problems of high energy consumption and sensor unreliability to provide accurate estimates of the user's trajectory and travel

time. CarTel also uses smart-phones and automobiles to collect and process data to be delivered to an internet server. In addition to monitor and mitigate traffic congestion, CarTel also use data collected to monitor and classify road surface conditions [44]. Other projects such as PEIR [91] use location data samples from phones to calculate personalized estimates of environmental impact and exposure. These applications which use sensed data to produce an area mapping of values can be easily modeled to use MapReduce to process data. Spatial alarms [8] can be used to enable the personalization of location-based services, they can remind users of the when they have arrived at a location of special interest.

Social applications are another popular class of applications that people are interested in and are particularly suited for smart-phones. Many users are very intimately connected to their phones. They carry them around, store contact lists and keep track of upcoming events in calendars. The closeness of users and their phones provides even more value by using the user's context to customize content.

- *Similarity Based Sharing* applications can be used to detect and automatically share similar multimedia items between users: music, pictures and videos. These items can be from similar in style, time and location or contents.

- *Event Summarization* applications can be used to produce a fuller experience from sporting events, conferences or political gathers. Data from multiple people can be gathered together to generate a more complete view.

- *Location and Status Based* applications can be used to notify friends of your current status. Other location based services can provide recommendations for venues in a

given area or notify you when your friends are nearby.

A few example projects include SoundSence [80], CenceMe [88], BikeNet [39], and Darwin Phones [87] and foursquare [51]. SoundSence is a project that uses a combination of supervised and unsupervised learning techniques to classify both general sounds like voice and music and also to discover novel sound events specific to users. SoundSence is designed for scalability and to run on the limited resources of phones. Likewise, CenceMe is also developed to run on cell phones, but its purpose is to allow members of a social network to share their sensing presence and status with friends in a secure and automatic manner. CenceMe captures the user's status by detecting her activity (e.g. sitting, walking, meeting friends), disposition, habits and surroundings automatically through classification algorithms. BikeNet allows cyclists to share information about themselves and the path they transverse in real-time using bike-to-bike sharing or through third party checkpoints. Darwin Phones is a collaborative reasoning and collaborative sensing system that reason about human behavior and context to infer the actions of users and groups of users. Location based social applications are also gaining popularity. Foursquare [52], an application which allows users to check into venues and gain badges, has gained over 500,000 users in its first year and grew 3,400% in 2010. These types of applications take advantage of the intimate connection between a smart-phone and their users.

For these types of systems, a MapReduce framework would allow the developers and designers to concentrate on the algorithms to analyze data instead of diverting their attention to the details of communications between devices. The framework allows the user to collect and process data locally and then collectively process the data from multiple

phones. For projects like SoundSence, CenceMe, BikeNet and Darwin Phones, the local tasks can perform basic classifications (or just feature extraction) of the user's actions or surroundings based on the sensed data, while data gathered from multiple phones can be processed together at the servers to produce a more accurate view of the data. Another very important feature is that the data collected from multiple users can then be re-distributed back to the phones for more accurate processing or feedback to users.

3.1.5 Scheduling with Failures

The problem of scheduling in the presence of failures has been studied in prior works, primarily in cluster-based and distributed system settings. 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 node fails, the execution is not interrupted; however, this requires extra nodes. Temporal redundancy re-executes tasks after they have failed, this requires extra time. Checkpointing [49] is used to limit the amount of work lost when failures occur, the checkpoint frequency must be carefully calibrated. In addition, Failure handling algorithms fall in *hard real-time* and *soft real-time* categories and further segmented into *online* and *offline* algorithms. Hard real-time systems make a guarantee that an application will complete by their deadline while soft-real time systems do not make a guarantee, instead they make a best effort attempt to complete applications by their deadlines. Offline algorithms pre-computes a schedule for tasks and application before the system is started; online algorithms decide how to schedule tasks while the system is running

Fault tolerance is achieved in [56] by reserving enough slack in the schedule to

tolerate transient and intermittent faults by re-executing the failed tasks without affecting the guarantees given to other tasks. Their approach only considers at most one fault with in a time interval. An optimal algorithm is provided for scheduling tasks offline and a linear time heuristic algorithm for dynamic scheduling.

An offline AI planner is used in [5] to generate feasible schedules for different potential errors in a hard real-time system. The system automatically creates fault-tolerant plans which are guaranteed to execute in hard real-time in the presence of failures from a user-specified list of potential faults.

The problem of checking the feasibility of a set of n aperiodic real-time tasks while allowing for at most k transient failures is considered in [7]. Both an offline and online version of their dynamic programming based algorithm is provided.

The replication of periodic hard real-time tasks in identical multi-processor environments is examined in [19]. Their approach is to replicate each task on K distinct processors using two different algorithms for minimizing the maximum utility of a system and for minimizing the number of processors required to derive feasible schedules. A 2-approximation algorithm was developed where there is an arbitrary number of processors. [58] considers a similar problem, but applies it to a system with a constant number of heterogeneous processors. A polynomial-time approximation scheme is presented to solve the problem of mapping recurring tasks to a heterogeneous set of processors such that timing and failure tolerance requirements are met.

Graceful system degradation when failures occur is achieved in [42] by choosing where replicas are placed. They create an objective function based on a utility model

which measures the functionality that the system is providing when some of its components have failed. This objective function is then incorporated into their TOAST task allocation tool which provides solutions to task allocation problems.

A mathematical approach is used in [110] to determine the optimal redundancy level for tasks. Their goal is to maximize a performance-related reliability measurement which can be adjusted based on how task success and failures are weighed.

An imprecise computation model were precision can be traded for timeliness is proposed in [6]. In this model, a task is divided into a mandatory part and an optional part. The mandatory portion of the task must be completed by its deadline while the optional part does not have this requirement. The optional portion refines the output of the mandatory part. A reward function is associated with the optional part of the task and a FT-Optimal framework is presented to compute a schedule which maximizes total reward and tolerates transient faults for the mandatory parts.

Probabilistic reliability under failures for a periodic real-time system on identical multiprocessor platforms is explored in [11]. They compute the minimum number of processors required to achieve some maximum tolerated probability of failure and also the maximum level of reliability given a set number of processors. Due to the probabilistic approach, no assumptions are made about the total number of failures that can occur.

MiscoRT provides a soft-real time scheduling system which uses both temporal and spatial redundancy. Tasks which are likely to fail are scheduled on multiple worker nodes and task which have already failed are reprocessed. The use of checkpointing is not explored in the system, but can be used to recover partial task results. Making hard-real

time guarantees would be close to impossible due to the many unpredictable sources of delay in the system setting.

3.2 System Settings

The system 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 3.1) 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 can be 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: *ready time* r_j is 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. $Laxity_j$ is computed as the difference between the Deadline of the application and its estimated execution time. The laxity value represents a measure of urgency for the application and is used to order the execution of the application tasks on the worker

nodes. The advantage of using the laxity value is that it gives an indication of how close the task is to missing its deadline; the task with the smallest laxity value has the higher priority. Negative laxity values indicate that the task is estimated to miss its deadline, different scheduling policies can handle these in different ways, one method is to allow applications which can meet their deadlines to proceed. MiscoRT targets soft real-time systems, where missing a deadline is not catastrophic for the system. The goal is to maximize the number of applications that meet their deadlines.

For each task t of an application A_j the system computes: 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 through profiling mechanisms [70].

The 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.

The scheduler is not concerned about the network topology of the system, similar to other cell phone based systems [89], it is assumed that if there is a connection to the

server, it is over a one-hop HTTP connection. While the system does not explicitly handle multi-hop networks, Any networking overhead is implicitly accounted for through the gathering of worker timing statistics.

3.3 Misco Design and Implementation

There are many unique characteristics of phones and their mobile environment which must be taken into account when designing a distributed system for them. The first and largest concern is the limited available battery power. Although energy use is a big concern, it is not a focus of this work. There are many other research groups focused on energy efficient computing and their efforts are orthogonal and their techniques can be applied to the system independently. However, there are some basic principles used to guide the design of the system. It is almost always better to perform local computation and storage than to transfer data over the network and WiFi connections are generally more energy efficient than cellular networks. In addition, network transfers, in practice, are not free and service providers frequently charge usage fees.

Mobility and failures also require consideration. Failures frequently occur due to device failures, software malfunctions and bad network coverage. Mobility [73] also acts as a failure mode. A phone is considered failed if it is disconnected from the network for an extended period of time.

3.3.1 Communications

One of the biggest challenges faced when designing Misco for mobile devices is the communication method used between phones and the server to transfer data and instructions. This communication problem, while important for traditional systems, is not as problematic as traditional systems are typically located in well connected data centers with high bandwidth, pairwise connectivity between each node. Many systems also provide a shared file system as part of their implementation, similar to HDFS, where Hadoop stores input and output data. For mobile, smart-phone systems, network usage is very costly in terms of both time (low bandwidth) and energy usage. For Misco, the phones do not communicate with each other, instead, they only communicate with a centralized server for both instructions and data. This is a realistic model of typical smart-phone usage as they generally connect to the Internet and an Internet server through either WiFi or a service provider's data plan (e.g. 3G).

All communications in the system happens using HTTP due to the ubiquitous nature of the protocol. It is very well supported by many devices, especially the devices with enough computation power to support processing. Another advantage of HTTP is it's built in end-to-end error correction and message ordering. For basic instruction passing between the server and workers, plain text HTTP requests and response can be used. A multi-part POST request is used for file uploads from the user or phone to the server.

In addition to communication technology, there are two main approaches to communicating between two parties: polling based and interrupt based. For the design,

a polling based approach is used despite the advantages interrupt based communication provides. The biggest attraction of interrupt based communication (and biggest drawback of polling) is the communication overhead. Less communication is required for interrupt based communication because data is sent only when it is required.

However, to support an interrupt based communications model, a server needs to be implemented and run on each phone to constantly listen for incoming requests or messages. This is not practical or desirable for many reasons: it adds complexity to the software required on the phone and uses up limited memory resources. Also, to implement such a system, each phone has to be reachable from other device in the network; this is unrealistic given the complexities imposed by firewalls and network mappings. For example, the IP and port number mapping to the phones are not consistent or guaranteed by the different service providers. An initial attempt was made to use Short Message Service (SMS) messages as a means to provide an interrupt based method to send instructions to the workers, but this method is not reliable and not all phones are equipped with such messaging plans or contain Subscriber Identity Module (SIM) cards.

The worker will simply poll the server for work whenever they are idle, the server will respond with work if available or *None* if there are no tasks. The phones will idle for a period of time and then retry requesting for a task. The implementation for such a polling method is extremely simple, and the idle polling frequency can be adjusted to try and match the task arrival frequency, but if it is not calibrated correctly, there can be inefficiencies resulting in energy waste and periods where idle capacity is not being used. However, there is no problem if the number of applications saturates the systems, in this

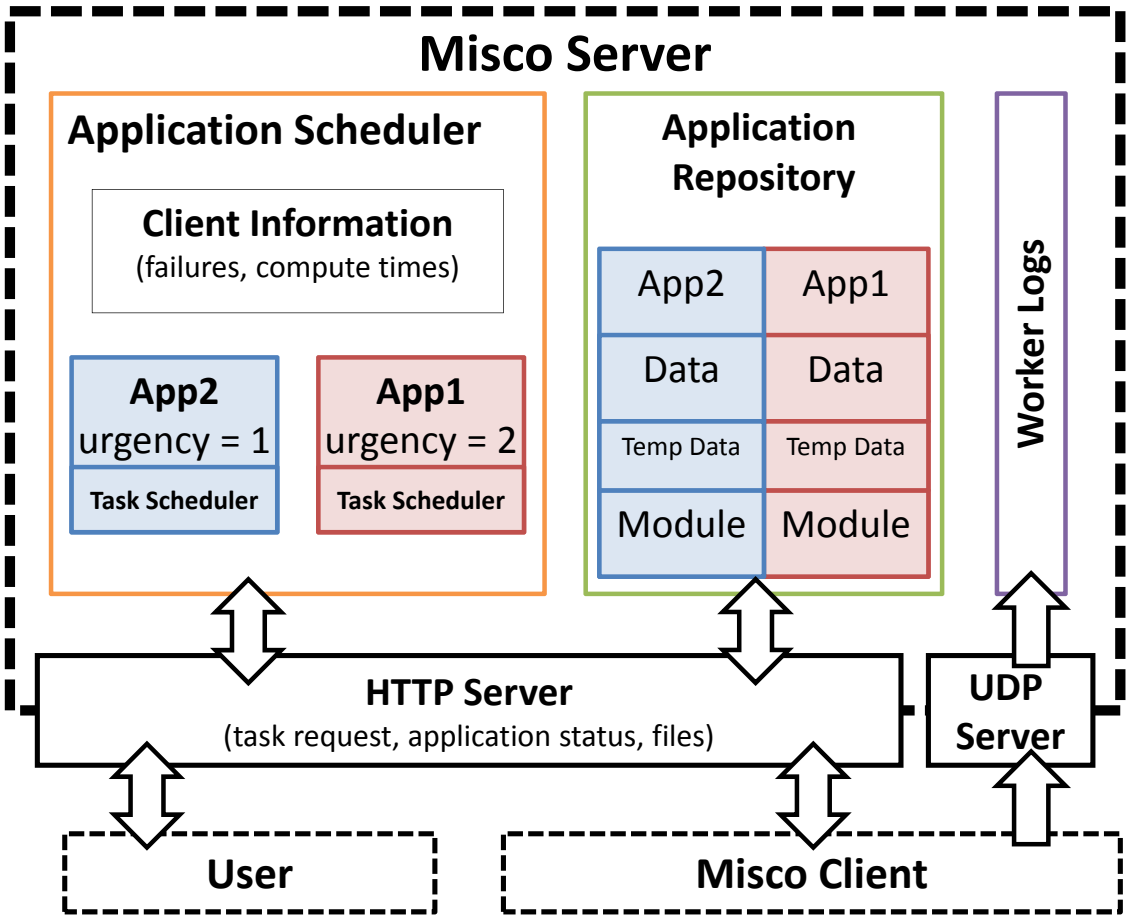


Figure 3.2: Architectural diagram for the Misco server.

scenario, there will always be work available.

3.3.2 Server

One of the fundamental design goals is scalability. Smart-phones are about one magnitude less powerful than desktop computers, but there are a lot more phones than computers. Current MapReduce clusters consist of thousands to tens of thousands of server nodes. In the case of phones, the target would be to reach numbers in the hundreds

of thousands or millions. One of the bottlenecks for current systems is the server which coordinates the tasks. The server should be as lightweight as possible. To this end, there is no need for monitoring all the workers and checking heartbeat messages for failures, the status of the workers is not important. The concern is with the status of the MapReduce tasks. In this way, the actual workers or number of workers in the system does not need to be known.

The Misco master server, shown in 3.3.2, is composed of several components responsible for storing application data, assigning map and reduce tasks to workers and interacting with the user for creating application and providing results. The server sits at a known IP address and listens to a pre-specified port. The server is implemented in Python and uses the *BaseHTTPServer* and *BaseUDPServer* combined with the *ThreadingMixIn* to provide multi-threaded request handling. Misco uses custom request handlers to deal with user and worker requests. To download data and Python module files from the server, the worker simply navigates to the proper Uniform Resource Locater (URL) containing the file name, similar to a standard HTTP server.

The process of creating an application in the server is initialized by making a HTTP request to the server. This request can be user generated through a form submission by a user on a browser, or by an external application which supplies the required fields. The server expects an application name, the input data file, the python module file, how to divide the input file and the number of partitions for the reduce phase. The user can also provide optional parameters for their map and reduce functions to use and provide timing details such as deadlines and estimated execution times. Additional

parameters are described in more detail later in Section 3.4.

Once this request has been submitted to the server, it will create the data structures to store the application's execution information. First, the input data is split into a number of pieces with each piece numbered correspondingly to a map task (each piece of data is the input to a map task). Different schedulers are free to use different data structures to keep track of tasks and their completion, in the most basic scheduler, an array is used to keep track of which tasks have been completed and the number of workers working on any particular task. This information is then used to decide on the next task to assign to a worker. A similar data structure is created to keep track of both map and reduce tasks. The number of map tasks is determined by the number of files the input is split into and the number of reduce tasks is determined as the number of partitions the user specified. These two structures are then passed into the scheduler component which places it into the pool of available applications.

The scheduling module is very flexible and can be replaced very easily. In the initial implementation, a simple First In, First Out (FIFO) scheduling technique was used. This is very similar to what many established MapReduce implementations are using. When a worker requests work from the server, an application, which has not yet been completed, will be picked from the pool of applications (FIFO order, in this case). Then, a task which has not been completed will be picked from that application and its details will be returned to the worker. This worker will be noted in the data structure. Note that all the map tasks from an application must be completed before any of it is reduce tasks can start. This restriction is a part of the traditional MapReduce model,

but there are recent research efforts to bypass this [24].

For more complicated schedulers, more complex algorithms are used for choosing which application and task to assign to workers. In addition to user provided timings, the scheduler can choose to track worker statistics such as processing power, failure rates and resource availability and use these factors to determine a scheduling technique to optimize its target parameters. Some schedulers will be concerned with meeting user specified application deadlines while others may only care about data throughput. Section 3.4 examines the MiscoRT scheduler in detail.

When a worker has completed a task, it will upload the results to the server. The server will pass this information to the scheduler, which can then mark off the completed task. The uploaded data from a map task is intermediary data and this data will be used as input for reduce tasks. The results from reduce tasks are considered to be the final results of the application. Once every task in an application is completed, the application is considered done and the results are considered complete. The server has the ability to notify an external process that the application has been completed by sending a User Datagram Protocol (UDP) message to an external process. This feature is very useful for automatic applications which want to perform further actions once the results are ready. Dowser is one such application, discussed later in Section 3.3.4.

Data storage on the server is straightforward, for each application, a folder is created to hold its Python module file. A separate folder is created for each of: the input data (split into separate files), the intermediary data returned from map tasks and the final result data returned from reduce tasks. To access any of these files, the worker

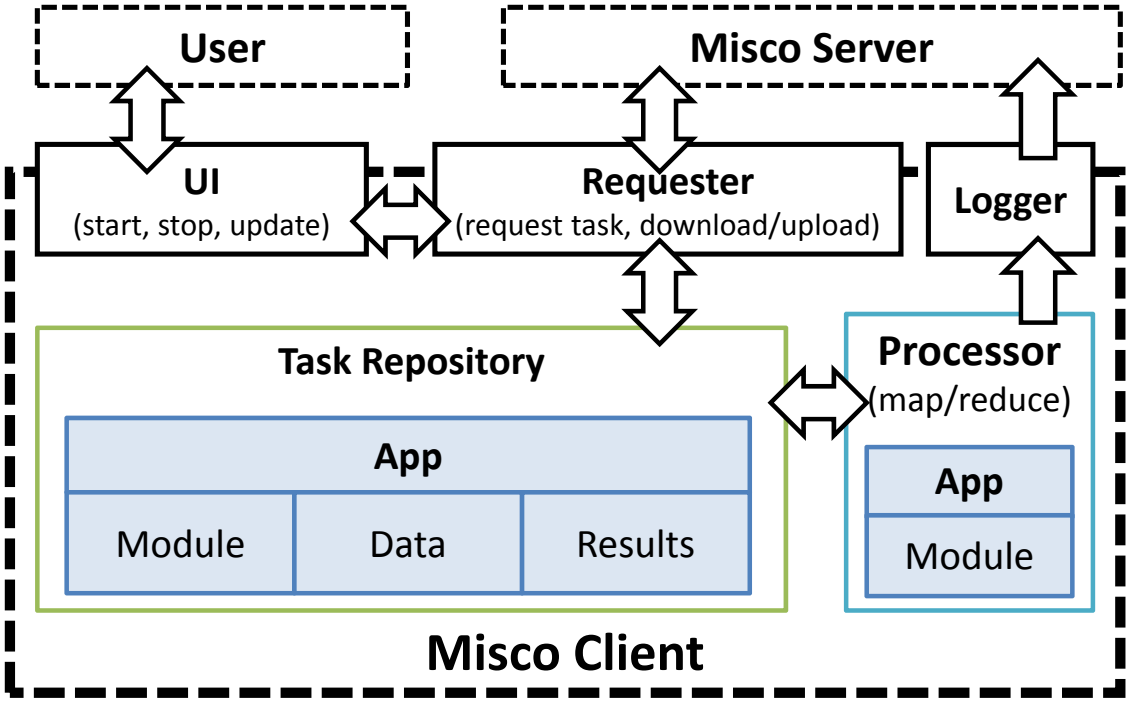


Figure 3.3: Architectural diagram for the Misco client.

simply requests it at the appropriate URL.

3.3.3 Client

Compared to the server, the worker is relatively simpler as it is not concerned with scheduling. Instead, it is only concerned with processing a single task at a time. Figure 3.3.3 shows an architectural diagram of the worker. The worker is also implemented in Python and can be run on any device that supports Python and Python's networking library.

Once the worker script is executed, no further user intervention should be neces-

sary unless the script or device fails in a way that requires the user to restart. The worker knows how to contact the server a priori and will immediately start requesting tasks. Whenever the worker is not processing a task, it is available for work and will contact the server with a request for work, passing the server the worker ID (for the phones, the International Mobile Equipment Identity (IMEI) number is used as an ID). The server will respond to the request with either *None* when there are no tasks for the worker to process or with a Python dictionary data structure containing information about the input data file to use, the function (map or reduce) to perform and any additional parameters the user specified when the application started. The worker will *eval()* this dictionary and download the required files from the server.

Once the worker has downloaded the files it needs, it will perform a dynamic import on the Python module file containing the map, reduce and accessory functions. The proper function will be executed with the input file passed in as a parameter. For additional flexibility, the user can supply an *InputReader* function for map tasks so that custom input data files can be used instead of the default behavior of reading the input file one line at a time. The user can also supply a custom *combiner function* which can be used to compress or pre-aggregate output data from the map function before sending the results to the server. An example of a useful pre-aggregate is demonstrated by the word count example, where the application counts the occurrences for each word in the document. The map function for *WordCount* simply takes each word and emits it as $< word, 1 >$ and then the numbers are added together in the reduce function. However, for common words, there may be a lot of duplicates (e.g. $< the, 1 >$). It would be much

more efficient to compress the intermediary data right after mapping by adding duplicates together, so if there are five instances of *the* in the input data for a map task, it would only output $< the, 5 >$.

The results from map tasks are stored in separate files with each file corresponding with a reduce partition. The results of a reduce task is a single results file. When the task is complete, the worker will upload the results to the server and request a new task. Data storage on the worker is even simpler than the server. The worker creates a single folder for each task. Downloaded input data is placed into this folder and the results generated are also placed into this folder. Once the results from the task are submitted to the server, this folder and its content are deleted.

There are several local optimizations performed on the workers. First, caching is allowed for the Python module files because a worker may be performing the same computation on different input data sets for the same application or even for different applications. If a worker determines that it already has a local copy of the module, it will not re-download it from the server. There was an issue with storing too many files on the phones, there were file locking problems that caused files to not be deletable later. Thus, whenever a task is finished, all generated data files are deleted before starting the next task.

3.3.4 Example Application: Dowser

As part of the Misco project, Dowser [35], an application which uses Misco as its processing core, was developed. Dowser is used to locate nearby areas of good internet connectivity. Figure 3.3.4 shows some sample data and calculated centroids on the UC

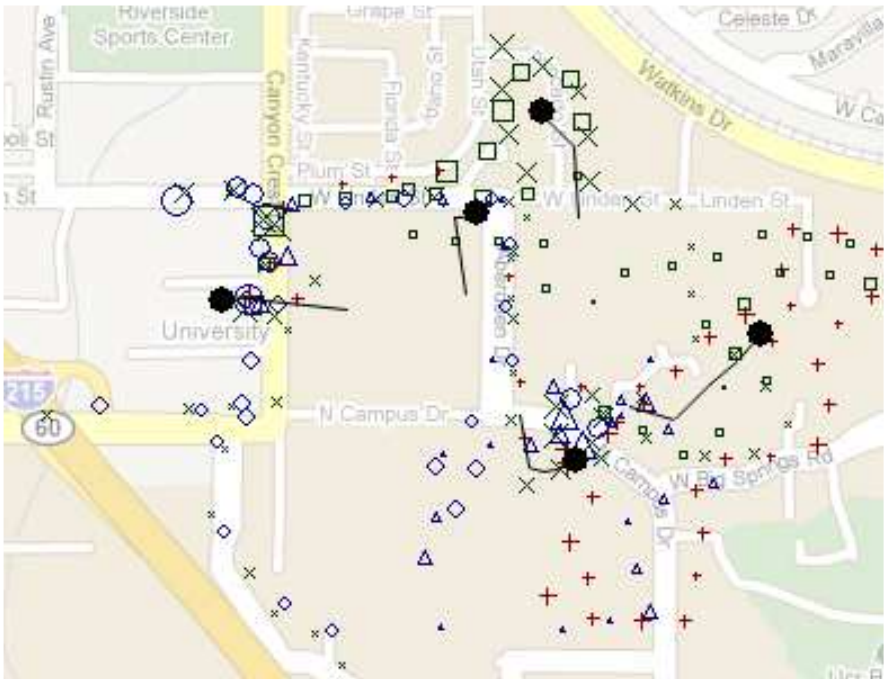


Figure 3.4: Dowser application

Riverside campus. Dowser is composed of three main components:

- Data collection component runs in the background of the smart-phones and periodically gathers and stores local connection strength information along with a GPS location and the current time stamp.

- Data processing component which is a distributed k-means clustering algorithm developed to run on top of MapReduce and use Misco.

- User Interface (UI) component which visualizes the user's current location and directs the user to nearby locations where areas with many good connections are located.

Focusing on the data processing component, a MapReduce application was developed which operates on the collected data in such a way that the k-means clustering is performed without actually transferring any data off the originating phones. The only data exchanged were locally calculated centroid locations information on each individual phone. This application demonstrates that computations can be performed where users do not need to send their collected data, this saves on network usage and also provides a degree of privacy. For the last stage of the application, a last round of MapReduce is run to disseminate the final results to the phones and they are then able to use these results in the UI. A detailed description for Dowser can be found in [35].

3.4 MiscoRT Scheduler

The main responsibility of the MiscoRT scheduler is to assign tasks to workers when they make requests. The scheduler uses a two-level scheme, as follows:

- The first-level scheduler, the *Application Scheduler*, determines the order of execution of the applications based on their urgencies and timing constraints. It estimates 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 adjusts their scheduling order to compensate for queuing delays and worker node failures.

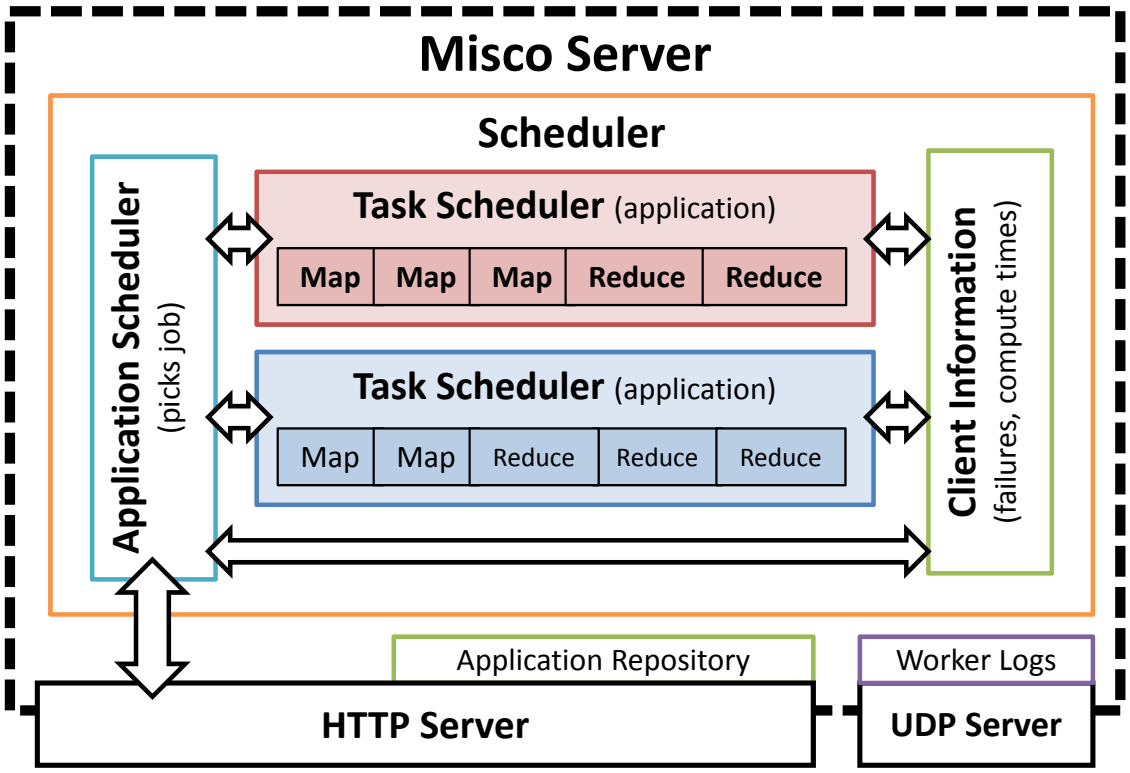


Figure 3.5: Architectural diagram of Misco scheduling system.

An architectural diagram of how the scheduler fits into the Misco server is shown in Figure 3.4. The remainder of this section explores the failure model used in the scheduler and the design of the application and task scheduler.

3.4.1 Failure Model

In this section a model to estimate the execution times of the applications under failures is presented. The failures of the worker devices are assumed to follow a Poisson distribution and 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 [37]. 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, these parameters are summarized 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

Single Task, Single Worker

The basic unit of work is a single task executing on a single worker W_i with processing time, including failures, C (superscripts and subscript are omitted 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 [77], the expected number of failures is computed using:

$$E[F] = \frac{\tau\lambda}{1 - \tau\lambda} \tag{3.1}$$

The amount of time to successfully complete a task 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:

$$C = \tau + \sum W + \sum D \tag{3.2}$$

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:

$$E[W] = \frac{\tau}{2} \tag{3.3}$$

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

$$w = E[C] = \tau + \frac{\tau}{2} * \frac{\tau\lambda}{1 - \tau\lambda} + \mu * \frac{\tau\lambda}{1 - \tau\lambda} \tag{3.4}$$

Multiple Tasks, Single Worker

In a series of T tasks, each with processing times $C_1, C_2, ..., C_T$, the times for the individual tasks are summed to derive the total processing time C.

$$C = \sum_{i=1}^{T} C_i \tag{3.5}$$

$$E[C] = \sum_{i=1}^{T} E[C_i] \tag{3.6}$$

Multiple Tasks, Multiple Workers

For 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.

Algorithm 1 The MiscoRT Application Scheduler

Input: Set of applications A in system

for all Application A_j in A **do**

calculate $Laxity_j$ of A_j

end for

Order A by $Laxity_j$

Task $\leftarrow$ TaskScheduler(A_j with smallest $Laxity_j$)

return Task

Since all workers are either processing a task or in a failure state, this can be modeled 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 $(1 \leq i \leq M)$ is:

$$\rho_i = \lceil \frac{1/w_i}{\sum_{k \in M} 1/w_k} * T \rceil \tag{3.7}$$

Then, the expected execution time for the application is:

$$exec\_time_j = E[C^j] = max_{i \in M}(\rho_i * w_i) \tag{3.8}$$

3.4.2 MiscoRT Application Scheduler

The MiscoRT application scheduler is used to determine which application from the pool of uncompleted applications to run. The pseudo code for the application scheduler is shown in Algorithm 1.

The 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 [71]. In LLF scheduling, each application is associated with a laxity value which represents its urgency. The laxity value $Laxity_j$ of an application A_j is computed as the difference between its deadline and the estimated execution time of the application under failures:

$$L_j = Deadline_j - current\_time - exec\_time_j \tag{3.9}$$

where the estimate of the application's execution time is computed using formula 3.8 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, the analytical model is used 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

Algorithm 2 The 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 $deadline_j$

  return T_i^j

else

  return *None*

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.

3.4.3 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. The pseudo code for the task scheduler is presented in Algorithm 2. When a worker requests a task, MiscoRT decides which task to run next, following these

steps:

Step 1: The application with the smallest laxity value is determined by the application scheduler, this application has the highest priority. The task scheduler checks if there are any unassigned tasks for this application. The primary insight of this is that an application completes when all of its tasks complete, thus all of the application tasks need to be scheduled for execution. To minimize the number of task replicas, tasks are reassigned only if the workers executing those tasks have been speculated to fail or if the task is not progressing as fast as it was estimated and will cause the application to miss its deadline.

Step 2: Check for task failures. 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. 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. This information can be obtained at run-time or by a user-defined threshold obtained from previous runs.

Step 3: Finally, check the progress of assigned tasks. Using formula 3.4, estimate the amount of time required for each task t to complete: $\epsilon_{t,i}^{j} = w_{i}^{j} - elapsed\_time_{t,i}$. If a task has been assigned to multiple workers, the minimum $\epsilon_{t,i}^{j}$ is used. The task with the

largest remaining execution time, $\epsilon_{t,i}^{j}$, is the slowest task and the scheduler checks if this task can complete before its deadline. 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. The task scheduler will assign the task to the new worker if this enables the application to meet its deadline.

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. Furthermore, under 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.

3.5 Results

3.5.1 Experiment Settings

An extensive set of experiments were used to evaluate the efficiency and performance of the scheduling scheme using applications of different sizes and various worker node failure rates.

The experimental platform consists of a testbed of 30 Nokia N95 8GB smartphones [92]. The Nokia N95 has ARM 11 dual CPUs at 332 Mhz, supports wireless

Figure 3.6: The Misco testbed of Nokia N95 8GB phones and a Linksys Router.

Table 3.1: 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

802.11b/g networks, Bluetooth and cellular 3g networks, 90 MB of main memory and 8 GB of local storage. The 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 the phones are connected via 802.11g to this router.

For the experiments, a set of 11 applications were used, 8 with 100KB input data and 3 with 1MB input data. The deadlines for the applications were set such that 5 applications have tight deadlines, 2 have medium and 3 have loose deadlines. These

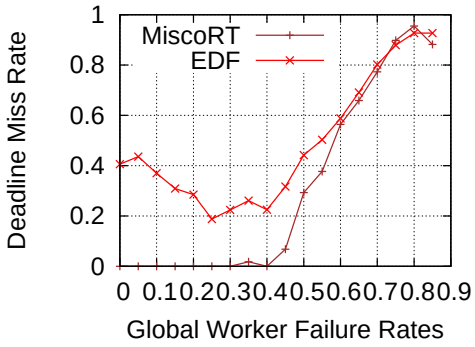


Figure 3.7: Application deadline miss rates when all worker failure rates vary.

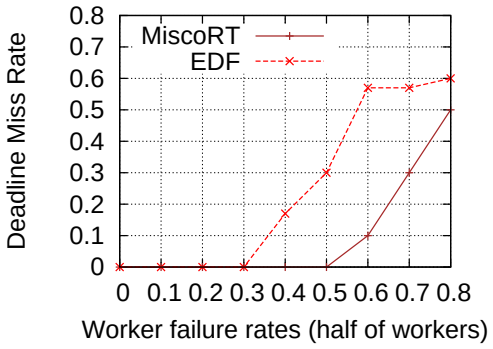


Figure 3.8: Application deadline miss rates when half of worker failure rates vary.

deadlines were derived by running the applications and observing their run times, the deadlines used were between 100 and 550s. The failure of worker nodes follows a Poisson distribution, as explained in section 3.4.1; to vary the failure rates of the workers, the failure arrival rates, λ, were adjusted. Each experiment is repeated 5 times. For each experiment, three different type of failure rates were used. First, every phone used the same failure rate and this rate is varied. Second, half of the workers were fixed and the other failure rates of the other phones were varied. Finally, a log-normal failure distribution was used among the workers to simulate the failure characteristics from user mobility [73]. The 4 sets of failure rates in the lognormal distributions are shown in table 3.1.

To provide a fair comparison, MiscoRT was compared with the ***Earliest Deadline First (EDF)*** scheduling policy, which is a 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. The EDF application scheduler was paired with a sequential task scheduler. The same experimental

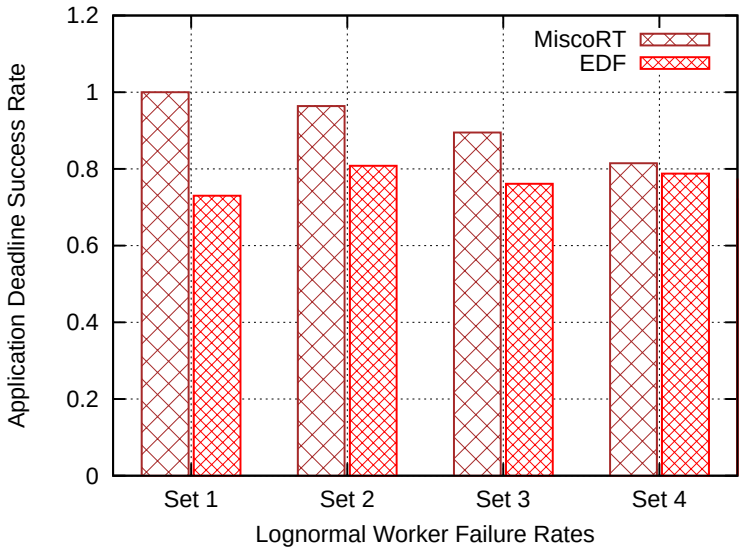


Figure 3.9: Application deadline miss rates when worker failure rates are lognormal.

parameters such as deadline and data input were used for the comparisons.

The performance of MiscoRT scheduling scheme was measured using the following metrics: *Miss Rate* represents the fraction of applications that miss their deadlines, *End-to-end time* measures the time the execution of the entire application set completes and *Wasted time* is the time spent by workers performing duplicate tasks as other workers.

Mobile Tourist Application

A mobile tourist application [4] [53] (location-based social networking application), was built to evaluate the performance of Misco. 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, a dataset from the Flickr photo sharing system was

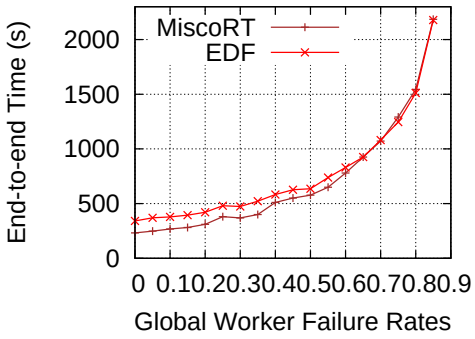


Figure 3.10: Application end-to-end times when all worker failure rates vary.

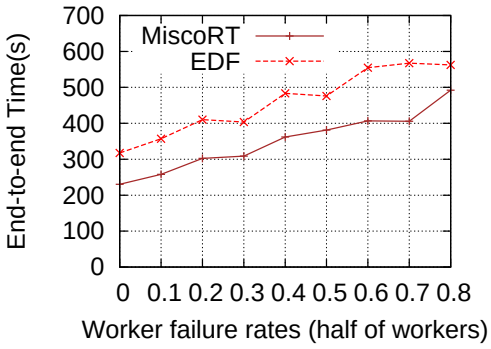


Figure 3.11: Application end-to-end times when half of worker failure rates vary.

compiled. 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.

The dataset consists of 50,000 image metadata (8.75 MB) taken from publicly available Flickr photos. *Santa Barbara* was used as the initial query for photos, then more pictures were downloaded from the owners of these pictures, 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

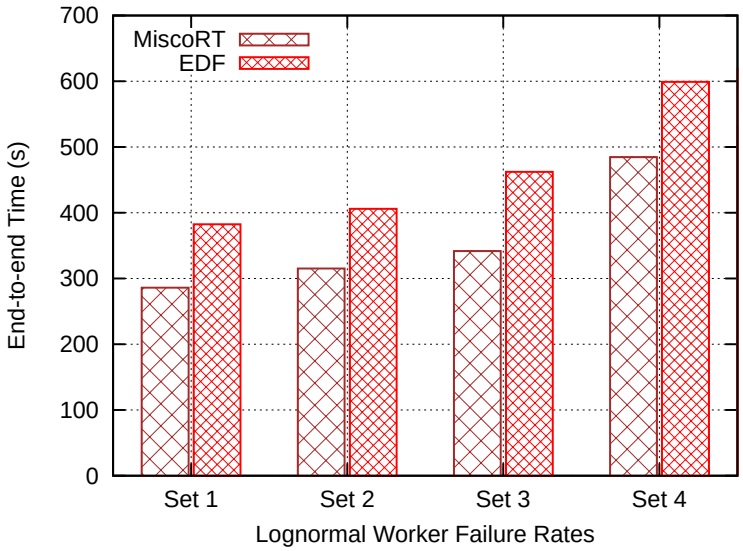


Figure 3.12: Application end-to-end times when worker failure rates are lognormal.

application that have recently seen wide adoption and fully exercises all aspects of the system and demonstrates the timeliness of the scheduler. More complex applications mainly differ in the functions performed in the map and reduce tasks and not in the system's execution sequence.

3.5.2 Performance

For the first set of experiments, the performance of MiscoRT is evaluated by measuring the deadline misses, end-to-end times and wasted times.

Figures 3.7, 3.8 and 3.9 shows the results for three different worker failure distributions. In Figure 3.7, 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. Figure 3.8 show better

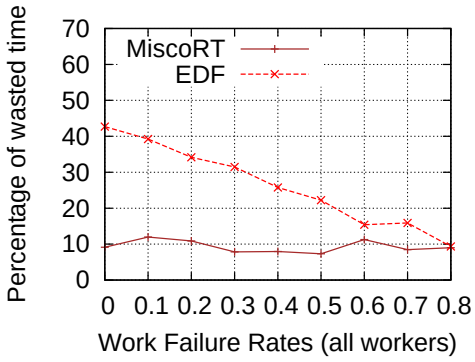


Figure 3.13: Wasted worker time when all worker failure rates vary.

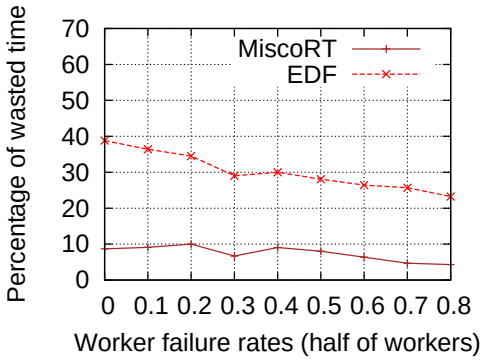


Figure 3.14: Wasted worker time when half of worker failure rates vary.

results for both schedulers because there is more worker resources and MiscoRT performs significantly better than EDF. Figure 3.9 also shows that MiscoRT outperforms EDF, but EDF improves as the amount of failures increase.

The end-to-end times of the applications are shown in Figures 3.10, 3.11 and 3.12. MiscoRT consistently performs better than the EDF scheduler. In the global worker failure rate case, 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 the 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. For the half workers failure case, MiscoRT is about 50% better at lower failure rates and 20% better at high failure rates. Log-normal distribution of failure rates show similar results.

The percentage of wasted time is shown in Figures 3.13 and 3.14, shows that at low failure rates, EDF performs poorly. With no worker failures, EDF wastes over 40% of processing time. As failures increase, the percentage of wasted time for EDF decreases

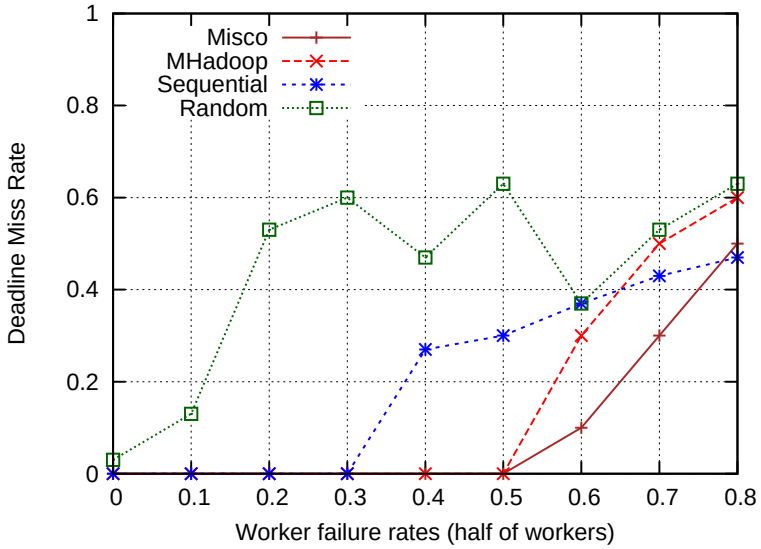


Figure 3.15: Application deadline miss rates for different task schedulers when half of the workers failure rates vary.

linearly, this is because when tasks are replicated to more than one node, the probability that multiple nodes fail is higher. MiscoRT maintains a constant waste throughout because the tasks are replicated based on predicted completion times and do not assign tasks when additional replicas are not beneficial.

MiscoRT with different Task Schedulers

To further illustrate the benefit of the MiscoRT, its task scheduler performance is compared with alternative task schedulers. The MiscoRT application scheduler is used as the first level scheduler and different task schedulers are used as the second level schedulers. The same set of experiments were performed as in Section 3.5.2. The following task schedulers were used:

The random task scheduler is a naive scheduler which picks a random task to execute. This scheduler has very low overhead, it does not store any information about

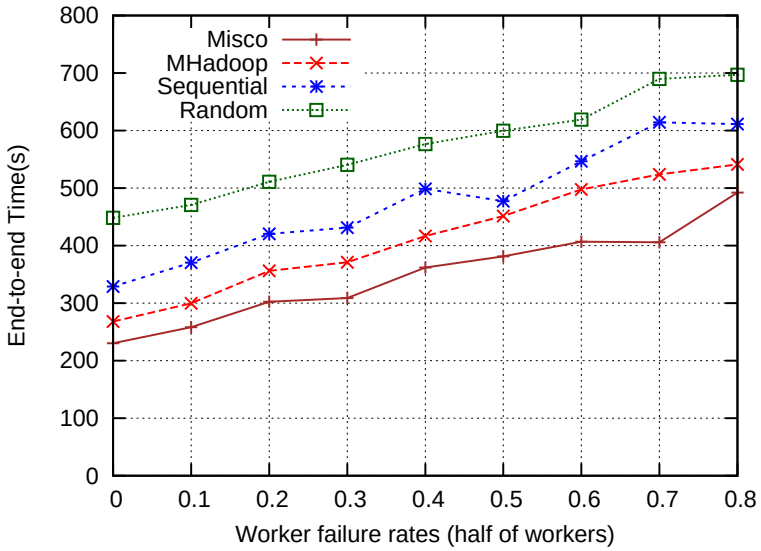


Figure 3.16: Application end-to-end times for different task schedulers when half of the workers failure rates vary.

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 task scheduler used by the popular Hadoop MapReduce framework [28]. Hadoop is designed for cluster-based environments where there is constant feedback from the workers informing the scheduler of their progress on tasks. Directly implementing such progress tracking in the system would be infeasible. To provide a fair comparison, the scheduler was modified while preserving the spirit of the original Hadoop scheduler. It attempts to speed up the execution time by re-assigning tasks only when the previous worker is taking more than the average amount

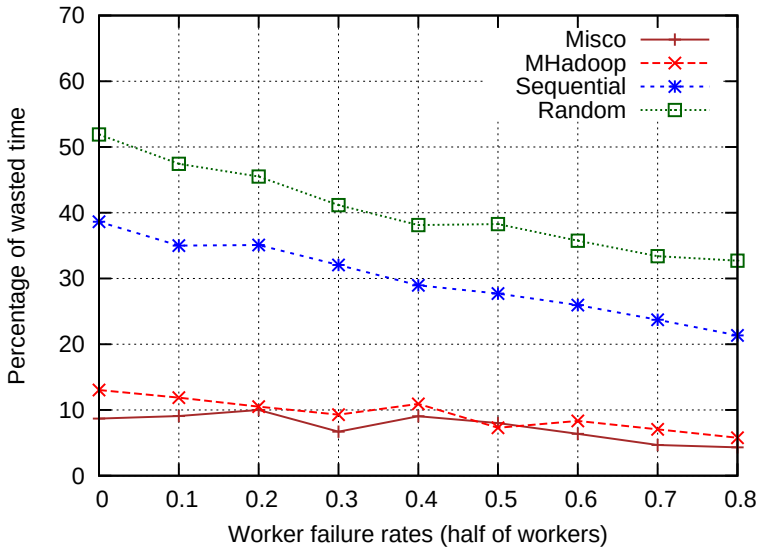


Figure 3.17: Wasted worker time for different task schedulers when half of the workers failure rates vary.

of time to complete that task.

Figures 3.15 and 3.18 show the application deadline miss rates for half worker and log-normal worker failure distributions. The MiscoRT task scheduler has a 25% to 40% higher success rate than the other task schedulers. Figures 3.16 and 3.19 show that MiscoRT out performs all other schedulers with the Modified Hadoop schedulers coming close in several scenarios. Finally, Figure 3.17 shows that the MiscoRT task scheduler maintains a low percentage wasted time comparable with that of modified Hadoop. Wasted worker time starts at around 10% and gradually decreasing to under 5% as failure rates increase. Both sequential and random perform much worse, starting at 37% and 52% respectively and decrease to 21% and 33%.

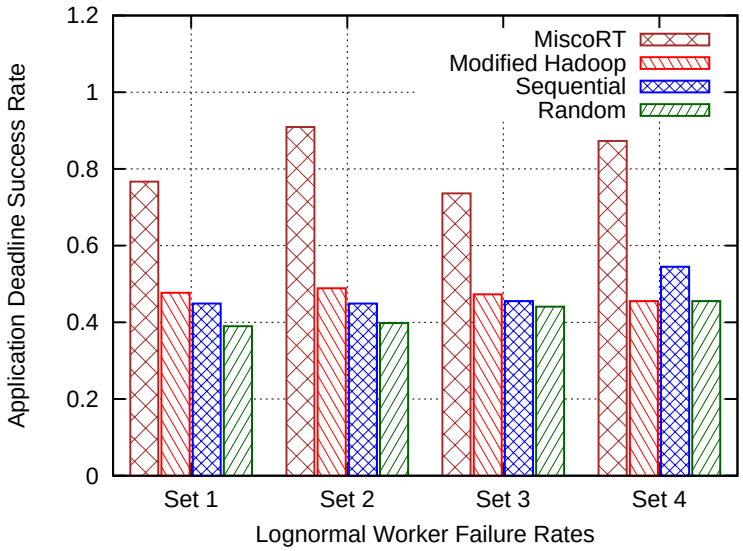


Figure 3.18: Application deadline miss rates for different task schedulers when worker failure rates are lognormal.

3.5.3 Model Validation

In this next experiment, the model is validated by comparing the predicted execution time, using the model described in section 3.4.1, with actual measured execution times. A single application consisting of 73 tasks was used and all worker nodes fail with the same rate. This failure rate was varied and the results in Figure 3.20 shows that the model is very accurate, its predicted times are very close to the measured times observed from running the application in the system, even at high failure rates.

3.5.4 Scalability

The scalability of the system is demonstrated by varying the number of applications while keeping the failure rate at 0. Figure 3.21 shows the End-to-end times, it increases linearly with the number of applications, as expected, this is due to having

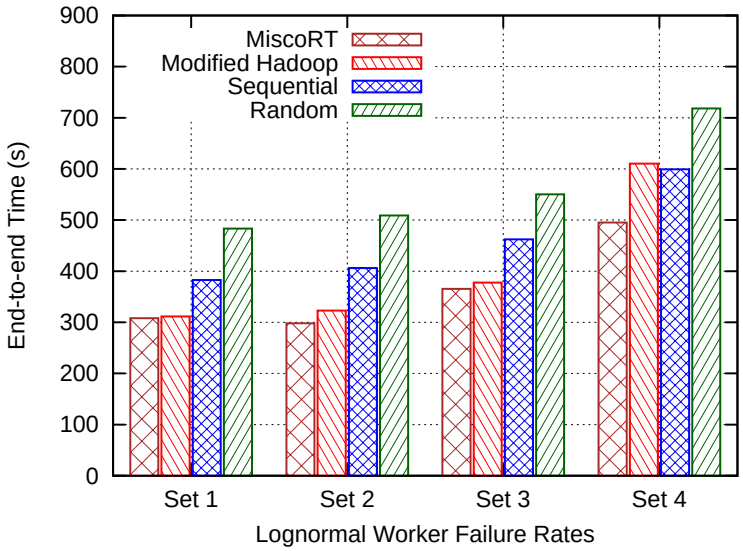


Figure 3.19: Application end-to-end times for different task schedulers when worker failure rates are lognormal.

a fixed processing power and linearly increase the amount of work. The Percentage of Wasted Time is shown in Figure 3.22, approximately 10% of task assignments generate duplicate results, this percentage remains constant even as the number of applications increase.

3.5.5 Deadline Sensitivity

In this experiment, the goal was to test the sensitivity of deadline value on the application miss rate. The tightness of the deadlines were varied by a constant factor ranging from the original deadlines to 20% of their values. The other parameters remained the same. A tighter deadline means that the applications have less time to execute. The global worker failure rate was set to 20%, but similar results have obtained for other failure rates. Figure 3.23 shows that both schedulers perform worse as the deadlines tighten, but Misco outperforms EDF consistently. When the deadlines are set very tight,

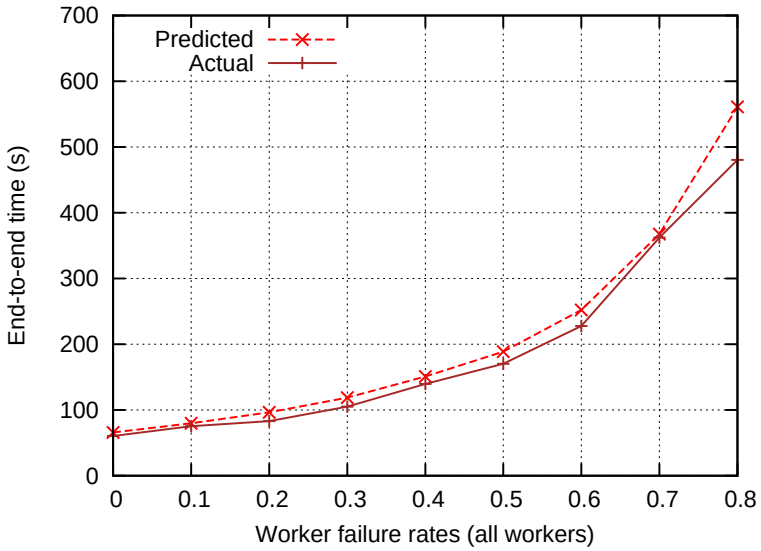


Figure 3.20: Validation for failure model.

at 0.2 of their original values, all applications miss their deadlines when using EDF while only 70% miss their deadline using Misco.

3.5.6 Overhead and Resource Usage

Finally, the overhead and resource usage of Misco is measured. The Nokia Energy Profiler [93] was used to monitor the CPU, memory and power consumption on the phones. The Misco system itself only requires 800KB of memory on the phones, very small compared the 90MB of available RAM. The experiment was performed on a worker tasked to perform 4 map tasks followed by 3 reduce tasks. The measurements starts 60 seconds before the actual experiment starts.

The power usage on the phones was monitored. Figure 3.24 shows the current draw used when running the system. Processing data requires 0.7 watts while network

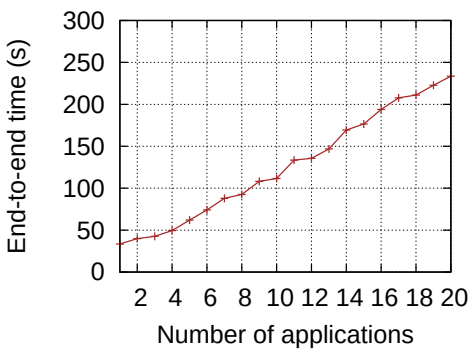


Figure 3.21: Performance of system as number of applications is increased.

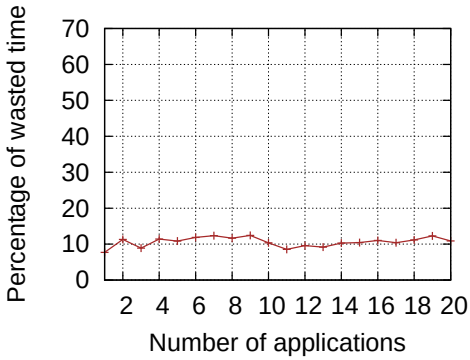


Figure 3.22: Wasted worker time as number of applications is increased.

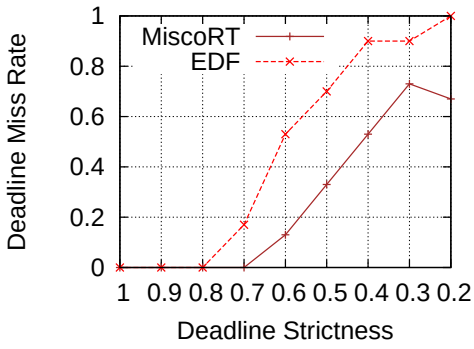


Figure 3.23: Sensitivity of system to deadline strictness.

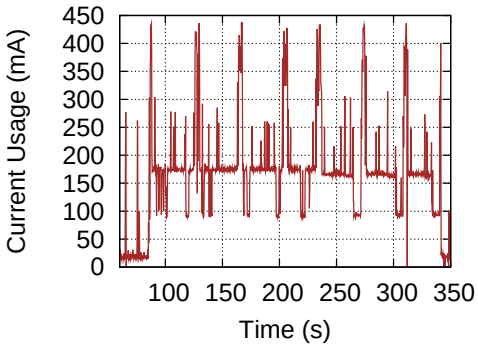


Figure 3.24: Phone client current draw

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, shown in Figure 3.25, for tasks is dependent on the application and also on any other programs or program schedulers running on the phone. Misco will gladly use any processing power available to it.

As the application runs, the memory usage is dependent on the user's modules and how much of the data is stored into memory. Figure 3.26 shows that during the map tasks, the key, value pairs are stored in memory and requires slightly more memory

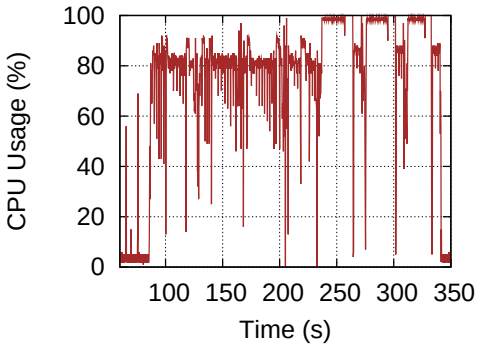


Figure 3.25: Phone client CPU usage

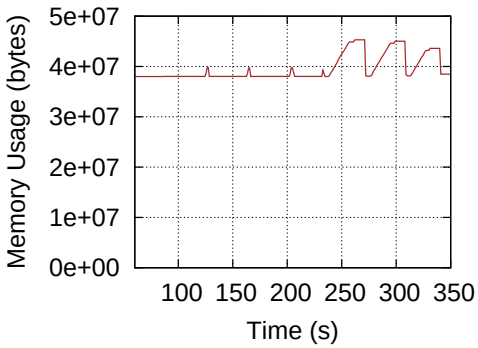


Figure 3.26: Phone client memory usage

than the input data pieces, up to an additional 300KB. The reduce partitions were larger and required an additional 700KB. These values are dependent on how the application is constructed and what data it stores during processing. The Misco system incurs very little overhead.

3.6 Discussion and Future Work

The idea of using smart-phones as part of distributed applications is fairly new and only recently been feasible due to technological advances in portable electronics. As a relatively new area, there are many areas that need to be explored such as energy, data locality, heterogeneity, real-time applications and security.

There has been some research in speeding up processing and providing incomplete or imprecise results. There are other approaches that speed up execution by optimizing the scheduling of applications and tasks. Research to providing faster results is very important for many of the monitoring applications, which must detect real-time problems, and for user facing applications, which must return results to the users within

a reasonable time frame.

3.6.1 Heterogeneous Devices

Since MapReduce is built on top of a collection of very loosely coupled components, it is reasonable to try increasing the performance and processing power of the system by allowing more devices to join the system. It would be relatively easy to extend the system to run on a variety of different devices with vastly different capabilities and processing powers. However, it is not clear how the work would be distributed, how capabilities of the system will be discovered and how to best allow the system to work together. For example, it would be impractical to process very large amounts of data on cell phones. Ideally, a large server farm would be used to do this processing. Conversely, it would be very difficult to use servers to gather and monitor noise over a city. Studying how to build harmonious and efficient heterogeneous systems is key to allowing for more portable and useful distributed systems.

3.6.2 Conserving Energy

One of the major concerns for developing systems for mobile devices is power usage. The biggest culprit of energy usage is network access. This fact makes for a very convincing case to fully apply the principles of data locality. Instead of transferring data to different machines for processing, try to perform the processing where the data is already located. This concept is not new and has already been investigated for data centers and there are methods to reduce data usage. The smart-phone based model is different. The majority of feasible application for this setting is not just using the phones

as processing units for raw data. Instead, the phones themselves are often the generator for the data, making it even more important for us to take advantage of data locality.

Conserving energy is not just constrained to the mobile space. It is also an area of study with a broad range of applications from electric vehicles to large data centers. Orthogonal research is being conducted on new, more efficient ways to generate and store energy. Renewable energy sources such as solar power and human generated kinematic energy are being developed to help power mobile electronics. Other interesting ideas such as wireless power supplies are also helping to solve this problem of providing energy. Research in providing more efficient energy will not just benefit this system, but has a large impact on many other areas.

3.6.3 Users and Their Phones

Unlike server machines and even general desktop computers, phones have a more personal connection with their users. Users generally own the phones they use and carry their phones with them at all times. This connection provides many advantages and disadvantages for using phones for distributed applications. Because phones are personal property, it is generally in the owner's best interests to keep their devices functioning properly. This eliminates the need for costly maintenance and power consumption of these devices. Also, the individual ownership of the devices encourages the development of portable and open standards without proprietary lock-in. A further advantage comes from the connection between owner and device. People are still the best at determining what events and things are interesting whether it's scenes to take pictures of or locations to congregate at. This user captured information is implicitly exciting and can be used

for applications many people are interested in.

However, there are quite a few disadvantages to using privately owned devices. The most significant would be the reluctance to provide personal computation resources for use in part of a distributed system. This can be addressed through providing applications which users are interested in participating in such as traffic or social applications, or by providing incentives for contribution. Another drawback of using mobile phones as a basis for applications is coverage. Since phones are with their users, there are likely areas with little or no coverage. This can be addressed by use hybrid systems composed of additional devices such as a wireless sensor network.

3.7 Summary

Mobile phones are becoming more powerful and more commonplace. Applications developed for these phones continue to evolve and advance. However, development tools for creating the types of distributed social and monitoring applications that people are demanding are difficult to use. This chapter looked at several current mobile applications and introduced the MapReduce framework as a solution for creating distributed applications for smart-phones and other devices. The efficiency of the MiscoRT system is demonstrated through experiments performed on a test-bed of Nokia N95 8GB smart-phones. As new developments for mobile devices progresses, the system is expected to be able to fully use any addition resources given to it and its performance will scale linearly.

There are many areas where it is not practical to use a mobile system, such as for processing very large quantities of data. The design of any system is based on a set

of trade-offs, on one end, specialized systems, such as databases, are highly optimized to perform specific computations. On the other end, completely open platforms allow for very flexible and complex applications, but only very basic optimization is possible. Using MapReduce as a basis for a distributed application framework was a choice on this spectrum of flexibility vs. optimizations. MapReduce has been proven to be very useful in large scale server environments and has spawned many more applications than was predicted. I hope that providing such a platform for mobile devices would also lead to similar benefits.

Chapter 4

Cacheflow: Scheduling Requests on Content Delivery Networks

The rapid growth in both Internet infrastructure and adoption has radically influenced every aspect of human society by enabling a wide range of applications and collaborative applications for business, commerce, entertainment and social networking. With the increase in rich and personalized content found in many web pages today, page load times are getting higher and maintaining user satisfaction has become a major concern.

People's needs to communicate and interact with each other has motivated better and more efficient ways to exhcange ideas, send and recieve messages, and share personal information with friends and family. Social networking applications have become the fastest growing phenomenon with a multitude of sites such as Facebook, Google+ and Twitter. Facebook, the leading social networking site, has over 800 million active users

who perform 3.9 trillion feed actions per day and serves over 200 billion web pages per month. Other sites also use personalized information to boost their business. Amazon, for example, list items and makes recommendatations to users based on their interests and purchasing history.

Loading web pages quickly isn't just a nicety for users. Studies have demonstrated the adverse effects of high latencies [62]: Amazon found that they lost 1% in sales for every 100ms of latency and Google found that an extra 500 ms dropped their search traffic by 20%. This latency is the result of an accumulation of individual latencies along an end-to-end network path. For example, when requesting a website, delays from the workstation, a DNS lookup, a cache miss at a proxy server, the network latency to the web server and processing on the web server all contribute to increasing the client's request response time.

Caching and content delivery networks (CDN) are two methods used to improve website response times. Caching is a popular and well studied method of improving network and system performance on the Internet. By placing copies of objects closer to the user, network latencies are reduced. However, traditional caching approaches are not well suited to handling the workload of sites which provide personalized contents such as Amazon and Twitter, or social networking sites such as Facebook and Google+. Caches are very effective at caching static web content where each user requests results in a single item being loaded. However, in the case of personalized website, a single page request can spawn hundreds of back end item requests on the server.

CDNs are a more recent approach that provides a similar function as proxy web

caches. They also store and serve client requests in a geo-distributed manner to reduce network latencies. However, they typically duplicate all the contents of the original servers to their nodes and can perform more sophisticated actions such as providing dynamic content. While CDNs are an effective way to increase client experience, they suffer from the same problem as proxy caches, they are not suited to handing the types of requests that social networks require.

In order to provide good response times, Facebook sets up large data centers with huge in memory caches to decrease server processing time. In order to store data at the scale that Facebook deals with requires a lot memory. This translates to a lot of servers. Servers are expensive. The energy required to run them is significant.

This chapter presents Cacheflow, a system for reducing client response time and improving server memory utilization by intelligently retriving items from multiple servers. The system is set in a CDN-like environment with shared cache knowledge between the server. Using the shared knowledge, servers collaborate to efficiently provide the items to satisfy a client's request. The primary objectives for the Cacheflow system are to (1) Provide better client latencies with same amount of memory resources and (2) Provide same client latencies with less memory resources.

4.1 Background and Related Works

Social networking sites such as Facebook [46], Google+ [57] and Twitter [107] serve content to millions of people, spread all over the world every day. Facebook [108] serves over 200 billion pages every month and processes over 3.9 trillion feed actions per

day in 2009. At the time of this writing [47], it has over 800 million active users, more than 75% of which are outside the United States.

A typical page on such a site usually consists of multiple objects personalized to the user. On a Facebook wall, for example, a user will see recent activities of their friends and topics that they choose to follow. These activities are represented by individual objects which generally follow a standard format of a picture and a short blurb of text. Each request for a page results in tens to hundreds of individual object requests. Since the page is personalized to each individual user, it is difficult to apply traditional web caching techniques more suitable for static pages. This structure of loading multiple personalized objects is shared by all social networking sites. Cacheflow is targeted to these settings.

The rest of this section explores related works in content delivery networks, distributed file systems and web caching.

4.1.1 Content Delivery Networks

Content Delivery Networks (CDNs) such as Akamai [94] and Limelight [66] have become a popular method of providing improved levels of performance, scalability and reliability for client-server applications. By locating contents and applications at geographically dispersed servers, the CDN can replicate and move data such that client requests can be serviced by a server at a location closer to it.

While most traditional CDN services focus on distributing static content such as web pages, images and large files, there have been improvements to provide support for dynamic applications and rich multimedia streaming. Currently, tens of thousands of CDN servers are deployed throughout the world, provide a major portion of Internet

traffic and make up a significant part of the Internet infrastructure. Akamai has $\sim 27,000$ data servers and ∼6,000 DNS servers while Limelight has $\sim 4,100$ data servers and ∼3,900 DNS servers.

In a CDN network, when a client first makes a request for an object, a DNS request is sent to its local DNS server. The local DNS server then contacts a DNS server in the CDN network which returns the IP of the data server which is closest to the client. The typical delay measured for performing the DNS lookup is from 130∼170 ms. This extra DNS lookup time was determined to be too costly and a different method of locating content was used in the Cacheflow system.

WhyHigh [75] is introduced as a tool to help diagnose latency problems affecting CDNs. Their approaches groups problem clients together by root cause such as inefficient routing and queuing of packets. This work is complementary to Cacheflow which provides a different method for a network to improve it's client response times.

Another technique for improving web page load times is introduced in [3]. The main issue addressed is the large RTTs between clients and servers and the number of RTTs required to load a page. Their approach is to study TCP and vary its initial congestion window parameter to decrease the time spent in the TCP slow-start phase. Similar to WhyHigh, this work is complementary. The principles of reducing RTT time and number of RTTs are used as a key part of Cacheflow.

Although a lot of inspiration was drawn from CDNs, such as requesting data from nearby servers, the problem setting is different. In the Cacheflow system, the content is not just single objects which are retrieved by the client, but requests which can possibly

spawn hundreds of object requests.

4.1.2 Distributed File Systems

There are several other types of systems which provide efficient methods of storing and accessing data. These include peer-to-peer (P2P) systems, structured P2P and distributed file systems. However, there is a wide range of focus for these systems including reliability, throughput, availability and security in addition to performance.

Unstructured P2P systems such as Gnutella [97] and Freenode [22] are composed of a large number of transient nodes. The main goal of these networks is the distribution of popular, often time large, files over a wide network. The network leverages the resources provided by all its connected nodes to distribute these files in a scalable way, each node downloads a piece of the file from multiple other nodes. Freenode expands on this idea of P2P file distribution but also adds anonymity for both writers and readers.

Structured P2P networks such as Pastry [98], Chord [104], and Tapestry [118] provide effective methods to store and access data in large overlay networks over the Internet. All these system are based on hashing an object to a key and then routing to the node responsible for the key range the object belong to. Although the different systems use different techniques, the basics of retrieving objects to serve user requests are the same.

Distributed file systems such Coda [102] provide a Unix style file system model to aid collaboration between physically dispersed users. Users are able to transparently make use of such a file system as if it were local. Bigtable [17] and the Google file system [55] provide an efficient data storage systems for structured data that scale to multiple

petabytes of data across thousands of commodity machines. Most of the distributed file systems are designed for use in controlled server cluster environments.

While these systems provide a good way to distribute data, the are not focused on the same problem area of Cacheflow, which is providing a low latency experience for clients access social network type sites.

4.1.3 Web Caching

Proxy caching and shared cache systems ([114] Internet Cache Protocol [111] [112], [1]) are shown to greatly increase cache rates, decrease Internet bandwidth and improve client latencies. However, the assumed system environment is different. In all existing works, it is assumed that clients are making requests for single items such as a web page, file, or image. In these systems, the techniques used rely on propagating a missed request to other caches or servers after an initial cache miss occurs. In system settings where each request leads to accessing hundreds of files, such an extra hop would be a source of an unacceptable delay. However, even with the disconnect in problem settings, there are many suggestions and lessons from these previous works on caching that are applicable.

In a study on Zipf-like distributions and web caches [14], it was found that web requests at a server follow a Zipf-like distribution where the relative probability of a request for the ith most popular item is proportional to $1/i^{\alpha}$. The study also showed that there is only a weak correlation between the frequency of a requested item and its size. From their results, they suggested that using independent clients requesting items following a Zipf-like distribution is an adequate model for web accesses. These results are

incorporated into the experiments for evaluating Cacheflow.

The problem of placing network caches in such a way to reduce the network traffic and minimize the average client delays is examined in [74]. Their setting allows for them to choose locations within the network to place these caches whereas in our setting, the locations of the servers is already determined. There may be an opportunity to use these techniques to bridge the network space between the servers and the end clients, but our servers were already distributed in a fashion to provide good for the clients.

Caching of heterogeneous sized objects is investigated in [1]. They presented a Pyramidal Selection Scheme which demonstrates much better performances than caches which do not use a size and cost based cache replacement and cache admission policies. However, they do point out that LRU is a very robust algorithm in practice when objects are uniform size.

The potential benefits from cooperative proxies are studied in [112] and they make several suggestions. For homogeneous user requests, taking advantage of the current state of other proxies can yield substantial benefits. My findings agree with this suggestion. However, in their setting, the client-to-proxy and proxy-to-proxy latencies are much smaller than proxy-to-server latencies, which does not apply in Cacheflow's setting where each proxy can act as the server.

Summary caches [48] are proposed as a scalable solution to sharing caches among web proxies. They use a bloom filter to economically store summary data regarding neighbor proxy's cache contents. However, they are most concerned about keeping the summary compact and lowering the overhead of summary sharing between proxies. Their

system achieves the same performance as the Internet Cache Protocol and does not take advantage of in memory caches like our system.

Shared memory caches such as Memcached [50] [100] create a key-value cache system using the main memories across multiple machines. The system is targeted toward commodity machines running on a local area network. Each machine runs a client which shares some portion of its memory with the entire system. This system is used for accessing individual files on local networks, Cacheflow deals with serving requests that access multiple objects over the internet.

4.2 System Settings

The system consists of:

- A collection of files. Each file is associated with a file name and a file size. No new files are introduced into the system and files are immutable.
- A Set consists of one or more files.
- A file may belong to multiple sets.

Server nodes and client nodes:

- Each server has a disk which contains all the files and also an in-memory cache which contains a subset of these files. Each server is also aware of all the set memberships.
- Each server is also aware of nearby servers and is updated on the cache occupancy of their nearby servers.

- When a server receives a request for a file which is not already in its cache, it must fetch it from disk. Each disk access incurs both a disk access time and a data transfer time.

- Clients make requests to the servers for sets. They are not aware of the contents of any of the sets. The client initially aware of only its local server.

This model is similar to real-world systems such as a Facebook wall. When a user requests for the wall to be loaded, different individual items which make up the wall (a set) must be loaded and these sets may be different for individual users but can contain items which are part of other user's requests.

4.3 System Design

The goal of the Cacheflow system is to provide an increased quality of service to users access web resources by lowering the total time required to complete each user request. This section details the main insights which guide the design decisions, the design of the major components of Cacheflow and also a cache hit rate model to predict cache hit rates.

The key insights which guild Cacheflow's design are:

- RTT times should be as low as possible and the number of trips required should be small.

- Disk seek times and read times are costly and should be avoided when possible, especially if there are multiple files involved.

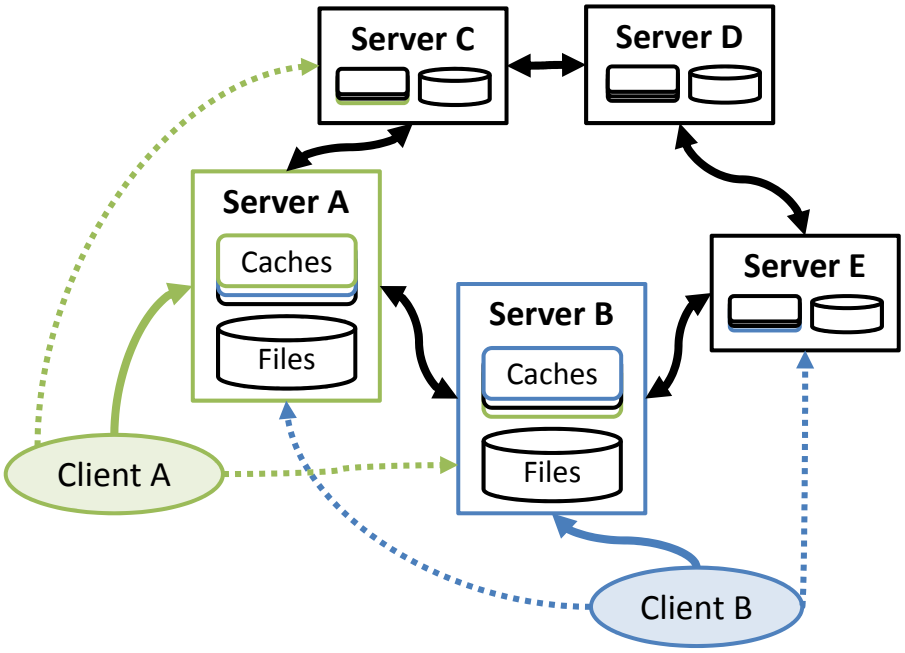


Figure 4.1: An overview diagram showing a group of servers with their neighbors. Also shows two clients and their local and remote servers.

- It may be faster to load files from a remote server's main memory than reading the file from a local server's disk.

The goal of the system is to achieve two results: 1) The same level of performance of existing systems using less total memory across servers or 2) a higher level of performance than existing systems while using the same amount of memory.

Our system consists of two main components, the Server and the Client, Figure 4.1 provides a basic overview of the server and client connections.

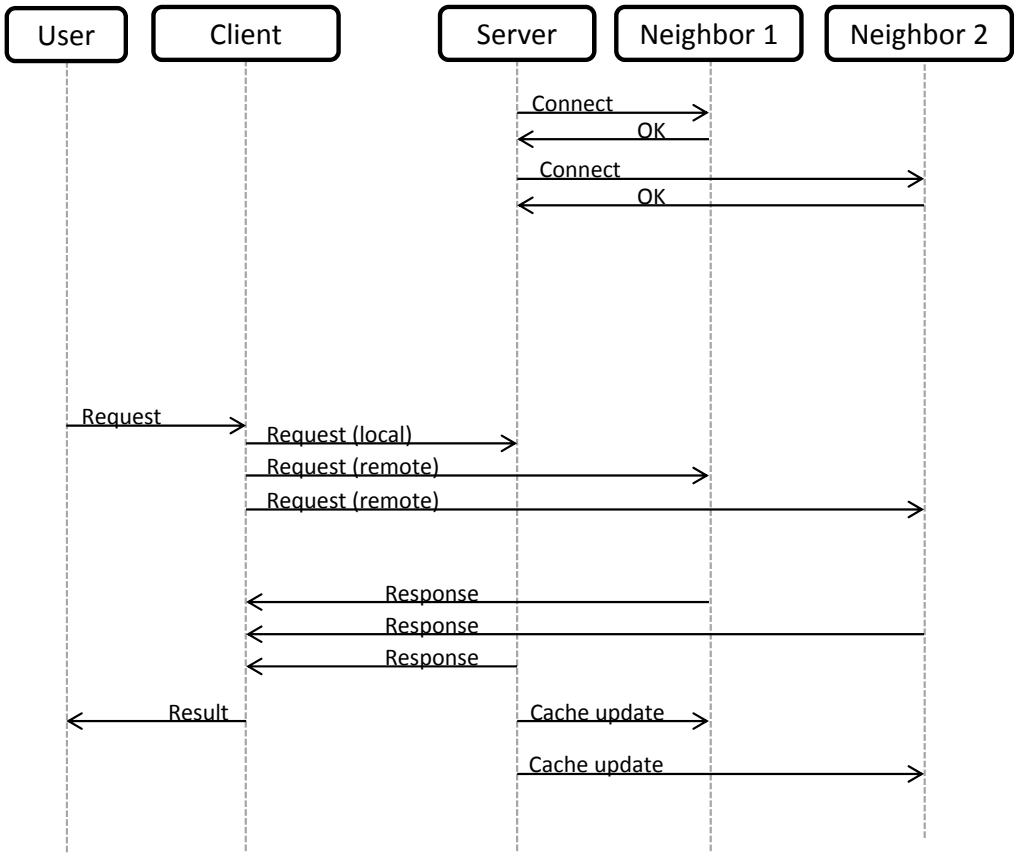


Figure 4.2: Time-line diagram demonstrating initial server setup and a client request being handled.

4.3.1 Server

The primary duty of the server is to act as a web-server, handling client requests and returning the files to fulfill the client requests. Each server keeps track of an adjustable number of nearby server, this list of servers is determined by an initial discovery phase. These nearby servers which are on the server's list will be referred to as neighbors.

There are many ways of determining the list of neighbors; they can be determined automatically or set manually. For Cacheflow, servers automatically determine their

neighbors. During the discovery phase, the server first determines the latency to each server in the system and sends a neighbor request to each of these servers in order of lowest to highest latency. This process stops when a preset number of neighbors have been reached. To avoid flooding the entire system with messages and to prevent the same pair of servers from trying to add each other simultaneously servers are only allowed to initialize connection attempts to those whose hostname lexicographically proceeds their own.

Once a list of nearby neighbor servers is established, any cache updates that occur will be sent to all neighbors. The sending of these messages can occur individually when a change occurs or batched together until some condition is met. Batched message can be sent once a certain number of update messages is reached, periodically, separated by some time period, or occur when some other even occurs. Cacheflow currently uses a link buffer value which indicates the number of messages to buffer before sending an update to the neighbor.

Each server maintains a view of the cache contents for each of its neighbors. Whenever a cache update message is received, the corresponding neighbor's cache view is updated. In addition to just updating the view for that neighbor, the server also uses these updates to influence its own cache replacement policy. If an item is added to a neighbor's cache which is also in the server's cache, the item is placed in a low-priority list which indicates files which can be removed from cache first.

There are three basic request types that the client makes: a request for a list of neighbors, a local request for a set and a remote request for a set. The first type is

handled simply by returning the list of neighbor server IPs (this avoids a potential DNS lookup for the client). The local and remote request types differ in that the client is closer to the server in the local case and thus, the server has more time to perform disk accesses to fulfill the request. The dynamics of local vs remote requests will be examined more thoroughly in Section 4.3.3.

Each server has an in memory cache used to store files, a least recently used (LRU) cache replacement strategy is used due to web accesses and file popularity generally following a Zipfian Distribution. When removing item from cache, before applying LRU, a check is first performed to see if any files are in the low-priority list (mentioned above) that can be remove first. By doing this, a large effective cache is created by reducing duplicate items across servers.

4.3.2 Client

The client requests sets of files. Initially, the client chooses a server by picking one with the lowest latency, this server will be referred to as the local server for the client. The client will make a one time request to the server to discover its neighbor servers, these servers will be referred to as remote servers.

For each request, the client sends it to the local server and each of the neighbor servers in parallel. It indicates in the request whether the request is local or remote. Once the client has received the results from the servers (or a timeout occurs), it will present the results to the user.

4.3.3 Request Handling

Since requests are sent to both remote and local servers simultaneously, returning duplicate results form multiple servers should be avoided where possible.

When a local request arrives a server, the server first gets all the files from the requested set which are in its cache. Then, it loads the files which are not in any neighbor cache from the disk. The server also checks each neighbor's cache contents.

A threshold value is introduced here to try to prevent the system from performing redundant work with no benefit. If there are any neighbors whose cache contains less than a threshold amount of files, these files are also loaded by the local cache and served by the server. A message is passed back to the client informing it of which remote server's cache content has been loaded. By doing this, the client can disregard any reply from the remote server. For a remote request, the server first checks to see if the number of files in its cache reaches the threshold, if it doesn't, the request is disregarded. Otherwise, it returns the files from the request set which are in its cache.

The combined effects of network latency and link buffers causes some inaccuracies in a server's view of neighbor caches. This, in turn, causes the server to mispredict some of the items which it needs to load from disk. The experiments in Section 4.5 show that most of the items are served as requested. Also, due to the nature of the social networking and retail suggestion applications, it is not critical for all the items to load.

The total amount of time per request can be determined by the maximum time required for each request to complete. In the following equation, rtt_{local} is the local RTT, $rtt_{neighborX}$ is the remote RTT for neighbor X, $time_m$ is the average per item-miss

handling time and m is the number of cache misses.

$$time_{req} = MAX(rtt_{local} + time_m * m, rtt_{remote1}, ..., r_{remoteN}) \tag{4.1}$$

4.3.4 Cache Hitrate Model

In this section, the expected hit rate for cache flow is modeled where the popularity of the files and requests follow a 20-80 type rule (simplified Zipfian distribution). It is assumed that all files have a uniform size.

N - total number of files

p - portion of files that are popular

S - number of items in a request set

q - portion of request set that are popular files

T - size of cache in number of files

E_n - Expected cache hit rate for nth request

E_n^p - Expected popular item cache hits for nth request

E_n^u - Expected unpopular item cache hits for nth request

T_n^p - Number of popular files in cache for nth request

T_n^u - Number of unpopular files in the cache for nth request

First, the expected number of hits are calcuated for any request by summing the expected number popular item cache hits and the unpopular item cache hits:

$$
\begin{aligned}
E_n^p &= ((q * S)/(p * N)) * T_n^p \\
E_n^u &= (((1-q)S)/((1-p)N)) * T_n^u \\
E_n &= E_n^p + E_n^u
\end{aligned}
\tag{4.2}
$$

Then, the number of popular and unpopular items in the cache can be calcuated through a set of recurrence equations given below. Each equation calculates the number of popular and unpopular files in the cache using the results of the previous iteration. The predicted hit rate is the expected hit rate after the number of items in the cache reaches T.

When $T_n^p + T_n^u < T - (S - E_n)$:

$$
\begin{aligned}
T_{n+1}^p &= (S * q - E_n^p) + T_n^p \\
T_{n+1}^u &= (S * (1-q) - E_n^u) + T_n^u
\end{aligned}
\tag{4.3}
$$

Else :

$$
\begin{aligned}
T_{n+1}^p &= (S * q - E_n^p) + T_n^p \\
&\quad - ((T_n^p - E_n^p)/(T_n^p + T_n^u - E_n)) * (T_n^p + T_n^u + (S - E) - T) \\
T_{n+1}^u &= (S * (1-q) - E_n^u) + T_n^u \\
&\quad - ((T_n^u - E_n^u)/(T_n^p + T_n^u - E_n)) * (T_n^p + T_n^u + (S - E) - T)
\end{aligned}
\tag{4.4}
$$

4.4 Implementation and Experiment Settings

This section provides the implementation details of Cacheflow and also explains some of the specific steps needed to prepare for the experiment environment. All of the

Useflow	Turns on and off the use of Cacheflow
Cache Size	Sets the size of the cache at each server
Files	Total number of files in system
Sets	Total number of sets in system
Files per Set	Number of files in each set
Number of Servers	Total number of servers to use
Clients per Server	Total number of client to connect to each server
Link Buffer	Number of cache status update messages to store at each server before sending them out
Threshold	Threshold for number of files in cache of neighbor before those files are loaded at the local server

Table 4.1: Parameters for Cacheflow system.

components of the system is implemented in Java. Most of the important parameters are coded in such a way that they are tunable using a configuration file. A list of these parameters and their purpose can be found in Table 4.1.

4.4.1 PlanetLab

In addition building the system, there are several particularities of the experiment platform, PlanetLab, had to be accounted for. PlanetLab is composed of a global collection of computers provided by universities and research organizations scattered around the world. There are approximately 900 registered nodes in the PlanetLab system at the time of this writing but not all of them are in a working state or capable of supporting the Cacheflow system. Users of PlanetLab are given a slice which is essentially a virtual machine at each node that the user wants to use. This resource sharing by multiple users results in unpredictable loads at the nodes. In addition, the statuses of the nodes and

their connections to each other are constantly fluctuating.

To produce usable nodes, the nodes are filtered through several steps. First, a Python PlanetLab API is used to determine all the nodes which have reported themselves to be in a boot state. Then, the up-times of the booted nodes are queried and the nodes which have been up for more than one day are kept. Next, Java and atd are installed onto these stable nodes and the nodes which fail to install the necessary software are cut. A single node with the lightest reported CPU load is selected from each physical site. Having multiple nodes from the same location participate in Cacheflow would lead to unrealistically low latency times for certain client-server pairings. After all the filtering stages, approximately 100 usable nodes remain. Of these 100 nodes which have passed the automatic filtering process, there are still several problem nodes which are manually filter out. These problems nodes consistently exhibit a large amount of errors (such as refusing connections and connection timeouts).

4.4.2 Controller

To reduce the effect of faulty nodes or nodes failing to start up properly, a controller component was created. The controller is responsible for keeping track of the states of the nodes and also selects which nodes to use in an experiment.

There is a single instance of the controller which sits at a reliable node and listens for TCP connections on a known port. Each server and client in the system knows the location of the controller. When a controller starts up, it first downloads the latest configuration files and parses the experiment settings. This includes the number of servers to use and how many clients to assign to each server. The controller then waits for nodes

to connect to it. A connecting node will establish a TCP connection with the controller and inform the controller of its role, this includes only its hostname in the case of a server or a hostname, local server and server latency in the case of a client. The controller keeps track of how many servers and clients have connected to it. It also keeps track of which servers the clients have selected as their local servers. Once the total number of servers and clients per server is reached, the controller sends a message to the servers telling them to perform their initialization. It then waits for the servers to respond with a ready message. When all the servers have responded, the controller informs a set of number of clients per server to start.

In addition to preparing and starting the servers and clients, the controller also sends period heartbeat messages to all the nodes. The server nodes reply with a status update message which includes the number of requests it has processed and also information regarding the last request (or current request) that they handled. The client nodes reply with the total number of requests they have handled, any failed requests and information regarding their last request (or current request). With this information the controller can provide a real-time display on the status of the system which was used heavily when debugging issues.

By using a controller, it is fairly certain the nodes being used are all running the system and in working condition. It also allows for the fine tuning of load distribution across servers and provides a valuable monitoring and debugging tools.

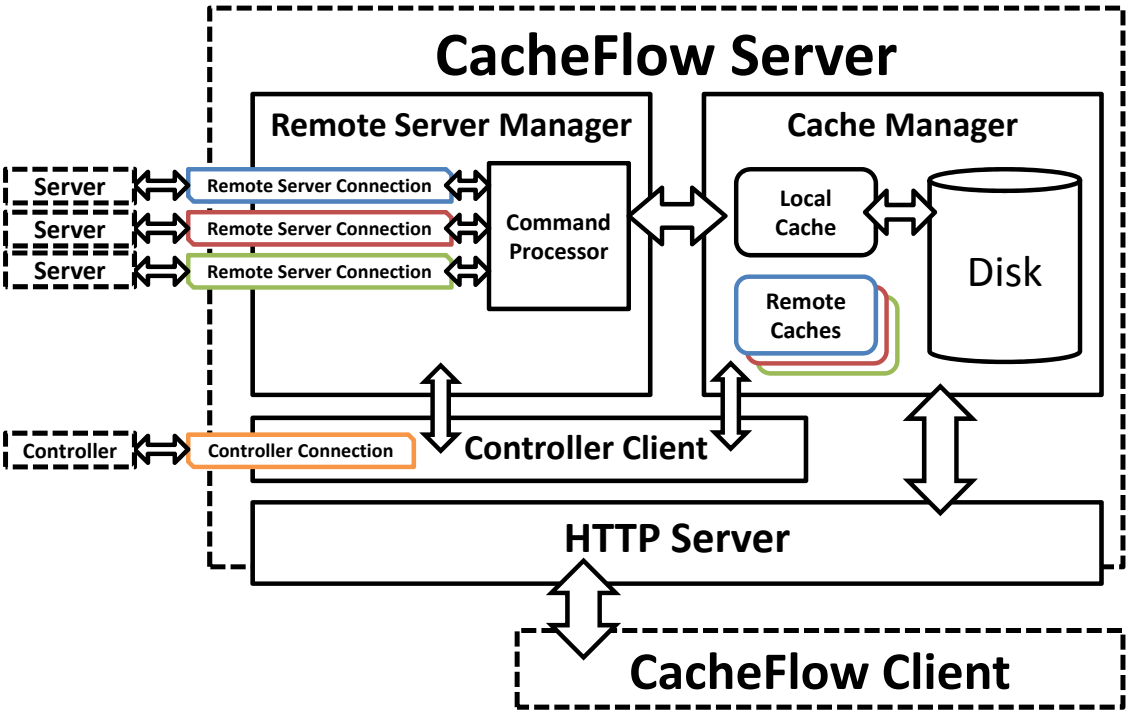


Figure 4.3: Architectural diagram of the Cacheflow Server

4.4.3 Server

Instead of implementing a web server from scratch, Cacheflow's server was extended from an existing web server, NanoHTTPD [41]. NanoHTTPD is a simple, small and multi-threaded. The web server's serve method is overloaded to intercept Cacheflow requests and inject Cacheflow result data into the response.

An architectural diagram of the server is shown in figure 4.3. The server is divided into two major components: the the cache manager and the remote server manager. The remaining parts of the server handle communication with the controller and interaction with the web server.

The *Cache Manager* is in charge of maintaining the local cache and the remote caches. It is also in charge of servicing requests which arrive via the HTTP server. The local cache is backed by a simulated disk which maintains a list of file names, their associated file sizes and the set memberships. The request handling process is detailed in Section 4.3.3. For the local cache, information regarding the file and the last access time is stored, for the remote caches, their cache contents is simply stored as a set of file names.

The *Remote Server Manager* sets up the connections between a server and its neighbors. It also maintains communications with neighbor servers, updates the remote cache contents and sends local cache updates to neighbors. The initial neighbor selection method is described in Section 4.3.1. A *RemoteConnection* object is created to interact with each neighbor using a TCP socket connection. All communications are carried out through these objects, any received messages are processed through a command processor. Status updates to caches are then passed by the command processor to the Cache Manager.

4.4.4 Client

The Cacheflow client is much simpler than the server. It has a component which communicates with the controller, informing the controller which server it has picked as a local server and then waiting for the start signal from the controller. The rest of the client is simply a loop which periodically makes requests for sets from the server. The requests are made to the local server and any remote servers simultaneously using multiple threads. Then the client waits for the results from all the servers.

Parameter	Default Value
Cache Size	30,000 KB
Files	5,000
File Size	50 KB (uniform)
Sets	1,000
Files per Set	100
Number of Servers	6
Clients per Server	3
Link Buffer	0
Threshold	0

Table 4.2: Default parameters used in experiments.

4.5 Results

This section explores the latencies (Round Trip Times RTTs) in the Planetlab system, statistics from a representative run of Cacheflow, the effects of cache size, link buffer size and file thresholds. The default parameter values can be found in Table 4.2 (there is a table describing the parameters in Table 4.1).

For the experiments, a number of servers is first chosen at random and clients are allowed to connect to them. Through experiments on PlanetLab, it was found that the system can consistently support 6 servers with 3 clients connecting to each server. If a higher number of servers or clients per server was attempted, there would frequently be cases where the desired distribution of clients to servers could not be met.

A total of 5,000 files where used with uniform file sizes of 50KB per file. The file sets were generated using a 80-20 distribution of files which is a simplified approximation

of a Zipf-like distribution (20% of the files were picked 80% of the time for each file in the set). 1,000 total sets were generated. [14] showed that web requests from clients follow a Zipf-like distribution and that a system may be sufficiently modeled by independent clients making requests following a Zipf-like distribution. The paper also goes on to find that there is a weak correlation between the access frequency of a web page and its size. This is particularly true in the Cacheflow setting considering the target applications. Facebook pages, for example, are generally composed of items with standardized formatting and thus are roughly equivalently sized. This format is similar in other systems such as Google+, Twitter and Amazon. Influenced by this, when making requests for sets, the clients also choose which set to request following a 80-20 distribution where 80% of the requests were from the same 20% of sets.

After the system starts, each client makes periodic requests every 5 seconds. A smaller request period was initially attempted, but resulted in server failures due to the temperamental nature of Planetlab nodes. This also keeps the hardware from saturating, allowing the request latencies due to RTT to dominate. For each set of parameters, the system was allowed to run for 30 minutes before being shutdown and the results taken.

4.5.1 RTT Measurements

In a real system, there would be many more servers than used during these experiments. By having more servers, clients would be closer to any given server than in experiments with only 6 servers. To compensate for the lack of available servers, an additional measurement experiment was performed to determine what the RTTs would be if more servers were used.

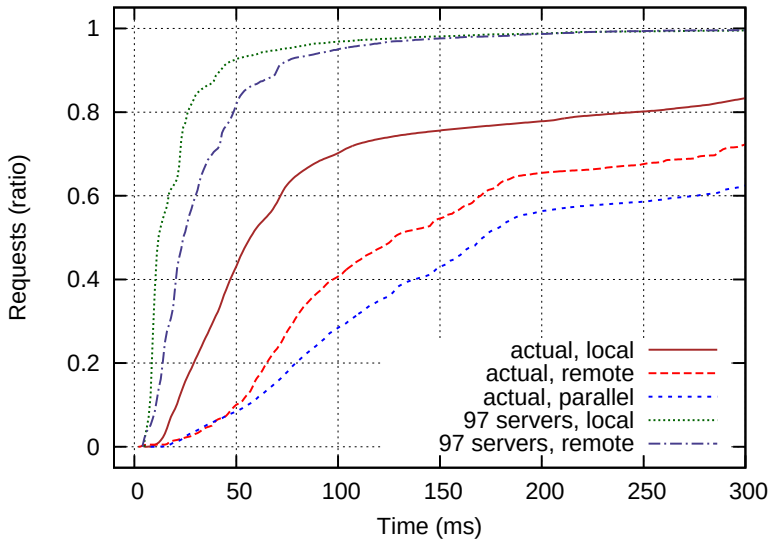


Figure 4.4: RTTs for client-servers for cases with 97 servers and 6 servers.

For this experiment, 98 nodes were available. 1 node was selected as the client node and the other 97 nodes acted as servers. Each server node picked 2 neighbor nodes which were closest to it. Then, the client node connected to its closest server node and the servers' 2 neighbors to measure their RTTs. This was repeated with each node taking a turn as the client node. In this fashion, the real world RTTs can be estimated if the system has 97 servers.

The RTTs were then compared with the actual RTTs measured during the course of the experiments. The resulting Cumulative Distribution Functions (CDFs) of the RTTs are shown in Figure 4.4. The figure also shows the CDF for parallel requests, labeled *actual, parallel*, this line represents the time required for all 3 requests to complete for the client. The 3 requests are run in separate threads, one is for the local server and two

Setting	Value	Description
useflow	true	Uses cacheflow
cachecapacity	30000	Size of Cache (KB)
nslinks	2	# of neighbor per servers
filedisk	disk-u-50.db	Simulated disk file
filesets	sets-et-1000.db	Simulated sets file
linkbuffer	600	# of cache status update messages to buffer
num sets	1000	Number of different request sets
set size	100	Number of files in a request set
num files	5000	Number of different files in system
file size	250000 KB (50 each)	Total and average file sizes

Table 4.3: Sample run experiment settings

are for the remote neighbor servers.

As the results show, the actual RTTs when using only 6 servers is much worse than the expected RTTs from an actual network where there are more servers. The average time for the 97 server case was 27ms for local and 37 ms for remote, with median values of 11 ms and 23 ms respectively. In the case of the test runs, the average local RTT is 133ms (57ms median) and average remote RTT is 204ms (128ms median). When in parallel, 279ms average, 179ms median. From these numbers, the latency in the experiment setting is about 5 times worse than the 97 server case. This value is used later to scale the performance numbers in Section 4.5.3.

server	reqs	local hits	remote hits
pluto.cs.brown.edu	2751	32	33
planetlab1.dtc.umn.edu	2790	33	33
planetlab-2.cs.colostate.edu	2771	33	33
nodeb.howard.edu	2755	33	34
planetlab1.cs.uoregon.edu	2762	33	33
planetlab2.georgetown.edu	2737	33	33

Table 4.4: Server statistics.

4.5.2 Sample Run

This section presents a representative experiment run. Table 4.3 shows the parameters used in this run. Table 4.4 contains the results seen at the servers. It shows the total number of requests served along with the number of cache hits when the request source is local or remote. Table 4.5 shows the mappings of clients to their local servers and also the measured RTT at the time the client selected the server. Finally, Table 4.6 contains the results seen by the clients. This table shows the number of requests, the number of failures caused by server problems or server timeouts (requests taking more than 1s), local cache hits, local disk accesses (misses at local and neighbor server caches), the remote cache hits and the total number of unique files actually downloaded. This total number of files is less than the expected 100 due to server mispredictions. The table also shows the average RTTs for local and remote requests. Figure 4.5 shows the CDF of the cache hits for local servers and for remote neighbor servers. The number of hits in the local cache is slightly higher than in remote caches.

#	client	server	RTT (ms)
1	planetlab1.cnis.nyit.edu	pluto.cs.brown.edu	14.9
2	planetlab4.rutgers.edu	pluto.cs.brown.edu	10.3
3	planetlabone.ccs.neu.edu	pluto.cs.brown.edu	2.3
4	planetlab-2.cs.uic.edu	planetlab1.dtc.umn.edu	12.7
5	planetlab4.cse.nd.edu	planetlab1.dtc.umn.edu	22.9
6	planetlab2.mnlab.cti.depaul.edu	planetlab1.dtc.umn.edu	12.5
7	pl-dccd-02.cua.uam.mx	planetlab-2.cs.colostate.edu	80.9
8	planetlab6.csres.utexas.edu	planetlab-2.cs.colostate.edu	37.4
9	ricepl-5.cs.rice.edu	planetlab-2.cs.colostate.edu	28.9
10	planetlab1.pop-rs.rnp.br	nodeb.howard.edu	158.0
11	planetlab2.cis.upenn.edu	nodeb.howard.edu	7.3
12	planetlab2.tsuniv.edu	nodeb.howard.edu	39.8
13	planet4.cs.ucsb.edu	planetlab1.cs.uoregon.edu	28.0
14	pl-node-1.csl.sri.com	planetlab1.cs.uoregon.edu	21.8
15	planetlab1.eecs.wsu.edu	planetlab1.cs.uoregon.edu	45.5
16	planet2.cs.rochester.edu	planetlab2.georgetown.edu	16.2
17	planetlab7.cs.duke.edu	planetlab2.georgetown.edu	10.5
18	planet11.csc.ncsu.edu	planetlab2.georgetown.edu	16.7

Table 4.5: Mapping between Clients and servers

#	client	reqs	fails	locH	locD	remH	files	locT	remT
1	cnis.nyit.edu	305	2	33	24	39	97.2	60	197
2	rutgers.edu	302	4	33	24	39	96.9	53	185
3	ccs.neu.edu	305	4	33	24	38	97.0	42	173
4	cs.uic.edu	303	9	33	24	39	97.0	46	91
5	cse.nd.edu	297	14	33	23	39	96.7	61	131
6	mnlab.cti.depaul.edu	298	11	34	23	39	97.1	44	79
7	cua.uam.mx	299	4	33	24	38	96.9	217	234
8	csres.utexas.edu	309	1	33	24	38	96.9	97	138
9	cs.rice.edu	312	0	34	24	38	97.2	71	128
10	pop-rs.rnp.br	286	8	34	23	39	97.1	331	370
11	cis.upenn.edu	306	9	33	23	39	96.6	31	51
12	tsuniv.edu	298	9	34	23	39	97.1	104	168
13	cs.ucsb.edu	307	1	33	24	38	96.8	78	191
14	csl.sri.com	304	3	33	23	39	96.8	80	218
15	eecs.wsu.edu	298	2	34	24	38	96.9	109	352
16	cs.rochester.edu	300	11	33	24	38	97.0	54	80
17	cs.duke.edu	298	14	33	24	39	97.1	42	81
18	csc.ncsu.edu	295	16	34	24	39	97.4	56	95

Table 4.6: Client Statistics. $fails$ - number of request failures or server timeouts, $locH$ - local cache hits, $locD$ - local disk access, $remH$ - remote cache hits, $files$ - unique files downloaded by client, $locT$ - RTT for local server, $remT$ - RTT for remote servers

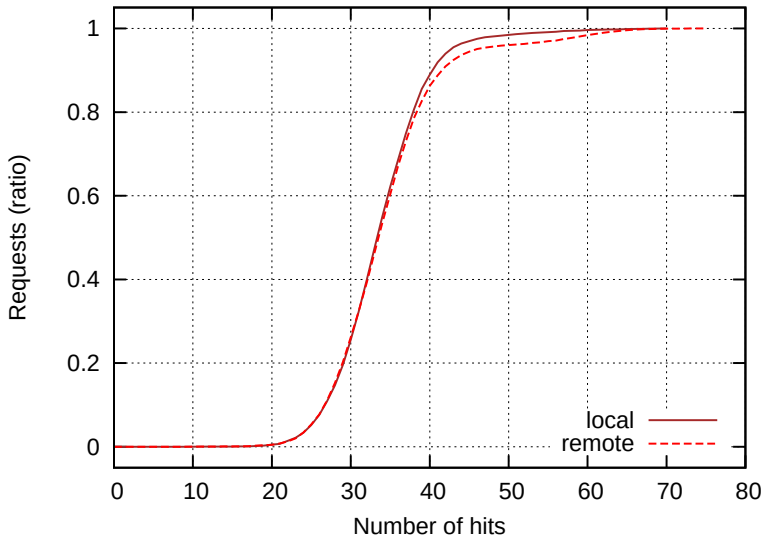


Figure 4.5: CDF for the number of cache hits at the local server and remote neighbor servers.

4.5.3 Cache Sizes

In this set of experiments, the cache size is varied while holding the other parameters to the default values. Figure 4.6 shows the overall cache hit rate with Cacheflow enabled, the cache rate at the local server only and the cache hit rate when not using Cacheflow. As the figure shows, Cacheflow greatly increases the cache hit rates. Both hit rate graphs grow logarithmically as expected [14]. The results also show that the amount of items cached at the local server is lower than without Cacheflow. This is due to the cache items being split between 3 servers with no duplicate items.

Since I did not want to speculate on the miss handling characteristics of a server (disk arrays, etc), I simply calculate what the average item-miss handling time would be required before Cacheflow starts to outperforms traditional systems. Due to the limited

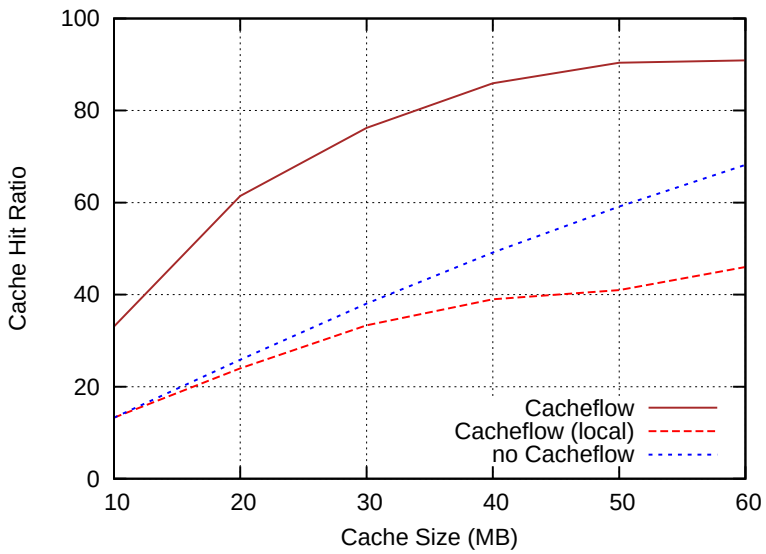


Figure 4.6: Hit rate when cache sizes are varied.

number of servers, the values were scaled down the numbers by a factor of 5, as determined in in Section 4.5.1. As Figure 4.7 shows, the per miss handling time for each item has to average below 1ms for non-Cacheflow systems to start performing better. For comparison, 6ms is the typical disk access time in a server disk array [109]. The equation used to determine the per item-miss time is:

$$time_m = (MAX(rtt_{local}, rtt_{remote1}, ..., r_{remoteN}) - rtt_{local})/hitrate \quad (4.5)$$

4.5.4 Model Validation

The results obtained in Section 4.5.3 are compared to hit rates obtained from applying the model in Section 4.3.4. As Figure 4.8 shows, the model is very accurate

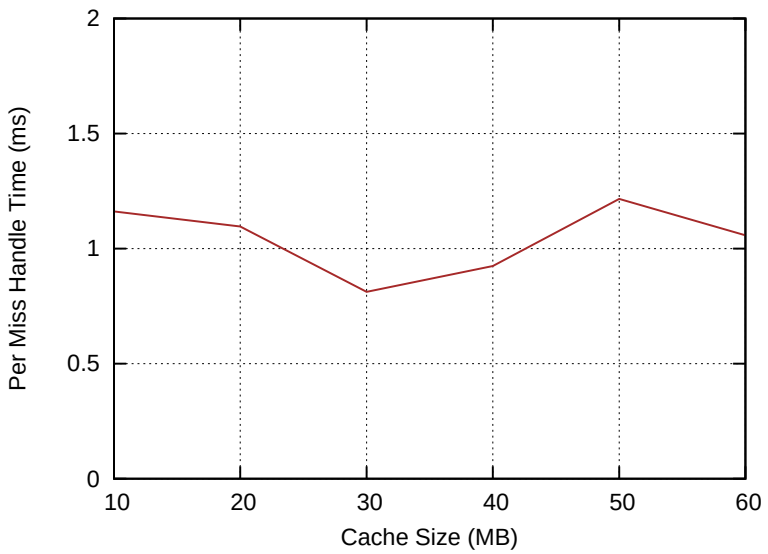


Figure 4.7: The minimum average per miss handling time required before Cacheflow outperforms non-Cacheflow system.

when predicting the hit rate for the non-Cacheflow case. Without Cacheflow setting of 2 neighbors, the model is less accurate. This is because while cache content is shared between a local server and its neighbors, the two neighbors do not share information with each other. In this case, the actual hit rate is somewhere between when the total cache size is 2 and 3 times the cache size of a single server.

4.5.5 Link Buffer

The link buffer value defines how many status messages to buffer together before actually sending a message to a neighbor server. The benefit of increasing this value is the reduction in the amount of messages sent, thus lowering the networking overhead for the system. The adverse effect is a decrease in cache hit rate due to mispredicting a neighbor's cache contents and not retrieving all the contents. Figure 4.9 shows that there

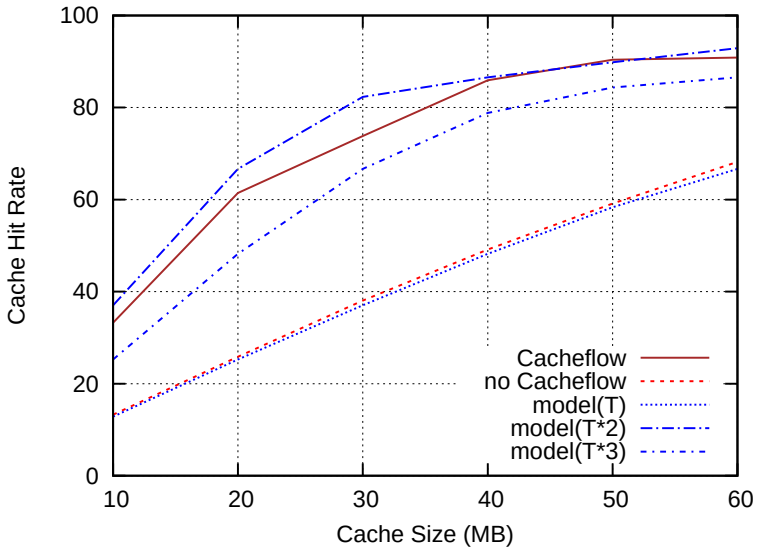


Figure 4.8: Validation of model, comparing hit rates predicted by the model to actual hit rates. There are three model hit rate lines, corresponding to cases were the used Cache size is 1x, 2x and 3x the actual cache size, T.

is a moderate drop in cache hit rate as the link buffer value is varied. Figure 4.10 shows the rate of mispredictions increase as the link buffer value is varied. There is always a bit of misprediction present in the system due to the time it takes messages to be transfered to and from neighbors.

Figure 4.11 takes a more detailed look at the distribution of mispredictions. This figure shows the CDF of the number of mispredictions per request for several different values of the link buffer. As the link buffer value increase, there are many more and larger mispredictions.

Cache status updates are generated only when the cache contents change. This means the number of cache messages is proportional to the miss rate and the rate of

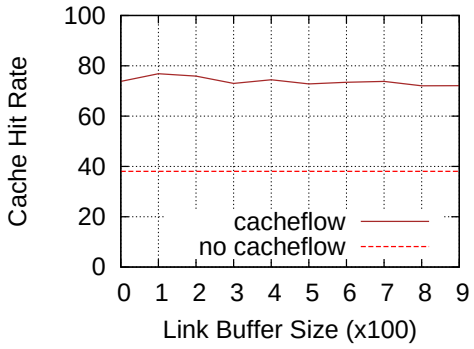


Figure 4.9: Hit rates when link buffer values are varied. The cache sizes are set to 30 MB

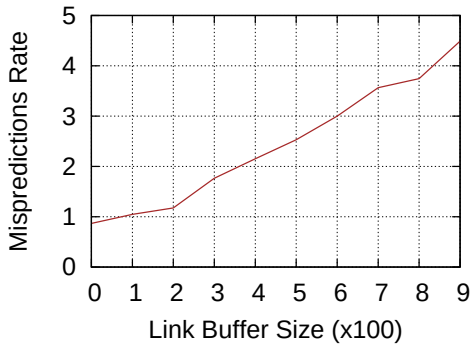


Figure 4.10: Ratio of mispredictions when link buffer values are varied. The cache sizes are set to 30 MB

sending status updates is the link buffer size divided by the number of messages per update. This link buffer is also limited by the packet size of the transmission protocol, in the case of TCP, the transmission window is dependent on link quality and the actual protocol.

4.5.6 Thresholds

The threshold value is used to prevent redundant requests to remote neighbors when it is faster to load data from the disk of a local server. The positive impact of using a threshold value depends on the ability to accurately predict the RTT between the client and the remote server. For this experiment, instead of predicting this RTT, the threshold value was just varied. This set of experiments use a cache size of 30 MB. Two link buffer values are used for comparison: 0 and 600.

Figure 4.12 shows the cache hit rate as the files threshold value is varied. There is a downward trend in the cache hit rate as the system uses less remote neighbors. At

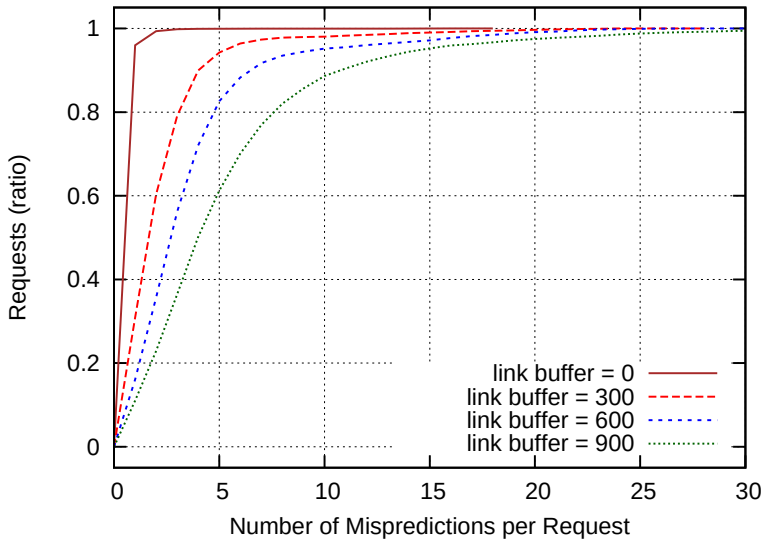


Figure 4.11: CDF of mispredictions with different link buffer values.

high threshold values, all the files are fetched locally and there is no difference between Cacheflow and no Cacheflow. When the link buffer value is at 600, hit rates are slightly lower than when there is no buffering of status messages, but is very slight (also shown in Figure 4.9).

The minimum average item-miss handling time that would be required before Cacheflow starts to outperform traditional systems is show in Figure 4.13. As the figure shows, the per-miss handling time is fairly constant until the threshold reaches 30, that's when the hit rates start to drop due to many requests only using the local server. When the link buffer is set to 600, it appears to perform better, but this is due to the mispredicted items which are not counted as misses.

Figure 4.14 shows the rate of mispredictions as the files threshold value is in-

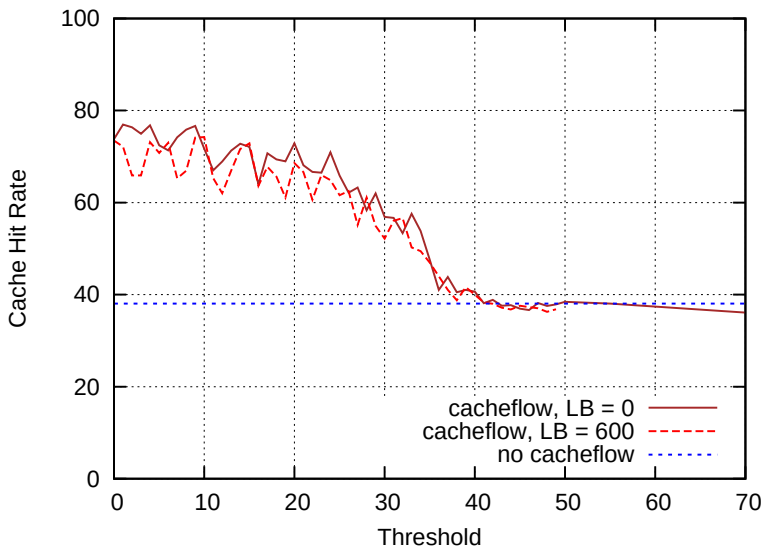


Figure 4.12: Cache hit rates when file threshold values are varied. The cache sizes are set to 30 MB

creased. Unlike increasing the link buffer value, increasing the threshold leads to a rapid decrease in mispredictions. As the threshold value increase, the use of neighbor caches also decreases, leading to this phenomenon. Figure 4.15 shows the ratio of all requests affected by the threshold increases as the threshold value is increased. The request ratio affected are smaller when the link buffer is higher.

4.6 Discussion and Future Work

In this section I discuss my experience designing Cacheflow, how to set the system parameters and several areas where the system can be improved.

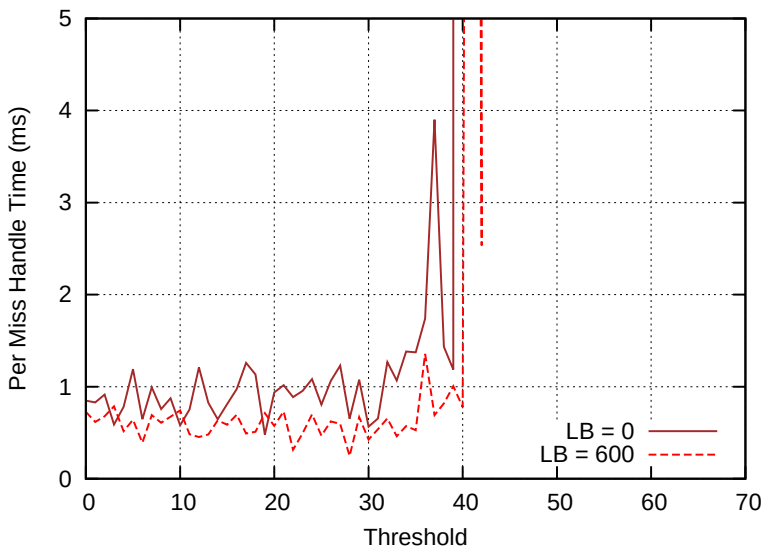


Figure 4.13: Per miss handling times when file threshold values are varied. The cache sizes are set to 30 MB

4.6.1 DNS Server

The initial design mimicked traditional CDNs more closely with two types of servers, a DNS server and a web server. The DNS server was in charge of keeping track of local and remote caches. When a client makes a request, the request is first sent to the local DNS server which returns the address of a web server that contains the largest portion of the request in cache.

Using this method, the results were only marginally better than just always accessing the local web server. The largest portion of time was taken by the two RTTs, one for accessing the DNS server and one for accessing the actual web server. The other major performance lose was due to loading all the files from one server, it seemed too costly to have the client request lists of specific files from individual servers. These issues

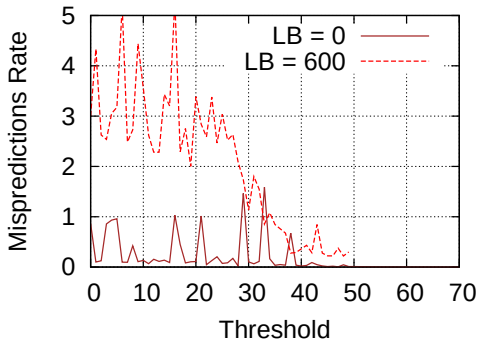


Figure 4.14: Ratio of mispredictions when file threshold values are varied. The cache sizes are set to 30 MB

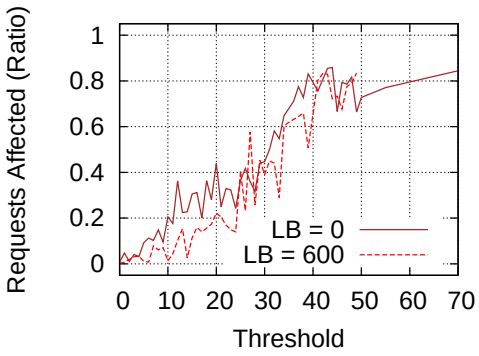


Figure 4.15: Number of requests affected by applying a file threshold.

were addressed to arrive at the final Cacheflow system.

Another drawback of having a separate DNS server was the updates that were required to be sent from the local web server to the local name server. Initially the servers sent status messages to each other using User Datagram Protocol (UDP) to avoid the TCP overhead. However, there was a significant amount of dropped messages and these messages have a large impact on the accuracy of the system. In particular, if a status message which indicates an item has been removed from a cache is lost, any requests for that item would result in a false positive. After switching to TCP connections between servers, no more messages were dropped, due to TCP's built in error handling. However, this sensitivity to dropped messages can be dealt with in several ways such as periodically synchronizing the cache contents or through a feedback system from the client.

4.6.2 Setting File Thresholds

Section 4.3.3 described a threshold mechanism for reducing the load on remote servers which do not contain enough items in cache. This threshold value can be set in several different ways. The main idea of this threshold value is that the disk access time required by the local server for the files which are in a remote server's cache should not exceed the time required to fetch the items from the remote server.

One way to set this value is to use measured network characteristics (RTT from client to neighbor server). However, network delays are frequently unpredictable which can cause the threshold values to be set imprecisely. In addition the information required is the RTT between the client and the remote server which may be hard to acquire. It may not be worth the effort to produce an accurate threshold value as small variations in threshold value do not have a large effect on cache hit rates.

4.6.3 Bloom Filters

Summary caches [48] use well known technique called Bloom filters [13] as a method of decreasing the overhead of their status messages and cache storage. Bloom filters work by using hashing techniques to indicate if an object is available at a location in a very compact way. The drawback of using bloom filters is that there can be false positives. However, this can be remedied by using more storage bits to represent caches. Using Bloom filters can be a great way to reduce both storage and transmissions overheads in Cacheflow. Also, from the resulting inaccuracies seen in the link buffer experiments (Section 4.5.5), the detrimental effect of the inaccuracies for using bloom filters are not

expected to be very high. This compact format for cache statuses can also lead to efficient cache synchronizations as discussed in Section 4.6.1.

4.6.4 Item Placement and Load Balancing

Cacheflow does not currently handle load balancing and this can lead to hot-spots in the system. This problem can be alleviated by better item placement in caches. A feedback system can be put into place so that popular items are split among the servers or even have multiple copies of a popular item.

Other methods of item placement such as pre-fetching items into cache, migrating items to areas where they are more popular and better cache admittance controls may lead to better performance. However, any such system is dependent on how accurately they can model and predict the workload and characteristics of the system.

4.7 Summary

In this chapter, the need for providing timely response for client requests on social networking and websites with personalization is explored. By taking advantage of multiple collaborating servers, Cacheflow is able to provide better response time for clients by lowering network latencies and reducing the number of request round trips.

Though a series of experiments, the efficiency of Cacheflow is demonstrated. The system is able to achieve much better cache hit rates using the same amount of memory overall and provide equivalent performance with the less memory. The hit rates for the case with two neighbors are the same as having between 2 and 3 times as much memory in

the non-Cacheflow case. Different application parameters for reducing system overhead is also shown to not have significant negative impacts on the cache hit rate. Misprediction rates only increased by 5% when 900 messages were buffered together.

Chapter 5

Conclusions

In this dissertation, I support the following thesis through the design an implementation of several systems: *In multiprocess systems, specialized schedulers that take the system characteristics into account can significantly improve the performance of the system over generalized schedulers.*

My contributions span across several different domains from low level single sensor system to high level distributed Internet based systems:

First, **Reordering Grouped Earliest Deadline First (RG-EDF)** is a scheduling policy developed to provide efficient quality of service for flash-equipped sensor devices. This system aims at improving storage quality by taking advantage of flash memory characteristics. Requests originating from multiple processes are combined together and reordered in a way that provides much better performance than existing systems. RG-EDF is implemented on a CC1010 sensor node with a SD flash card attached. Experiment results show that RG-EDF and G-EDF perform up to twice as well as FIFO and EDF

schedulers. Peak throughput is 225% better and request end-to-end times are twice as good.

Second, **MiscoRT** is a real-time application and task scheduler for the Misco system. Misco is a MapReduce framework developed for smart-phones. MiscoRT uses a two-level approach to schedule tasks so that applications meet their deadlines. The scheduler incorporates an expected failure model to predict execution times in inherently unstable settings. Misco and MiscoRT are implemented and tested on a testbed of Nokia NSeries smart-phones.

Extensive experiment results demonstrate that Misco is efficient, has low overhead and out performs its competitors. The system has a very modest memory overhead, only 800KB of memory out of the 90MB available. For performance, MiscoRT is shown to complete applications up to 32% faster. Further, MiscoRT scales linearly to the number of workers available in the system.

Finally, **Cacheflow** is a system for reducing client response times and improving server memory utilization in content delivery systems for social networking and personalized services type sites. The system intelligently retrieves items from multiple servers simultaneously, reducing the total number of round trips required. Through experiments performed on PlanetLab, Cacheflow is shown to provide better client latencies using the same amount of memory resources and provides the same client latencies with less memory resources.

From experiment results, Cacheflow is shown to greatly improve cache hit rates. With two neighbors per server, the resulting hit rate is between a system with double to

triple the amount of memory. It is also shown that this resulting hit rate increase leads to improvements in client response time as long the average miss handler time is greater than 1 ms, much lower than traditional disk access times of around 6 ms. The results also show that overhead reducing techniques such as buffering out puts and applying file thresholds do not have significant adverse effects on the quality of service provided.

Each one of my works exploit the underlying characteristics to improve upon the performance of the system. As hardware and infrastructure continue to improve in computation power, speed, ubiquity and sensing capabilities, applications are expected to become more useful. As a result, they are becoming more complex and requires more raw input data. Specialized scheduling techniques are critical in allowing the systems to meet the ever increasing demands of these applications.

Bibliography

[1] Charu Aggarwal, Joel L. Wolf, and Philip S. Yu. Caching on the world wide web. *IEEE Trans. on Knowl. and Data Eng.*, 11:94–107, January 1999.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38:393–422, 2002.

[3] Mohammad Al-Fares, Khaled Elmeleegy, Benjamin Reed, and Igor Gashinsky. Overclocking the yahoo! cdn for faster web page loads. In *Internet Measurement Conference 2011*, 2011.

[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] Ella M. Atkins, Tarek F. Abdelzaher, Kang G. Shin, and Edmund H. Durfee. Planning and resource allocation for hard real-time, fault-tolerant plan execution. *Autonomous Agents and Multi-Agent Systems*, 4(1-2), 2001.

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

[7] Hakan Aydin. On fault-sensitive feasibility analysis of real-time task sets. In *International Real-Time Systems Symposium*, 2004.

[8] Bhuvan Bamba, Ling Liu, Arun Iyengar, and Philip S. Yu. Distributed processing of spatial alarms: A safe region-based approach. In *ICDCS, Washington DC, US*, 2009.

[9] A. Banerjee, A. Mitra, W. Najjar, D. Zeinalipour-yazti, V. Kalogeraki, and D. Gunopulos. Rise co-s : High performance sensor storage and co-processing architecture. In *IEEE Sensor and Ad Hoc Communications and Networks*, Santa Clara, CA, 2005.

[10] Berkeley. mobile millennium. http://traffic.berkeley.edu/.

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

[12] Timothy Bisson, Scott A. Brandt, and Darrell D. E. Long. Nvcache: Increasing the effectiveness of disk spin-down algorithms with caching. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 422–432, Monterey, CA, September 2006.

[13] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[14] Lee Breslau, Pei Cue, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *In INFOCOM*, pages 126–134, 1999.

[15] David N. Breslauer, Robi N. Maamari, Neil A. Switz, Wilbur A. Lam, and Daniel A. Fletcher. Mobile phone based clinical microscopy for global health applications. *PLoS ONE*, 4, 07 2009.

[16] E. Carrera and R. Bianchini. Improving disk throughput in data-intensive servers. In *International Symposium on High-Performance Computer Architecture*, Madrid, Spain, February 2004.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *USENIX*, pages 205–218, 2006.

[18] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.*, 3:837–863, November 2004.

[19] Jian-Jia Chen, Chuan-Yue Yang, Tei-Wei Kuo, and Shau-Yin Tseng. Real-time task replication for fault tolerance in identical multiprocessor systems. In *RTAS, Bellevue, US*, 2007.

[20] Shimin Chen and Steven W. Schlosser. Map-reduce meets wider varieties of applications. Technical report, Intel, 2008.

[21] Chipcon. Chipcon cc1010, http://www.keil.com/dd/chip/3506.htm.

[22] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, volume 2009, pages 46–66, 2001.

[23] CMU. Cmucam, http://cmucam.org/.

[24] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[25] Crossbow. mica2, http://www.xbow.com/products/product_pdf_files/ wireless_pdf/mica2_datasheet.pdf.

[26] Crossbow. stargate, http://www.xbow.com/products/product_pdf_files/ wireless_pdf/6020 − 0049 − 01_b_stargate.pdf.

[27] Crossbow. Crossbow technology. http://www.xbow.com/, May 2011.

[28] Doug Cutting. Hadoop core. http://hadoop.apache.org/core/, May 2011.

[29] Hui Dai, Michael Neufeld, and Richard Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *ACM Conference on Embedded Networked Sensor Systems*, pages 176–187, Baltimore, MD, USA, 2004.

[30] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. Mapreduce: simplified data processing on large clusters. In *Symposium on Opearting Systems Design & Implementation*. USENIX Association, 2004.

[31] Peter Desnoyers, Deepak Ganesan, and Prashant Shenoy. Tsar: a two tier sensor storage architecture using interval skip graphs. In *SenSys*, San Diego, California, USA, 2005.

[32] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. Diskseen: Exploiting disk layout and access history to enhance i/o prefetch. In *USENIX*, 2007.

[33] Adam Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen, and Ville H. Tuulos. Misco: a mapreduce framework for mobile systems. In *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '10, pages 32:1–32:8, New York, NY, USA, 2010. ACM.

[34] Adam J. Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen, and Ville Tuulos. Scheduling for real-time mobile mapreduce systems. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, pages 347–358, New York, NY, USA, 2011. ACM.

[35] Adam Ji Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikäinen, Ville H. Tuulos, Sean Foley, and Curtis Yu. Data clustering on a network of mobile smartphones. In *SAINT*, pages 118–127. IEEE Computer Society, 2011.

[36] Adam Ji Dou, Song Lin, and Vana Kalogeraki. Real-time querying of historical data in flash-equipped sensor devices. In *IEEE Real-Time Systems Symposium*, pages 335–344. IEEE Computer Society, 2008.

[37] Charles Ebeling. An introduction to reliability and maintainability engineering, 1997.

[38] B. Eich. Tracemonkey: Javascript lightspeed. http://weblogs.mozillazine.org/ roadmap/archives/2008/08/tracemonkey_javascript_lightsp.html, 2008.

[39] Shane B. Eisenman, Emiliano Miluzzo, Nicholas D. Lane, Ronald A. Peterson, Gahng-Seop Ahn, and Andrew T. Campbell. Bikenet: A mobile sensing system for cyclist experience mapping. *ACM Trans. Sen. Netw.*, 6:6:1–6:39, January 2010.

[40] Peter R. Elespuru, Sagun Shakya, and Shivakant Mishra. Mapreduce system over heterogeneous mobile devices. In *International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2009.

[41] Jarno Elonen. Nanohttpd. http://elonen.iki.fi/code/nanohttpd/, August 2011.

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

[43] V8 JavaScript Engine. Google. http://code.google.com/apis/v8/intro.html.

[44] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. The pothole patrol: using a mobile sensor network for road surface monitoring. In Dirk Grunwald, Richard Han, Eyal de Lara, and Carla Schlatter Ellis, editors, *MobiSys*, pages 29–39. ACM, 2008.

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

[46] Facebook. Facebook. http://www.facebook.com.

[47] Facebook. Facebook statistics. http://www.facebook.com/press/info.php?statistics, 2011.

[48] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *IEEE/ACM Transactions on Networking*, pages 254–265, 1998.

[49] Thomas Huining Feng and Edward A. Lee. Real-time distributed discrete-event execution with fault tolerance. In *RTAS, St. Louis, US*, 2008.

[50] Brad Fitzpatrick. Distributed cacheing with memcached. http://www.linuxjournal.com/article/7451, 2004.

[51] foursquare. foursquare. http://foursquare.com.

[52] foursquare. So we grew 3400% last year... http://blog.foursquare.com/2011/01/24/2010infographic/.

[53] J. Freyne, A. Brennan, B. Smyth, D. Byrne, A. Smeaton, and G. Jones. Automated murmurns: The social mobile tourist application. In *Int. Conference on COmputational Science and Engineering, vol. 4, pp. 1021-1026*, 2009.

[54] Gartner. Gartner says worldwide mobile device sales to end users reached 1.6 billion units in 2010; smartphone sales grew 72 percent in 2010. http://www.gartner.com/it/page.jsp?id=1543014, February 2011.

[55] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system, 2003.

[56] Sunondo Ghosh, Rami Melhem, and Daniel Mosse. Enhancing real-time schedules to tolerate transient faults. In *RTSS, Pisa, Italy*, 1995.

[57] Google. Google+. http://plus.google.com, 2011.

[58] Sathish Gopalakrishnan and Marco Caccamo. Task partitioning with replication upon heterogeneous multiprocessor systems. *RTAS, San Jose, US*, 2006.

[59] Ilya Grigorik. Collaborative map-reduce in the browser. http://www.igvita.com/2009/03/03/collaborative-map-reduce-in-the-browser/, March 2009.

[60] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *PACT, Toronto, Canada*, 2008.

[61] 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*, 2003.

[62] Todd Hoff. Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/latency-everywhere-and- it-costs-you-sales-how-crush-it, 2009.

[63] R. Honicky. N-smarts: Advanced sensing. http://www.cs.berkeley.edu/~honicky/nsmarts/sensing.shtml, 2010.

[64] HTC. Droid incredible technical specifications. http://www.htc.com/us/products/droid-incredible-verizon/#tech-specs, 2010.

[65] Prashant Shenoy Huan Li and Krithi Ramamritham. Scheduling messages with deadlines in multi-hop real-time sensor networks. In *RTAS, CA, US*, 2005.

[66] Cheng Huang, Angela Wang, Jin Li, and Keith W. Ross. Measuring and evaluating large-scale cdns. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, IMC '08, pages 15–29, New York, NY, USA, 2008. ACM.

[67] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. Cartel: a distributed mobile sensor computing system. In *In 4th ACM SenSys*, pages 125–138, 2006.

[68] Intel. Intel imote, http://www.xbow.com/products/product_pdf_files/wireless_pdf/mica2_datasheet.pdf.

[69] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *ACM Symposium on Operating Systems Principles*, Banff, Alberta, Canada, 2001.

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

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

[72] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX Technical Conference on UNIX and Advanced Computer Systems*, 1995.

[73] Minkyong Kim and David Kotz. Extracting a mobility model from real user traces. In *In Proceedings of IEEE INFOCOM*, 2006.

[74] P. Krishnan, Krishnan Danny Raz, and Yuval Shavitt. The cache location problem. *IEEE/ACM Transactions on Networking*, 8:568–582, 2000.

[75] Rupa Krishnan, Harsha V. Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize cdn performance. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pages 190–201, New York, NY, USA, 2009. ACM.

[76] Purushottam Kulkarni, Deepak Ganesan, and Prashant Shenoy. The case for multi–tier camera sensor networks. In *NOSSDAV*, pages 141–146, Stevenson, Washington, USA, 2005.

[77] Albert Leon-Garcia. *Probability and Random Processes for Electrical Engineering (2nd Edition)*. McGraw Hill, 1993.

[78] Chuanpeng Li and Kai Shen. Competitive prefetching for concurrent sequential i/o. In *EuroSys*, pages 189–202, 2007.

[79] Ming Li, Deepak Ganesan, and Prashant Shenoy. Presto: feedback-driven data management in sensor networks. In *ACM/USENIX Symposium on Networked Systems Design and Implementation*, pages 23–23, San Jose, CA, 2006.

[80] Hong Lu, Wei Pan, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. Soundsense: scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*, MobiSys '09, pages 165–178, New York, NY, USA, 2009. ACM.

[81] Christopher Lumb, Jiri Schindler, Gregory R. Ganger, Erik Riedel, and David F. Nagle. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 87–102, San Diego, CA, 2000.

[82] Liqian Luo, Qing Cao, Chengdu Huang, Tarek Abdelzaher, John A. Stankovic, and Michael Ward. Enviromic: Towards cooperative storage and retrieval in audio sensor networks. In *ICDCS '07*, page 34, Washington, DC, USA, 2007. IEEE Computer Society.

[83] Dimitrios Lymberopoulos and Andreas Savvides. Xyz: a motion-enabled, power aware sensor node platform for distributed sensor network applications. In *IPSN '05*, page 63, Los Angeles, CA, 2005.

[84] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[85] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD*, 2003.

[86] Eugene Marinelli. Hyrax: Cloud computing on mobile devices using mapreduce. http://www.contrib.andrew.cmu.edu/ emarinel/masters_thesis/, 2009.

[87] 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.

[88] Emiliano Miluzzo, Nicholas D. Lane, Kristóf Fodor, Ronald Peterson, Hong Lu, Mirco Musolesi, Shane B. Eisenman, Xiao Zheng, and Andrew T. Campbell. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 337–350, New York, NY, USA, 2008. ACM.

[89] Emiliano Miluzzo, Nicholas D. Lane, Kristof Fodor, Ronald A. Peterson, Hong Lu, Mirco Musolesi, Shane B. Eisenman, Xiao Zheng, and Andrew T. Campbell. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *SenSys*, 2008.

[90] E. Mumolo. Prediction of disk arm movements in anticipation of future requests. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, College Park, Maryland, October 1999.

[91] Min Mun, Sasank Reddy, Katie Shilton, Nathan Yau, Jeff Burke, Deborah Estrin, Mark H. Hansen, Eric Howard, Ruth West, and Péter Boda. Peir, the personal environmental impact report, as a platform for participatory sensing systems research. In Krzysztof Zielinski, Adam Wolisz, Jason Flinn, and Anthony LaMarca, editors, *MobiSys*, pages 55–68. ACM, 2009.

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

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

[94] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44:2–19, August 2010.

[95] Mohammad Rahimi, Rick Baer, Obimdinachi I. Iroezi, Juan C. Garcia, Jay Warrior, Deborah Estrin, and Mani Srivastava. Cyclops: in situ image sensing and interpretation in wireless sensor networks. In *SenSys '05*, pages 192–204, San Diego, California, USA, 2005.

[96] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *International Symposium on High-Performance Computer Architecture*, 2007.

[97] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6:2002, 2002.

[98] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, 2001.

[99] Moray Rumney. Imt-advanced: 4g wireless takes shape in an olympic year, 2008.

[100] Paul Saab. Scaling memcached at facebook. http://www.facebook.com/note.php?note_id=39391378919, 2008.

[101] Mastooreh Salajegheh, Yue Wang, Kevin Fu, Anxiao (andrew Jiang, and Erik Learned-miller. Exploiting half-wits: Smarter storage for low-power devices. In *In Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST 11*, 2011.

[102] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, David, and C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459, 1990.

[103] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, 0:1–10, 2010.

[104] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Transactions on Networking*, pages 149–160, 2001.

[105] Arvind Thiagarajan, Lenin Ravindranath, Katrina LaCurts, Samuel Madden, Hari Balakrishnan, Sivan Toledo, and Jakob Eriksson. Vtrack: accurate, energy-aware road traffic delay estimation using mobile phones. In *SenSys, Berkeley, US*, 2009.

[106] Ville Tuulos. Disco. http://discoproject.org/, May 2011.

[107] Twitter. Twitter. http://twitter.com, 2011.

[108] Amin Vahdat. Presentation summary, high performance at massive scale: Lessons learned at facebook. http://idleprocess.wordpress.com/2009/11/24/ presentation-summary-high-performance-at -massive-scale-lessons-learned-at-facebook/, 2009.

[109] Elizabeth Varki, Arif Merchant, Jianzhang Xu, and Xiaozhou Qiu. Issues and challenges in the performance analysis of real disk arrays. *IEEE Trans. Parallel Distrib. Syst.*, 15:559–574, June 2004.

[110] Fuxing Wang, Krithi Ramamritham, and John A. Stankovic. Determining redundancy levels for fault tolerant real-time systems. *IEEE Trans. Comput.*, 44(2), 1995.

[111] Duane Wessels and Kim Claffy. Internet cache protocal (icp), version 2. http://ds.internic.net/rfc/rfc2186.txt, 1998.

[112] Kang won Lee, Sambit Sahu, Khalil Amiri, and Chitra Venkatramani. Understanding the potential benefits of cooperation among proxies: Taxonomy and analysis, 2001.

[113] David Woodhouse. Jffs: The journalling flash file system. In *Ottawa Linux Symposium*, http://sourceware.org/jffs2/, 2001.

[114] Jianliang Xu, Jiangchuan Liu, Bo Li, and Xiaohua Jia. Caching and prefetching for web content distribution. *IEEE Comput. Sci. Eng.(CiSE), Special Issue on Web Engineering*, 6:54–59, 2004.

[115] YAFFS. Yet another flash file system. http://www.yaffs.net, May 2011.

[116] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.

[117] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. Microhash: An efficient index structure for flash-based sensor devices. In *FAST 2005*, San Fransisco, CA, Dec 2005.

[118] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.