TOWARDS ENERGY-EFFICIENT, FAULT-TOLERANT, AND LOAD-BALANCED MOBILE CLOUD

A Dissertation

by

CHIEN-AN CHEN

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

DOCTOR OF PHILOSOPHY

Chair of Committee,	Radu Stoleru
Committee Members,	Anxiao Jiang
	Jyh-Charn Liu
	Narasimha Reddy
Head of Department,	Dilma Da Silva

December 2015

Major Subject: Computer Engineering

ABSTRACT

Recent advances in mobile technologies have enabled a new computing paradigm in which large amounts of data are generated and accessed from mobile devices. However, running resource-intensive applications (e.g., video/image storage and processing or map-reduce type) on a single mobile device still remains off bounds since it requires large computation and storage capabilities. Computer scientists overcome this issue by exploiting the abundant computation and storage resources from traditional cloud to enhance the capabilities of end-user mobile devices. Nevertheless, the designs that rely on remote cloud services sometimes underlook the available resources (e.g., storage, communication, and processing) on mobile devices. In particular, when the remote cloud services are unavailable (due to service provider or network issues) these smart devices become unusable. For mobile devices deployed in an infrastructureless network where nodes can move, join, or leave the network dynamically, the challenges on energy-efficiency, reliability, and load-balance are still largely unexplored.

This research investigates challenges and proposes solutions for deploying mobile application in such environments. In particular, we focus on a distributed data storage and data processing framework for mobile cloud. The proposed mobile cloud computing (MCC) framework provides data storage and data processing services to MCC applications such as video storage and processing or map-reduce type. These services ensure the mobile cloud is energy-efficient, fault-tolerant, and load-balanced by intelligently allocating and managing the stored data and processing tasks accounting for the limited resources on mobile devices. When considering the load-balance, the framework also incorporates the heterogeneous characteristics of mobile

cloud in which nodes may have various energy, communication, and processing capabilities. All the designs are built on the *k-out-of-n computing* theoretical foundation. The novel formulations produce a reliability-compliant, energy-efficient data storage solution and a deadline-compliant, energy-efficient job scheduler. From the promising outcomes of this research, a future where mobile cloud offers real-time computation capabilities in complex environments such as disaster relief or warzone is certainly not far.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

In this section, we describe the problems that motivate this research. We then introduce the research challenges and how these challenges are addressed, followed by a list of main contributions.

1.1 Motivation and Challenges

In the traditional cloud computing, clients (personal computer, laptop, smartphone, etc.) offload computation or data to remote service providers such as Google or Amazon to perform computation-intensive tasks that are infeasible on the local devices. One important characteristic of the traditional cloud computing is that it relies on an infrastructural network between the remote server and the clients, so the performance of the Internet has great impacts on the service quality. Clients suffer from random disruption or long delay that occur in the middle of the route or at the remote servers. In remote areas where the Internet is unavailable, the cloud applications relying on remote servers are simply impossible to operate.

Recent advances in the design and the deployment of mobile cloud systems tap into the increasingly abundant sensing, storage, processing, and communication capabilities available on smart devices. In particular, a class of mobile cloud systems consisting entirely of intermittently connected mobile devices has been conceived and prototyped by researchers [49] [46] [91] [75]. Such infrastructureless and autonomous mobile cloud systems are not only interesting from a theoretical point of view, as they pose the most *challenging design settings*, but also important in enabling real-world applications. The applications, including, but not limited to, 1) Tactical cloud systems for military operation; there is currently an effort by the U.S. military to equip soldiers with commodity smartphones or tablets [31, 73]). 2) Dis-

aster relief situations where the network infrastructure is damaged by water, fire, or wind [33, 70]. 3) Crowded events like football game or New Year's celebration where the cellular network is overloaded and congested [11, 90]. 4) Situations when cellular communication or cloud service is expensive; if data are only to be shared locally, it is unnecessary to relay data to remote cloud, which incurs multiple layers of extra cost. For computation-intensive tasks such as video encoding/decoding and big data processing that are infeasible on a single mobile device [31, 82], autonomous mobile cloud provides a platform where a collection of mobile devices cooperate to complete these complicated tasks.

There are many challenges in bringing the computation-intensive tasks to mobile environments. Mobile devices are resource-constrained in terms of communication bandwidth, processing power, reliability, and energy. While conserving the resources on mobile devices is imperative, system performance should not be overlooked. This research aims to realize a distributed data storage and data processing system in an infrastructureless mobile cloud in which the services provided by the mobile devices must be energy-efficient, fault-tolerant, and load-balanced. Since most mobile devices are battery-powered, *energy consumption* for data distribution, data retrieval, and data processing must be minimized. *Reliability* is a major challenge in mobile cloud; unstable wireless links can make a node unreachable or cause network partitions, and device failures may occur due to energy depletion or physical damage. As mobile cloud is highly heterogeneous and mobile devices have various processing power, communication technologies, energy capacities, and operating systems, efficiently integrating and utilizing these resources is challenging. Specifically, the services need to consider *load-balance*, ensuring that no node is overloaded with tasks beyond its capability or becomes a hot-spot that bottlenecks the system performance. Finally, *security* is also a concern as the stored data often contain sensitive information and

they should not be compromised if the devices are captured or stolen [45] [104]. To address the aforementioned issues, the k-out-of-n mobile cloud computing framework is proposed.

The framework was evaluated through both simulation and real-world implementations. We participated in the disaster response exercises and military technology conference host by Texas A&M Engineering Extension Service (TEEX) [89] and and Naval Post Graduate School respectively to demonstrate our development. The framework is implemented as a *Service* in Android operating system and it provides mobile applications data storage or data processing services. We implemented an application *Media Share* for the disaster response exercises. During the exercise, responders used this application to store and share media files (pictures or videos). The stored media files can later be processed upon request to identify and extract frames that contain human figures. Instead of manually searching through hundreds of images and videos, *Media Share* completes the job in minutes using our MCC framework. The feedback from real responders and military personals validates the usefulness of this application and guides the development of the k-out-of-n computing framework.

1.2 Dissertation Statement

Mobile devices are resources-constrained in terms of energy, processing, bandwidth, and reliability. To extend the usability of mobile devices, most existing solutions rely on the Internet to access resources from remote cloud. However, in situations when remote cloud is unavailable (e.g., due to congested network, cloud outage, or natural disaster), these mobile devices become useless. To address these issues, this research proposes a mobile cloud computing (MCC) framework aimed at an infrastructureless network where all participating nodes are mobile devices.

The mobile cloud framework provides distributed data storage and data processing services to mobile applications and ensures the services are energy-efficient, fault-tolerant, and load-balanced.

This study investigates the potentials and challenges of big data services in mobile environments. The proposed framework minimizes the energy consumption and enables computation-intensive data processing jobs in a mobile cloud consisting of only mobile devices. Exploiting this MCC framework, applications requiring data storage or data processing services can be migrated to and deployed in a mobile environment quickly.

1.3 Main Contributions

The contributions of this research are outlined as follows:

A distributed data storage framework (the k-out-of-n storage) in an infrastructureless mobile cloud.

- It presents a mathematical model for both optimizing energy efficiency and fulfilling the fault-tolerant requirements of data storage.
- It presents an efficient algorithm for estimating the communication cost in a mobile ad-hoc network where nodes can leave, join, or move freely.
- It presents a simple and distributed protocol to monitor the mobile ad-hoc network topology.
- It formulates a data caching problem that is designed considering the file access pattern and user mobility to improve the energy efficiency and data availability.
- It presents a centralized solution and a distributed solution for solving the data caching problem; the centralized solution obtains an optimal solution and the distributed solution obtains an approximated solution in shorter time.

- It presents a parameter selection algorithm that chooses/adjusts parameter (k, n) dynamically to maintain the system optimality (energy-efficiency, reliability, and load-balance).

- It presents a mechanism for interpreting parameter (k, n) such that the data reliability and the data redundancy can be considered separately. The mechanism allows better flexibility and energy-efficiency when maintaining the storage system.

A distributed data processing framework (the k-out-of-n data processing) over the k-out-of-n storage in an infrastructureless mobile cloud.

- It formulates an optimization problem for task allocation and task scheduling. The solution ensures that the data processing job is energy-efficient and fault-tolerant.

- It presents a heuristic solver based on the Tabu Search technique to efficiently obtain an approximated a solution.

- It implements the Hadoop generic file system interface such that Hadoop-MapReduce can process the data stored in our k-out-of-n storage. With this interface, any existing MapReduce application can be ported to mobile environments quickly.

- It implements a new scheduling algorithm for Hadoop jobTracker to minimize the data transferring energy when taskTrackers retrieve data blocks from the k-out-of-n storage.

A distributed data storage and data processing framework in a heterogeneous mobile cloud.

- It presents a lightweight distributed framework for data storage & data processing in a heterogeneous mobile cloud. The framework is scalable to larger networks and the degree of heterogeneity.

- It presents a distributed and low-overhead load-balancing algorithm that heuristically reduces the system-wide *load imbalance*.

- It presents an energy-efficient agent-based search algorithm that explores and discovers resources such as data fragments and free processors in a mobile cloud.

- It implements a media sharing and processing Android application based on the heterogeneous MCC framework.

1.4 Organization

This dissertation is organized as follows. In Section 2, the state of the art and prior works related to this dissertation are reviewed. In Section 3, the high-level architecture and the main operations of this MCC framework are introduced. Section 4 describes the k-out-of-n distributed data storage framework, Section 5 describes the k-out-of-n distributed data processing framework, and Section 6 presents the integrated k-out-of-n data storage and data processing framework in a heterogeneous mobile cloud. Finally, Section 7 concludes this dissertation and illustrates its future perspective.

2. LITERATURE REVIEW

In this section, we review the backgrounds and the related works of this research. Three major topics, *distributed data storage*, *distributed data processing*, and *mobile cloud* are discussed.

2.1 Distributed Data Storage

A distributed file system (DFS) is a file system constructed of a collection of network-connected nodes. The purpose of DFS is to support the same kind of file sharing when users are physically dispersed in a distributed system [66]. This topic has long been studied and most challenges were considered [34, 41, 62, 76, 88, 94, 101]. In early 1990, the research and development focused on file system deployed in data center where nodes are servers connected with high bandwidth LAN, which is also the most common architecture adopted in industry today. A different type of distributed file system, Peer-to-Peer (P2P) file system, which targets on millions of PCs connected with the Internet, was introduced in 2000. P2P is a class of applications that take advantage of resources – storage, cycles, content, human presence – available at the edge of the Internet [93]. Conventional client-server systems are asymmetric in which servers are usually more powerful than the clients, while in P2P system, each peer may be both client or server and hardware specifications can be highly nonuniform. Also, most conventional distributed file system requires a centralized node that coordinates all nodes, stores the meta data, and maintains the file directory. Many P2P file systems, however, are completely decentralized and do not need a centralized node. Some famous P2P file systems are [9, 39, 40, 77, 83, 85, 95].

In the past decade, due to the need of retaining "Big Files" in long-term storage devices, big companies Google and Yahoo developed distributed file systems that

can handle terabyte of files efficiently and relievablely. Google designed and implemented Google File System (GFS) [34] in early 2002. GFS was designed to be fault-tolerant on inexpensive commodity hardware serving a large number of clients. On top of GFS, Google developed BigTable in 2005 aimed to store and manage structured data that may be petabytes large across hundreds or thousands of machines. Although not a relational database, BigTable resembles a high-performance and scalable database that stores data in a multi-dimensional sorted map [12]. Yahoo developed a distributed, scalable, and fault-tolerant file system called Hadoop Distributed File System (HDFS) in 2005. Similar to GFS, HDFS aims to store terabyte of data across hundreds or thousands of commodity hardware [94]. HDFS was originally designed to store search engine data, but was quickly improved to support analytic for various production applications. Many Hadoop-related projects are still being actively developed on top of HDFS. E.g., HBase, Hive, ZooKeeper, etc.

Different from GFS or HDFS which are designed for traditional computers deployed in a static network with high bandwidth, our mobile distributed file system (MDFS) runs on resource-constrained mobile devices in which the battery capacity is limited, the memory is small, and the communication bandwidth is low. Furthermore, the mobile network is dynamic, meaning the topology changes constantly when the mobile nodes move. P2P file system is similar to our MDFS in a sense that MDFS is also decentralized and it can be deployed in a network with nonuniform nodes. However, P2P file system operates over the Internet while MDFS primarily operates in an mobile ad-hoc network. Also, nodes in P2P file systems generally have constant energy source and are not mobile. All these differences pose new research challenges to MDFS.

Erasure coding has always been a popular fault-tolerant technique in distributed data storage system. Researchers proposed solutions for achieving higher reliability

in dynamic networks. Comparing to another common method for improving the data reliability, data replication, erasure coding achieves higher mean time to failure and lower bandwidth requirement (network traffic) [100]. Although erasure coding technique outperforms replication technique in many aspects, due the simplicity of replication technique and low cost storage medias, replication technique are still widely adopted (e.g., GFS and HDFS). Dimakis et al. proposed several erasure coding algorithms for maintaining a distributed storage system in a dynamic network [23–25]. They encode a file into segments and allocate segments to storage nodes to maximize the probability of successfully recover a file. A different allocation scheme can also minimize the expected data recovery delay. Leong et al. proposed an algorithm for optimal data allocation that maximizes the recovery probability [65, 65]. Aguilera et al. proposed a protocol to efficiently adopt erasure code for better reliability with low data redundancy [1]. Their solution achieves efficient concurrent data update. Ali et al. proposed Raptor codes based distributed storage algorithms for collecting data in a wireless sensor network in which k nodes sense data and distribute the data to n nodes ($k \leq n$) [3]. Cooley et al. proposed Lincoln Erasure Code (LEC) that is applicable to large-scale distributed storage across thousands of nodes. LEC was shown to provide higher performance in terms of encoding/decoding throughput and network scalability [3]. Rashmi et al. proposed Minimum Bandwidth Regenerating code and Minimum Storage Regenerating code based on a common product-matrix framework [84]. MDFS adopts the similar Maximum Distance Separable (MDS) code as these prior works, but our objective function, minimizing the system-wide communication energy, was never considered before. Furthermore, MDFS considers a dynamic network while most prior works consider a static network with stable network connectivity.

MDFS conservers data transferring energy by placing data fragments in selected

storage nodes such that the data can be retrieved with the lowest communication energy. Data allocation problem in distributed storage system had been well studied in the traditional cloud. Several works also optimized latency and communication costs. Alicherry and Lakshman proposed a 2-approximation algorithm for selecting optimal data centers in a distributed cloud with the objective to minimize the maximum distance, or latency, between the selected data centers. The same algorithm is also used to select rack in each data center [2]. Beloglazov et al. It presents an architectural framework and principles for energy-efficient cloud computing for data centers. It solved the problem by applying their Modified Best Fit Decreasing algorithm. Their scheduling algorithm considers quality of service and energy consumption characteristics of the devices [8]. Liu et al. proposed an Energy-Efficient Scheduling (DEES) algorithm for data grid that supports real-time and data-intensive application. DEES integrates the process of scheduling tasks and data placement considering data locations and application properties. It saves energy by reducing data replications and task transfer [68]. [92] proposed cloudlet seeding, a strategic placement of high performance computing assets in wireless ad-hoc network such that computational load is balanced. *CAROM* (Cache A Replica On Modification) is an ensemble of replication and erasure codes to provide efficient and reliable distributed file system. CAROM ensures a low bandwidth, low redundancy, and low latency file system [69]. Most of these solutions, however, are designed for high performance servers in a static network. Our solution focuses on resource-constrained mobile devices in a dynamic network.

Distributed data storage on mobile devices in an infrastructureless network have also been looked at. For example, *STACEE* [78] creates a peer to peer storage system from connected laptops and mobile devices, with an explicit goal to minimize the total energy consumption while maximizing user satisfaction. *WhereStore* [96] is a

location-based data storage for smart devices interacting with the cloud. It uses each device's location history to determine what data to replicate locally. *Phoenix* [81] is a distributed communication and storage protocol aiming to make efficient use of storage space and communication bandwidth while maximizing the longevity of stored data. *PP2db* is a privacy-preserving and scalable distributed file system targeted at mobile network. It supports the anonymous but trusted exchange of Quality of Experience (QoE) information [20]. Anderson et al. investigates a mobile P2P architecture that can reconcile the decentralized operation of P2P file sharing with the interests of network operators such as traffic control [4]. Huchton et al. [46] proposed a k-Resilient Mobile Distributed File System for mobile devices targeted primarily for military operations. Using the Reed Solomon code and Shamir's key sharing algorithm, the file system stores file fragments securely on the smartphones. Neither the data reliability or data maintenance are explicitly considered in these works. Our MDFS considers the data reliability and energy-efficiency in an integrated manner, which was never attempted before. MDFS also studies the storage maintenance problem and propose an algorithm to continuously monitor and maintain the system optimality.

2.2 Distributed Data Processing

The volume and the velocity of data generation has never stopped growing. Researchers and computer scientists have developed different techniques to store, process, and maintain these data. In order to adapt to such fast-growing technology, one of the objectives is to leverage low-cost commodity hardware as computation platform instead of using expensive supercomputers. The computation capability of a cluster built upon commodity hardware can be scaled up or upgraded much easier than a supercomputer. With this low cost computation platform, companies

can afford to build their in-house data center consisting of hundreds or thousands of processors. Three most popular distributed data processing frameworks at the time this dissertation is written are *Hadoop* [29], *Storm* [97], and *Spark* [106], all licensed under Apache Software Foundation [5]. *Hadoop* is designed for batch data processing in which each job processes a fixed size file stored in the Hadoop Distributed File System; *Storm* is designed for real-time stream data processing in which it continuously processes an unbounded stream of data generated from sources such as social network feed or video stream; *Spark* is a more general data processing framework that can handle both batch processing and steam data processing. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches [30].

As commodity hardware are not designed for high reliability or long Mean Time To Failure, Hadoop, Storm, and Spark are all built with fault-tolerant mechanisms to recover from node failures. When a cluster consists of thousands of commodity hardware such as hard drives, memory, power supply, and processors, component failures become the norm rather than the exception [34]. Here we briefly introduce the fault-tolerance mechanism used by different frameworks. In Hadoop, a jobTracker on the NameNode continuously monitors each deployed job that is concurrently executed by many taskTrackers on different processor nodes. If any of the taskTracker fails to respond the heartbeat message for a period of time, the jobTracker considers this processor fails and reschedules the unfinished tasks to other taskTrackers. In case a jobTracker node (NameNode) dies, all the running jobs are lost (Single point of failure). Recent improvement on Hadoop has allowed a Checkpoint node to periodically backups the NameNode's meta data information. In Strom, each worker is monitored by both the node's Supervisor and the Nimbus; Supervisor can restart a worker if it dies, and Nimbus can reassign a worker to other nodes in case the node

fails. Nimbus is similar to the JobTracker in Hadoop and is responsible for assigning tasks to nodes and monitoring for failures. Each node runs a Supervisor daemon that listens for tasks assigned to this node and monitors the local worker processes. In case Nimbus or Supervisor dies, they can be safely restarted without affecting the ongoing workers, meaning that the topology can continue executing without Nimbus or Supervisors' presence. Nimbus or Supervisor can be restarted using the states kept in ZooKeeper. However, if any worker also dies during this period, there is no way to restart or reassign the lost task. Sparks's fault-tolerance mechanism relies on Resilient Distributed Dataset (RDD), which is an immutable, deterministically re-computable, and distributed dataset. In case a worker node fails, a lost RDD can be reconstructed using the lineage information. A transformation applied to a RDD produces another RDD, and the lineage information records how a RDD was derived. However, in case the driver node (master) running the Spark Streaming application fails, all workers and their in-memory data are lost. The major differences between our k-out-of-n data processing framework and these existing works are that they are designed to operate in static and stable network environment in which data are transferred though high-bandwidth LAN and nodes are powered by constant energy source. Most importantly, they all face the single point of failure problem to some degree.

We now look at various techniques used to achieve fault-tolerance in distributed computing. In general, fault-tolerance includes "fault-detection" step and "fault-recovery" step. In this dissertation, we assume a *fail-fast* model meaning that once an error occurs on a node, it fails immediately and does not attempt to repair itself. The detection of fail-fast fault is simple and straightforward. There are three most common fault-tolerant techniques in modern distributed computing systems, *checkpointing, replication, and rescheduling*. Checkpointing is the process of peri-

odically storing the states of a running process on a reliable storage. The stored data, naming *checkpoint*, allows the system to recover to an earlier state in case of a failure [10, 38, 50, 53, 86]. One of the challenge in ensuring the correctness of a checkpoint is *Synchronization*. If a checkpoint occurs when there are still in-transit messages, the saved states may be inconsistent. [27] describes three checkpointing strategies, *coordinated checkpointing, uncoordinated checkpoint, and communication-induced checkpointing*. In coordinated checkpointing, processes synchronize with each other and save states at the same time to ensure the consistency. In uncoordinated checkpointing, each process can schedule to save state independently. This strategy is more efficient, but has the risk of inconsistency. Communication-induced checkpointing reduces the overhead in process synchronization by only coordinating several critical checkpoints.

Replication technique achieves fault-tolerance by duplicating a task to multiple instances and concurrently running these instances on multiple processors. It ensures that the task can complete successfully as long as one of the processor finishes its assigned tasks [63, 67, 98, 99, 107]. How to place the replicas and how to ensure the consistency between replicas have great impacts on the overall reliability and system performance. Task replication incurs communication delay and network traffic. A large portion of this dissertation focuses on optimizing data allocation and task allocation to improve the reliability, energy-efficiency, and load-balance performance. Rescheduling technique reassigns a failed task to another available processor. It does not incur extra storage or computation resources like checkpointing or replication, but it takes longer for recovering a time-consuming task. As a result, this strategy is not suitable for jobs that consist of long-running processes. If processes are synchronized, then the entire job may pause in order to wait for the failed process to restart [47, 48, 51, 103]. Depending on the available resources and the types of applications, one

fault-tolerant technique may be better than another. Hadoop and Storm belong to rescheduling, and Spark belongs to checkpointing. The fault-tolerance technique in our k-out-of-n data processing framework is a combination of checkpointing and rescheduling; when a node fails, the unfinished tasks on this node are reassigned to other processors nodes. This schedule is predetermined before the processing starts to ensure the energy-efficiency and load-balance. During the job execution, each processor node performs checkpointing by announcing its execution state (the tasks that it has completed) so that other nodes know what to restart if this node fails.

Distributed data processing in mobile cloud has also been studied. In 2009, Marinelli introduced a Hadoop based platform Hyrax [71] for distributed data processing on smartphones. In particular, the Hadoop taskTracker and DataNode processes were ported to Android phones. The jobTrakcker and NameNode still run on a regular computer. Hyrax runs in an infrastructural network where all nodes connect to a single access point router. Another MapReduce framework based on Python, Misco [54] was implemented on Nokia mobile phones. It has a similar server-client model where the server keeps track of various user jobs and assigns them to workers on demand. Yet another server-client model based MapReduce system was proposed over a cluster of mobile devices [26] where the mobile client implements MapReduce logic to retrieve work and produce results from the master node. Similar to Hyrax, the server of Misco also runs on a regular computer. P2P-MapReduce [72] describes a prototype implementation of a MapReduce framework which uses a peer-to-peer model for parallel data processing in dynamic cloud topologies. It exploits p2p model to manage node churn, node failure, and job recovery in a distributed manner. At each time, a small subset of nodes are assigned as master nodes and others are assigned as master nodes. The role may change dynamically depending on the network condition and to ensure the desired master/slave ratio.

Several prior works had attempted to deploy mobile clouds in wireless ad-hoc networks. Huerta-Canepa and Lee proposed a virtual cloud computing framework [49] targeting at an ad-hoc network consisting of mobile phones. The framework detects nearby nodes that have the same movement pattern, and creates a virtual resource provider on the fly among these nearby nodes. *MobiCloud* [42] treats mobile devices in an ad-hoc network as service nodes to provide traditional cloud computing services. To build a trustworthy MANET communication, MobiCloud addresses trust management, secure routing, and risk management in the network. It supports information dissemination, routing, and localization functions in MANET. Continuing the *MobiCloud* work, Huang et al. [43] proposes data processing framework in mobile cloud through trust management and private data isolation. In this work, the mobile cloud has tree virtual domains, the cloud mobile and sensing domain, the cloud trusted domain, and the cloud public service and storage domain. Finally, *Scavenger* [61] is a cyber-foraging system that eases the development of distributed processing applications in a mobile cloud setting. It intelligently schedules and allocates tasks considering data locality, device capability, and task complexity. The previous research focused only on the parallel processing of tasks on mobile devices using the MapReduce framework without addressing the real challenges that occur when these devices are deployed in the mobile environment. Different from most of the existing works, our distributed data processing algorithm aims to achieve energy-efficiency, load-balance, and fault-tolerance in mobile cloud.

2.3 Mobile Cloud

Cloud computing is an aggregation of computation resources such as storage, computation, and software in which clients request services over the internet. Based on the model of the service, it can generally be categorized into three types, Infras-

tructure as a service (IaaS), Platform as a service (PaaS), and Software as a service (SaaS). Mobile cloud Computing (MCC) can be defined as an integration of cloud computing technology with mobile devices to make the mobile devices resource-full in terms of computational power, memory, storage, energy, and context awareness. Mobile cloud computing is the outcome of interdisciplinary approaches comprising mobile computing and cloud computing [28, 56]. A mobile cloud may or may not consist of the traditional cloud services; in an infrastructural mobile cloud, mobile devices usually access service from or offload computation to remote servers over the Internet; in an infrastructural-less or ad-hoc mobile cloud, a group of mobile devices and computing nodes form a self-sustained cloud in a local network. The mobile cloud studied in this work belongs to the ad-hoc mobile cloud. Mobile cloud computing is similar to conventional cloud computing because they both enable resource sharing between physically separated service providers and clients. In mobile cloud computing, however, mobile devices face more limitations such as computation, storage, connectivity, energy, and mobility, which incurs more problems that do not exist in traditional cloud.

The architecture of MCC application can be classified into 3 types, *remote cloud server*, *virtual resource cloud*, and *cloudlet* [28]. Most of the commercial mobile applications use the *remote cloud server* architecture in which mobile applications completely offload computations and data storage to remote servers hosted in data centers. E.g., Google Map, Gmail, Facebook App, SoundHound, Yelp, etc. This type of MCC is similar to traditional cloud computing because mobile devices simply act as a thin client connecting to remote servers over the Internet. The major disadvantage of this architecture is that the mobile applications become unusable when there is no Internet or the remote server is down. In *virtual resource cloud*, a collection of mobile devices form a peer-to-peer network and each mobile device

provides services to or request services from other mobile nodes. This type of MCC does not rely on the Internet or remote servers, and it can function in infrastructural or infrastructureless local network. However, because all services (e.g., storage and computation) are provided by resources-constrained mobile devices, the complexity of the tasks that virtual resource cloud can perform still lags behind remote cloud server. E.g., Hyrax [71], Scavenger [61], MobiCloud [42], and Serendipity [91]. *Cloudlet* MCC combines the architecture from both remote cloud and virtual resource cloud. Mobile devices offload workload to the nearby computing nodes (cloudlet) such as routers or PC, and these stationary computing nodes have stable connection to the Internet or remote cloud. Cloudlet itself can also provide services to mobile devices, and computation or data intensive tasks are offloaded to remote servers. In such architecture, MCC can function in a local network without the Internet, and it can reach the more powerful remote cloud if the Internet is available. The mobile devices in cloudlet again act as a thin client connecting to the cloudlet. E.g. Cloudlet [87], PocketCloudlet [60], and Cloudlet Seeding [92].

In MCC, mobile nodes offload data or computation to cloud. Depending on the MCC architecture and the types of applications, different offloading strategies have been proposed. The most common offloading strategies in current research are *Client-Server Communication, Virtualization, and Mobile Code* [28]. For applications using *Client-server communication* strategy, mobile clients use remote procedure call (RPC) or remote method invocation (RMI) to offload tasks to servers in the cloud. This offloading method is similar to the service provision in traditional cloud in which a remote cloud server provides a unique API to service all mobile clients from different platforms. Most commercialized mobile applications adopt this simple method. It is also used in many academic works such as Hyrax [71], Huerta-Canepa's work [49], Cuckoo [55], and Carmen [57]. *Virtualization* method abstracts virtual

resources such as hardware, operating system, or application in a VM-layer. Mobile clients offload tasks through memory page migration, which provides a seamless and non-interrupted vision of task execution. Identical codes can be executed on both servers and clients over the VM-layer, and developers have the finest control of what to offload and what not to offload [15,80]. Virtualization solution is the most comprehensive solution, but its realization is usually complicated, error-prone, and causes significant overhead. Some example projects are CloneCloud [14], ThinkAir [58], and Clone2Clone [59]. The third offloading strategy *Mobile Code* allows a client to transfer a piece of code to server and server can run the code directly. Some existing Internet browser technologies such as JavaScript and ActiveX have used this method for decade. Example projects in MCC are MAUI [21], Balan's work [6], Zhang's work [108], and Scavenger [61].

A MCC framework is usually designed under a specific objective according to its applications. Due to the limited resources on mobile devices, performance objectives such as energy, security, latency, or system lifetime, are often being optimized. In some cases, a framework may try to achieve multiple objectives simultaneously. Here we briefly describe several prior works with different design objectives. *CloneCloud* [14] uses a combination of static analysis and dynamic profiling to determine whether offloading a thread can reduce the execution time or energy use for a target computation and communication environment. Scavenger [61] proposes a mobile code approach to ease the development of cyber foraging (MCC) applications. Developers can easily create highly mobile, distributed, and parallel mobile applications on Scavenger system. *MobiCloud* [42] proposes a MCC computation framework in mobile ad-hoc network that provides a fundamental trust model including identity management, key management, and security data access policy. *MAUI* [21] supports fine-grained code offload (mobile code) to maximize energy savings. It de-

cides at runtime whether to execute a method remotely or locally under the real-time networking and CPU conditions. *Carmen* [57] manages the connectivity of a set of mobile nodes and enables mobile users to coordinate system resources (computation) across a mobile cloud. *ThinkAir* [58] exploits the smartphone virtualization technique to provide method-level computation offloading. *ThinkAir* provides on-demand resource allocation to reduce execution time and energy consumption on mobile devices.

3. SYSTEM ARCHITECTURE

This section describes the architecture of our Mobile Cloud Computing (MCC) framework in high level. As shown in Fig. 3.1, the k-out-of-n MCC framework is built across network layer, transport layer, and application layer. The majority of the functions are implemented above the transport layer, but the framework accesses routing information and link quality information from the network layer. The expected outcome of this framework is an *energy-efficient*, *fault-tolerant*, and *load-balanced* distributed data storage & processing system for a dynamic network. Figure 3.2 shows how the framework interacts with applications. The framework runs on all nodes in the mobile cloud and provides services to applications that aim to: (1) store data in mobile cloud reliably such that the energy consumption for transferring the data is minimized (the k-out-of-n data storage); (2) reliably process the stored data such that energy consumption for processing the data is minimized (the k-out-of-n data processing); and (3) support heterogeneous network that consists of nodes of different hardware capabilities. As an example, an application running in a mobile ad-hoc network generates many media files and these files must be stored reliably resilient to device failures, i.e. the files are recoverable even if one or several devices in the network fail. At later time, the application may query for information like the number of an object appearing in a set of video files. Without loss of generality, we assume a stored data object immutable, meaning that it is stored once, and will be retrieved or accessed for processing multiple times later.

As shown in Figure 3.2, applications generate *data* and our framework stores data in a set of mobile nodes. For higher data reliability and availability, each data is encoded and partitioned into *fragments*; the fragments are distributed to a set of

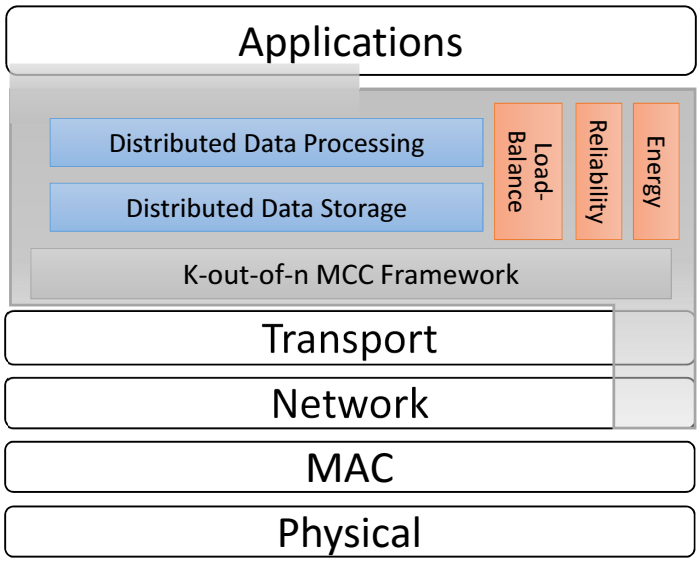


Figure 3.1: A cross-layer mobile cloud computing framework for distributed data storage and data processing.

storage nodes and any subset of k different fragments can recover the original file. There are two options for processing the stored data, either using our k-out-of-n Data Processing component or using the Hadoop MapReduce component. For k-out-of-n data processing component, applications provide processing *functions* and each function is instantiated to multiple *tasks* that process a set of data concurrently on multiple nodes. We call a set of tasks instantiated from one function a *job* and nodes executing the tasks are *processor nodes*. For Hadoop MapReduce component, our framework provides the same interface as Hadoop Distributed File System (HDFS) such that any existing MapReduce application can directly access and process files stored in the k-out-of-n data storage. *Client nodes* are the nodes requesting data allocation or processing operations. A node can have any combination of roles from storage node, processor node, or client node, and any node can retrieve data fragments from any storage node.

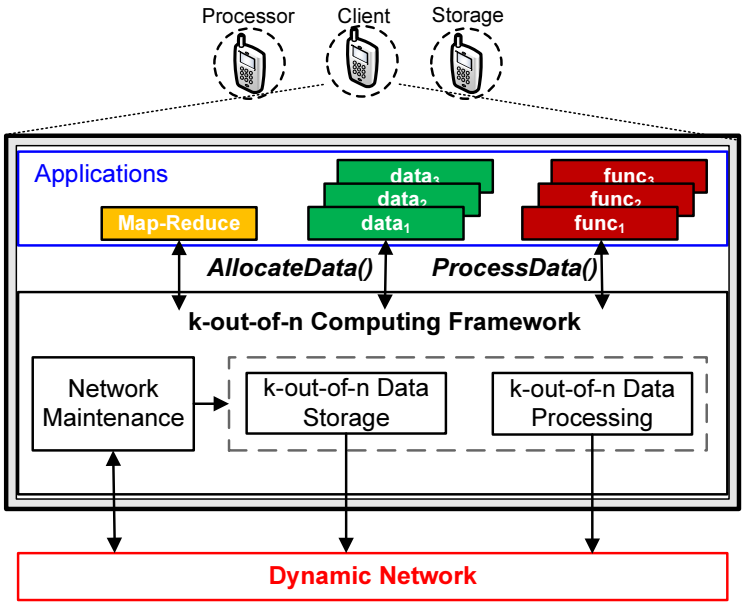


Figure 3.2: System architecture of the mobile cloud computing framework. The framework runs on all nodes and it provides data storage and data processing services to applications, e.g., fault-tolerant storage, video processing, Hadoop Map-Reduce applications.

3.1 Operations in the k-out-of-n Computing Framework

Topology Change, *Data Creation*, *Data Retrieval*, and *Data Processing* are the four primary operations in the k-out-of-n computing framework. An operation is triggered by requests from the applications and each operation consists of a set of components. Figure 3.3 shows the main operations and their corresponding components. *Network Maintenance Component* continuously monitors the network topology, including the reliability (or failure probability) of each node, and the neighbors of each node. Upon a significant change of network condition, network maintenance component notifies the framework and the framework reevaluates the system parameters and optimal data allocation. Except for the network maintenance component, all other components are triggered only when specific events occur. In the rest of this section,

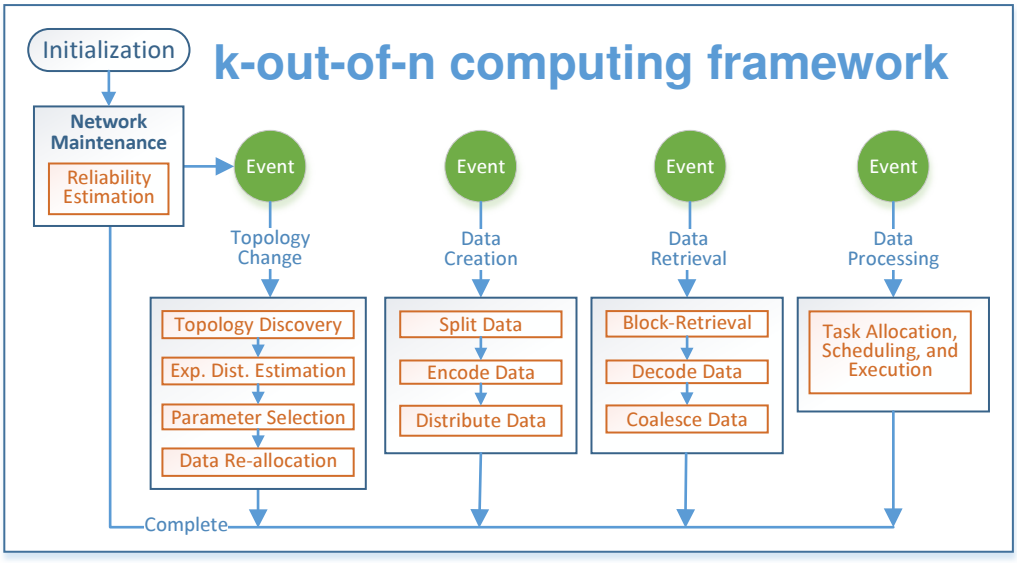


Figure 3.3: The operations supported by the k-out-of-n computing framework. The maintenance component continuously runs at the background while all other components are triggered only when specific events occur. Each event triggers a sequence of operations that are completed by a set of nodes.

we briefly introduce these four data operations. More detail will be covered in the later section.

3.1.1 Network Maintenance and Topology Change Event

When the network maintenance component detects a significant change of network condition due to mobility, node failures, or depleted energy, it initiates a sequence of procedures to reexamine the network. *Topology Discovery Component* floods a discovery packet to the network and each node replies with its neighbor table and estimated reliability. Similar to traditional distributed computing settings, *reliability* is defined as the probability that a stored data object can be recovered successfully or a processing job completes within a predefined time period. *Reliability Estimation* component on each node estimates its own reliability based on the *residual energy*, *mobility*, and *application-specific factors* assigned by the administrator.

A node is considered failed if other nodes in the network cannot communicate with it, which may be caused by hardware/software failure, depleted battery, or physical damage. Once a node fails, its stored data or assigned tasks are no longer accessible. Based on the reliability of nodes and network topology, *Expected Transmission Estimation* component computes the expected distance matrix D, in which each element D_{ij} indicates the expected energy consumption for node i to send a unit packet to node j. *Parameter Selection* component then searches for parameter (k, n) that can achieve data reliability requirement and load-balance requirement. Finally, *Data Reallocation* component tries to restore the system optimality by reallocating data fragments. It computes the most energy-efficient strategy for reallocating data fragments if possible. In situations when too many fragments and storage nodes fail, *Data Redistribution* is necessary, which invokes a *Data Retrieval* followed by a *Data Creation*. The purpose is to recover the original data, regenerate the data fragments, and redistribute the data fragments.

3.1.2 Data Creation Event

Data Creation event is triggered when a client node requests to store a file. A file is first split into blocks and each block is encrypted and encoded into multiple data fragments. Splitting a file into blocks allows a file to be recovered partially and thus allows multiple blocks of a file to be recovered/processed concurrently on different processor nodes. We use Shamir's key sharing algorithm and Reed-Solomon error correction code to encrypt and encode each data block. The top of Figure 3.4 illustrates how a file is split into blocks and encoded into fragments. Once all the key and data fragments are ready, they are sent to and stored on a set of selected storage nodes such that the stored data are reliable and can be accessed energy-efficiently.

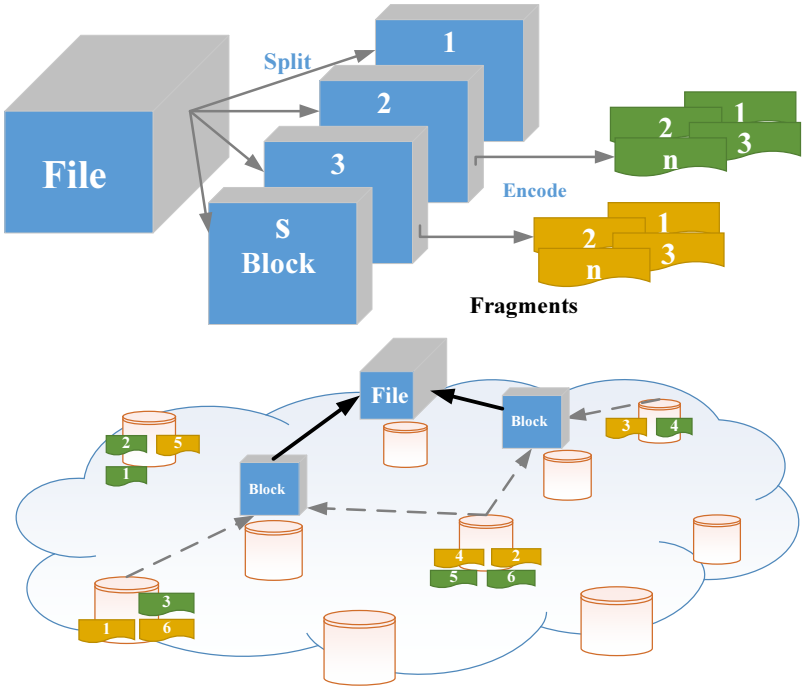


Figure 3.4: Data creation event (top) and data retrieval/processing event (bottom). A big file is split into blocks, encoded into data fragments, and distributed to the network. During the data retrieval or data processing events, each block can be retrieved and recovered independently. As a result, blocks of a file can be processed concurrently on multiple processors.

3.1.3 Data Retrieval Event

Data Retrieval event is triggered when a client requests for either a file read operation or a data processing operation. Given the data blocks of a file, *Block Retrieval* component locates and retrieves data and key fragments from the storage nodes. Once all the required fragments are successfully retrieved, *Data Decoding* component decodes and decrypts the fragments, and recovers the data blocks. The original file can be reconstructed by coalescing all the recovered data blocks.

3.1.4 Data Processing Event

Data processing event is triggered when a client requests to process one or multiple files stored in the distributed storage. A processing job can have one or multiple tasks and each task is assigned to one of the selected processor nodes. Specifically, each task corresponds to one data block that is to be retrieved and processed on a processor node. The *k-out-of-n data processing component* achieves fault-tolerance by replicating each task to multiple task instances such that the job can complete successfully as long as k or more of the processor nodes finish their assigned tasks. To reduce the energy consumption for retrieving and processing the tasks, tasks are allocated and scheduled to processor nodes in a way to minimize the expected data transmission & data processing energy. In addition, when an optional deadline constraint is provided, the framework ensures that the expected job makeSpan meets the deadline.

4. DISTRIBUTED DATA STORAGE*

In this section, we present the k-out-of-n data storage framework that supports fault-tolerant and energy-efficient remote storage under a dynamic network topology, i.e., mobile cloud. Note that part of this section is reprinted from the previously published papers.* Without loss of generality, in this section, we assume that the mobile cloud consists of only homogeneous mobile nodes where all nodes have identical hardware capability and form a mobile ad-hoc network. The more complicated heterogeneous network will be studied in the later section. Each node can move, join, or leave the network freely. The mobility of the nodes is primarily due to human movements, e.g., that of soldiers or disaster responders. The network provides mobile cloud services for applications that need energy-efficient and reliable distributed storage. We assume all nodes can be used for storing data, and any node can be a *client node* that requests storage services from the mobile cloud.

We integrate the k-out-of-n reliability mechanism into the mobile cloud. k-out-of-n is a well-studied topic in reliability control [16] that ensures a system of n components operates correctly as long as k or more components work. More specifically, we investigate how to store data in a mobile cloud with the k-out-of-n reliability such that: 1) the system-wide communication energy for distributing and retrieving the

*Reprinted with permission from "Resource allocation for energy efficient k-out-of-n system in mobile ad hoc networks" by Chien-An Chen, Myounggyu Won, Radu Stoleru, and Geoffrey Xie, *International Conference on Computer Communications and Networks, 2013*, Copyright © 2013, IEEE.
Reprinted with permission from "Energy-Efficient, Fault-Tolerant Data Storage & Processing in Dynamic Networks" by Chien-An Chen, Myounggyu Won, Radu Stoleru, and Geoffrey Xie, *International Symposium on Mobile Ad Hoc Networking and Computing, 2013*, Copyright © 2013, Association for Computing Machinery, Inc.
Reprinted with permission from "Energy-Efficient, Fault-Tolerant Data Storage & Processing in Mobile Cloud" by Chien-An Chen, Myounggyu Won, Radu Stoleru, and Geoffrey Xie, *IEEE Transactions on Cloud Computing, 2015*, Copyright © 2015, IEEE.

stored data is minimized; and 2) the stored data meets the data reliability requirement during a predefined mission completion time. In this proposed framework, a data object is encoded and partitioned into n fragments, and then stored on n different storage nodes. As long as k or more of the n nodes are available, the data object can be successfully recovered. The parameters k and n determine the degree of reliability. Smaller k/n ratio achieves higher reliability with the cost of higher data redundancy. System administrator may select these parameters depending on the data reliability requirements and storage capacity. The framework also provides an adaptive *Parameter Selection* algorithm that adjusts the parameters based on the reliability requirement and the network condition.

In the rest of the section, we first formulate the k-out-of-n data storage problem. We explain in detail how each component in the k-out-of-n data storage framework works. A variation of the data storage problem that uses splitting (k, n) parameters is then present. To further improve the performance, we propose a caching algorithm designed specifically for the k-out-of-n data storage. Finally, we show the evaluation results.

4.1 Formulation of the k-out-of-n Data Storage Problem

We consider a dynamic network with N nodes denoted by a set $V = \{v_1, v_2, ..., v_N\}$. We assume nodes are time synchronized. For security purpose, only a set of n selected storage nodes can store data and other nodes cannot cache any retrieved data, i.e., a client deletes the retrieved file immediately after reading it. Before the application-predefined mission completion time T, each client accesses each created file once. For convenience, we will use i and v_i interchangeably hereafter. The network is modeled as a graph $G = (V, E)$, where E is a set of edges indicating the communication links among nodes. Each node has an associated *failure probability*

$P[f_i]$ where f_i is the event that causes node v_i to fail.

Relationship Matrix R is a $N \times N$ matrix defining the relationship between nodes and storage nodes. More precisely, each element R_{ij} is a binary variable – if R_{ij} is 0, node i will not retrieve data from storage node j; if R_{ij} is 1, node i will retrieve fragment from storage node j. *Storage node list* X is a binary vector containing storage nodes, i.e., $X_i = 1$ indicates that v_i is a storage node.

The *Expected Transmission Time Matrix* D is defined as a $N \times N$ matrix where element D_{ij} corresponds to the ETT for transmitting a fixed size packet from node i to node j considering the failure probabilities of nodes in the network, i.e., multiple possible paths between node i and node j. The ETT metric [18] has been widely used for estimating transmission time between two nodes in one hop. We assign each edge of graph G a positive estimated transmission time. Then, the path with the shortest transmission time between any two nodes can be found. However, the shortest path for any pair of nodes may change over time because of the dynamic topology. ETT, considering multiple paths due to nodes failures, represents the "expected" transmission time, or "expected" transmission energy between two nodes.

$$R_{opt} = \arg\min_{R} \sum_{i=1}^{N} \sum_{j=1}^{N} D_{ij} R_{ij} \tag{4.1}$$

$$\text{Subject to:} \quad \sum_{j=1}^{N} X_j = n \tag{4.2}$$

$$\sum_{j=1}^{N} R_{ij} = k \ \forall i \tag{4.3}$$

$$X_j - R_{ij} \geq 0 \ \forall i \tag{4.4}$$

$$X_j \text{ and } R_{ij} \in \{0, 1\} \ \forall i, j \tag{4.5}$$

Equations 4.1 - 4.5 formulate the data allocation problem as an ILP. In this problem, we are interested in finding n storage nodes denoted by $S = \{s_1, s_2, ...s_n\}, S \subseteq V$ such that the *total expected transmission cost* from any node to its k closest storage nodes – in terms of ETT – is minimized (Eq. 4.1). The first constraint (Eq. 4.2) selects exactly n nodes as storage nodes; the second constraint (Eq. 4.3) indicates that each node has access to k storage nodes; the third constraint (Equation 4.4) ensures that j^{th} column of R can have a non-zero element if only if X_j is 1; and constraints (Equation 4.5) are binary requirements for the decision variables.

4.2 Energy-Efficient and Fault-Tolerant Data Allocation

This section describes the main components in the k-out-of-n data allocation framework.

4.2.1 Topology Discovery

Topology Discovery is executed during the network initialization phase or whenever a significant change of the network topology is detected (as detected by the Network Monitoring component). During Topology Discovery, one delegated node floods a *request* packet throughout the network. Upon receiving the request packet, nodes reply with their neighbor tables and failure probabilities. Consequently, the delegated node obtains global connectivity information and failure probabilities of all nodes. This topology information can later be queried by any node.

4.2.2 Failure Probability Estimation

We assume a fault model in which faults caused only by node failures and a node is inaccessible and cannot provide any service once it fails. The failure probability of a node estimated at time t is the probability that the node fails by the mission completion time T. We define the *effective time* $T_e = T - t$ as

the time interval during which the estimated failure probability is effective. A node estimates its failure probability based on the following events/causes: energy depletion, temporary disconnection from a network (e.g., due to mobility), and application-specific factors. We assume that these events happen independently. Let f_i be the event that node i fails and let f_i^B, f_i^C, and f_i^A be the events that node i fails due to energy depletion, temporary disconnection from a network, and application-specific factors respectively. The failure probability of a node is as follows: $P[f_i] = 1 - \left(1 - P\left[f_i^B\right]\right)\left(1 - P\left[f_i^C\right]\right)\left(1 - P\left[f_i^A\right]\right)$. We now present how to estimate $P[f_i^B], P[f_i^C]$, and $P[f_i^A]$.

4.2.2.1 Failure by Energy Depletion

Estimating the remaining energy of a battery-powered device is a well-researched problem [102]. We adopt the remaining energy estimation algorithm in [102] because of its simplicity and low overhead. The algorithm uses the history of periodic battery voltage readings to predict the battery remaining time. Considering that the error for estimating the battery remaining time follows a normal distribution [64], we find the probability that the battery remaining time is less than T_e by calculating the cumulative distributed function (CDF) at T_e. Consequently, the predicted battery remaining time x is a random variable following a normal distribution with mean μ and standard deviation σ. μ is simply the estimated battery remaining time at current time t, and σ is obtained from historical data.

$$
\begin{aligned}
P\left[f_i^B\right] &= P\left[\text{Rem. time} < T_e \mid \text{Current Energy}\right] \\
&= \int_{-\infty}^{T_e} f(x;\mu;\sigma^2)dx, \quad f(x;\mu;\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}
\end{aligned}
$$

4.2.2.2 Failure by Temporary Disconnection

Nodes can be temporarily disconnected from a network, e.g., because of the mobility of nodes, or simply when users turn off the devices. The probability of temporary disconnection differs from application to application, but this information can be inferred from the history: a node gradually learns its behavior of disconnection and cumulatively creates a probability distribution of its disconnection. Then, given the current time t, we can estimate the probability that a node is disconnected from the network by the time T as follows: $P\left[f_i^C\right] = P\left[\text{Node } i \text{ disconnected between } t \text{ and } T\right]$.

4.2.2.3 Failure by Application-dependent Factors

Some applications require nodes to have different roles. In a military application for example, some nodes are equipped with better defense capabilities and some nodes may be placed in high-risk areas, rendering different failure probabilities among nodes. Thus, we define the failure probability $P[f_i^A]$ for application-dependent factors. This type of failure is, however, usually explicitly known prior to the deployment.

4.2.3 Expected Transmission Time Computation

It is known that a path with minimal hop-count does not necessarily have minimal end-to-end delay because a path with lower hop-count may have noisy links, resulting in higher end-to-end delay. Longer delay implies higher transmission energy. As a result, when distributing data or processing the distributed data, we consider the most energy-efficient paths – paths with *minimal transmission time*. When we say path p is the shortest path from node i to node j, we imply that path p has the lowest transmission time (equivalently, lowest energy consumption) for transmitting a packet from node i to node j. The shortest distance then implies the lowest

transmission time.

Given the failure probability of all nodes, we calculate the ETT matrix D. However, if failure probabilities of all nodes are taken into account, the number of possible graphs is extremely large, e.g., a total of 2^N possible graphs, as each node can be either in failure or non-failure state. Thus, it is infeasible to deterministically calculate ETT matrix when the network size is large. To address this issue, we adopt the *Importance Sampling technique*, one of the Monte Carlo methods, to approximate ETT. The Importance Sampling allows us to approximate the value of a function by evaluating multiple samples drawn from a sample space with known probability distribution. In our scenario, the probability distribution is found from the failure probabilities calculated previously and samples used for simulation are snapshots of the network graph with each node either fails or survives. The function to be approximated is the ETT matrix, D.

A sample graph is obtained by considering each node as an independent Bernoulli trial, where the success probability for node i is defined as: $p_{X_i}(x) = (1 - P[f_i])^x P[f_i]^{1-x}$, where $x \in \{0, 1\}$. Then, a set of sample graphs can be defined as a multivariate Bernoulli random variable B with a probability mass function $p_g(b) = P[X_1 = x_1, X_2 = x_2, ..., X_n = x_n] = \prod_{i=1}^{N} p_{X_i}(x)$. $x_1, x_2, ..., x_n$ are the binary outcomes of Bernoulli experiment on each node. b is an $1 \times N$ vector representing one sample graph and $b[i]$ in binary indicating whether node i survives or fails in sample b.

Having defined our sample, we determine the number of required Bernoulli samples by checking the variance of the ETT matrix denoted by $Var(E[D(B)])$, where the ETT matrix $E[D(B)]$ is defined as follows: $E[D(B)] = \left(\sum_{j=1}^{K} b_j p_g(b_j)\right)$ where K is the number of samples and j is the index of each sample graph.

In Monte Carlo Simulation, the true $E[D(B)]$ is usually unknown, so we use

the ETT matrix estimator, $\tilde{D}(B)$, to calculate the variance estimator, denoted by $\widehat{Var}\left(\tilde{D}(B)\right)$. The expected value estimator and variance estimator below are written in a recursive form and can be computed efficiently at each iteration:

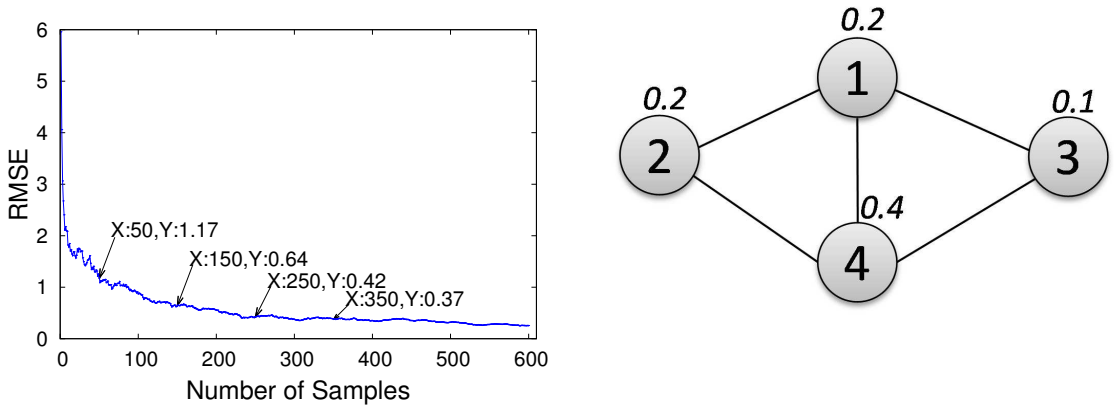
$$\tilde{D}(B_K) = \frac{1}{K}\left((K-1)\,\tilde{D}(B_{K-1}) + b_K\right)$$
$$\widehat{Var}\left(\tilde{D}(B_K)\right) = \frac{1}{K(K-1)}\sum_{i=1}^{K}\left(b_j - \tilde{D}(B_K)\right)^2$$
$$= \frac{1}{K}\left(\frac{1}{K-1}\sum_{j=1}^{K}(b_i)^2 - \frac{K}{K-1}\left(\tilde{D}(B_K)\right)^2\right)$$

Here, the Monte Carlo estimator $\tilde{D}(B)$ is an unbiased estimator of $E[D(B)]$, and K is the number of samples used in the Monte Carlo Simulation. The simulation continues until $\widehat{Var}\left(\tilde{D}(B)\right)$ is less than $dist\_var_{th}$, a user defined threshold depending on how accurate the approximation has to be. We chose $dist\_var_{th}$ to be 10% of the smallest node-to-node distance in $\tilde{D}(B)$.

Figure 4.1 compares the ETT found by Importance Sampling with the true ETT found by a brute force method in a network of 16 nodes. The *Root Mean Square Error* (RMSE) is computed between the true ETT matrix and the approximated ETT matrix at each iteration. It is shown that the error quickly drops below 4.5% after the 200^{th} iteration.

4.2.4 The k-out-of-n Data Allocation

After the ETT matrix is computed, the k-out-of-n data allocation is solved by ILP solver. A simple example of how the ILP problem is formulated and solved is shown here. Considering Figure 4.1(b), distance Matrix D is a 4×4 symmetric matrix with each component D_{ij} indicating the expected distance between node i



(a) (b)

Figure 4.1: (a) Root Mean Square Error (RMSE) of each iteration of Monte Carlo Simulation. (b) A simple graph of 4 nodes. The number above each node indicates the failure probability of the node.

and node j. Let's assume the expected transmissions time on all edges are equal to 1. As an example, D_{23} is calculated by finding the probability of two possible paths: $2 \rightarrow 1 \rightarrow 3$ or $2 \rightarrow 4 \rightarrow 3$. The probability of $2 \rightarrow 1 \rightarrow 3$ is $0.8 \times 0.8 \times 0.9 \times 0.4 = 0.23$ and the probability of $2 \rightarrow 4 \rightarrow 3$ is $0.8 \times 0.6 \times 0.9 \times 0.2 = 0.08$. Another possible case is when all nodes survive and either path may be taken. This probability is $0.8 \times 0.8 \times 0.6 \times 0.9 = 0.34$. The probability that no path exists between node 2 and node 3 is (1-0.23-0.08-0.34=0.35). We assign the longest possible ETT=3, to the case when two nodes are disconnected. D_{23} is then calculated as $0.23 \times 2 + 0.08 \times 2 + 0.34 \times 2 + 0.35 \times 3 = 2.33$. Once the ILP problem is solved, the binary variables X and R give the allocation of data fragments. In our solution, X shows that nodes $1 - 3$ are selected as storage nodes; each row of R indicates where the client nodes should retrieve the data fragments from. E.g., the first row of R shows that node 1 should retrieve data fragments from node 1 and node 3.

$$D = \begin{pmatrix} 0.6 & 1.72 & 1.56 & 2.04 \\ 1.72 & 0.6 & 2.33 & 2.04 \\ 1.56 & 2.33 & 0.3 & 1.92 \\ 2.04 & 2.04 & 1.92 & 1.2 \end{pmatrix}$$

$$R = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} \quad X = (1\ 1\ 1\ 0)$$

4.2.5 Distributed Network Monitoring

The Network Monitoring component monitors the network topology as well as each nodes' reliability continuously on each node. Whenever a client node needs to create a file, the Network Monitoring component provides the client with the most recent topology information immediately. When there is a significant topology change, it notifies the framework to update the current solution. We first give several notations. A term s refers to a state of a node, which can be either U and NU. The state becomes U when a node finds that its neighbor table has drastically changed or its reliability has dropped significantly; otherwise, a node keeps the state as NU. We let p be the percentage of entries in the neighbor table that has changed, r_{est} be the estimated reliability when the last data allocation problem was solved, and r_{cur} be the current estimated reliability. A set $\mathcal{ID}$ contains the node IDs with p greater than τ_1 or $r_{est} - r_{cur}$ greater than τ_r. τ_1 and τ_r are thresholds indicating a significant topology change or reliability change.

Algorithm 1: Distributed Network Monitoring

At each beacon interval:
if *($p > \tau_1$ or $r_{est} - r_{cur} > \tau_r$) and $s \neq U$* **then**
  $s \leftarrow U$
  Put +ID to a beacon message
end
if $p \leq \tau_1$ *and* $r_{est} - r_{cur} \leq \tau_r$ *and* $s = U$ **then**
  $s \leftarrow NU$
  Put −ID to a beacon message
end

Upon receiving a beacon message on V_i
for *each ID in the received beacon message* **do**
  if $ID > 0$ **then**
    $\mathcal{ID} \leftarrow \mathcal{ID} \bigcup \{\text{ID}\}$
  else
    $\mathcal{ID} \leftarrow \mathcal{ID} \setminus \{\text{ID}\}$
  end
end
if $|\{\mathcal{ID}\}| > \tau_2$ **then**
  Notify V_{del} and V_{del} initiate topology discovery
end
Add the ID in $V_i's$ beacon message

The Network Monitoring component is simple yet energy-efficient as it does not incur significant communication overhead – it simply piggybacks node ID on a beacon message. The protocol is depicted in Algorithm 1. We predefine one node as a *topology_delegate* V_{del} who is responsible for maintaining the global network information. When a node's neighbour table or reliability changes significantly, the node changes its state to U and piggybacks its ID on a beacon message. Upon receiving a beacon message, nodes check the IDs in it. For each ID, nodes add the ID to set $\mathcal{ID}$ if the ID is positive; otherwise, remove the ID. If a client node finds that the size of set $\mathcal{ID}$ becomes greater than τ_2, a threshold for "significant" global network

condition change, the node notifies V_{del}; and V_{del} executes the Topology Discovery protocol. To reduce the amount of traffic, client nodes request the global network information from V_{del}, instead of running the topology discovery by themselves. After V_{del} completes the topology update, all nodes reset their status variables back to NU, set $p = 0$, $r_{est} = r_{cur}$.

4.3 Reliability-Compliant and Energy-Aware Data Replication

In a traditional k-out-of-n storage system, files are encoded using a single (k, n) parameter for both data encoding and fragment placement or storage: each file is encoded as n data fragments and each fragment is distributed to a unique storage node. This simple scheme, however, is not always suitable for the mobile cloud setting, where both the network topology and the failure probabilities of individual nodes may change rapidly and unpredictably. In other words, the system's reliability and its energy performance will fluctuate. Consequently, a single assignment of the (k, n) parameter is unlikely to be sufficient, and adjusting (k, n) on-demand requires a full cycle of data retrieval, data recovery, data encoding, and fragment placement, which has a major impact on the energy performance. Instead, we propose a proactive approach using two separate (k, n) parameters for data replication and fragment storage, respectively, so that the system can remain energy-efficient and reliable with *minimal maintenance cost* as nodes join and leave the mobile cloud. The maintenance cost here includes the energy for reallocating, regenerating, and redistributing the data fragments. The details are described below.

4.3.1 Fragment Parameter and Storage Parameter

We formally define (k_f, n_f) and (k_s, n_s) as the *fragment parameter* and *storage parameter*, respectively. n_f is the number of data fragments that a data block is encoded into, while n_s is the total number of storage nodes for placing these frag-

ments. k_f is the minimal number of data fragments required for recovering the data object and k_s is the minimal number of storage nodes that a client needs to contact in order to recover a data object. k_s should meet a *feasibility* requirement in that *any subset of k_s storage nodes provide k_f or more distinct data fragments.* As an example, in Figure 3.4 where $(k_s, n_s) = (3, 4)$ and $(k_f, n_f) = (4, 6)$, the yellow block is encoded into $n_f = 6$ data fragments and these fragments are stored on $n_s = 4$ different storage nodes. To recover the yellow block, a client needs to retrieve at least $k_f = 4$ distinct data fragments. Our algorithm ensures that any $k_s = 3$ storage nodes contains $k_f = 4$ distinct fragments. Note that for the security purpose, we assume that data fragments can not be cached on clients and retrieve fragments have to be deleted once clients finish using the file.

In this manner, the reliability of data retrieval depends *entirely* on the *storage parameter* and the failure probabilities of the storage nodes.(This claim is substantiated in the next subsection.) In other words, we can determine a starting value of (k_s, n_s) from the application data reliability requirement and only need to adjust this parameter (and/or select a different set of storage nodes) to adapt to the changing network topology. As such, we then carefully choose a single fragment parameter (k_f, n_f) for which a range of expected (k_s, n_s) values is feasible, so that when the storage parameter or the set of storage nodes changes, the system simply reallocates the existing fragments and thus avoids going through the full cycle of decoding and re-encoding the data object.

4.3.2 Estimating Data Storage Reliability from (k_s, n_s)

Given the storage parameter (k_s, n_s), the locations of the storage nodes, and the reliability of each storage node, we can estimate the reliability of the data object. Suppose n is the set of n_s selected storage nodes, c is the subset of functional nodes

in n, and $\bar{c} = c \setminus n$ is the subset of failed nodes in n. We consider the probability that k_s or more storage nodes remain functional ($k_s \leq |\mathrm{c}| \leq n_s$).

For each size of $|c|$, there are $\binom{n_s}{|c|}$ combinations that need to be considered. Equation 4.6 evaluates the reliability of a system with parameter (k_s, n_s). $\mathbf{S_i}$ is the reliability of a system of exactly i functional nodes; R_l is the reliability of the l^{th} node in c; Q_m is the failure probability of the m^{th} node in $\bar{c}$. Although this is a straightforward computation, its time complexity is $O(n!)$ because the combinations of subset c in Equation 4.7 can be large. Furthermore, because we are to search for a single storage parameter in Equation 4.6 that satisfies the reliability requirement, there are $\sum_n^N n$ possible (k_s, n_s) pairs that need to be considered (N is the network size). To account for this computation infeasibility on mobile devices, we propose an approximation algorithm to pre-compute the reliability offline and perform a table-lookup at run time. More specifically, we discretize all the variables in Equation 4.6 and Equation 4.7, and build a table of reliability with respect to different (k_s, n_s) and nodes reliability.

We first approximate R_l and Q_m by the *mean reliability* r and the *mean failure probability* $(1 - r)$. We then discretize parameters k_s, n_s, and r so that a set of reliabilities can be pre-computed and stored in a table. The reliability calculation in Equation 4.6 is thus simplified to $R(k_s, n_s) = \sum_{i=k_s}^{n_s} \binom{n_s}{i} r^i (1-r)^{n_s - i}$. This simplified reliability computation can further be written recursively as $R(k_s, n_s) = R(i, j) = (1-r)R(i, j-1) + rR(i-1, j-1)$. The recursive form allows the table to be built efficiently with dynamic programming. Table 4.1 is an example. We will show in the evaluation section (Figure 4.12a) that this approximation can accurately guide the searching procedure to derive a good storage parameter.

	n_s=5	n_s=6	n_s=7	n_s=8	n_s=9
k_s=1	0.92	0.95	0.99	0.999	0.999
k_s=2	0.82	0.85	0.87	0.89	0.92
k_s=3	0.72	0.79	0.82	0.84	0.85

Table 4.1: Reliability Lookup Table. A 2-D slice of a 3-D lookup table. The reliability r in this 2-D table is 0.8

$$R(k_s, n_s) = \sum_{i=k_s}^{n_s} \mathbf{S_i} \tag{4.6}$$

$$\mathbf{S_i} = \sum_{j=1}^{\binom{n_s}{i}} \prod_{l \in c} R_l \prod_{m \in \bar{c}} Q_m \tag{4.7}$$

where $\forall c \subset n$ and $|c| = i$

4.3.3 Determining (k_s, n_s)

Storage parameter (k_s, n_s) and the allocation of n_s storage nodes are obtained by solving the k-out-of-n storage allocation problem (Eq. 4.8-4.12). A set of *Candidate Storage Parameters* is first selected from the table lookup, and a single storage parameter is then found by solving the optimization problem. Suppose the reliability requirement is 0.8. By checking Table 4.1, we find a set of (k_s, n_s) that meets the reliability requirement (values ≥ 0.8). Although multiple k_s in each column may satisfy the reliability requirement, only the one with the highest k_s is selected, as using lower k_s incurs higher data redundancy and maintenance cost. We call these shaded cells in Table 4.1 as *candidate storage parameters* $((2,5),(2,6),(3,7)...)$.

To select a single storage parameter, we consider how the storage parameter

affects the load distribution. Assume a data object of size $\mathbf{s}$ bytes is stored into a system of N nodes and each node requests the data object once during the effective time period $\mathbf{T_e}$. The total number of fragments transmitted during $\mathbf{T_e}$ is Nk_f (each node downloads k_f fragments) and each storage node delivers on average Nk_f/n_s fragments, or equivalently $Nk_f/n_s \times \mathbf{s}/k_f = N\mathbf{s}/n_s$ bytes (each data fragment is approximately $\mathbf{s}/k_f$ bytes). To limit the traffic on each storage node, each storage should not transmit more than $\mathbf{M}$ bytes within $\mathbf{T_e}$. This *bandwidth constraint* is then written as $N\mathbf{s}/n_s \leq M$ or $n_s \geq N\mathbf{s}/M$.

We now describe how a new k-out-of-n storage allocation problem using split parameters is formulated. The objective function, Equation 4.8, minimizes the data retrieval time. Note that the "transmission time" here indicates the "transmission energy" as the RF energy consumption is proportional to the radio transmitting time [13]. Hereafter, we use the terms transmission energy and transmission time interchangeably when referring to the energy. In Equation 4.8, D_{ij} is the expected transmission time for node j to send a byte to node i, index c is the file creator, R_{ij} is a decision variable indicating whether client node i retrieves data from storage node j, and X_j is a decision variable indicating whether node j is selected as a storage node. The expected transmission time matrix D is estimated based on [22]. $D_{ij}R_{ij}$ indicates total data retrieval cost, $D_{jc}X_j$ indicates the data creation cost, and $\mathbf{s}/f_k(\sum_{j=1}^{N} X_j)$ is the size of each data fragment. Constraint 1 (Equation 4.9) enforces a lower bound on n_s according to the bandwidth constraint; constraint 2 (Equation 4.10) ensures that each client is assigned enough storages nodes (k_s or more); function $f_k(n_s)$ returns the corresponding k_s of the given n_s in the candidate storage parameters; constraint 3 (Equation 4.11) ensures that X_j is 1 if and only if j^{th} column of R has non-zero values.

The solution for the optimization problem gives a single storage parameter (k_s, n_s)

and the allocation of the storage nodes (X_i) such that the system minimizes the transmission energy while meeting the reliability requirement. This combinatorial optimization problem is solved by a Tabu Search heuristic [35]. The detail of how we formulate the Tabu Search will be described in Section 5.4.

$$R_{opt} = \arg\min_{R} \ \mathbf{s}/f_k(\sum_{j=1}^{N} X_j) \times (\sum_{i=1}^{N}\sum_{j=1}^{N} D_{ij}R_{ij} + \sum_{j=1}^{N} D_{jc}X_j) \tag{4.8}$$

Subject to:

$$\sum_{j=1}^{N} X_j \geq N\mathbf{s}/M \tag{4.9}$$

$$\sum_{j=1}^{N} R_{ij} \geq f_k(\sum_{j=1}^{N} X_j) \ \forall i \tag{4.10}$$

$$NX_j \geq \sum_{i=1}^{N} R_{ij} \geq X_j \ \forall j \tag{4.11}$$

$$X_j \text{ and } R_{ij} \in \{0,1\} \ \forall i,j \tag{4.12}$$

4.3.4 Determining (k_f, n_f)

Fragment parameter (k_f, n_f) controls how a data object is encoded and the level of data redundancy. We aim to reduce the data maintenance cost when selecting the fragment parameter. Recall that a storage parameter is deemed *feasible* if the n_f data fragments can be distributed to the n_s selected storage nodes and any subset of k_s storage nodes contains at least k_f data fragments. When a topology change occurs, the number of available fragments n_f may change (nodes fail or leave the network) or the storage parameter may be updated. If the current (k_f, n_f) still has a

Algorithm 2: FeasibilityTest

Input: (k_s, n_s, k_f, n_f)
Output: $feasible$
$feasible = false$
$uf = n_s - (n_f \mod n_s)$
if $uf \geq k_s$ **then**
  if $\lfloor \frac{n_f}{n_s} \rfloor k_s \geq k_f$ **then**
    $feasible = true$
  end
else
  if $\lfloor \frac{n_f}{n_s} \rfloor uf + \lceil \frac{n_f}{n_s} \rceil (k_s - uf) \geq k_f$ **then**
    $feasible = true$
  end
end
return $feasible$

feasible allocation, then the system can be repaired by reallocating data fragments. If no feasible allocation is possible, the data object needs to be re-distributed. Note that an infeasible allocation does not mean the data is lost; it simply indicates that the reliability requirement imposed by the storage parameter cannot be satisfied by the current parameter settings. The data can still be recovered as long as k_f data fragments are available.

We derive a simple feasibility test to check whether a feasible allocation exists for the given (k_s, n_s) and (k_f, n_f). Algorithm 2 tries to distribute n_f data fragments uniformly to n_s storage nodes. Once all n_f fragments have been allocated, we find the subset of k_s storage nodes with the *least number of fragments*; if this subset contains n_f or more fragments, the current parameter settings have at least one feasible allocation. Otherwise, there is no feasible allocation. uf indicates the number of "under-filled nodes", in which each node has $\lfloor \frac{n_f}{n_s} \rfloor$ fragments (assume $n_f \geq n_s$). uf helps identify the k_s storage nodes that has the fewest data fragments.

$$
\begin{aligned}
& uf = n_s - (n_f \mod n_s) \\
& k_f \leq \begin{cases} \lfloor \frac{n_f}{n_s} \rfloor k_s & \text{if uf} \geq k_s \\ \lfloor \frac{n_f}{n_s} \rfloor uf + (k_s - uf) \lceil \frac{n_f}{n_s} \rceil & \text{otherwise} \end{cases}
\end{aligned} \tag{4.13}
$$

$$
\begin{aligned}
& \hat{n_s} + \alpha \hat{n_s} \geq n_s \geq \hat{n_s} - \alpha \hat{n_s} \geq 0 \\
& \hat{r} \geq r \geq \hat{r} - \beta \hat{r} \geq R_{req} \\
& \alpha, \beta \geq 0; \quad n_s \in integer
\end{aligned} \tag{4.14}
$$

Similar to the feasibility test, Equation 4.13 finds the largest feasible k_f when given the storage parameter and n_f. The rationale is that the subset of k_s storage nodes with the fewest data fragments must have at least k_f fragments in order to have a feasible allocation.

Overall, we prefer a fragment parameter that can support multiple possible storage parameters. The fragment parameter must be carefully selected to adapt to the most possible storage parameters while keeping the data redundancy low. Naturally, the possible storage parameters are those in the *candidate storage parameters*. Suppose $(\hat{k_s}, \hat{n_s})$ is the selected storage parameters and $\hat{r}$ is the mean reliability of the selected storage nodes. We select a (k_f, n_f) that can adapt to the storage parameters in the range specified in Equation 4.14. α determines the range of storage parameters that (k_f, n_f) can support, and β determines the mean reliability reduction that (k_f, n_f) can tolerate. R_{req} is the minimum reliability requirement of the application.

For example, if we refer to Table 4.1 and set $(\hat{k_s}, \hat{n_s}) = (3, 7)$, $\hat{r} = 0.8$, $\alpha = 0.2$, and $\beta = 0.1$. Assume the table contains 21 discrete reliability uniformly spread in $[0, 1]$. The mean reliability that the fragment parameter needs to support is

$r = 0.8$ and $r = 0.75$ because these are the only two discrete reliability within range $0.8 \geq r \geq 0.72$. For $r = 0.8$, the storage parameters in the range are $(2, 6)$, $(3, 7)$ and $(3, 8)$ (by table lookup); for $r = 0.75$, another three storage parameters $(2, 6)$, $(2, 7)$ and $(3, 8)$ are in the range. The selected fragment parameter needs to support all these storage parameters.

To determine a single fragment parameter $(\hat{k_f}, \hat{n_f})$, $\hat{n_f}$ is set to the maximum n_s in Equation 4.14, i.e., $\hat{n_f} = \lceil (1 + \alpha)\hat{n_s} \rceil$. Using this $\hat{n_f}$ and Equation 4.13, we find the feasible fragment parameters for each candidate storage parameter. E.g, given $(k_s, n_s) = (3, 7)$ and $\hat{n_f} = 8$, the feasible k_f are 1, 2, and 3. In this manner, we obtain an intersection of all the feasible fragment parameters. In the intersection set, the one with the highest k_f is selected. This selected $(\hat{k_f}, \hat{n_f})$ is feasible to all storage parameters in the range specified by Equation 4.14.

4.3.5 Fragment Re-allocation

When the topology maintenance component detects a significant topology change, the system re-evaluates the storage and fragment parameters, and the allocations of the data fragments. If there are still sufficient data fragments, the system simply moves the fragments from the old storage nodes to the new storage nodes. In this section, we present an algorithm based on the *minimum-cost flow problem* to reallocate the fragments with minimal transmission energy.

The problem is formulated as a directed graph shown in Figure 4.2 where the left side is the set of current storage nodes and the right side is the set of newly selected storage nodes. All current nodes are connected to a virtual source node and all new nodes are connected to a virtual destination node.

Each arc is associated with a cost a_{ij} and a capacity c_{ij}, represented by (a_{ij}, c_{ij}); the arcs between the current nodes and the new nodes are assigned infinite capacity

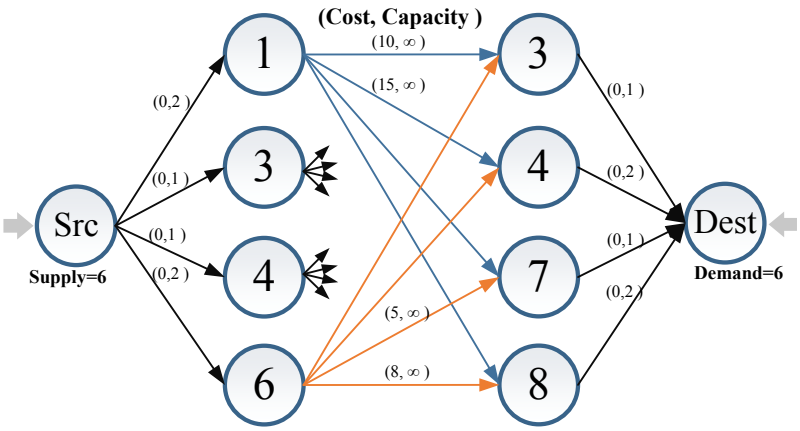


Figure 4.2: The Minimum-cost flow problem formulation

and their costs are set to the D_{ij}; the arcs from the virtual source to the current nodes are assigned zero cost and their capacities are equal to the number of fragments on the current node; the arcs from the new storage nodes to the virtual destination are assigned zero cost and their capacities are equal to the number of required fragments on the new node. The *supply* of the virtual source and the *demand* of the virtual destination are both set to the total number of fragments to be transferred.

The optimization problem is expressed as a Linear Programming problem in Equations 4.15-4.17; the objective function (Equation 4.15) minimizes the cost for sending supply from the source to the destination; x_{ij} is the decision variable indicating the number of fragments node i sends to node j. Equation 4.16 ensures that the flow conservation property on all nodes except the source and the destination; V is the magnitude of supply/demand; Equation 4.17 ensures the flow on each arc does not exceed the capacity. This particular linear programming problem is solved efficiently in polynomial time by the *network simplex algorithm* [79].

$$x_{opt} = \arg\min_{x} \sum_{i=1}^{n_s} \sum_{j=1}^{n_s} a_{ij} x_{ij} \tag{4.15}$$

Subject to:

$$\sum_{j} x_{ji} - \sum_{j} x_{ij} = \begin{cases} V, & i = src, \\ 0, & i \neq src, dest, \\ -V, & i = dest. \end{cases} \qquad \forall i \tag{4.16}$$

$$0 \leq x_{ij} \leq c_{ij} \tag{4.17}$$

4.4 Caching for the k-out-of-n Distributed Storage

In this section, we explore data caching for k-out-of-n computing in mobile cloud environments, with the goal of distributing data in a way that the expected future energy consumption for nodes to retrieve data is minimized, while preserving reliability. More specifically, we propose to place data caches (in addition to the originally stored data) based on the actual data access patterns and the network topology. We formulate the cache placement optimization problem and propose a centralized caching framework *(CC)* that optimally solves the problem and a distributed solution that approximates the optimal solution. The distributed caching framework *(DC)* learns data access patterns by sniffing packets and informing a resident cache daemon about popular data items.

The k-out-of-n distributed storage system so far assumes the network is homogeneous and all nodes have equal probability to request each file. However, in reality, not all nodes request all files and some files may be requested only by a small portion of nodes. For instance, given a network of rectangle shape, if the files are requested only by client nodes located at the shorter edges of the rectangle, it will be extremely energy inefficient to place storage node at the center of the network. Additionally, for

security concerns, client nodes are not allowed to keep the decoded files locally and nodes always need to retrieve the file fragments from the storage nodes whenever a file is needed for reading. This security constraint causes an unavoidable high energy consumption and heavy network traffics. To address these challenges, we propose to cache some "popular" data fragments in the network and allow client nodes to retrieve file fragments from nearby caching nodes instead of always going to the farther storage nodes.

Our caching strategy is designed based on two observations: temporal locality of file access and the group mobility exhibited by nodes. Temporal locality of file access means that a file recently accessed by a node is likely to be accessed again by the same node in the near future. Thus collecting statistics, i.e., how files were accessed by nodes in the past, lays ground for predicting the future. Group mobility exhibited by nodes indicates that nodes often move as a group instead of moving individually. As a result, placing cached data within a group of nodes that tend to move together can also greatly improve the performance.

4.4.1 Cache Placement Formulation

Now we are ready to formulate the cache placement optimization problem. The objective of the problem is to minimize the total expected distance from every potential user to its k cache agents. For convenience, we omit the file index w and represent the file as F in the problem formulation. Based on the previous definition, two mapping variables are defined: $\mathbf{x_i^l}$ is a binary variable indicating whether node v_i is a cache agent for fragment f_l, and $\mathbf{y_{ij}^l}$ is a binary variable indicating whether node v_j is assigned to node v_i for retrieving fragment f_l. The following Integer Linear Program (ILP) expresses our cache placement problem.

The first constraint (Equation 4.19) indicates that up to K fragment copies will

$$\text{Minimize} \quad \sum_{l \in F} \sum_{i \in V} \sum_{j \in U} D_{ij} y_{ij}^{l} r_j \tag{4.18}$$

$$\text{s.t.} \quad \sum_{l \in F} \sum_{i \in V} x_i^l \leq K \tag{4.19}$$

$$\sum_{l \in F} \sum_{i \in V} y_{ij}^l \geq k, \forall j \in U \tag{4.20}$$

$$x_i^l \geq y_{ij}^l, \forall i \in V, \forall j \in U, \forall l \in F \tag{4.21}$$

$$x_{s_l}^l = 1, \forall l \in F \tag{4.22}$$

$$\sum_{l \in F} x_i^l \leq \min\{k-1, L_i - A_i\}, \forall i \in V \tag{4.23}$$

$$x_i^l, y_{ij}^l \in \{0,1\}, \forall i \in V, \forall j \in U, \forall l \in F \tag{4.24}$$

be placed on the cache agents for this file. Parameter K is determined based on the file popularity and buffer size. How it is calculated is described in Equation 4.25 and Equation 4.26. The second constraint (Equation 4.20) ensures that each potential user has accesses to at least k different fragment caches. The k parameter is configured by the application depending on the reliability requirement. The third constraint (Equation 4.21) makes sure that if a potential user is assigned to a node for a particular fragment, then the node must be a cache agent for that fragment. Equation 4.22 ensures that the service centers are also cache agents. Equation 4.23 creates a buffer limit on each cache agent. Also, for security purposes, less than k cached fragments can be created for each file. The last constraint (Equation 4.24) is the binary requirement for the decision variables.

We adopt the findings from [52] to help determine the number of caches for each file, K_w, given the file's popularity and the nodes' buffer size. In Equation 4.25 below, n is the number of service centers selected when a file is created, ϕ represents the correlation between a file's popularity and the total number of its cached fragments,

and r_w is the request frequency of file F_w. The minimum number of caches for each file is n because each file is encoded and distributed to n service centers at the creation time. Equation 4.26 defines a user-configured variable η to represent the percentage of occupancy allowed on cache agents' buffer. Combining Equation 4.25 with Equation 4.26 and configuring a proper η, we can then solve for ϕ.

$$K_w = \max\left\{n, \phi \cdot \left(\sum_{i \in U} r_i\right)^{2/3}\right\} \tag{4.25}$$

$$\eta = \frac{\text{total \# of fragment copies of all files}}{\text{overall buffer size of all nodes}} = \frac{\sum_w K_w}{\sum_i L_i} \tag{4.26}$$

Based on this, files with higher popularity are given higher priority when selecting the cache placement. In specific, given a collection of files $\{F_w\}$, the cache placements for each file is determined one by one based on its K_w value. The process repeats until all files are associated with a specific K_w.

4.4.2 Distributed Caching Framework

The goal of distributed caching framework (DC) is to allow each individual node to make its own caching decision without the need of global information. The distributed algorithm does not collect the topology information or all files' access pattern, and is robust to node failures.

To learn the file access pattern, the network layer sniffs the passing-by packets and delivers the packets of interest to the middleware. In this way, Cache Daemon (CDaemon) learns the file request frequency and by whom the file is requested. Once nodes have identified the popular fragments needed to be cached, DC algorithm needs to determine where to place the cache fragments to maximize the energy saving. The intuition is to select the nodes that are closest to the file requestors in terms of hop

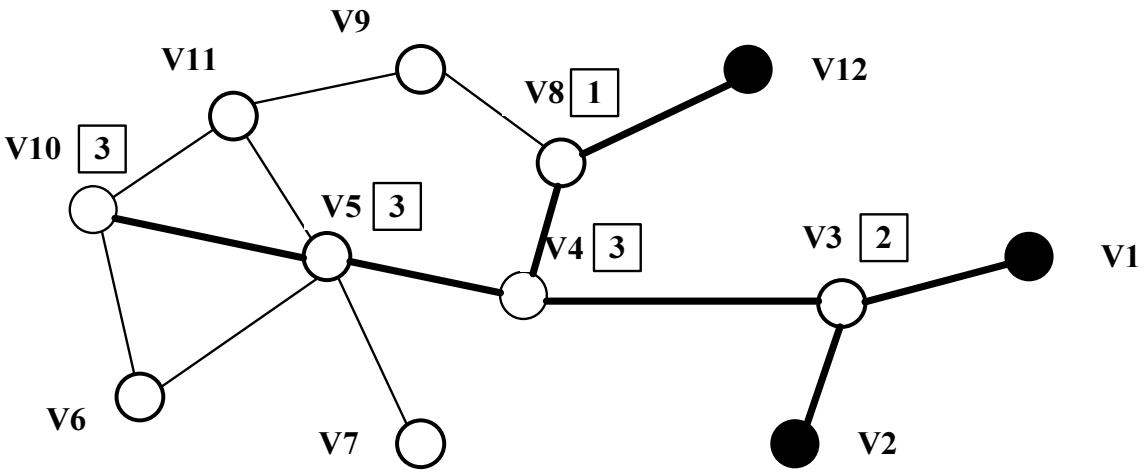


Figure 4.3: An example of cache placement in distributed caching framework. The number in the square indicates the frqReq counter for a specific fragment. Black circles are fragment requesters.

count. Following the logic of our framework, we anticipate the node that first observes the popular fragment to be the closest one to the users group. All the intermediate nodes on the route cooperate to determine the best cache agent. An example is illustrated in Figure 4.3.

4.4.2.1 When to create a cache fragment?

A new fragment cache is added when we find the counter associated with the fragment exceeds a predefined threshold θ. Suppose θ is set to 3 in Figure 4.3. After v_1, v_2, and v_{12} make the same *fragReq* destined for v_{10}, v_4, v_5, and v_{10} will update their counter of the requested fragment to 3. This will trigger *DC* to add a new fragment cache. Parameter θ has a great impact on the system performance as it affects the frequency that the cached fragments are updated. We determine θ by estimating the average number of requests that each fragment cache will serve. If the number of actual requests exceeds the predefined value, then an extra fragment cache is necessary.

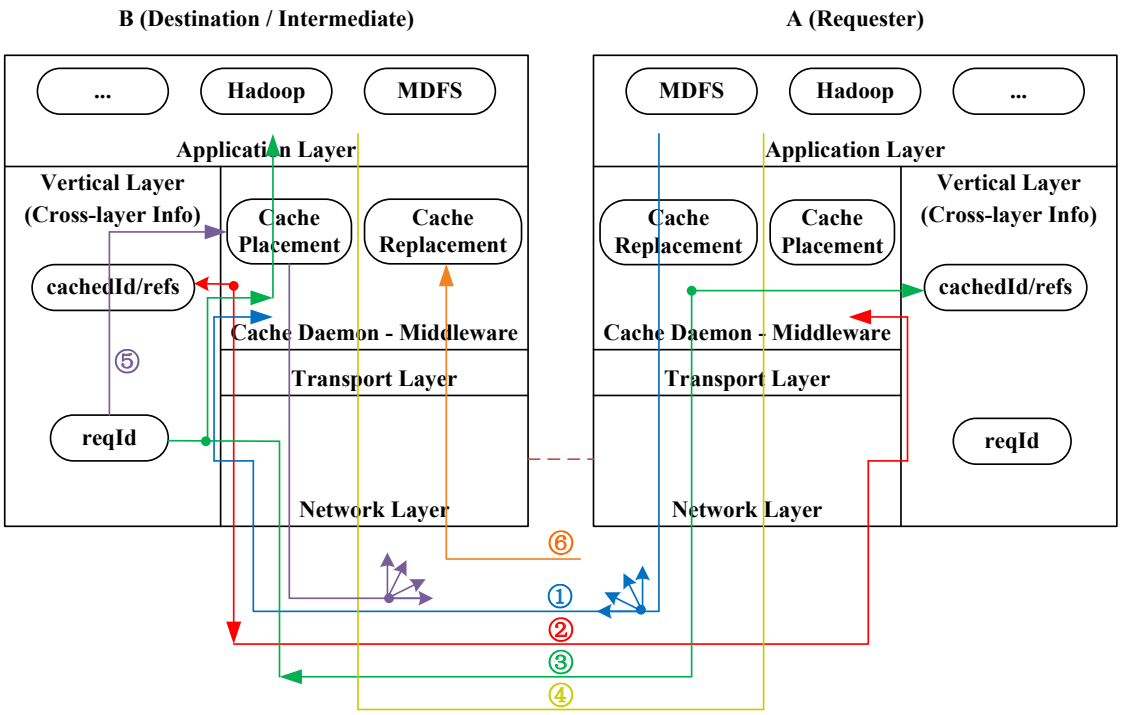


Figure 4.4: System architecture of cross-layer design for proposed distributed caching framework. 1) *fileReq*: broadcast a file request; 2) *fileRep*: unicast a file reply (may require a route discovery); 3) *fragReq*: unicasting a fragment request (may require a route discovery); 4) data transmission via TCP; 5) cache placement: broadcast an exchange request (*exReq*) to one-hop neighbors, unicast an exchange reply (*exRep*), and unicast an exchange confirmation (*exCfm*); 6) cache replacement.

In the previous example, when v_4, v_5 and v_{10} all reach the threshold defined by θ, only one of them should initiate its cache placement module. Since our objective is to minimize the distance from the file requestors to the fragment cache, v_4 seems to be the best candidate among the three. From the observation that the node closest to the file requestors reaches the threshold earlier than other candidate nodes (v_5 and v_{10}), v_4 can actively notify other candidates NOT to cache the fragment.

4.4.2.2 How to select a cache agent

Although only v_4 will initiate its cache placement module, any node in its vicinity has a chance to be selected as the cache agent. v_4 coordinates with all its 1-hop neighbors and determines the best cache agent by comparing their *qualification scores*, defined in Equation 4.27.

$$score(i) = I(i) \cdot \left\{ \alpha \cdot (1 - Pf_i) + (1 - \alpha) \cdot \frac{L_i - A_i}{L_i} \right\} \tag{4.27}$$

In Equation 4.27, $I(i)$ is an indicator variable showing whether adding the new fragment cache will violate the security constraint on v_i, Pf_i is the failure probability of node v_i, L_i is the buffer capacity of node v_i, A_i is the number of cached items on v_i, and α is a weight parameter in the range $(0, 1)$. We define the score in such a way to eliminate the nodes that may violate the security constraint, and give the nodes with lower failure probability or more buffer space higher score.

4.5 Evaluation

In this section, we first present the hardware implementation results on Android smartphones. More extensive simulation results in larger-scale network are then shown.

4.5.1 Real-World Implementation

This section investigates the feasibility of running our framework on real hardware. We compare the performance of our framework with a random data allocation and processing scheme (Random), which randomly selects storage/processor nodes. Specifically, to evaluate the k-out-of-n data allocation on real hardware, we implemented a Mobile Distributed File System (MDFS) on top of our k-out-of-n comput-

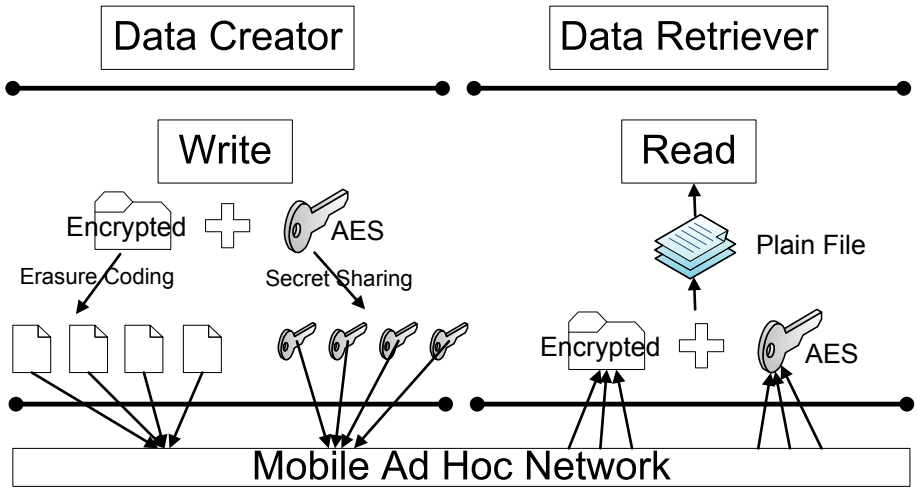
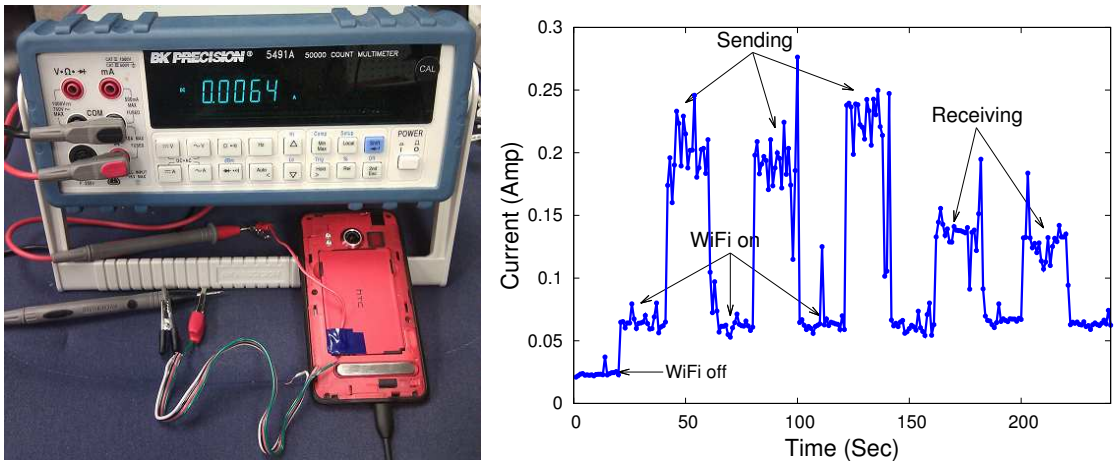


Figure 4.5: An overview of our Mobile Distributed File System (MDFS).

ing framework. We also test our k-out-of-n data processing by implementing a face recognition application that uses our MDFS.

Figure 4.5 shows an overview of our MDFS. Each file is encrypted and encoded by erasure coding into n_1 data fragments, and the secret key for the file is decomposed into n_2 key fragments by key sharing algorithm. Any *maximum distance separable code* can may be used to encoded the data and the key; in our experiment, we adopt the well-developed Reed-Solomon code and Shamir's Secret Sharing algorithm. The n_1 data fragments and n_2 key fragments are then distributed to nodes in the network. When a node needs to access a file, it must retrieve at least k_1 file fragments and k_2 key fragments. Our k-out-of-n data allocation allocates file and key fragments optimally when compared with the state-of-art [46] that distributes fragments uniformly to the network. Consequently, our MDFS achieves higher reliability (since our framework considers the possible failures of nodes when determining storage nodes) and higher energy efficiency (since storage nodes are selected such that the energy consumption for retrieving data by any node is minimized).

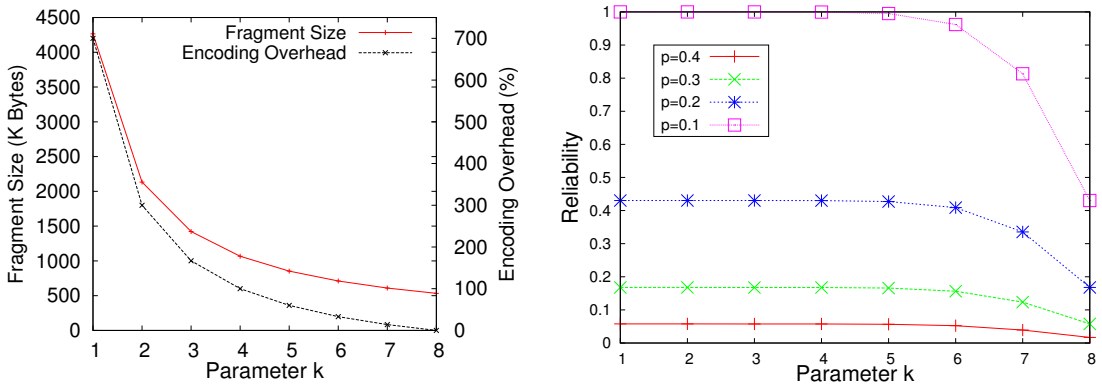


(a) (b)

Figure 4.6: (a) Energy measurement setting. (b) Current consumption on Smartphone in different states.

We implemented our system on HTC Evo 4G Smartphone, which runs Android 2.3 operating system using 1G Scorpion CPU, 512MB RAM, and a Wi-Fi 802.11 b/g interface. To enable the Wi-Fi AdHoc mode, we rooted the device and modified a config file – wpa_supplicant.conf. The Wi-Fi communication range on HTC Evo 4G is 80-100m. Our data allocation was programmed with 6,000 lines of Java and C++ code.

The experiment was conducted by 8 students who carry smartphones and move randomly in an open space. These smartphones formed an Ad-Hoc network and the longest node to node distance was 3 hops. Students took pictures and stored in our MDFS. To evaluate the k-out-of-n data processing, we designed an application that searches for human faces appearing in all stored images. One client node initiates the processing request and all selected processor nodes retrieve, decode, decrypt, and analyze a set of images. In average, it took about $3-4$ seconds to process an image of size 2MB. Processing a sequence of images, e.g., a video stream, the time may

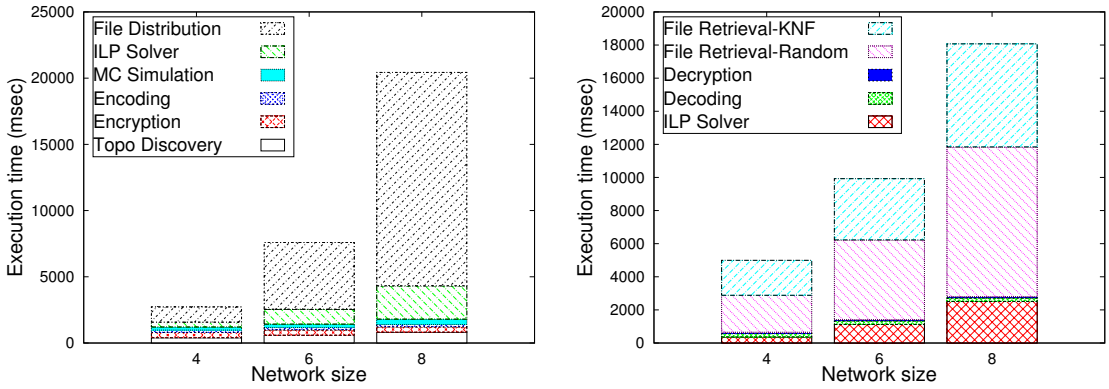


(a) (b)

Figure 4.7: (a) A file of 4.1 MB is encoded with n fixed to 8 and k swept from 1 to 8. (b) Reliability with respect to different k/n ratio and failure probability.

increase in an order of magnitude. The peak memory usage of our application was around 3MB. In addition, for a realistic energy consumption model in simulations, we profiled the energy consumption of our application (e.g., WiFi-idle, transmission, reception, and 100%-cpu-utilization). Figure 4.6a shows our experimental setting and Figure 4.6b shows the energy profile of our smartphone in different operating states. It shows that Wi-Fi component draws significant current during the communication(sending/receiving packets) and the consumed current stays constantly high during the transmission regardless the link quality.

Figure 4.7a shows the overhead induced by encoding data. Given a file of 4.1MB, we encoded it with different k values while keeping parameter $n = 8$. The left y-axis is the size of each encoded fragment and the right y-axis is the percentage of the overhead. Figure 4.7b shows the system reliability with respect to different k while n is constant. As expected, smaller k/n ratio achieves higher reliability while incurring more storage overhead. An interesting observation is that the change of system reliability slows down at $k = 5$ and reducing k further does not improve the



(a) (b)

Figure 4.8: Execution time of different components with respect to various network size.

reliability much. Hence, $k = 5$ is a reasonable choice where overhead is low ($\approx 60\%$ of overhead) and the reliability is high($\approx 99\%$ of the highest possible reliability).

To validate the feasibility of running our framework on a commercial smartphone, we measured the execution time of our MDFS application in Figure 4.8a. For this experiment we varied network size N and set $n = \lceil 0.6N \rceil$, $k = \lceil 0.6n \rceil$, $k_1 = k_2 = k$, and $n_1 = n_2 = n$. As shown, nodes spent much longer time in distributing/retrieving fragments than other components such as data encoding/decoding. We also observe that the time for distributing/retrieving fragments increased with the network size. This is because fragments are more sparsely distributed, resulting in longer paths to distribute/retrieve fragments. We then compared the data retrieval time of our algorithm with the data retrieval time of random placement. Figure 4.8b shows that our framework achieved 15% to 25% lower data retrieval time than Random. To validate the performance of our k-out-of-n data processing, we measured the completion rate of our face-recognition job by varying the number of failure node. The face recognition job had an average completion rate of 95% in our experimental

setting.

4.5.2 The k-out-of-n Data Storage Simulation

We conducted simulations to evaluate the performance of our k-out-of-n framework (denoted by KNF) in larger scale networks. We consider a network of $400 \times 400 \text{m}^2$ where up to 45 mobile nodes are randomly deployed. The communication range of a node is 130m, which is measured on our smartphones. Two different mobility models are tested – Markovian Waypoint Model and Reference Point Group Mobility (RPGM). Markovian Waypoint is similar to Random Waypoint Model, which randomly selects the waypoint of a node, but it accounts for the current waypoint when it determines the next waypoint. RPGM is a group mobility model where a subset of leaders are selected; each leader moves based on Markovian Waypoint model and other non-leader nodes follow the closest leader. Each mobility trace contains 4 hours of data with 1Hz sampling rate. Nodes beacon every 30 seconds.

We compare our KNF with two other schemes – a greedy algorithm (Greedy) and a random placement algorithm (Random). Greedy selects nodes with the largest number of neighbors as storage/processor nodes because nodes with more neighbors are better candidates for cluster heads and thus serve good facility nodes. Random selects storage or processor nodes randomly. The goal is to evaluate how the selected storage nodes impact the performance. We measure the following metrics: consumed energy for retrieving data, consumed energy for processing a job, data retrieval rate, completion time of a job, and completion rate of a job. We are interested in the effects of the following parameters – *mobility model*, *node speed*, k/n *ratio*, τ_2, and *number of failed nodes*, and *scheduling*. The default values for the parameters are: $N = 26$, $n = 7$, $k = 4$, $\tau_1 = 3$, $\tau_2 = 20$; our default mobility model is RPGM with node-speed 1m/s. A node may fail due to two independent factors: depleted energy

or an application-dependent failure probability; specifically, the energy associated with a node decreases as the time elapses, and thus increases the failure probability. Each node is assigned a constant application-dependent failure probability.

We first perform simulations for the k-out-of-n data allocation by varying the first four parameters and then simulate the k-out-of-n data processing with different number of failed nodes. We evaluate the performance of data processing only with the number of node failures because data processing relies on data retrieval and the performance of data allocation directly impacts the performance of data processing. If the performance of data allocation is already bad, we can expect the performance of data processing will not be any better.

The simulation is performed in Matlab. The energy profile is taken from our real measurements on smartphones; the mobility trace is generated according to RPGM mobility model; and the linear programming problem is solved by the Matlab optimization toolbox.

4.5.2.1 Effect of Mobility

In this section, we investigate how mobility models affect different data allocation schemes. Figure 4.9 depicts the results. An immediate observation is that mobility causes nodes to spend higher energy in retrieving data compared with the static network. It also shows that the energy consumption for RPGM is smaller than that for Markov. The reason is that a storage node usually serves the nodes in its proximity; thus when nodes move in a group, the impact of mobility is less severe than when all nodes move randomly. In all scenarios, KNF consumes lower energy than others.

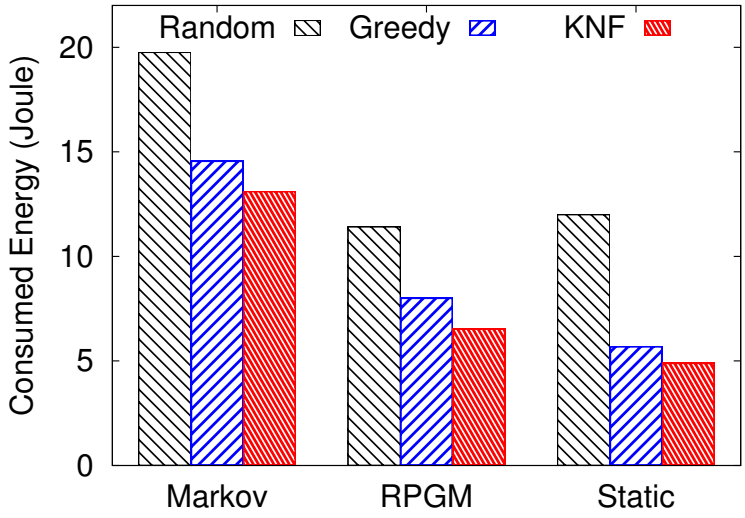


Figure 4.9: Effect of mobility on energy consumption. We compare the three different allocation algorithms under different mobility models.

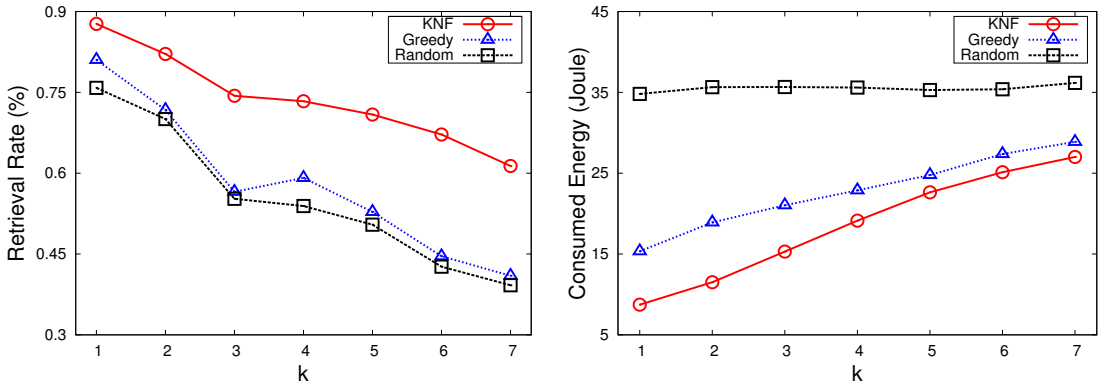


Figure 4.10: (a) Effect of k/n ratio on data retrieval rate when $n = 7$. (b) Effect of k/n ratio on energy efficiency when $n = 7$.

4.5.2.2 Effect of k/n Ratio

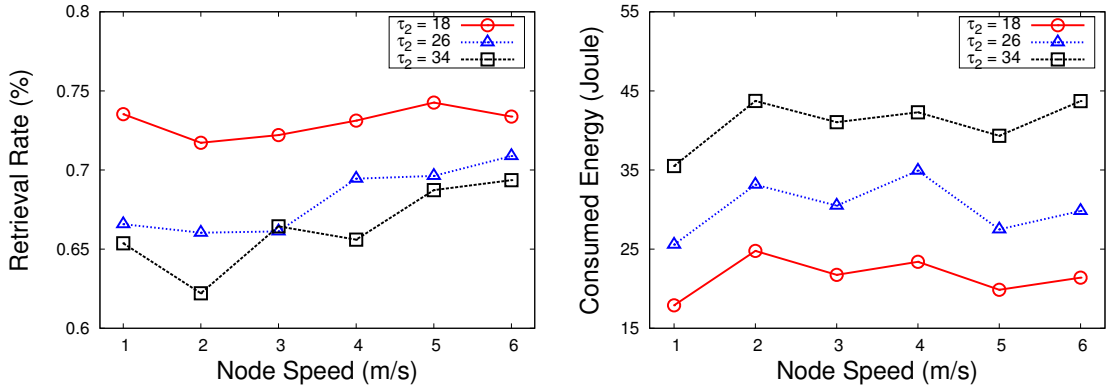
Parameters k and n, set by applications, determine the degree of reliability. Although lower k/n ratio provides higher reliability, it also incurs higher data redun-

dancy. In this section, we investigate how the k/n ratio (by varying k) influences different resource allocation schemes. Figure 4.10a depicts the results. The data retrieval rate decreases for all three schemes when k is increased. It is because, with larger k, nodes have to access more storage nodes, increasing the chances of failing to retrieve data fragments from all storage nodes. However, since our solution copes with dynamic topology changes, it still yields 15% to 25% better retrieval rate than the other two schemes.

Figure 4.10b shows that when we increase k, all three schemes consume more energy. One observation is that the consumed energy for Random does not increase much compared with the other two schemes. Unlike KNF and Greedy, for Random, storage nodes are randomly selected and nodes choose storage nodes randomly to retrieve data; therefore, when we run the experiments multiple times with different random selections of storage nodes, we eventually obtain a similar average energy consumption. In contrast, KNF and Greedy select storage nodes based on their specific rules; thus, when k becomes larger, client nodes have to communicate with some storage nodes farther away, leading to higher energy consumption. Although lower k/n is beneficial for both retrieval rate and energy efficiency, it requires more storage and longer data distribution time. A 1MB file with $k/n = 0.6$ in a network of 8 nodes may take 10 seconds or longer to be distributed (as shown in Figure4.8b).

4.5.2.3 Effect of τ_2 and Node Speed

Figure 4.11a shows the average retrieval rates of KNF for different τ_2. We can see that smaller τ_2 allows for higher retrieval rates. The main reason is that smaller τ_2 causes KNF to update the placement more frequently. We are aware that smaller τ_2 incurs overhead for relocating data fragments, but as shown in Figure 4.11b, energy consumption for smaller τ_2 is still lower than that for larger τ_2. The reasons are, first,



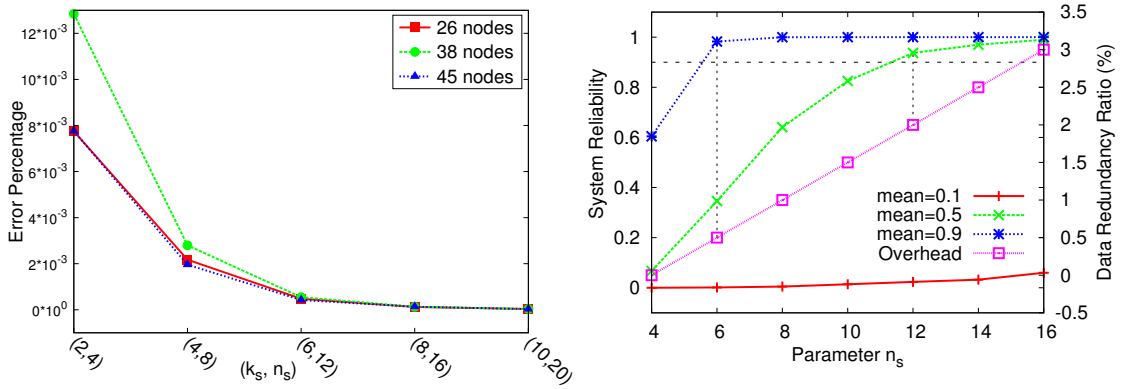
(a) (b)

Figure 4.11: (a) Effect of τ_2 and node speed on data retrieval rate. (b) Effect of τ_2 and node speed on energy efficiency.

energy consumed for relocating data fragments is much smaller than energy consumed for inefficient data retrieval; second, not all data fragments need to be relocated. Another interesting observation is that, despite higher node speed, both retrieval rates and consumed energy do not increase much. The results confirm that our network monitoring component works correctly: although nodes move with different speeds, our component reallocates the storage nodes such that the performance does not degrade much.

4.5.3 Reliability-Compliant Data Replication Evaluation

In this section, we employ both synthetic traces and a Dartmouth Network Trace [37]. Specifically, we are interested in the accuracy of reliability estimation and data maintenance/reallocation energy. These metrics are measured for different network sizes, number of storage/processor nodes, and number of failure nodes. We first evaluate the performance of our solution for data storage and maintenance. Given an application reliability requirement, we compare the energy consumption for data storage and maintenance using either the traditional single (k, n) parameter, or



(a) (b)

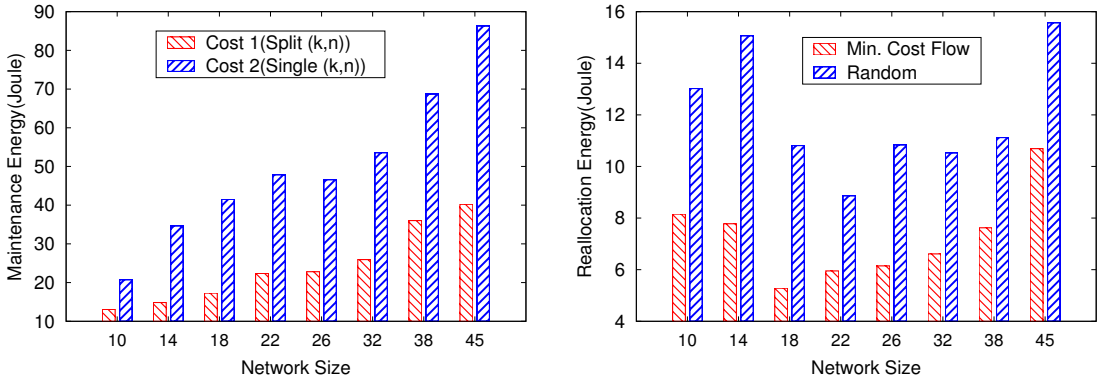
Figure 4.12: (a) The error of system reliability estimation in different network sizes. (b) The effect of (k_s, n_s) on the system reliability and data redundancy.

our proposed *variable data fragmentation*. When evaluating our framework on the real network trace, we also compare the performance with Hadoop Distributed File System (HDFS).

4.5.3.1 Synthetic Network Trace

Figure 4.12a depicts the accuracy of the table-lookup reliability estimation. Given the storage parameter (k_s, n_s), we compare the true reliability computed with Equation 4.6 and the approximated reliability estimated by table lookup. The results show that the error is less than 1% and decreases as n_s increases. When more than half of the nodes in the network are selected, the error drops to below 0.1%. The error decreases with n_s because a larger selected subset is better approximated by the mean network reliability.

Figure 4.12b shows the effects of storage parameter (k_s, n_s) on the system reliability and data redundancy. In this experiment, the network size is 16, k_s is fixed at 4, (k_f, n_f)=(k_s, n_s), and n_s varies from 4 to 16. Four networks with different mean reliabilities are evaluated. Data redundancy is defined as the increase in size



(a) (b)

Figure 4.13: (a) Performance comparison of unsplit storage parameter, i.e. $(k_s, n_s) = (k_f, n_f)$, and splitting storage parameter. (b) Compare the energy consumption of random reallocation and min-cost flow reallocation.

after a data object is encoded by erasure coding. It is obvious that both the system reliability and the data redundancy increase with n_s. Data redundancy is essentially the price of higher system reliability. Assume the reliability requirement is 0.9. The network with 0.9 mean reliability needs only 50% redundancy while the network with 0.5 mean reliability needs 200% redundancy in order to achieve the same reliability requirement.

The purpose of having split storage and fragment parameters is to provide more flexibility and reduce the maintenance cost. If a single fragment parameter can adapt to several storage parameters, the chance of expensive data re-encoding and re-distribution will be lower. The maintenance cost here includes the energy consumption for data reallocation, data retrieval, data encoding, and data redistribution. We estimate the energy consumption based on the energy profile measured from a real smartphone (HTC Evo). Figure 4.13a shows that split scheme reduces the overall maintenance energy by around 50%. Figure 4.13b considers only the "data reallocation energy" from the total maintenance energy and compares the performance

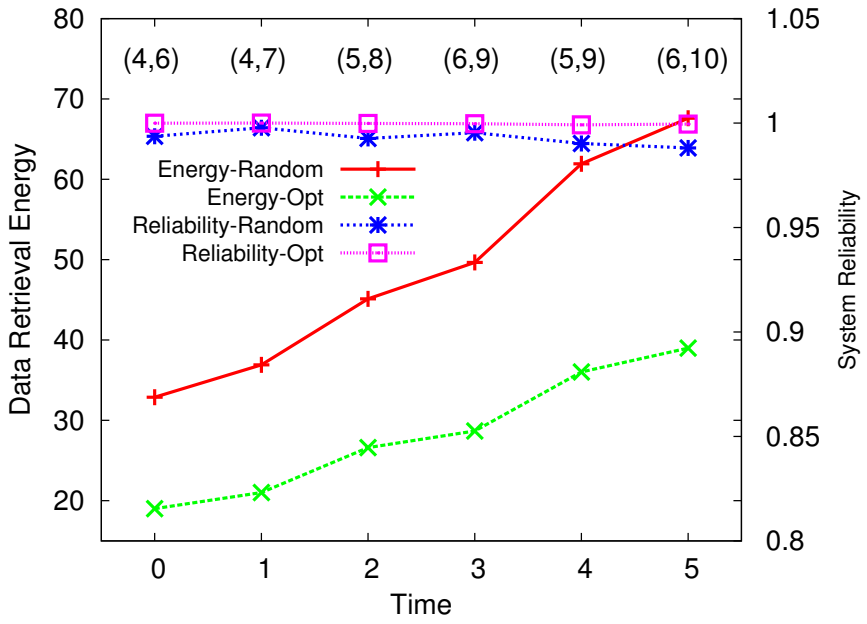


Figure 4.14: The effectiveness of the maintenance algorithm. The data retrieval energy increases as the time elapses, but the system reliability remains almost constant due to the updated storage parameter.

between *minimum-cost flow reallocation* and *Random* reallocation. *Random* scheme randomly pairs up the new storage node and the old storage node when reallocating fragments. An interesting observation is that network size 10 and 14 have relatively high reallocation energy. The reason is that these two networks have higher data redundancy and more topology changes; when the network size is small, the ratio n_s/k_s is usually higher in order to achieve the system reliability requirement, causing higher redundancy ratio n_f/k_f. More redundant data thus increases the overall reallocation cost.

In our network trace, each node's reliability gradually decreases as time elapsed. Upon a topology change event, the maintenance algorithm may update the storage parameter or data allocation to maintain the system optimality. Figure 4.14 compares the data retrieval energy of our solution and *Random* solution. In *Random*,

n_s storage nodes are randomly selected and clients access data from their closest k_s storage nodes. The x-axis represents the time the maintenance algorithm is triggered and the updated storage parameter is shown at the top of the plot; the left and the right y-axis represent the overall data retrieval energy and system reliability at each time point respectively. The energy efficiency of both schemes degrade as the time elapses because routes between clients and storage nodes become less reliable and the average retrieval time becomes longer. One interesting observation that the system reliability remains almost constant in both schemes. This is because the updated storage parameter keeps the system reliability high regardless of the data allocation.

4.5.3.2 Dartmouth Network Trace

The Dartmouth Outdoor Dataset [37] includes the GPS locations and routing tables of 41 laptops moving in a $255 \times 365 \text{m}^2$ athletic field for 1.5 hours. During the experiment, 7 laptops failed to generate any data, and another 8 laptops became inactive after 30-40 minutes. Less than 26 nodes completed the entire experiment, as most of the laptops reached the end of battery life. This realistic trace serves as a good model for evaluating our solutions.

A set of files are created at the beginning of the experiment, and each file is split into $4MB$ data blocks. We then evaluate the average maintenance energy (reallocation, retrieval, and redistribution) of each data block. Figure 4.15 shows the maintenance energy at three different times.

Similar to Figure 4.13, we compare the maintenance cost of single and of split storage parameter solutions. We then compare the energy consumption of the min-cost flow reallocation and *Random* reallocation. At the beginning of the experiment, all nodes have high reliability and the network has good connectivity. As a result, very low data redundancy is necessary to achieve the reliability requirement and the

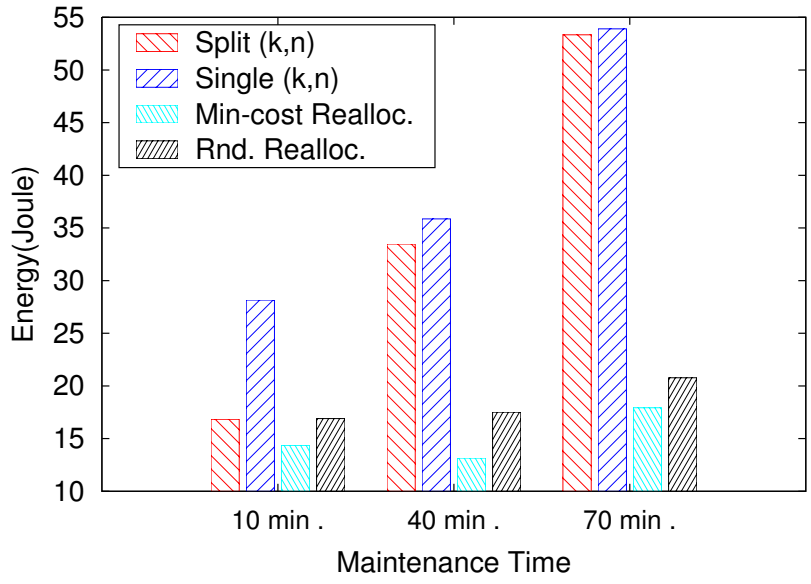
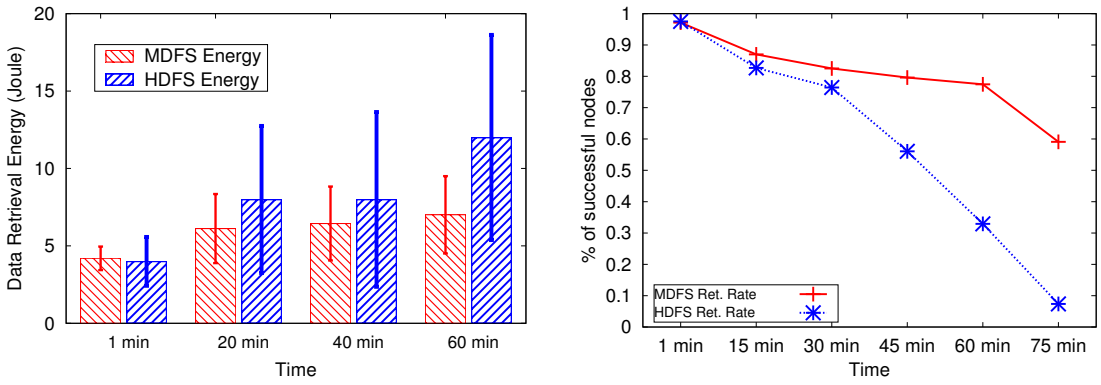


Figure 4.15: Maintenance energy of Dartmouth dataset at different times.

overall maintenance cost is low. As the time elapses, nodes move towards wider area, some nodes become inactive, and most nodes' reliability drop. These cause higher data redundancy, unstable connectivity, and thus higher maintenance energy. When data re-distribution is inevitable, e.g., at 40 and 70 minutes, the performance of split storage parameter becomes close to the single storage parameter. Overall, the min-cost flow fragment reallocation achieves 20-50% lower energy than random fragment reallocation.

Figure 4.16 compares the performance of our storage system and the *Hadoop Distributed File System* (HDFS) in Dartmouth Network Trace. In our storage system (MDFS), a file is encoded, distributed, and maintained based on the framework presented in Sec. 4.3. HDFS replicates each data block 3 times to 3 different nodes. Figure 4.16a evaluates the energy consumption for each node to retrieve the file at different times. The y-axis is the *mean retrieval energy* of all nodes in the network,



(a) (b)

Figure 4.16: Comparison of our storage system (MDFS) with HDFS. (a) Mean data retrieval energy from each node in the network at different times. (b) Data retrieval rate – the percentage of nodes that can recover the data.

and the error bar indicates the standard deviation of data retrieval energy among all nodes. It can be seen that our storage framework achieves better energy efficiency, and the variance of data retrieval energy among clients nodes is also much lower than HDFS. This is because our framework allocates data considering the energy efficiency and the data fragments are more evenly distributed in the network. Figure 4.16b estimates the number of nodes that can successfully retrieve the stored data at different times. The y-axis is the percentage of nodes that can successfully retrieve the stored data. In HDFS, the data becomes unavailable if all 3 nodes fail or if a client node can not access any of the 3 storage nodes. Because our framework continuously maintains the stored data, client nodes in MDFS have much higher probability to successfully recover a file than HDFS.

4.5.4 Caching Algorithm Evaluation

Figure 4.17 depicts the performance metrics of running *NC*, *CC*, *DC* and *IC* algorithms with varying number of nodes in the network. An immediate observa-

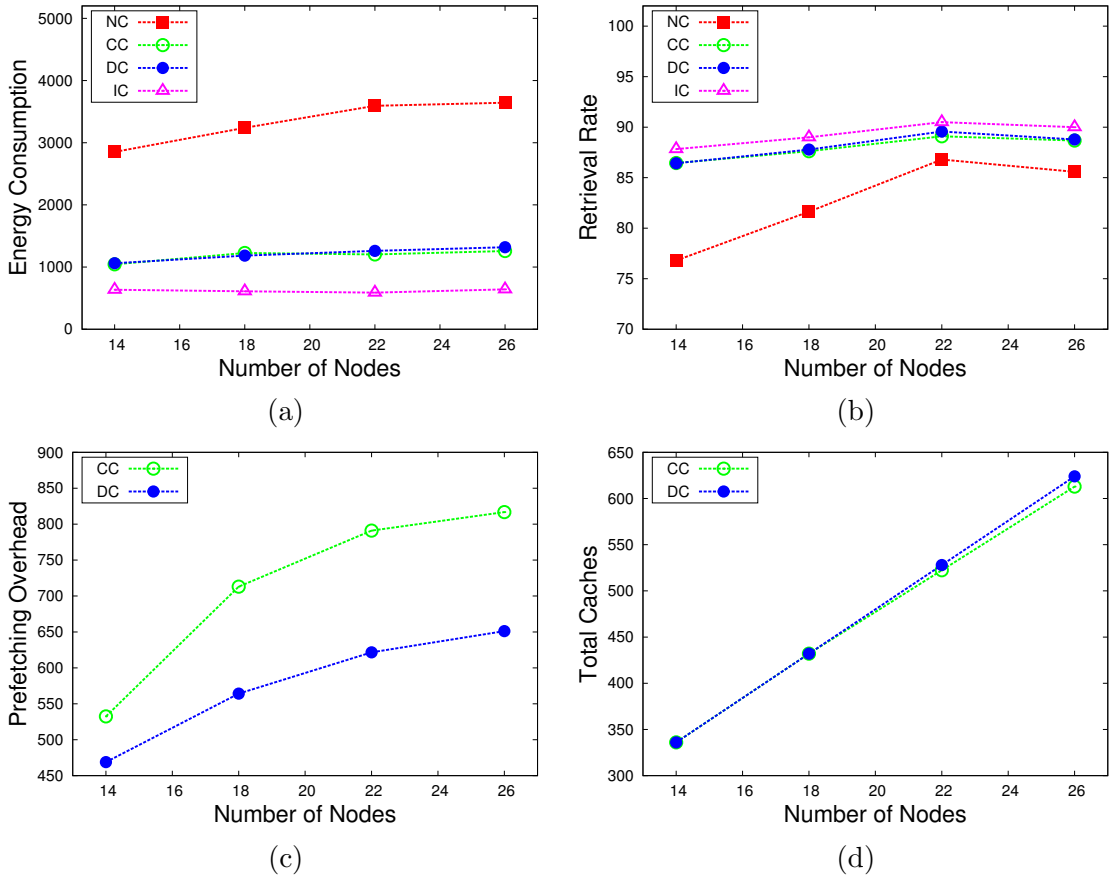


Figure 4.17: Effect of nodes number on (a) Energy Consumption; (b) Retrieval Rate; (c) Prefetching Overhead; (d) Total Caches. The test scenario is based on 12 files, 600 requests, and the buffer size is set to be holding up to 24 fragments.

tion is that the performance of *NC* is highly subjective to the number of nodes and their movements. For energy consumption, more nodes usually implies more hops, therefore more energy consumption. When there are not enough nodes in the area and the network density is relatively low (below 22 in our simulation), adding more nodes to the network only forms paths with more hops, thus slightly increasing the energy consumption for retrieving the data. After the network density has reached a "saturated" point (22 in our simulation), the chance of nodes finding better or shorter paths to cache agents increases and thus the total energy consumption starts

to decrease. As for the retrieval rate, higher number of nodes generally provides more candidate cache agents and thus improves the data availability. Another interesting observation is that with caching enabled, the fluctuation of both energy and retrieval rate reduces because the file requests become more likely to be fulfilled by the nearby cache agents rather than the service centers farther away. As there are always cached fragments somewhere in the network, the failures of the service centers do not significantly bring down the performance of the system. After examining *CC*, *DC* and *IC*, it is clear that our proposed *DC* is much more effective under all circumstances.

We consider network size up to 26 nodes in this simulation because: 1) a mobile cloud of 26 nodes is realistic in a real-world environment; 2) in order to compare the performance between the centralized solution (CC) and the distributed solution (DC), 26 is close to the largest network size that the CC problem can be solved effectively by the Matlab optimization toolbox; 3) given the fixed network dimensions (400 m^2) and the file access pattern, the energy consumption and the data retrieval rate already reach steady state at 22 nodes. Thus, adding more nodes does not affect the result much.

5. DISTRIBUTED DATA PROCESSING*

In this section, we present the k-out-of-n data processing framework that is built on top of the k-out-of-n data storage framework in section 4. Note that part of this section is reprinted from the previously published papers.* The objective is to provide applications a fault-tolerant and energy-efficient data processing service that can process the files stored in our distributed storage. We again assume a homogeneous mobile cloud in which all nodes have identical hardware capability and form a mobile ad-hoc network. The more complicated heterogeneous network will be studied in the next section. Similar to the data allocation problem, a set of n_p processor nodes are selected for processing the stored data and all tasks can be completed as long as k_p or more of the n_p processor nodes can finish the assigned tasks. The tasks are assigned to and scheduled on processor nodes considering the data transfer energy and node's reliability. The parameters k_p and n_p determine the degree of reliability of a processing job (different from the (k, n) used for data storage). System administrator may select these parameters depending on the reliability requirement. The framework also provides an adaptive *Parameter Selection* algorithm that adjusts the parameters based on the reliability requirement and the network condition.

In the rest of the section, we first formulate the k-out-of-n data processing problem. The task allocation problem and the task scheduling problem are then explained

*Reprinted with permission from "Energy-Efficient, Fault-Tolerant Data Storage & Processing in Dynamic Networks" by Chien-An Chen, Myounggyu Won, Radu Stoleru, and Geoffrey Xie, *International Symposium on Mobile Ad Hoc Networking and Computing, 2013*, Copyright © 2013, Association for Computing Machinery, Inc.
Reprinted with permission from "Hadoop MapReduce for Tactical Clouds" by Johnu George, Chien-An Chen, Radu Stoleru, Geoffrey Xie, Tamim Sookoor, and David Bruno, *International Conference on Cloud Networking, 2014*, Copyright © 2014, IEEE.
Reprinted with permission from "Energy-Efficient, Fault-Tolerant Data Storage & Processing in Mobile Cloud" by Chien-An Chen, Myounggyu Won, Radu Stoleru, Geoffrey Xie, *IEEE Transactions on Cloud Computing, 2015*, Copyright © 2015, IEEE.

in detail. The 2-level Tabu-Search solver is present to solve these optimization problems. After understanding our k-out-of-n data processing framework, we explain how the Hadoop MapReduce component is integrated into our framework. The section finishes with the evaluation results.

5.1 Formulation of the k-out-of-n Data Processing Problem

We consider a dynamic network of N nodes denoted by a set $V = \{v_1, v_2, ..., v_N\}$. The problem formulation shares the same settings and notations used in section 4.1. In addition, here we define several new notations used for modeling the task scheduling. *Scheduling Matrix* S is an $L \times N \times M$ matrix where element $S_{lij} = 1$ indicates that task j is scheduled at time l on node i; otherwise, $S_{lij} = 0$. l is a relative time referenced to the starting time of a job. Since all tasks are instantiated from the same function, we assume they spend approximately the same CPU time on any node. Given the terms and notations, we are ready to formally describe the k-out-of-n data processing problems.

The objective of this problem is to find n_p nodes in V as processor nodes such that energy consumption for processing a job of M tasks is minimized. In addition, it ensures that the job can be completed as long as k_p or more processors nodes finish the assigned tasks. Before a processor node starts processing a data object, assuming the correctness of erasure coding, it first needs to retrieve and decode k data fragments because nodes can only process the decoded plain data object, but not the encoded data fragment. Because the network is homogeneous, each task consumes the same CPU time and CPU energy on any processor node. However, a task still has different data retrieval energy on different processor nodes. The data retrieval energy depends on the data fragments location and network topology. We will model a more general case in which both CPU energy and communication energy

are considered in the next section.

Before formulating the problem, we define some functions: (1) $f_1(i)$ returns 1 if node i in S has at least one task; otherwise, it returns 0; (2) $f_2(j)$ returns the number of instances of task j in S; and (3) $f_3(z, j)$ returns the transmission cost of task j when it is scheduled for the z^{th} time. We now formulate the k-out-of-n data processing problem as shown in Equation 5.1 - 5.6.

The objective function (Eq. 5.1) minimizes the total transmission cost for all processor nodes to retrieve their tasks. l represents the time slot of executing a task; i is the index of nodes in the network; j is the index of the task of a job. We note here that T^r, the *Data Retrieval Time Matrix*, is a $N \times M$ matrix, where the element T^r_{ij} corresponds to the estimated time for node i to retrieve task j. T^r is computed by summing the transmission time (in terms of ETT available in D) from node i to its k closest storage nodes of the task.

$$\text{minimize} \quad \sum_{l=1}^{L} \sum_{i=1}^{N} \sum_{j=1}^{M} S_{lij} T^r_{ij} \tag{5.1}$$

$$\text{Subject to:} \quad \sum_{i}^{N} f_1(i) = n \tag{5.2}$$

$$f_2(j) = n_p - k_p + 1 \ \forall j \tag{5.3}$$

$$\sum_{l=1}^{L} S_{lij} \leq 1 \ \forall i, j \tag{5.4}$$

$$\sum_{i=1}^{N} S_{lij} \leq 1 \ \forall l, j \tag{5.5}$$

$$\sum_{j=1}^{M} f_3(z_1, j) \leq \sum_{j=1}^{M} f_3(z_2, j) \ \forall z_1 \leq z_2 \tag{5.6}$$

The first constraint (Eq. 5.2) ensures that n nodes in the network are selected as

processor nodes. The second constraint (Eq. 5.3) indicates that each task is replicated n_p-k_p+1 times in the schedule such that any subset of k processor nodes must contain at least one instance of each task. The third constraint (Eq. 5.4) states that each task is replicated at most once to each processor node. The fourth constraint (Eq. 5.5) ensures that no duplicate instances of a task execute at the same time on different nodes. The fifth constraint (Eq. 5.6) ensures that a set of all tasks completed at earlier time should consume lower energy than a set of all tasks completed at later time. In other words, if no processor node fails and each task completes at the earliest possible time, these tasks should consume the least energy.

5.2 Energy-Efficient and Fault-Tolerant Data Processing

The k-out-of-n data processing problem is solved in two stages – *Task Allocation* and *Task Scheduling*. In the Task Allocation stage, n_p nodes are selected as *processor nodes*; each processor node is assigned one or more tasks; each task is replicated to n_p-k_p+1 different processor nodes. An example is shown in Figure 5.1a. However, not all instances of a task will be executed – once an instance of the task completes, all other instances will be canceled. The task allocation can be formulated as an ILP as shown in Equations 5.7 - 5.11. In the formulation, $\overline{R_{ij}}$ is a $N \times M$ matrix which predefines the relationship between processor nodes and tasks; each element $\overline{R_{ij}}$ is a binary variable indicating whether task j is assigned to processor node i. $\overline{X}$ is a binary vector containing processor nodes, i.e., $X_i = 1$ indicates that v_i is a processor node. The objective function minimizes the transmission time for n processor nodes to retrieve all their tasks. The first constraint (Eq. 5.8) indicates that n of the N nodes will be selected as processor nodes. The second constraint (Eq. 5.9) replicates each task to (n_p-k_p+1) different processor nodes. The third constraint (Eq. 5.10) ensures that the j^{th} column of R can have a non-zero element if only if X_j is 1; and

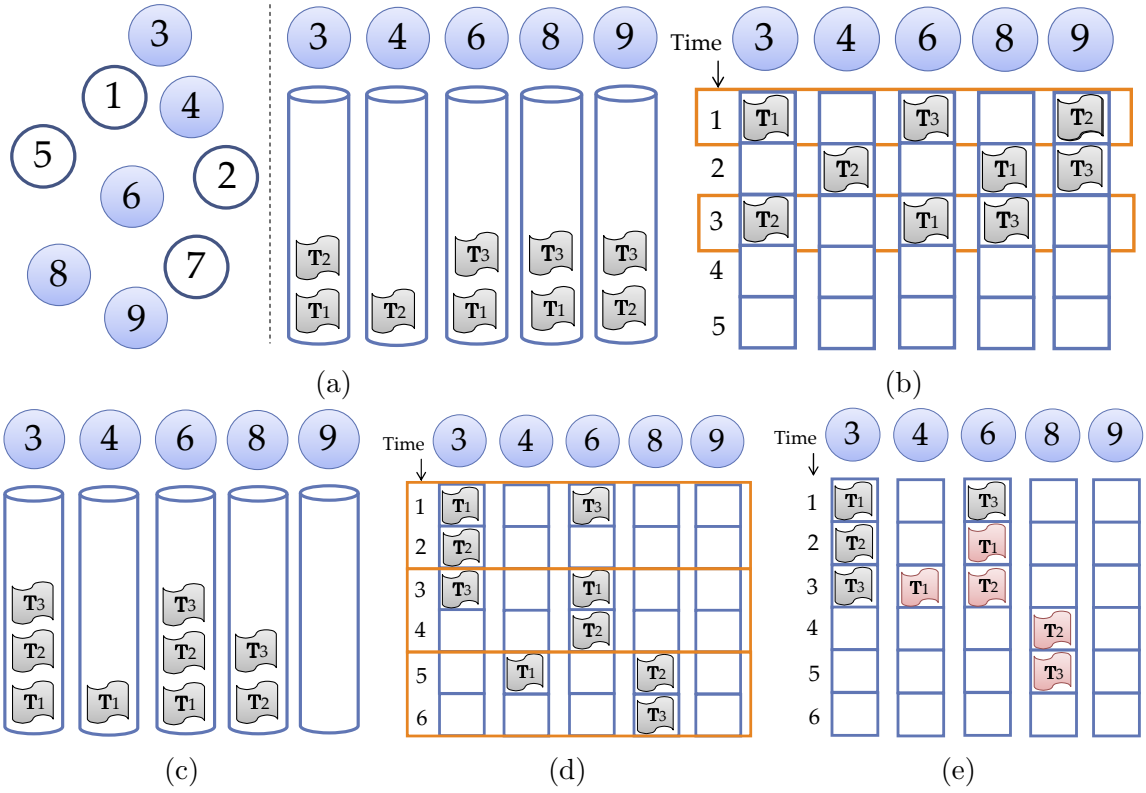


Figure 5.1: The k-out-of-n data processing example with $N = 9, n_p = 5, k_p = 3$. (a) and (c) are two different task allocations and (b) and (d) are their tasks scheduling respectively. In both cases, node $3, 4, 6, 8, 9$ are selected as processor nodes and each task is replicated to 3 different processor nodes. (e) shows that shifting tasks reduce the job completion time from 6 to 5.

the constraints (Eq. 5.11) are binary requirements for the decision variables.

Once processor nodes are determined, we proceed to the Task Scheduling stage. In this stage, the tasks assigned to each processor node are scheduled such that the energy and time for finishing at least M distinct tasks is minimized, meaning that we try to shorten the job completion time while minimizing the overall energy consumption. The problem is solved in three steps. First, we find the minimal energy for executing M distinct tasks in $\overline{R_{ij}}$. Second, we find a schedule with the minimal energy that has the shortest completion time. As shown in Figure 5.1b, tasks 1 to 3

$$\overline{R_{opt}} = \arg\min_{\overline{R}} \sum_{i=1}^{N} \sum_{j=1}^{M} T_{ij}^{r} \overline{R_{ij}} \tag{5.7}$$

$$\text{Subject to:} \quad \sum_{i=1}^{N} \overline{X_i} = n \tag{5.8}$$

$$\sum_{i=1}^{N} \overline{R_{ij}} = n_p - k_p + 1 \ \forall j \tag{5.9}$$

$$\overline{X_i} - \overline{R_{ij}} \geq 0 \ \forall i \tag{5.10}$$

$$\overline{X_j} \text{ and } \overline{R_{ij}} \in \{0, 1\} \ \forall i, j \tag{5.11}$$

are scheduled on different nodes at time slot 1; however, it is also possible that tasks 1 through 3 are allocated on the same node, but are scheduled in different time slots, as shown in Figure 5.1c and 5.1d. These two steps are repeated n-k+1 times and M distinct tasks are scheduled upon each iteration. The third step is to shift tasks to earlier time slots. A task can be moved to an earlier time slot as long as no duplicate task is running at the same time, e.g., in Figure 5.1d, task 1 on node 6 can be safely moved to time slot 2 because there is no task 1 scheduled at time slot 2.

The ILP problem shown in Equations 5.12 - 5.15 finds M unique tasks from $\overline{R_{ij}}$ that have the minimal transmission cost. The decision variable W is an $N \times M$ matrix where $\overline{R_{ij}} = 1$ indicates that task j is selected to be executed on processor node i. The first constraint (Eq. 5.13) ensures that each task is scheduled exactly one time. The second constraint (Eq. 5.14) indicates that W_{ij} can be set only if task j is allocated to node i in $\overline{R_{ij}}$. The last constraint (Eq. 5.15) is a binary requirement for decision matrix W.

Once the minimal energy for executing M tasks is found, among all possible schedules satisfying the minimal energy budget, we are interested in the one that

$$W_E = \arg\min_{W} \sum_{i=1}^{N} \sum_{j=1}^{M} T_{ij} \overline{R_{ij}} W_{ij} \tag{5.12}$$

$$\text{Subject to:} \quad \sum_{i=1}^{N} W_{ij} = 1 \ \forall j \tag{5.13}$$

$$\overline{R_{ij}} - W_{ij} \geq 0 \ \forall i, j \tag{5.14}$$

$$W_{ij} \in \{0, 1\} \ \forall i, j \tag{5.15}$$

$$\underset{Y}{\text{minimize Y}} \tag{5.16}$$

$$\text{Subject to:} \sum_{i=1}^{N} \sum_{j=1}^{M} T_{ij} \times \overline{R_{ij}} \times \overline{W_{ij}} \leq E_{min} \tag{5.17}$$

$$\sum_{i=1}^{N} \overline{W_{ij}} = 1 \quad \forall j \tag{5.18}$$

$$\overline{R_{ij}} - \overline{W_{ij}} \geq 0 \ \forall i, j \tag{5.19}$$

$$Y - \sum_{j=1}^{M} \overline{W_{ij}} \geq 0 \quad \forall i \tag{5.20}$$

$$\overline{W_{ij}} \in \{0, 1\} \quad \forall i, j \tag{5.21}$$

has the minimal completion time. Therefore, the minimal energy found previously, $E_{min} = \sum_{i=1}^{N} \sum_{j=1}^{M} T_{ij} \overline{R_{ij}} W_E$, is used as the "upper bound" for searching a task schedule.

If we define $L_i = \sum_{j=1}^{M} \overline{W_{ij}}$ as the number of tasks assigned to node i, L_i indicates the completion time of node i. Then, our objective becomes to *minimize the largest number of tasks in one node*, written as $\min \left\{ \max_{i \in [1,N]} \{L_i\} \right\}$. To solve this *min-max* problem, we formulate the problem as shown in Equations 21 - 26.

The objective function minimizes integer variable Y, which is the largest number

Algorithm 3: Schedule Re-arrangement

```
L=last time slot in the schedule
for time t = 2 → L do
    for each scheduled task J in time t do
        n ← processor node of task J
        while n is idle at t − 1 AND
        J is NOT scheduled on any node at t − 1 do
            Move J from t to t − 1
            t = t − 1
        end
    end
end
```

of tasks on one node. $\overline{W_{ij}}$ is a decision variable similar to W_{ij} defined previously. The first constraint (Eq. 5.17) ensures that the schedule cannot consume more energy that the E_{min} calculated previously. The second constraint (Eq. 5.18) schedules each task exactly once. The third constraint (Eq. 5.20) forces Y to be the largest number of tasks on one node. The last constraint (Eq. 5.21) is a binary requirement for decision matrix $\overline{W}$. Once tasks are scheduled, we then rearrange tasks – tasks are moved to earlier time slots as long as there is free time slot and no same task is executed on other node simultaneously. Algorithm 3 depicts the procedure. Note that the k-out-of-n data processing framework guarantees that k_p or more functional processor nodes can complete all M tasks of a job. In general, *it is also possible* that a subset of processor nodes of size less than k_p complete all M tasks.

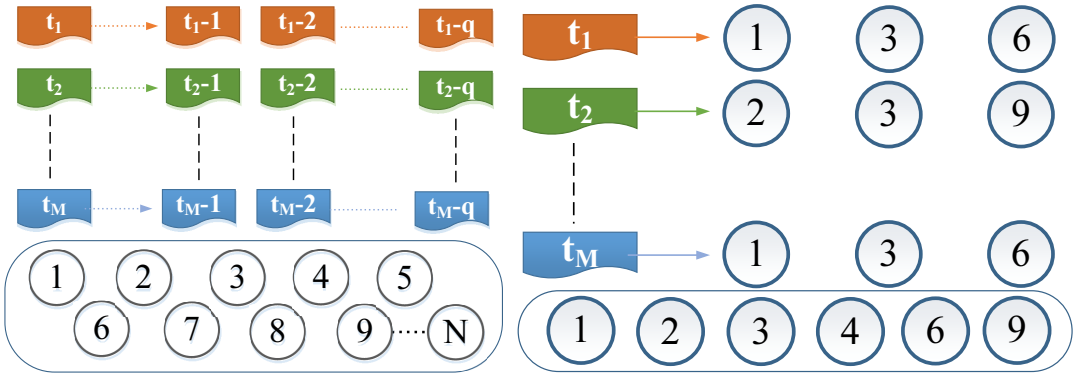
5.3 Deadline-Compliant Energy-Aware Task Scheduling

We now study a different and more complicated problem formulation where a soft job deadline is added as a constraint and the (k_p, n_p) parameter is selected by the problem solver based on the reliability requirement. The rationale behind the "soft deadline" is that we ensure "expected job makespan" meets a given deadline.

Although meeting a hard deadline is also possible, considering that the type of job is not extremely time-critical, a soft deadline allows a more energy-efficient solution (task schedule). Similar tradeoff of soft or hard deadline was studied in prior works [74, 105]. Clients submit jobs to process the stored files, e.g., object recognition or statistical analysis. A job is partitioned into multiple tasks, where each task processes one data block. The objective is to assign and schedule these tasks to a set of n_p selected processor nodes such that the job is *fault-tolerant*, *energy-efficient*, and *completes within a deadline*. Similar to storage parameter (k_s, n_s) in section 4.3.1, a *processor parameter* (k_p, n_p) determines the reliability of a job; a job succeeds if k_p or more processors complete their assigned tasks. To ensure the k-out-of-n reliability, each task is replicated to (n_p-k_p+1) *task instances* and each task instance is assigned to a processor node. In this manner, at least one instance of a task will execute as long as no more than n_p-k_p processor nodes fail. (k_p, n_p) is selected by the same procedure as selecting (k_s, n_s); given the reliability requirement of a job, a set of *Candidate Processor Parameters* is found by table lookup (Table 4.1), and a single processor parameter in this set is then selected by solving the data processing optimization problem. The solution of the problem selects a single processor parameter (k_p, n_p), selects n_P processor nodes, and creates a schedule $\mathbb{S}$. The following subsections describe how the problem is formulated and solved.

5.3.1 Energy-Efficient and Load-Balanced Task Allocation

Each task involves *retrieving* and *processing* a data block. Assuming that the network is homogeneous and a task requires the same CPU processing time on any node, we minimize only the *transmission time* (thus, energy) for retrieving tasks. Although each task is replicated multiple times, most task instances are simply standby and will not retrieve or process data objects – once an instance of the task completes, all



(a) (b)

Figure 5.2: (a) The network has N nodes and each task has q replications. t_1-1 is the first instance of task 1 and t_M-q is the last instance of task M. (b) (k_p,n_p) is selected as (4,6). Nodes 1, 2, 3, 4, 6, and 9 are selected as processor nodes by the task scheduling algorithm and each task is assigned to (n_p-k_p+1=3) different processor nodes.

other instances of the task are canceled. Figure 5.2a and 5.2b illustrate an example of task allocation where M tasks are to be processed and each task has q replicated instances ($q = n_p - k_p + 1$).

Equations 5.22-5.31 formulate the data processing optimization problem that finds a minimal energy solution constrained on a soft job deadline. Assume the network has N nodes and the job has M tasks. X and $\mathbb{S}$ are the decision variables. X is a binary vector containing the selected processor nodes, i.e., X_i=1 indicates that node i is selected as a processor node. *Schedule matrix* $\mathbb{S}$ is a $N \times M \times N$ matrix which defines how the task instances are allocated and scheduled, i.e., $\mathbb{S}_{ijl}$=1 indicates that task j is assigned to processor node i at *level* l. A *level* is a time slot in which each task is scheduled once. Each level is represented by a $(N \times M)$ matrix (the first two dimensions of $\mathbb{S}$) where each row is a processor and each column is a task.

The objective function (Equation 5.22) minimizes the *Expected Retrieval Time*

$$\mathbb{S}_{opt} = \arg\min_{\mathbb{S}} ERet(\mathbb{S}) \tag{5.22}$$

$$\text{Subject to:} \quad n_p \leq n_{max} \tag{5.23}$$

$$\sum_{i=1}^{N}\sum_{l=1}^{N} \mathbb{S}_{ijl} = q \ \forall j \tag{5.24}$$

$$\sum_{i=1}^{N} \mathbb{S}_{ijl} = 1 \ \forall j, \forall l \leq q \tag{5.25}$$

$$\sum_{l=1}^{N} \mathbb{S}_{ijl} \leq 1 \ \forall i, \forall l \leq q \tag{5.26}$$

$$\sum_{i=1}^{N}\sum_{j=1}^{M}\sum_{l=q+1}^{N} \mathbb{S}_{ijl} = 0 \tag{5.27}$$

$$EMakespan(\mathbb{S}) \leq deadline \tag{5.28}$$

$$MX_i \geq \sum_{j=1}^{M} \mathbb{S}_{ijl} \geq X_i$$
$$\forall i, \forall l \leq q \tag{5.29}$$

$$n_p = \sum_{i=1}^{N} X_i \ ; k_p = f_k(n_p);$$
$$q = n_p - k_p + 1 \tag{5.30}$$

$$X_i \text{ and } \mathbb{S}_{ijl} \in \{0, 1\} \ \forall i, j, l \tag{5.31}$$

of the job. Function $ERet(\mathbb{S})$ takes a schedule matrix $\mathbb{S}$ as the input and evaluates the total expected time for retrieving the job; how $ERet(\mathbb{S})$ is evaluated will be described in section 5.3.3. Constraint 1 (Equation 5.23) allows at most n_{max} nodes to be selected as the processor nodes. Constraint 2 (Equation 5.24) replicates each task q times ($q = n_p - k_p + 1$). Constraint 3 (Equation 5.25) ensures that each task is scheduled exactly once in each level. Constraint 4 (Equation 5.27) indicates that the tasks are only scheduled in the first $n_p - k_p + 1$ levels, and level $n_p - k_p + 2$

to level N are all zeros. Note that n_p and k_p are unknown variables. Constraint 5 (Equation 5.28) ensures that the expected job completion time is within the deadline. Function $EMakespan(\mathbb{S})$ returns the *expected job makespan* of the job; how it is evaluated will be explained in section 5.3.3. Constraint 6 (Equation 5.29) ensures that $X_i = 1$ if and only if there is at least one task assigned to processor i in each level. Constraint 7 (Equation 5.30) defines the helping variables n_p and k_p; n_p is the number of the selected processor nodes and it is equal to the number of 1 appearing in X; function $f_k(n_p)$ returns the corresponding k_p of the given n_p by checking the candidate processor parameters.

5.3.2 Fault-Tolerant and Minimal Expected Makespan Task Allocation

The data processing problem can be slightly modified to find either the *minimal-energy* solution or *minimal job makespan* solution. If the deadline in Equation 5.28 is set to infinity, the solution achieves minimal energy. If we change the objective function (Equation 5.22) to $EMakespan(\mathbb{S})$ and remove the deadline constraint, the solution achieves minimal expected makespan. Specifically, the new objective function becomes $\mathbb{S}_{opt} = \arg\min_{\mathbb{S}} EMakespan(\mathbb{S})$.

5.3.3 Computation of Expected Retrieval Time and Expected Job Makespan

The rationale behind the "expected time" is that a task may execute on one of the (n_p-k_p+1) assigned processor nodes and each task instance has certain probability to complete the job depending on the scheduled time and the reliability of its processor node. Before explaining the expected retrieval time, we first explain how the tasks are scheduled in $\mathbb{S}$. The tasks are scheduled according to the following rules: 1) each processor node may have multiple task instances, but no more than one instance of a same task can be scheduled on a single processor node; 2) each task should be scheduled exactly once in each level. Task instances in lower levels have higher

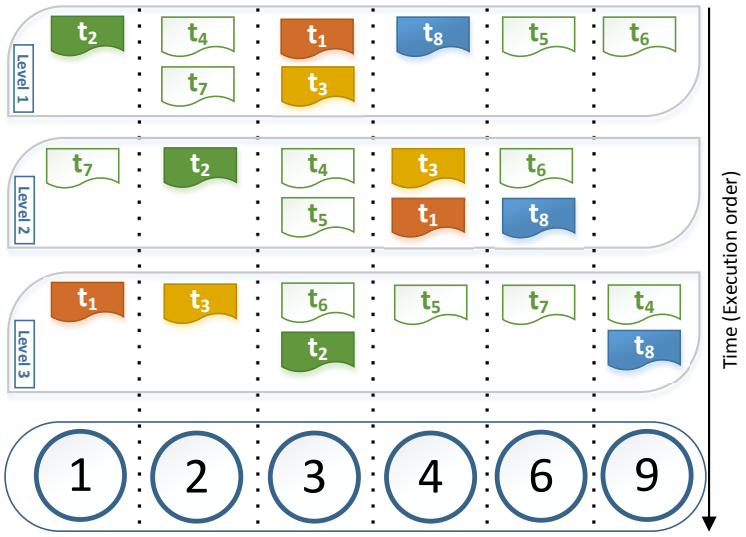


Figure 5.3: An example of task schedule $\mathbb{S}$. The job has $M = 8$ tasks, and each task has 3 replicated instances. Schedule $\mathbb{S}$ has 3 levels and each level contains 8 unique tasks.

priority to task instances in higher levels, and a task instance in a higher level may execute only if all instances of the same task in lower levels have failed. As shown in Figure 5.3, a task instance in level 2 may start only if the processor node of the same task fails in level 1; a task instance in level 3 may start only if the processor nodes of the same task fail in both level 1 and level 2. We now define few notations: *ECT(*$\mathbb{S}$*, t)* and *ERT(*$\mathbb{S}$*, t)* are the *Expected Completion Time* and the *Expected Retrieval Time* of task t in schedule $\mathbb{S}$ respectively. R_i and $\overline{R_i}$ represent the reliability and the failure probability of each processor node, where i is the node index.

Using task 1 in Figure 5.3 as an example, task 1 has three instances (t_1^1, t_1^2 and t_1^3), and they are assigned to node 3, node 4, and node 1 in level 1, level 2, and level 3 respectively. The probability that task 1 finishes at level 1 is R_3, the reliability of node 3. The probability that task 1 finishes at level 2 is $\overline{R_3}R_4$, the probability that node 3 fails and node 4 successfully completes the assigned tasks. The probability

that task 1 finishes at level 3 is $\overline{R_3 R_4} R_1$, the probability that both node 3 and node 4 fail while node 1 succeeds. We denote *RetT(t)* as the retrieval time of the task instance t, which is evaluated based on the size and the locations of the data fragments.

Assume that instances of task 1 have the lowest priority and always execute after other tasks in the same processor. The completion time of task 1 in node 3 at level 1 is $RetT(t_1^1)+RetT(t_3^1)$, the completion time of task 1 in node 4 at level 2 is $RetT(t_3^2)+RetT(t_1^2)$, and the completion time of task 1 in node 1 at level 3 is $RetT(t_1^3)$. Note that when considering the completion time of a task in one level, the retrieval time of all other tasks on the same processor need to be considered. The time wasted by the failed task instances also need to be considered. Finally, the *expected completion time* of task 1 is:

$$
\begin{aligned}
&\mathbf{ECT}(\mathbb{S}, \mathbf{t_1}) = R_3(RetT(t_1 - 3) + RetT(t_3 - 1)) \\
&+ \overline{R_3} R_4(RetT(t_1 - 3) + RetT(t_3 - 1) \\
&+ RetT(t_3 - 2) + RetT(t_1 - 2)) \\
&+ \overline{R_3 R_4} R_1(RetT(t_1 - 3) + RetT(t_3 - 1) + \\
&\quad RetT(t_3 - 2) + RetT(t_1 - 2) + RetT(t_1 - 1))
\end{aligned}
$$

In this manner, we compute the expected completion time of all tasks. The *Expected Job Makespan* is the longest expected completion time among all tasks, i.e., $\mathbf{EMakespan}(\mathbb{S}) = \max_{\mathbf{i}} \mathbf{ECT}(\mathbb{S}, \mathbf{t_i})$.

The *Expected Retrieval Time* is computed similarly, except that we do not need to consider the retrieval time spent by other tasks in the same processor. E.g.,

when calculating the retrieval time of task 1 in level 2, only *RetT(t_1-2)* needs to be considered. The *expected retrieval time* of task 1 in Figure 5.3 is:

$$\mathbf{ERC}(\mathbb{S}, \mathbf{t_1}) = R_3(RetC(t_1^1)) + \overline{R_3}R_4(RetC(t_1^1) + RetC(t_1^2))$$
$$+ \overline{R_3R_4}(RetC(t_1^1) + RetC(t_1^2) + RetC(t_1^3))$$

In this manner, we compute the expected retrieval time of each task, and the *Expected Retrieval Time* of the job is the summation of the expected retrieval time of all tasks, i.e., $\mathbf{ERet}(\mathbb{S}) = \sum_{\mathbf{i}} \mathbf{ERT}(\mathbb{S}, \mathbf{t_i})$.

5.4 Tabu Search Solver

As the conventional optimization toolbox cannot efficiently solve our integer constraint programming problem, we adopt *Tabu Search*(TS) heuristic for solving our optimization problems. Tabu search was first proposed by Glover in 1986 to solve various combinatorial problem that appeared in operations research [36], and was later shown to perform well in solving *facility location* problem and *job shop scheduling* problem in many literatures [35] [17]. In this section, we briefly describe how the k-out-of-n data processing problem is solved by tabu search heuristic. We skip the k-out-of-n data allocation problem in the previous section as it is a simpler version of the data processing problem and the solver works similarly.

We proposed a 2-stages TS search algorithm in which two TS procedures cooperate to explore two decision variables, X and $\mathbb{S}$. The first TS procedure explores the number of processor nodes (n_p) in the candidate processor parameters and the possible subset of n_p processor nodes among all N nodes in the network; the second TS procedure takes the selected n_p processor node in the first procedure and

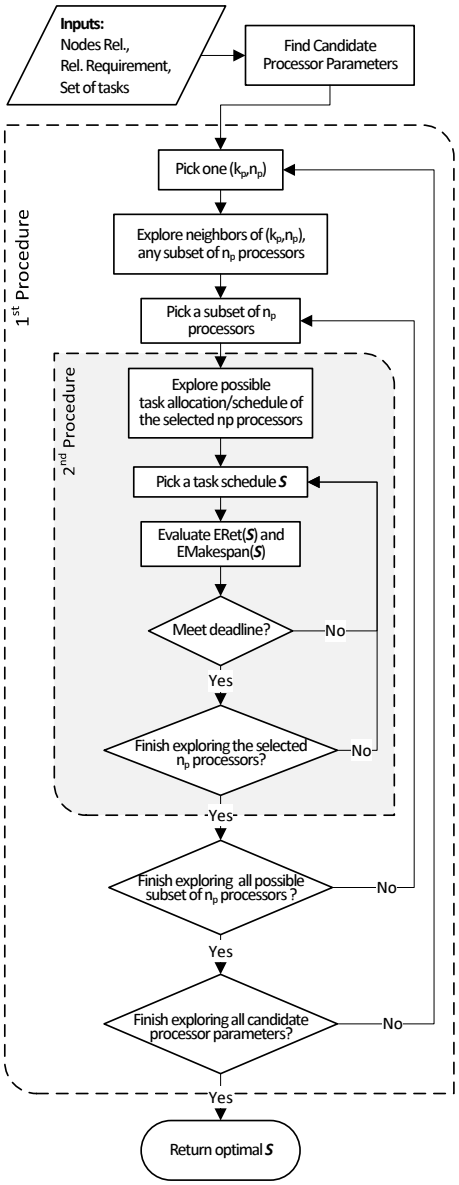


Figure 5.4: The flow chart describes how the 2-stages Tabu Search solves the k-out-of-n data processing problem. The first TS procedure explores the number of processor nodes n_p and the possible subset of n_p nodes; the second TS procedure explores the possible task allocation and task scheduling.

explores the possible task allocation and task scheduling; when the lowest energy schedule that satisfies the deadline constraint is found, the second procedure returns

the schedule $\mathbb{S}$ back to the first procedure. This cycle repeats until the stopping criteria is reached. The stopping criteria is either all the feasible solutions have been explored or an stable optimal solution is reached. The flowchart in Fig. 5.4 outlines the TS search procedure. The solution of this heuristic approach is "near optimal" and may just be a local minimal. However, our evaluation shows that a near optimal solution with the quality 5% worse than the true optimal can be reached with only 1% of the computation time.

5.5 Hadoop MapReduce Integration

Hadoop is a scalable platform that provides distributed storage and computational capabilities on clusters of commodity hardware. Hadoop MapReduce is a popular open source programming framework for cloud computing [29]. The framework splits the user job into smaller tasks and runs these tasks in parallel on different nodes, thus reducing the overall execution time when compared with a sequential execution on a single node. This architecture however, fails in the absence of external network connectivity, as it is the case in military or disaster response operations. This architecture is also avoided in emergency response scenarios where there is limited connectivity to cloud, leading to expensive data upload and download operations. In such situations, wireless mobile ad-hoc networks are typically deployed [32].

Building Hadoop on a mobile network enables the devices to run data intensive computing applications without direct knowledge of underlying distributed systems complexities. We developed the Hadoop MapReduce framework over MDFS and studied its performance in a real heterogeneous mobile cluster. The Hadoop MapReduce cloud computing framework meets our processing requirements for several reasons: 1) in the MapReduce framework, as the tasks are run in parallel, no single mobile device becomes a bottleneck for overall system performance; 2) the MapRe-

duce framework handles resource management, task scheduling and task execution in an efficient fault tolerant manner. It also considers the available disk space and memory of each node before tasks are assigned to any node; 3) Hadoop MapReduce has been extensively tested and used by large number of organizations for big data processing over many years. We implement the generic file system interface of Hadoop for MDFS which makes our system inter-operable with other Hadoop frameworks like HBase. There are no changes required for existing HDFS applications to be deployed over MDFS.

5.5.1 MDFS Hadoop Component

The file system functionality of each cluster node is split across three layers- MDFS Client, Data processing layer and Network communication layer. User applications invoke file system operations using the *MDFS client*, a built-in library that implements the MDFS file system interface. The MDFS client provides file system abstraction to upper layers. The user does not need to be aware of file metadata or the storage locations of file fragments. Data Processing layer manages the data and control flow of file system operations. The functionality of this layer is split across two daemons- *Name Server* and *Data Server*. *MDFS Name Server* is a lightweight MDFS daemon that stores the hierarchical file organization or the namespace of the file system. All file system metadata including the mapping of a file to its list of blocks is also stored in the MDFS Name Server. The Name Server has the same functionality as Hadoop NameNode. The Name Server is always updated with any change in the file system namespace. *MDFS Data Server* is a lightweight MDFS daemon instantiated on each node in the cluster. It coordinates with other MDFS Data Server daemons to handle MDFS communication tasks like neighbor discovery, file creation, file retrieval and file deletion. On startup, it starts a local RPC server

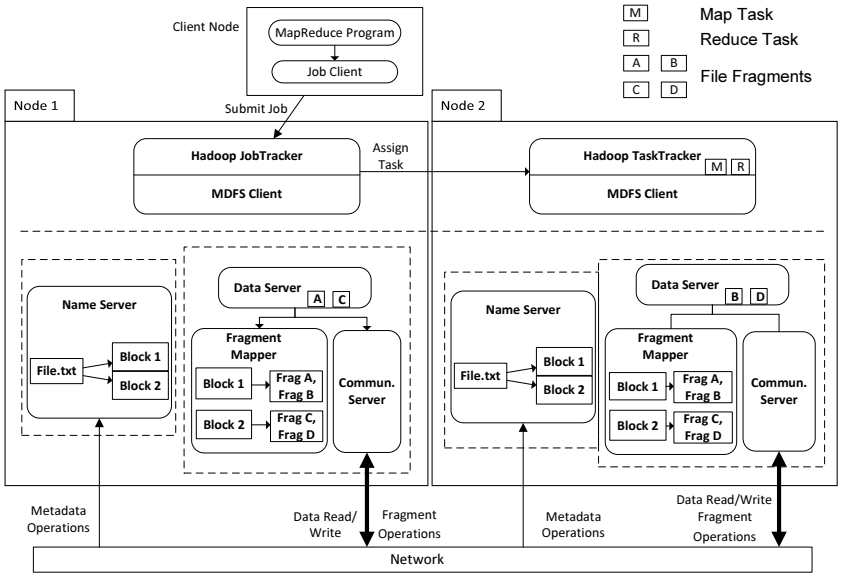


Figure 5.5: Distributed Mobile Hadoop architecture.

listening on the port defined by mdfs.dataservice.rpc-port in the configuration file. When the user invokes any file system operation, the MDFS client connects to the local Data Server at the specified port and talks to it using the MDFS Data Protocol. *Network communication layer* handles the communication between the nodes in the network. It exchanges control and data packets for various file operations. This layer abstracts the network interactions and hides the complexities involved in routing packets to various nodes in case of dynamic topologies like in MANETs.

5.5.2 Mobile Hadoop Architecture

We propose a distributed Mobile Hadoop architecture where there is no central entity to manage the cluster. As shown in Figure 5.5, every participating node runs a Name Server and a Fragment Mapper. After every file system operation, the update is broadcasted in the network so that the local caches of all nodes are synchronized. Moreover, each node periodically syncs with other nodes by sending

broadcast messages. Any new node entering the network receives these broadcast messages and creates a local cache for further operations. This architecture has no single point of failure and no constraint is imposed on the network topology. Each node can operate independently, as each node stores a separate copy of the namespace and fragment mapping. The load is evenly distributed across the cluster in terms of metadata storage when compared to the centralized architecture. However, network bandwidth is wasted due to the messages broadcast by each node for updating the local cache of every other node in the network. As the number of nodes involved in processing increases, this problem becomes more severe, leading to higher response time for each user operation.

5.5.3 Energy-Aware Task Scheduling

There are many challenges in bringing data locality to MDFS. Unlike native Hadoop, no single node running MDFS has a complete data block; each node has at most one fragment of a block due to security reasons. Consequently, the default MapReduce scheduling algorithm that allocates processing tasks closer to data blocks does not apply. MDFS, however, could find a processor node (taskTracker) that is closest to the fragments of a data block. In particular, we consider hop count as an estimator for the transmission cost between nodes. Knowing the network topology (from the topology maintenance component in MDFS) and the locations of each fragment (from the fragments mapper), we could estimate the total hop count for each node to retrieve the closest k fragments of the block. Smaller total hop count indicates lower transmission time, lower transmission energy, and shorter job completion time.

We now describe how to find the minimal cost (hop-count) for fetching a block from a taskTracker. $C_i^\star(b)$ is defined as the minimal cost of fetching block b at node

Algorithm 4: Task Scheduling

Input: S_b, F_b, d, k, i
Output: $X, C_i^\star$
$C_i^\star = 0$
$X \longleftarrow$ A new $1 \times N$ array initialized to 0
$D \longleftarrow$ A new $1 \times N$ array
for *j=1 to N* **do**
    D[j].node=j
    D[j].cost=$d_{ij} \times F_b(j) \times (S_b/k)$
    if $D[j].cost == 0$ **then**
        D[j].cost=N^2 // Just assign a big number
    end
end
$D \longleftarrow$ Sort D in increasing order by D.cost
for *i=1 to k* **do**
    X[D[i].node]=1
    $C_i^\star$ += D[i].cost
end
return $X, C_i^\star$

i. Let F_b be a $1 \times N$ binary vector where each element $F_b(i)$ indicates whether node i contains a fragment of block b (note that $\sum_{i=1}^{N} F_b(i) = n \quad \forall b$); S_b is the size of block b; $d_{i,j}$ is the distance (hop count) between node i and node j; X is a $1 \times N$ binary decision variable where $X_j = 1$ indicates that node j sends a data fragment to node i.

$$C_i^\star(b) = \min \sum_{j=1}^{N} (S_b/k) F_b(j) d_{i,j} X_j, \quad s.t. \sum_{j=1}^{N} X_j = k$$

$C_i^\star(b)$ can be solved by algorithm 4. Once $C_i^\star(b)$ for each node is found, the processing task for block b is then assigned to node p where $p = \arg\min_i C_i^\star(b)$.

5.6 Evaluation

In this section, we first present the evaluation results for the k-out-of-n Data Processing framework. The simulations are conduced on both synthetic data trace and realistic data trace. We then show the system implementation results for mobile Hadoop.

5.6.1 The k-out-of-n Data Processing Simulation

The simulation environment for the k-out-of-n data processing is identical to the environment for the k-out-of-n data storage. For readers' convenience, we repeat the setting again here. We consider a network of $400\times400\text{m}^2$ where up to 45 mobile nodes are randomly deployed. The communication range of a node is 130m, which is measured on our smartphones. Two different mobility models are tested – Markovian Waypoint Model and Reference Point Group Mobility (RPGM). Markovian Waypoint is similar to Random Waypoint Model, which randomly selects the waypoint of a node, but it accounts for the current waypoint when it determines the next waypoint. RPGM is a group mobility model where a subset of leaders are selected; each leader moves based on Markovian Waypoint model and other non-leader nodes follow the closest leader. Each mobility trace contains 4 hours of data with 1Hz sampling rate. Nodes beacon every 30 seconds.

5.6.1.1 Effect of node failures in the k-out-of-n data processing

This section investigates how the failures of processor nodes affect the energy efficiency, job completion time, and job completion rate. We first define how Greedy and Random work for data processing. In Greedy, each task is replicated to n-k+1 processor nodes that have the lowest energy consumption for retrieving the task, and given a task, nodes that require lower energy for retrieving the task are scheduled

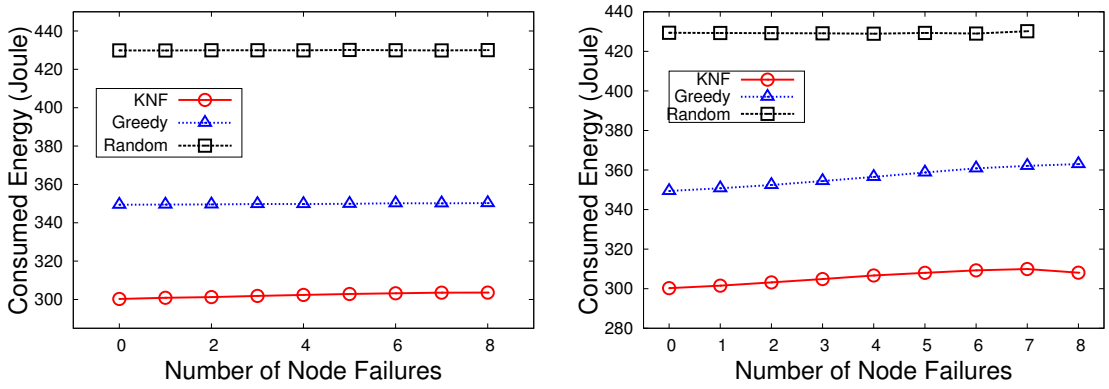


Figure 5.6: (a) Effect of node failure on energy efficiency with fail-slow. (b) Effect of node failure on energy efficiency with fail-fast.

earlier. In Random, the processor nodes are selected randomly and each task is also replicated to n_p-k_p+1 processor nodes randomly. We consider two failure models: fail-fast and fail-slow. In the fail-fast model, a node fails at the first time slot and cannot complete any task, while in the fail-slow model, a node may fail at any time slot, thus being able to complete some of its assigned tasks before the failure.

Figure 5.6a and Figure 5.6b show that KNF consumes 10% to 30% lower energy than Greedy and Random. We observe that the energy consumption is not sensitive to the number of node failures. When there is a node failure, a task may be executed on a less optimal processor node and causes higher energy consumption. However, this difference is small due to the following reasons. First, given a task, because it is replicated to n_p-k_p+1 processor nodes, failing an arbitrary processor may have no effect on the execution time of this task at all. Second, even if a processor node with the task fails, this task might have completed before the time of failure. As a result, the energy difference caused by failing an additional node is very small. In the fail-fast model, a failure always affects all the tasks on a processor node, so its

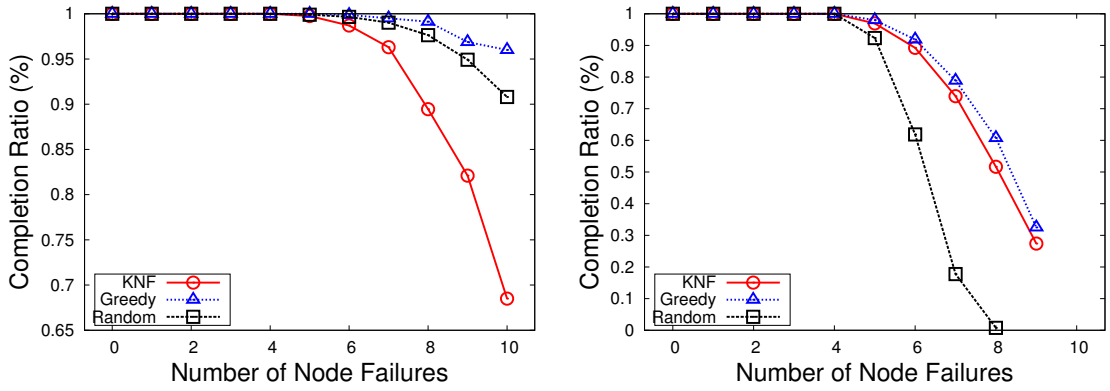
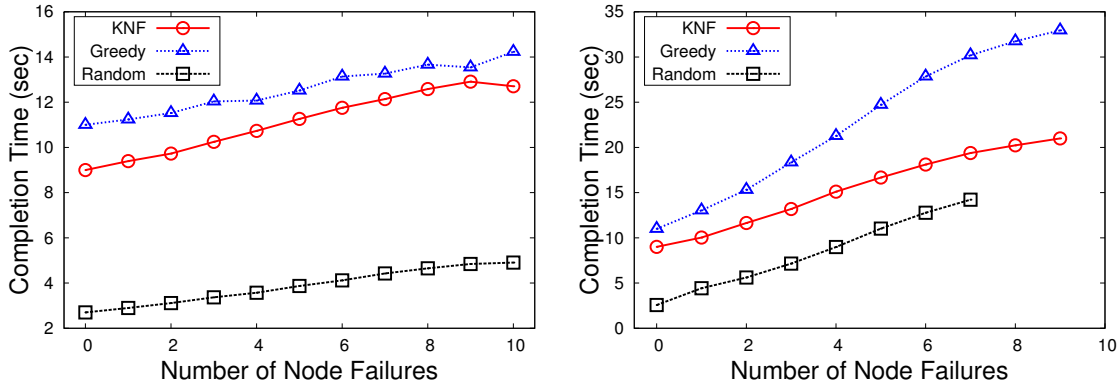


Figure 5.7: (a) Effect of node failure on completion ratio with fail-slow. (b) Effect of node failure on completion ratio with fail-fast.

energy consumption increases faster than the fail-slow model.

In Figure 5.7a and Figure 5.7b, we see that the completion ratio is 1 when no more than $n_p - k_p$ nodes fail. Even when more than $n_p - k_p$ nodes fail, due to the same reasons explained previously, there is still chance that all M tasks complete (tasks may have completed before the time the node fails). In general, for any scheme, the completion ratio of the fail-slow model is higher than the completion ratio of the fail-fast model. An interesting observation is that Greedy has the highest completion ratio. In Greedy, the load on each node is highly uneven, i.e., some processor nodes may have many tasks but some may not have any task. This allocation strategy achieves high completion ratio because all tasks can complete as long as one such high load processor nodes can finish all its assigned tasks. In our simulation, about 30% of processor nodes in Greedy are assigned all M tasks. Analytically, if three of the ten processor nodes contain all M tasks, the probability of completion when 9 processor nodes fail is $1 - \binom{7}{6}/\binom{10}{9} = 0.3$. Note that load-balancing is not an objective in this problem formulation as main objectives is energy-efficiency. We will consider



(a) (b)

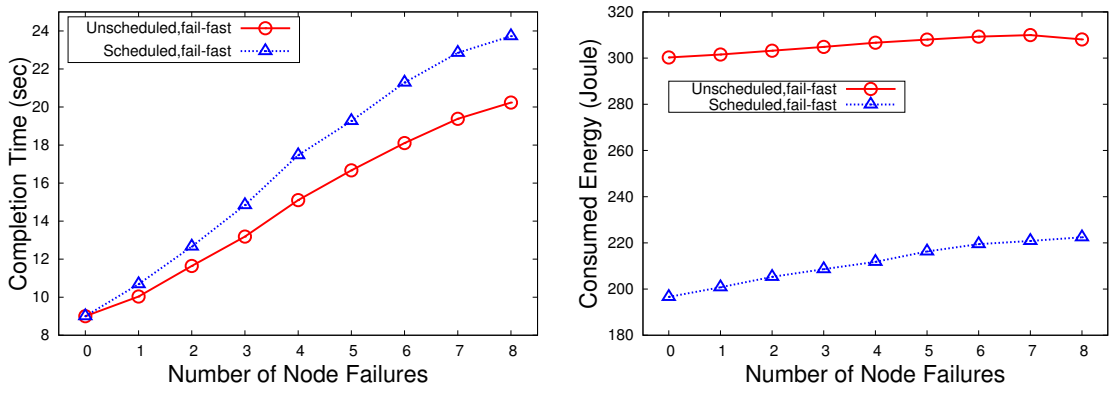
Figure 5.8: (a) Effect of node failure on completion time with fail-slow. (b) Effect of node failure on completion time with fail-fast.

the more complicated load-balancing problem in the later section.

In Figure 5.8a and Figure 5.8b, we observe that completion time of Random is lower than both Greedy and KNF. The reason is that both Greedy and KNF try to minimize the energy at the cost of longer completion time. Some processor nodes may need to execute much more tasks because they consume lower energy for retrieving those tasks compared to others. On the other hand, Random spreads tasks to all processor nodes evenly and thus results in lowest completion time.

5.6.2 Effect of Scheduling

Figure 5.9a and Figure 5.9b evaluate the performance of KNF before and after applying the scheduling algorithms to the k-out-of-n data processing. When the tasks are not scheduled, all processing nodes try to execute the assigned tasks immediately. Since each task is replicated to n_p-k_p+1 times, multiple instances of a same task may execute simultaneously on different nodes. Although concurrent execution of a same task wastes energy, it achieves lower job completion time. This is because when there is node failure, the failed task still has a chance to be completed on other processing



(a) (b)

Figure 5.9: (a) Comparison of performance before and after scheduling algorithm on job completion time. (b) Comparison of performance before and after scheduling algorithm on job consumed energy.

node in the same time slot, without affecting the job completion time. On the other hand, because our scheduling algorithm avoids executing same instances of a task concurrently, the completion time will always be delayed whenever there is a task failure. Therefore, scheduled tasks always achieve minimal energy consumption while unscheduled tasks complete the job in shorter time. The system reliability, or the completion ratio, however, is not affected by the scheduling algorithm.

5.6.3 Deadline-Aware Task Scheduling Simulation

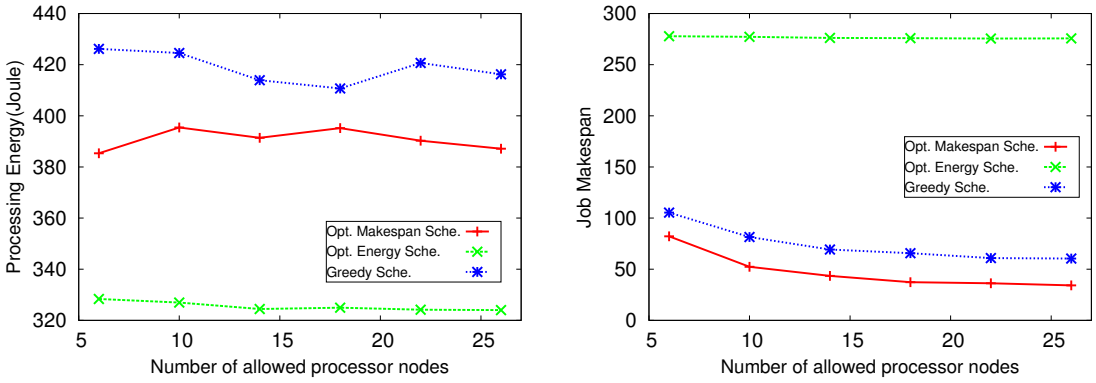
In this section, we employ both synthetic traces and a Dartmouth Network Trace [37]. Specifically, we are interested in the data processing energy and data processing makespan. These metrics are measured for different network sizes, number of storage/processor nodes, and number of failure nodes. We first evaluate the performance of our solution for data processing in mobile cloud. We measure the overall energy consumption and job makespan for completing a job, and compare the performance of the minimal-makespan and minimal-energy solutions, when the

application has a given processing deadline requirement (e.g., constraint in Equation 5.28). It is worth noting that the performance of data allocation (Section 4.2 – 4.3) directly impacts the performance of data processing as data processing requires accessing the stored data (as we discuss in the Conclusions section, we leave the joint optimization of data storage and data processing in mobile cloud for future work). We use a *Greedy* data processing algorithm for performance comparison. The idea of *Greedy* is to first select n_{max} nodes that potentially have the lowest task retrieval time, and then schedule tasks in a round-robin manner to achieve both load-balance and energy efficiency. The *Greedy* selects n_{max} nodes that contain the largest number of data fragments as the processor nodes. To achieve fault-tolerance, each task is also replicated n_p-k_p+1 times. At each level, processor nodes choose the tasks to execute in round-robin manner; each processor node always picks an unscheduled task that has the lowest retrieval energy.

5.6.3.1 Synthetic Network Trace

Throughout the data processing evaluations, we use a job consisting of 50 tasks (and thus 50 data blocks). The size of each data block is between 4MB-8MB, and is created at different times by our proposed data storage algorithm for mobile cloud. During the data processing operation, if a processor node fails, task instances scheduled before the failure time can still complete, but all later task instances fail.

Figures. 5.10a and 5.10b depict the effects of *the number of allowed processors* on the processing energy and job makespan in a 26 nodes network. The max number of allowed processors is determined by n_{max} in Equation 5.23. The solver finds a subset of processors and schedules tasks on these nodes. We compare the solutions of two different objective functions, *minimal-energy* and *minimal-makespan*. The greedy solution is also included. As shown, and expected, the optimal energy schedule has

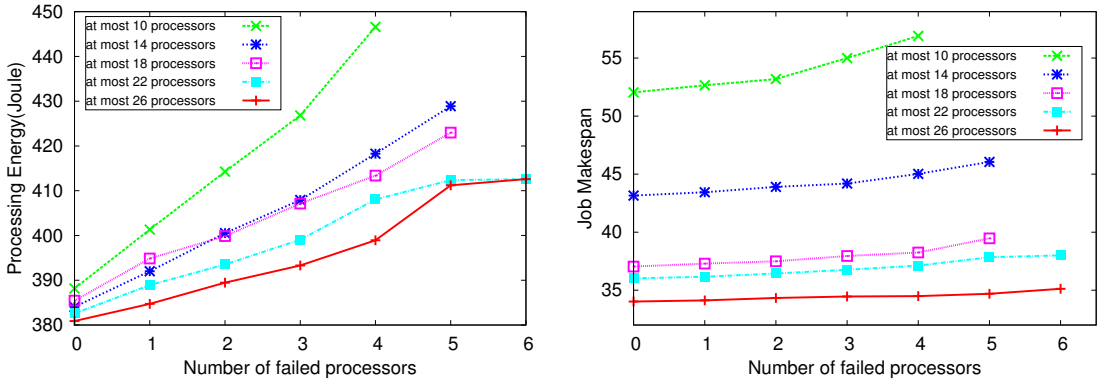


(a) (b)

Figure 5.10: The number of allowed processor nodes versus processing energy and job makespan. (a) Processing energy (b) Job makespan

the lowest energy and the optimal makespan solution has the lowest job makespan. However, the optimal energy schedule also has the worst job makespan, ∼4x higher than optimal makespan.

An interesting observation is that the optimal energy solution is not affected by the number of allowed processor nodes, as shown in both figures. This is because it only selects a few "most energy-efficient" processor nodes, regardless of the n_{max} constraint. As a result, its energy and job makespan remain almost constant once the optimal subset of processors is found. The tradeoff of minimizing the energy is the highly unbalanced task allocation and long job makespan. A similar behavior is also shown in minimal job makespan solution. The job makespan decreases gradually as n_{max} increases, but the improvement is not proportional to the number of added processor nodes. Again, although more processor nodes are available, the solver may choose only a small subset of nodes as processors. Besides, due to the dynamic nature of Mobile Cloud, distributing tasks to more nodes instead of clustering tasks on few highly reliable processors may cause more failures and unstable links.

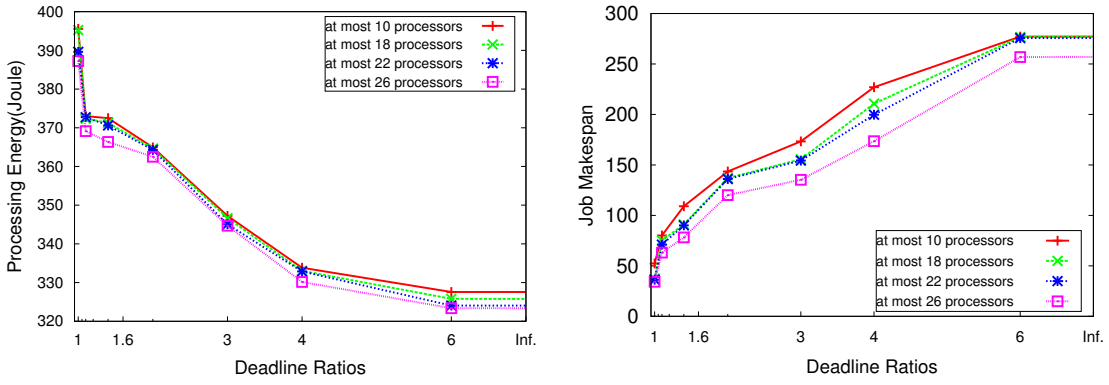


(a) (b)

Figure 5.11: Effect of node failures on processing energy and job makespan. (a) Processing energy (b) Job makespan (second)

Figures. 5.11a and 5.11b show how the processing energy and job makespan are affected by processor failures. The experiment uses 26 nodes and solutions for six different number of allowed processor nodes (n_{max}) are presented. Note that the x-axis shows only the number of failed processors, but other non-processor nodes may also fail and affect the overall performance. Both processing energy and job makespan increase when processor nodes fail because the tasks on failed processor nodes need to be retrieved and executed again on other processors. Also, node failures cause unstable links, so they increase both energy and time for retrieving tasks. The increment of energy and makespan grows faster as the number of failed processors increase. However, when comparing the solutions of different n_{max}, we see a more stable and resilient behaviour of solution with more processor nodes.

In Figures. 5.12a and 5.12b, we plot the processing energy and job makespan using *minimal-energy* constrained on the deadline. Numbers on the x-axis are the ratios between the deadline and the minimal job makespan ($minMakespan$); i.e., ratio 1.4 corresponds to "$deadline = 1.4 \times minMakespan$". $minMakespan$ is first found



(a) (b)

Figure 5.12: Effect of deadline constraint on: (a) Processing energy (b) Job makespan

by solving the *minimal-makespan* problem. The last index *Inf.* indicates *infinite deadline* constraint, which is essentially the minimal energy solution. Again, we use 26 nodes and compare the solutions of four different number of allowed processor nodes.

An interesting observation is that the processing energy drops significantly when the deadline ratio increases from 1 to 1.1, and it then slowly approaches the minimal energy solution. From the minimal makespan solution at x=1 to the minimal energy solution at x=*Inf.*, around 40% of the energy drop occurs in x=$[1, 1.1]$. This result has two important implications: 1) a slight increment of the deadline constraint from the *minMakespan* can significantly reduce the processing energy; 2) our minimal-energy solution is effective and is insensitive to the deadline constraint when the deadline ratio is above 1.1. As for the job makespan, it gradually increases with the deadline ratio and approaches a constant value when the processing energy reaches the minimal.

	5 min.	20 min.	30 min.	40 min.
Avg. N2N dist.	1.84	2.51	3.22	3.27
Avg.Degree	6.97	4.86	4.63	4.11
MCC	20	29	28	26
# of failures	3	3	5	8

Table 5.1: Statistics of Network Trace. Average node to node distance (hop-count), average degree of nodes, size of the maximal connected component, and number of failed nodes.

5.6.3.2 Dartmouth Network Trace

The Dartmouth Outdoor Dataset [37] includes the GPS locations and routing tables of 41 laptops moving in a $255 \times 365 m^2$ athletic field for 1.5 hours. During the experiment, 7 laptops failed to generate any data, and another 8 laptops became inactive after 30-40 minutes. Less than 26 nodes completed the entire experiment, as most of the laptops reached the end of battery life. This realistic trace serves as a good model for evaluating our solutions.

A file of 50 blocks is stored in the mobile cloud at the beginning of the experiment and an identical processing job is performed at four different times, 5, 10, 30, and 40 minutes. The mobility of nodes is given by the Dartmouth Outdoor Dataset [37]. Three different solutions, minimal-energy, minimal-makespan, and *Greedy* are compared in Figure 5.13, at different times. Some important network statistics at each processing time are summarized in Table 5.1. It shows the average node to node distance in terms of hop-count, the average degree of each node, the size of the maximal connected component (MCC), and the number of failed nodes.

Node failures are expected in Mobile Cloud (e.g., data loss, broken links, task re-execution, unstable links, and network partition). Also, when the physical size of the network increases due to mobility, the average node to node distance (hop-

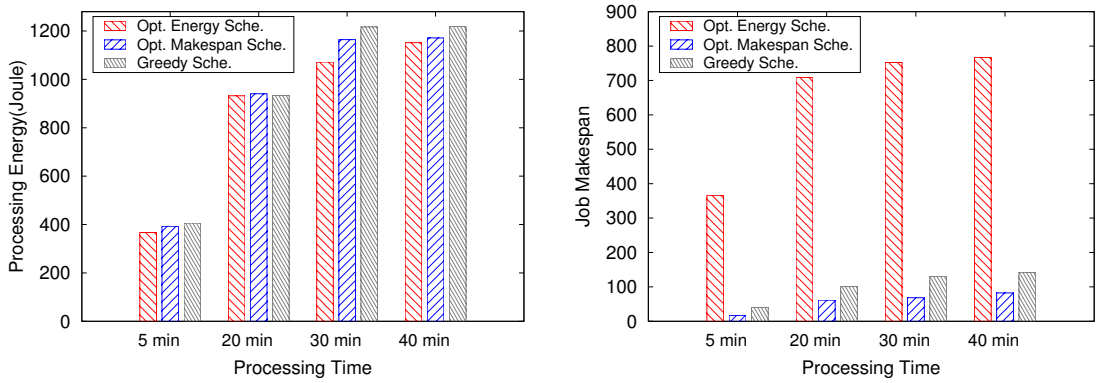


Figure 5.13: Performance evaluation of data processing on Dartmouth Outdoor Dataset. (a) Processing energy (b) Job makespan

count) increases and the number of reachable nodes in the network decreases (some nodes may leave the network). All these changes make the data retrieval and data processing in Mobile Cloud more difficult, and thus impair the data processing energy as well as job makespan. When all 35 nodes are operational, close to each other, and have good network connectivity at the beginning of the experiment, both processing energy and job makespan achieve the minimal. However, as the network topology changes with time, the performance of all three solutions degrades. One interesting result is that all three solutions have very close data processing energy. A possible explanation is that the network spreads very uniformly and node to node distance has low variance, so the data allocation does not have significant impact on the data retrieval energy. As for the job makespan, minimal-energy solution again has the worst job makespan, and the *Greedy* has slightly longer job makespan than the minimal-makespan solution.

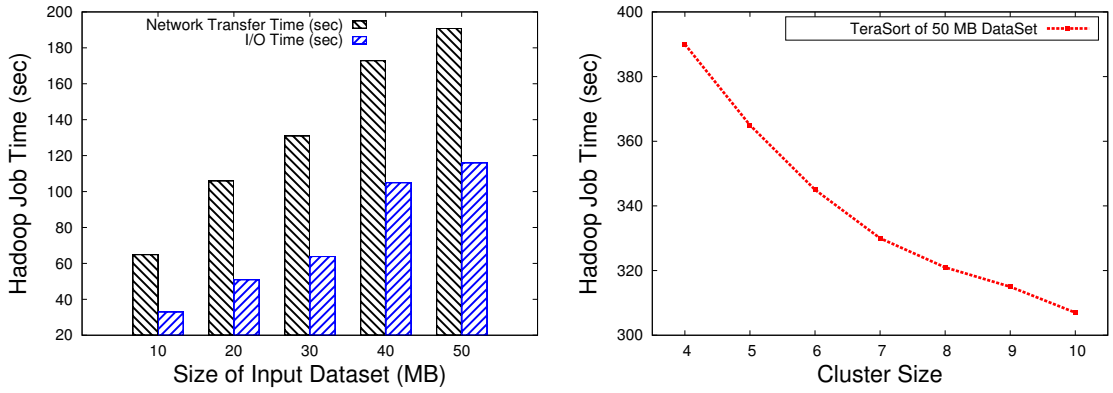
5.6.4 Mobile Hadoop

To measure the performance of mobile Hadoop on mobile devices, we ran Hadoop benchmarks on a heterogeneous 10 node mobile wireless cluster consisting of 1 personal desktop computer (Intel Core 2 Duo 3 GHz processor, 4 GB memory), 10 netbooks (Intel Atom 1.60 GHz processor, 1 GB memory, Wi-Fi 802.11 b/g interface) and 3 HTC Evo 4G smartphones running Android 2.3 OS (Scorpion 1Ghz processor, 512 MB RAM, Wi-Fi 802.11 b/g interface). We used TeraSort, a well-known benchmarking tool that is included in the Apache Hadoop distribution. Our benchmark run consists of generating a random input data set using TeraGen and then sorting the generated data using TeraSort.

5.6.4.1 Effect of dataset size and cluster size

Figure 5.14a shows that processing time is 70% smaller than the network transmission time for TeraSort benchmark. So, tasks have to be sufficiently long enough to compensate the overhead in task setup and data transfer for maximum throughput. For real world clusters, the optimal value of block size will be experimentally obtained.

The cluster size determines the level of possible parallelization in the cluster. As the cluster size increases, more tasks can be run in parallel, thus reducing the job completion time. Figure 5.14b shows the effect of cluster size on job completion time. For larger files, there are several map tasks that can be operated in parallel depending on the configured block size. So the performance is improved significantly with increase in cluster size as in the figure. For smaller files, the performance is not affected much by the cluster size, as the performance gain obtained as part of parallelism is comparable to the additional cost incurred in the task setup.



(a) (b)

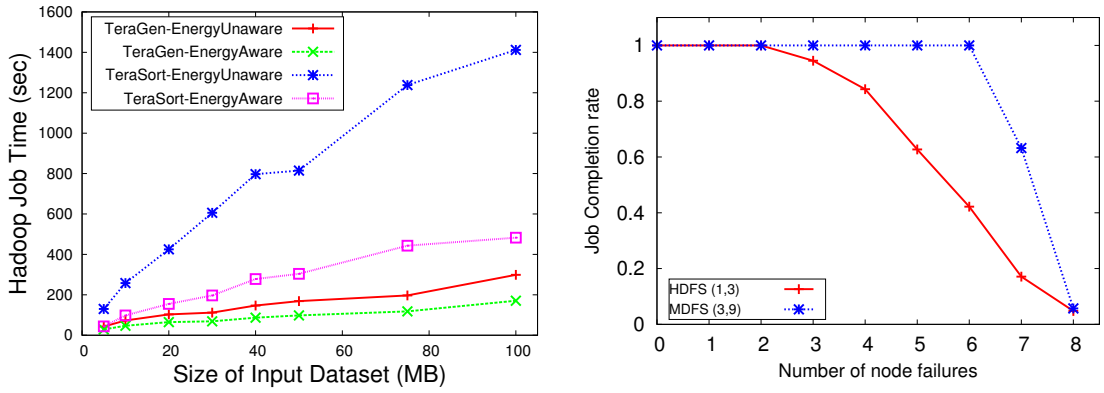
Figure 5.14: (a) Job Completion time versus Input dataset size (b) Cluster size on Job Completion Time

5.6.5 Energy-Aware Task Scheduling v.s. Random Task Scheduling

As mentioned in Section 5.5.3, our energy-aware task scheduling assigns tasks to taskTrackers considering the locations of data fragments. The default task scheduling algorithm in Map-Reduce component is ineffective in mobile ad-hoc network as the network topology in a traditional data center is completely different from a mobile network. Figure 5.15a compares the job completion time between our energy-ware scheduling algorithm and a random task scheduling. The default Map-Reduce task scheduling in a mobile ad-hoc network is essentially a random task allocation. In both *TeraGen* and *TeraSort* experiments, our scheduling algorithm effectively reduces the job completion time by more than 100%. Lower job completion time indicates lower data retrieval time and lower data retrieval energy of each taskTracker.

5.6.6 Effect of Node Failures on MDFS and HDFS

In this section, we compare the fault-tolerance capability between MDFS and HDFS. We consider a simple failure model in which a task fails with its processor node and a taskTracker can not be restarted once it fails. In HDFS, each data block



(a) (b)

Figure 5.15: (a) Comparison of new task scheduling algorithm versus Random (b) Comparison of job completion rate between HDFS and MDFS

is replicated to 3 different nodes, and HDFS can tolerate to lose at most 3 data nodes; in MDFS, each data block is encoded and stored to 9 different nodes ($k = 3, n = 9$), and MDFS can tolerate to lose up to 6 data nodes. There are total 12 nodes in the network. Note that although HDFS can tolerate to lose at most 3 data nodes, it does not mean that the job would fail if more than 3 nodes fail; if the failed node does not carry the data block of the current job, it does not affect the taskTracker; as a result, we see completion rate gradually drops from 1 after more than 3 nodes fail. Figure 5.15b shows that MDFS clearly achieves better fault-tolerance when 3 or more nodes fail.

6. HETEROGENEOUS MOBILE CLOUD

In this section, we study and design an *energy-efficient, fault-tolerant, and load-balanced* distributed data storage and data processing framework for *Heterogeneous Mobile Cloud (HMC)*. Different from the previous sections, mobile nodes now can have various hardware specifications in terms of communication interfaces, processing capabilities, and energy capacities. We will look at the k-out-of-n data storage and data processing frameworks in an integrated manner and propose a *decentralized algorithm* that allocates *data, computation, and communication* resources in a mobile cloud. The framework is scalable to larger network and can adapt to heterogeneous network. With the same requirements as the previous sections, fault-tolerance ensures that the stored data are resilient to node failures, energy-efficiency ensures that the framework minimizes the system-wide energy consumption, and load-balance ensures that each node is allocated proper workload according to its available communication, processing, and energy resources. We refer to this Mobile Storage & Processing System as *MSPS*.

MSPS greedily minimizes the *standardized energy* consumption while upholding the system-wide *load imbalance*. *Standardized energy* is a value in (0,1) defined as the ratio between the *consumed energy* and the *energy capacity*. *Consumed energy* is the energy consumed by radio communication or data processing since the application starts; *energy capacity* is the amount of energy that the mobile device can store. The rationale is that each node may have different energy capacity and nodes with higher energy capacity should be able to contribute more. E.g., a generator-powered tablet has much higher energy capacity than a battery-powered tablet; when both tablets process the same task, the standardized energy of the first tablet is much lower than

the standardized energy of the second tablet. In other words, two nodes consuming the same amount of standardized energy may have very different workloads, but they both run out of energy when their standardized energy reaches 1. *MSPS* keeps the rate of standardized energy usage roughly equal among all nodes and *balances* the energy consumption in a way such that all nodes have approximately equal operational time. The *load* of a node is defined as the standardized energy consumption in a short period of time, and the *load imbalance* indicates how different the load of all nodes in the network is. *MSPS* heuristically reduces the *load imbalance* to prolong the system lifetime. Furthermore, the utilization rate of the wireless communication and CPUs are also considered when allocating tasks, to avoid deteriorating the performance while conserving energy.

In the rest of the section, we first illustrate the architecture of the heterogeneous Mobile Storage & Processing System, followed by its design and details. We then present the agent-based search algorithm used for efficient request dissemination. Finally, the evaluation results of MSPS are shown.

6.1 MSPS Architecture and Problem Formulation

We consider a heterogeneous mobile cloud of N nodes, where each node may have different hardware capabilities or energy capacity. Nodes communicate with each other using any available wireless interface. *MSPS* supports three major data operations: *data creation*, *data retrieval* and *data processing*. When creating a new file, the file creator encodes a file using *erasure coding* in which a file is encoded into n data fragments, and any subset of k data fragments together can recover the original file; each data fragment is then sent to a different storage node. When another node needs to read the file, it searches and retrieves k of the n data fragments from the network to recover the file. Any node can submit a job to process a subset of

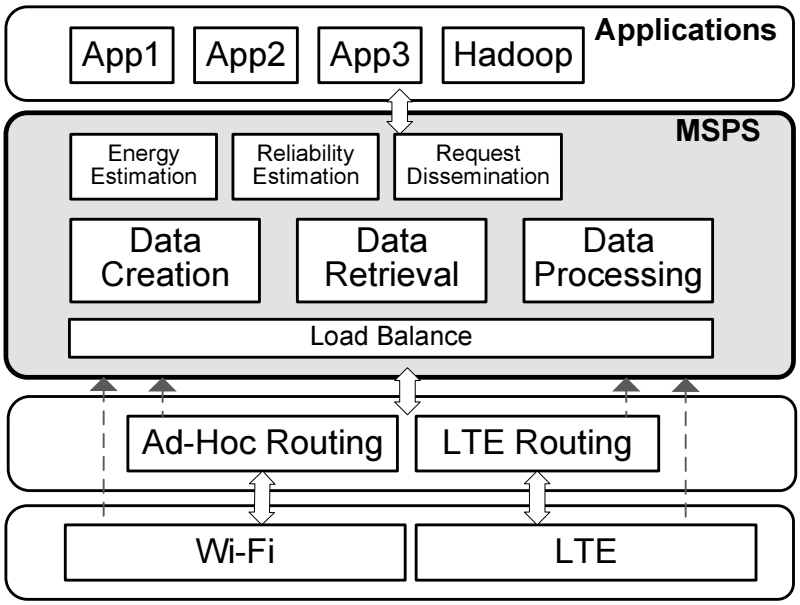


Figure 6.1: Mobile Storage & Processing System (MSPS) System Architecture

stored files. A processing job consists of multiple independent tasks, where each task corresponds to processing a single file on a selected processor node.

We make the following assumptions when designing *MSPS*. Nodes are mobile, and can depart or join a network freely. Each node's hardware capabilities are profiled so that the data transferring and data processing energy are known. All processing functions are also profiled so that the number of instructions of a processing task can be estimated. A system-wise distributed directory service allows nodes to know the available files. Files stored in *MSPS* are shared by all nodes in the network, and every node can create, retrieve, or process the stored files. Files are immutable – once created, they can be deleted, but can not be modified.

Figure 6.1 illustrates the system architecture. *MSPS* serves as a middleware that provides applications data storage and data processing services. *MSPS* accesses the routing information and the link quality information from the network layer and

MAC layer. The three major data operations, *data creation*, *data retrieval*, and *data processing*, access this cross-layer information when allocating communication and processing tasks. As an example, we assume WiFi and LTE are the only two communication interfaces. The *Energy Estimation* component estimates the energy from communication and processing; communication part includes the energy for sending/receiving a packet through various wireless interfaces based on the current link quality and the hardware specification of the node; processing energy is estimated from the processing speed and hardware specification of the node. The *Reliability Estimation* component estimates the reliability of a node based on the current remaining energy, connectivity, and mobility. The *Load-balance* component monitors the utilization rate of communication, CPU, and energy resources on each node. Data operations use this information to avoid overloading when allocating tasks. The *Request Dissemination* component disseminates request messages to discover storage nodes, file fragments, or processor nodes in the network. The *Agent-based search* algorithm is implemented in this component.

We now formulate *MSPS* in Equations (6.1)–(6.4) as an optimization problem. The *energy load* $L_i(t)$ of node i at time t is defined as the standardized energy consumed during time $[t - T_{LB}, t)$. T_{LB} is the time period in which the *load* is of interest. The *Load Imbalance LI(t)* at time t is defined in Equation 6.5 where $L_{15}(t)$ is the mean load of the nodes in top 15%, and $L_{\mu}(t)$ is the mean load of all nodes. The objective is to minimize the standardized energy consumption constrained on the utilization rate of energy (Eq. 6.2), communication (Eq. 6.3), and CPU resources (Eq. 6.4). This optimization problem is solved for each data operation. Given a data operation request, the objective function (Eq. 6.1) finds a subset of nodes $X \subset N$ to perform the task such that the cumulative standardized energy $E_i^{std}(task, t)$ of all participating nodes at time t is minimized. Using the data creation task shown in

$$X_{opt} = \arg\min_{X} \sum_{i \in X} E_i^{std}(task, t) \tag{6.1}$$

Subject to:

$$\frac{L_i(t)}{L_\mu(t)} - 1 \leq S^{LI} \ \forall i \in X \tag{6.2}$$

$$U_i^{com}(t) \leq S^{com} \ \forall i \in X \tag{6.3}$$

$$U_i^{cpu}(t) \leq S^{cpu} \ \forall i \in X \tag{6.4}$$

Figure 6.2, as an example, node 11 selects nodes 2, 3, 9, 10, and 11 to participate in the data creation task. Note that this optimization problem is considered in a *local sense*, meaning that it may not consider all nodes in the network when selecting X_{opt}. Because *MSPS* does not assume or acquire global information, the subset X is selected only from the nodes discovered by the *request dissemination* procedure.

A task may require nodes to transfer data, store data, or process data. S^{LI}, S^{com}, and S^{cpu} are the predefined thresholds for energy, communication, and CPU resource utilization rates. Equation 6.2 indicates that a selected node should not have an energy load higher than the network average by ratio of S^{LI}. $U_i^{com}(t)$ is a value in $[0, 1]$ representing the average utilization rate of a communication interface on node i during time $[t - T_{LB}, t)$. Depending on the available wireless interfaces of a node, $U_i^{com}(t)$ may be WiFi, LTE, or other possible interfaces. Similar to $U_i^{com}(t)$, $U_i^{cpu}(t)$ represents the CPU average utilization rate on node i. Equations 6.3 and 6.4 ensure that a node will not be assigned more tasks than its communication or processing capabilities.

$$LI(t) = \frac{L_{15}(t)}{L_\mu(t)} - 1 \tag{6.5}$$

6.2 MSPS Design

This section presents the details of each data operation (i.e., data creation, data retrieval, and data processing). We then present the load-balance algorithm and explain how it is integrated with the data operations to make *MSPS* both energy-efficient and load-balanced.

6.2.1 Data Creation

A node starts the *Data Creation* operation when it needs to store or share a file in *MSPS*. We refer to this node as *File Creator*. Here we assume the storage parameter (k, n) is specified by the application based on its reliability requirement. A file is encoded into n data fragments, and each fragment is sent to a selected storage node. The file creator first uses the *request dissemination* component to discover nodes that can store data fragments. A node receiving the request replies with its routing table, reliability, and the current energy profile. The energy profile contains $L_i(t)$, $U_i^{com}(t)$, $U_i^{cpu}(t)$ and the standardized energy for transferring data using various wireless interfaces available on the node. The details of request dissemination procedure and energy profile are explained in later sections.

One additional requirement for data creation is ensuring that the stored files are reliable and have high availability. Without reliable data, the entire *MSPS* is useless. The Data creator needs to choose storages nodes that are more reliable, i.e., nodes that are less likely to fail in the near future. *Data Reliability* of a file can be estimated from the reliability of its storage nodes. $R_x(t)$ indicates the reliability of node x at time t. Assume $X = \{x_1, x_2, ..., x_n\}$ is a set of n nodes and $\mathbf{c}$ is a subset of X that contains i nodes. Equation 6.6 calculates the probability of exactly i functioning nodes and $(n - i)$ malfunctioning nodes in X; it considers all possible subsets $\mathbf{c}$ in X. Equation 6.7 calculates the probability of k or more functioning nodes in X,

i.e., it is the reliability of a file stored in X. This reliability estimation has time complexity $O(n!)$ due to the combination $\binom{n}{i}$ term in Equation 6.6. Note that the time parameter t is ignored for clarity.

$$S_i = \sum_{j=1}^{\binom{n}{i}} \prod_{l \in \mathbf{c}} R_l \prod_{m \in \bar{\mathbf{c}}} (1 - R_m) \tag{6.6}$$

$$\text{where } \mathbf{c} \subset X, \bar{\mathbf{c}} = X \backslash \mathbf{c}, \|X\| = n, \text{ and } \|\mathbf{c}\| = i$$

$$R^{(k,n)}(X) = \sum_{i=k}^{n} S_i \tag{6.7}$$

Using the routing information and the energy profile, the file creator accumulates information about the standardized energy of nodes on the path and estimates the total standardized energy for delivering a fragment to a potential storage node. When a node is reachable through multiple wireless interfaces, the file creator always chooses the interface that has the lowest cumulative standardized data transferring energy. Assume $E^{std}_{creator}(x_i, t)$ gives the minimal standardized energy when sending a fragment from the file creator to node i. Equation (6.8) describes a simplified optimization problem for data creation without considering the load balance constraints (Equations 4.2–4.4). The additional constraint ensures that the created data meets the reliability requirement r_{req}.

$$X_{sto} = \arg\min_{X} \sum_{x \in X} E^{std}_{creator}(x) \tag{6.8}$$

$$\text{s.t.:} \quad R^{(k,n)}(X) \geq r_{req} \ , \ \|X\| = n$$

Solving this problem, however, is very challenging due to the complexity of relia-

bility estimation $R^{(k,n)}(\cdot)$. In order to quickly find a feasible solution, we propose an approximated solution. Instead of exhaustively searching for a set of storage nodes X from the network and evaluating its data reliability $R^{(k,n)}(X)$, we consider only nodes with reliability no less than r_{min} such that any subset of n nodes guarantees to satisfy the data reliability constraint. r_{min} enforces a minimal reliability to the selected storage nodes and effectively eliminates the reliability constraint. The objective function (Equation 6.8) now only needs to choose n nodes with the lowest $E^{std}_{creator}(x)$.

To find r_{min}, we solve the equation $R^{(k,n)}(X) = r_{req}$ in which the reliability of each node $R_x(\cdot)$ is substituted with r_{min}. This equation can be solved efficiently using root-finding methods such as bisection, interpolation, or Newton's method. Since $R^{(k,n)}(X)$ is a continuous function that monotonically increases with $R_x(\cdot)$, any subset X with all nodes' reliability higher than r_{min} must have $R^{(k,n)}(X)$ higher than r_{req}. The data creation procedure is described in these three steps:

1. Solve $R^{(k,n)}(X) = r_{req}$ where $R_x(\cdot) = r_{min}$
2. Find all x such that $R_x(\cdot) \geq r_{min}$. These nodes are candidate storage nodes.
3. Sort all candidate storage nodes in increasing order with respect to $E^{std}_{creator}(x)$. The first n nodes are then selected as storage nodes.

6.2.2 Data Retrieval

A node needs k data fragments of a file in order to decode and recover the original file. Although any subset of k fragments can recover the file, the file requester tries to minimize the standardized energy for retrieving k data fragments. The *Request Dissemination* component discovers the available data fragments and collects information from nearby nodes. Nodes receiving the request reply informa-

tion about the fragments they carry, the routing tables, and their energy profiles. Similar to data creation, the file requester uses the collected information to estimate the total standardized energy for retrieving a data fragment from a potential storage node. Each fragment again may be sent through multiple wireless links between the file requester and the storage node, but the file requester considers only the link with minimal standardized energy. $E^{std}_{request}(x)$ represents the minimal standardized energy for downloading a data fragment from node x to file requester. Assume $F = \{f_1, f_2, ..., f_k\}$ represents a set of fragments that the file requester chooses to retrieve, and $x_{sto}(f)$ gives the storage node of fragment f. Equation 6.9 describes a simplified data retrieval optimization problem without the load balance constraints. Essentially, the k fragments with the lowest $E^{std}_{request}(x_{sto}(f))$ are retrieved. To help improve data availability and data reliability, the retrieved data fragments are cached on the file requestor. These cached fragments can later serve other file requesters.

$$F_{opt} = \arg\min_{F} \sum_{f \in F} E^{std}_{request}(x_{sto}(f)) \tag{6.9}$$
$$\text{s.t.:} \quad \|F\| = k$$

6.2.3 Data Processing

Any node can submit a data processing job to process/analyze a set of files. $T = \tau_1, \tau_2, ..., \tau_M$ indicates a set of M tasks that the job requester submits. Each task is assigned a selected processor node. This processor node retrieves, recovers, and then processes the file. Similar to other data operations, the *job creator* uses *request dissemination* component to announce a job. A node receiving the job request estimates its standardized energy for retrieving and processing a task and sends this

information to the job requester. A node has 0 file retrieval energy if it has k or more cached fragments. If a node has no cached fragments of a task, it estimates the task retrieval energy using the most expensive communication interface. Let $E^{std}_{proc}(x, \tau)$ represent the minimal standardized energy for node x to retrieve and process task τ. Y is a $1 \times M$ decision variable in which Y_i indicates the processor node for task τ_i. Equation 6.10 describes a simplified data processing optimization problem without the load balance constraints. Each task τ_i is essentially assigned to the node $Y_i = \arg\min_x E^{std}_{proc}(x, \tau_i)$.

$$Y_{opt} = \arg\min_{Y} \sum_{i=0}^{M} E^{std}_{proc}(Y_i, \tau_i) \tag{6.10}$$

6.2.4 Load Balancing

To avoid overloading a small number of nodes or causing performance bottleneck, *MSPS* considers *load-balancing* when allocating data operation tasks. In this section, we explain how *MSPS* detects load imbalance and reduces load imbalance. The system-wide load imbalance $LI(t)$ is formally defined in Equation 6.5. $LI(t) = 0$ if a system is perfectly balanced, and $LI(t)$ increases towards positive infinity when a system becomes more imbalanced. Our goal is to keep each individual load $L_i(t)$ as close to the mean load $L_\mu(t)$ as possible. Specifically, *MSPS* avoids assigning more tasks to a node if its current load is much greater than the system mean load. The load balancing algorithm is integrated into each data operation such that the decisions made not only minimize the standardized energy, but also lower the system-wise load imbalance.

The mean load $L_\mu(t)$, however, cannot be evaluated exactly because *MSPS* does

not have global information. Instead, a *sample mean load* $\hat{L_\mu}(t)$ estimated from local information is used to approximate the population mean $L_\mu(t)$. $\hat{L_\mu}(t)$ is calculated for each data operation using the information that the search agent discovered. Since nodes can move freely and a data operation can be initiated from any region of the network, $\hat{L_\mu}(t)$ estimation is not biased to any subset of nodes.

When describing the data operations in previous sections, we focused only on minimizing the standardized energy (Equation 6.8 – Equation 6.10). The load balance constraints Equation (4.2) – Equation (4.4) were neglected for clarity. To solve the complete optimization problem with load balance constraints, we first transform the constrained optimization problem into an unconstrained optimization problem using a penalty method. The constraints are combined into a penalty function $g(i,t)$ multiplied to the objective function, as shown in Equation 6.11. Specifically, $g(i,t)$ is multiplied to the simplified objectives functions (Equation 6.8 – Equation 6.10). The new solution avoids assigning tasks to nodes with high utilization rate (communication, processing, or energy), and thus gradually alleviates the load on nodes with $L_i(t) > L_\mu(t)$.

The characteristic of the penalty function is that it is zero if all the constraints in Equation (4.2) – Equation (4.4) are satisfied, and grows exponentially if any constraint is violated. α is a positive real value that controls how fast the penalty function grows and $H(\cdot)$ is the Heaviside step function that limits the domain of interest. For example, assume the communication utilization and the CPU utilization constraints are satisfied, but the load on node i is slightly higher than the mean load ($L_i(t) = 0.6,\ L_\mu(t) = 0.5,\ S^{LI} = 1,\ \alpha = 1$). The objective function will be penalized by a factor of $1 + \frac{1}{5}e^{\frac{5}{4}}$. The rationale behind this transformation is that it relaxes the hard constraints and allows a solution to violate slightly if the advantages of violation (energy saving) is considerably higher than the disadvantage

$$X_{opt} = \arg\min_{X} \sum_{i \in X} E_i^{std}(task, t) \times (1 + g(i, t)) \tag{6.11}$$

$$\begin{aligned} g(i,t) = {} & (\tilde{L}) e^{\frac{\alpha}{|1-\tilde{L}|}} (H(\tilde{L} - S^{LI}) - H(\tilde{L} - 1)) \\ & + (\tilde{U_1} - S^{com}) e^{\frac{\alpha}{|1-\tilde{U_1}|}} (H(\tilde{U_1} - S^{com}) - H(\tilde{U_1} - 1)) \\ & + (\tilde{U_2} - S^{cpu}) e^{\frac{\alpha}{|1-\tilde{U_2}|}} (H(\tilde{U_2} - S^{cpu}) - H(\tilde{U_2} - 1)) \end{aligned} \tag{6.12}$$

$$\tilde{L} = \frac{L_i(t) - L_\mu(t)}{L_\mu(t)},\ \tilde{U_1} = U_i^{com}(t),\ \tilde{U_2} = U_i^{cpu}(t)$$

of violation (load imbalance). Furthermore, the solution of the optimization problem can be obtained quickly using greedy search. However, there is an unavoidable tradeoff between energy saving and load balancing because load balance constraints (or penalty function) may prevent a task from being assigned to the most energy efficient node. The level of load imbalance that an application can tolerate and how fast *MSPS* moves towards a balanced state are all controlled by the parameters in $g(i, t)$.

6.2.5 Reliability Estimation

The reliability of each node is estimated based on the failure probability model proposed in section 4.2.2. Three main factors, energy depletion, temporary departure from the network (e.g., due to mobility), and application-specific factors, are used to to determine the reliability of a node at a given time. The reliability of a node at time t is defined as the probability that the node remains functional from time t to time $t + T_{rel}$. T_{rel} is the time period in which the stored data or the processing tasks are effective, i.e., the stored data or a processing job meets the application reliability requirement only in time $[t, t + T_rel]$. As a result, the reliability of a node generally

decreases with time.

6.2.6 Energy Profile

Each node keeps an *energy profile* that tracks the node's energy capacity, remaining energy, and power consumption of each wireless interface. When receiving a request, a node estimates its standardized energy for transferring the requested data using the energy model proposed in [44]. The power of each wireless link, e.g., Bluetooth, Wi-Fi, 3G, or LTE, is modeled by $P = \alpha_u t_u + \alpha_d t_d + \beta$. t_u and t_d are uplink throughput (Mbps) and downlink throughput respectively; α_u, α_d, and β are experimentally obtained fitting coefficients (mW/Mbps). For data processing energy, the CPU of each node is profiled in advance so that the energy consumption for processing a file can be estimated from the processing function and the file size.

6.3 Request Dissemination

All three primary operations, *data creation*, *data retrieval*, and *data processing*, send request messages to explore the network. Data creation sends *storage request* to discover suitable storage nodes, data retrieval sends *file request* to find fragments of a file, and data processing sends *job request* to find processor nodes for processing a set of files. The simplest solution is broadcasting a request through the LTE network, but it affects all nodes in the network and is too costly. Another naive solution is flooding a request through the Wi-Fi network. However, flooding a message in a multi-hop Wi-Fi network incurs traffic burst (broadcast storm problem) if the messages traverse too many hops, or may fail to find sufficient resources (storage nodes or fragments) if the messages are not transmitted far enough. In this section, we propose an *agent-based search* algorithm that explores resources using mobile agents such that the desired resources can be found with high probability without causing too much overhead.

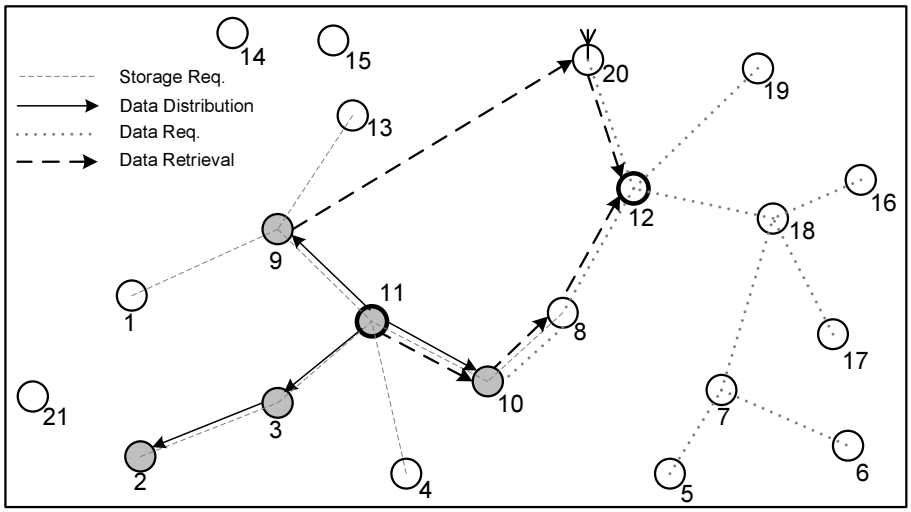


Figure 6.2: Data Creation & Data Retrieval. Node 11 creates a file with (k,n)=(3,5). Dash line indicates storage request and reply messages; solid arrow line represents data distribution flow. Nodes 2, 3, 9, 10, 11 are selected as storage nodes. Node 12 retrieves the file. Dotted line indicates data request messages; dash arrow line represents data retrieval flow. 2 fragments are retrieved through Wi-Fi network from node 10 and node 11, and 1 fragment is retrieved through cellular network from node 9.

6.3.1 Agent-Based Search

The goal of the search algorithm is to explore resources in the network as well as collect information such as routing table and energy profile from other nodes. The *Search Initiator* is the node that starts a search task; it dispatchs one or multiple *agents* that explore in different regions of the network to discover the desired resources. Each agent is assigned a *target resource* value that indicates the quantity of the resource it needs to find. When an agent finds sufficient resources or reaches a node that has no more unexplored neighbors, the agent replies its collected information back to the search initiator. One agent can *fork* into multiple children agents which share the target resource of the parent agent. Each child agent is then dispatched to different regions of the network to accomplish the parent agent's search

Algorithm 5: Agent-based Search

```
agent arrives node :
/* agent and node exchange information                                    */
agent.collectInfo(node)
node.collectInfo(agent)
if node.resource ≥ agent.targetRsc then
    node.resource -= agent.targetRsc
    agent.targetRsc = 0
else
    agent.targetRsc -= node.resources
    node.resource = 0
end
if TargetRsc == 0 || node.unexploredNeib.isEmpty() then
    reply(agent, requester)
else
    agent.targetRsc = agent.targetRsc / unexploredNeib.size()
    for n ∈ node.unexploredNeib do
        send(agent, n)
    end
end
```

task. The *update function* updates information on both the search agent and the node that the agent visits; whenever an agent visits a node, it tells the node the upstream nodes that it has visited; the node also tells the agent its information such as the data transferring energy or data processing energy of the requested task. If a search agent needs to continue (because target resource has not been completely found and there is still unexplored neighbors), it transfers the target resource to each child search agent. The number of the child agents is determined by the number of unexplored neighbor nodes. The agent-based search procedure is depicted in Algorithm 5.

The search algorithm, however, does not guarantee to find sufficient resources in one pass. If the network topology is sparse or the resources are distributed very

non-uniformly, some agents may reply and terminate without finding enough target resources. If the search initiator fails to find enough resources, it starts another search iteration. Knowing the amount of deficient resources and the boundary of the previous search (where the search agents terminated), the search initiator simply starts a new search from the boundary nodes. The search continues until sufficient resources have been found or the entire network has been explored. The procedure is outlined as follows:

1. Search initiator starts the Agent-based Search.
2. Search initiator waits for replies from the terminated agents.
3. Search completes when enough resources are found; otherwise, if there are still unexplored nodes, the search initiator starts a new Agent-based search from the boundary nodes.
4. Repeat steps 2 – 3 until sufficient target resources are found or the entire network has been explored.

Using storage request as an example, we now present a simple agent-based search in Figure 6.2. Assume all nodes in the network are valid potential storage nodes. Node 11 is the file creator (search initiator) that needs to find 7 storage nodes. It creates a search agent with target resource 7. Since node 11 itself is also a valid "resource" for storing fragments, the update function immediately updates the target resource to $7 - 1 = 6$, meaning one unit of the target resource has been found. The update function than forks the agent into 4 children agents destined to nodes 3, 4, 9 and 10 respectively; each child agent's target resource is set to 6/4 as they share the parent agent's target resource. When the children agents reach their destined nodes, the target resource of each agent is updated to $3/2-1 = 1/2$, indicating nodes 3, 4, 9

and 10 are all valid resources (storage nodes). At node 9, the agent again forks into 2 child agents with target resource set to $\frac{1/2}{2} = 1/4$; these two agents are sent to node 1 and node 13. Agents at node 3 and node 10 proceed to node 2 and 8 respectively with their updated target resource 1/2. The agent at node 4 terminates because it has no more new nodes to explore. When the children agents reach nodes 1, 2, 8, and 13, their target resources are updated to negative values, indicating that they have found the desired target resources and no longer need to proceed. In this simple example, the search initiator successfully finds 9 resources $(11, 3, 4, 9, 10, 2, 1, 13, 8)$, satisfying the initial target resource of 7.

We now describe in more detail how the agent-based search is used by the *storage discovery*, *file discovery* and *processor discovery* procedures.

Storage discovery: Given a file encoded with parameter (k, n), the target resource is n storage nodes that satisfy the reliability constraint r_{min}. When an agent arrives at a node, the *collectInfo()* function exchanges information between the search agent and the visited node. In particular, the search agent needs to know if the resource on this node has been claimed by other agents of the same search task. The node also learns from the search agent the upstream nodes that this agent has visited. Other search agents arriving this node later will not dispatch children agents to those visited nodes again.

File discovery: Given a file encoded with parameter (k, n), the target resource is k data fragments. A node can provide a fragment resource only if this fragment belongs to the requested file, and the same fragment has not been discovered by this or other search agents in the upstream nodes. Because the same fragments may be cached on multiple nodes, each fragment should only be counted once towards the target resource. ***Processor discovery***: Given a job of M files to process, the goal is to find at least one processor node for each task. A node may process one or

Simulator		
Phy. Interfaces:WiFi, LTE	Comm. Range: WiFi($\leq$ 160m), LTE($\leq$ 1500m)	
Mobility: Rnd Waypoing	Moving Speed: 0-4m / sec.	Size: 50 nodes
Routing: AODV	File size: 2MB, (k,n)=(3,9)	Field: $800m^2$
Hardware Implementation		
Devices: Nexus 5, HTC One, Note 2, Galaxy S3...		Size: $\leq$12 nodes
Picture file $\leq$ 3MB, Video file $\leq$ 30MB		Field $300m^2$

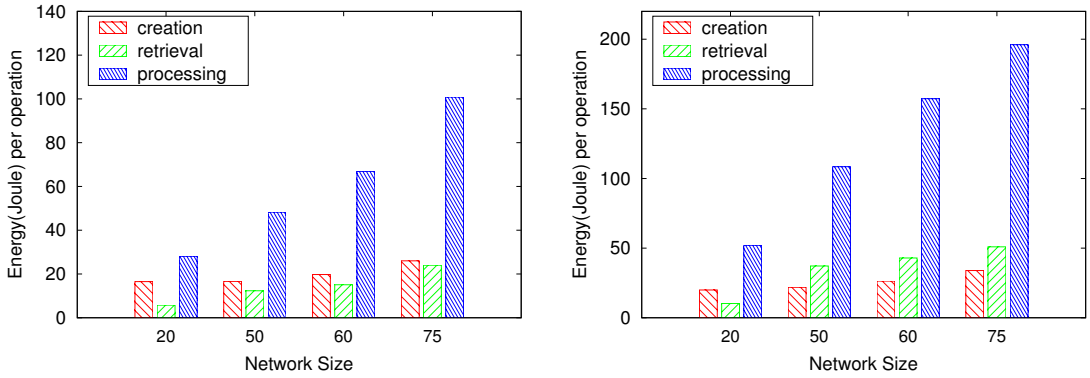
Table 6.1: MSPS Evaluation Settings.

more files depending on its state and capabilities. The target resource is M, and a node can provide a processing resource if it can retrieve and process a task without overloading itself. Besides giving the number of processing resource this node can provide, each node also informs the search agent the cached fragments it carry for the requested tasks. A node with cached fragments can have significantly lower task retrieval energy.

6.4 Evaluation

We evaluate our *MSPS* through extensive simulations on Jist/Swans [7] network simulator and through a real-world implementation on Android phones. In simulation, we are interested in the *energy efficiency, system-wide load imbalance, and the system lifetime* under various heterogeneous networks; in hardware-based evaluation, we want to know the feasibility of our algorithm and understand how it performs on modern smartphones. The default settings for simulator and hardware implementation are listed in Table 6.1.

We first look at the overall energy consumption of data operations (creation, retrieval, and processing) for different network sizes. The performance of *MSPS* is compared with a simple *Random* algorithm that selects storage nodes or processor nodes in a random manner. We then show how *MSPS* saves communication energy



(a) (b)

Figure 6.3: (a) Data operations in *MSPS*. (b) Data operation with random allocation.

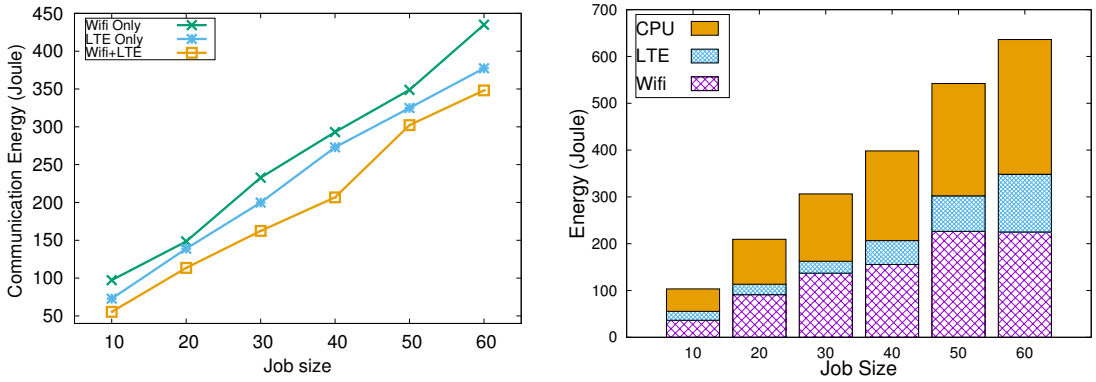
by utilizing multiple wireless interfaces (Wi-Fi and LTE). The load balance performance of *MSPS* is then studied under different heterogeneous networks that consist of nodes with different processing capabilities and battery capacities. The agent-based search algorithm is then benchmarked by measuring the number of resources that the search agent successfully finds and the number of packets the search algorithm sends during a resource discovery procedure. Finally, we demonstrate the feasibility of our algorithm by implementing an Android application *MediaShare* based on *MSPS* that shares and processes multimedia files (pictures and videos) on a group of smartphones. We evaluate and collect data of this application during 2015 Summer Institute on Flooding [19].

6.4.1 Energy Consumption of Data Operations

We first look at the energy consumption for each data operation for different network sizes. The energy is measured by the wireless interfaces on/off states in the MAC layer, so it includes the overhead for the entire network stack. Each node has two wireless interfaces, Wi-Fi and LTE, that can operate alternatively, but not simultaneously. Three types of nodes are considered: the high performance nodes

(HPC) have the largest battery capacity (10,000 mAh), CPU power (2 Watt), and processing throughput (1 MB/sec); the low performance nodes (LPC) have the lowest battery capacity (2,100 mAh), CPU power (0.5 Watt), and processing throughput (0.75 MB/sec); and the medium performance nodes (MPC) have all the hardware capabilities in-between HPC and LPC. In Figure 6.3, the Y-axis shows the cumulative energy of all nodes in the network for conducting a single data operation. Each 2MB file is encoded with $(k, n) = (3, 9)$, so a file creator needs to find at least 9 storage nodes and a file requester needs to find at least 3 data fragments. Each processing job processes 5 files stored in *MSPS*.

The energy consumption of data operations increases with the network size due to the energy overhead from the additional nodes. When the network density increases, the radio interference also causes lower throughput and thus higher communication consumption. In general, data creation consumes higher energy than data retrieval because each creation distributes 9 fragments while each retrieval downloads only 3 fragments. However, since data creator usually finds storage nodes in its nearby neighbors while data requester may retrieve fragments from storage nodes far away, their energy consumption difference is small. The performance of data processing is expected to be the highest because each task involves retrieve a file and process a file. We observe that the energy consumption of all three data operations increase almost linearly with the network size, which demonstrates the scalability of *MSPS* in larger networks. We also compare *MSPS* (Figure 6.3a) with a random allocation scheme (Figure 6.3b) in which storage nodes and processor nodes are selected in a random manner. *MSPS* outperforms the random scheme by at least 20-40%.



(a) (b)

Figure 6.4: (a) Comparison of using WiFi only, LTE only, or both for data processing. (b) Energy consumption of different components.

6.4.2 Effects of Communication Interfaces

In this section, we study the energy savings gained from using multiple communication interfaces. Figure 6.4 depicts the communication energy of various job sizes when using different communication interfaces. LTE consumes approximately $4-6$ times higher power than WiFi, but LTE takes advantages of longer communication range, more stable links, and higher throughput. Since our primary objective is to minimize energy consumption, *MSPS* prioritizes Wi-Fi when nodes are within short range. However, when two nodes are multiple hops away, the cumulative energy for sending, relaying, and receiving a packet in the Wi-Fi network may exceed the energy of using LTE network. Figure 6.4a shows that on average, using only Wi-Fi is the least energy efficient option. It is because in a network of 50 nodes spread across 800m^2 area, the hop-count distance between two nodes can be as high as 7 hops. It is also likely that the Wi-Fi network disconnects temporarily and some nodes become unreachable from others. Figure 6.4b shows the breakdown of energy consumption for processing a job. The fact that each individual energy consumption increases

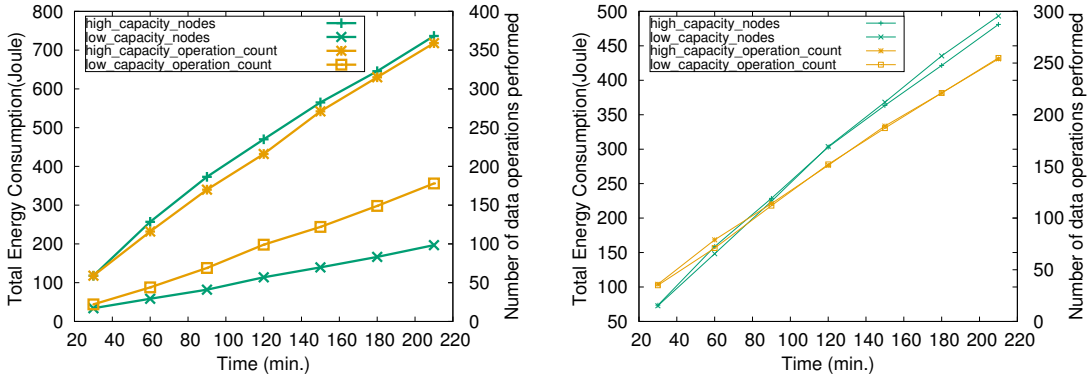


Figure 6.5: (a) Energy consumption versus number of tasks assigned on processor nodes with different energy capacities. (b) Same as (a), but without using standardized energy.

almost linearly with the job size shows that *MSPS* highly scalable.

6.4.3 Performance of the Load-Balance Algorithm

Figure 6.5 shows how *MSPS* allocates communication and processing tasks considering the energy capacities of each node. 50 nodes of three different types HPC, MPC, and LPC are deployed. Figure 6.5a shows the energy consumption and the number of tasks (send/receive fragments or process files) assigned to different types of nodes. The result shows that HPC nodes receive about 2 times more tasks and consume $3-4$ times more energy than the LPC nodes during the entire operation. This is our desired behavior as more tasks are pushed to nodes with higher energy or processing resources. Figure 6.5b shows the results from the same experiment without using *standardized energy*. This way, *MSPS* neglects the differences of energy capacities and evenly allocates tasks to each node. Note that the load-balance algorithm still ensures each node receives approximately the same workload.

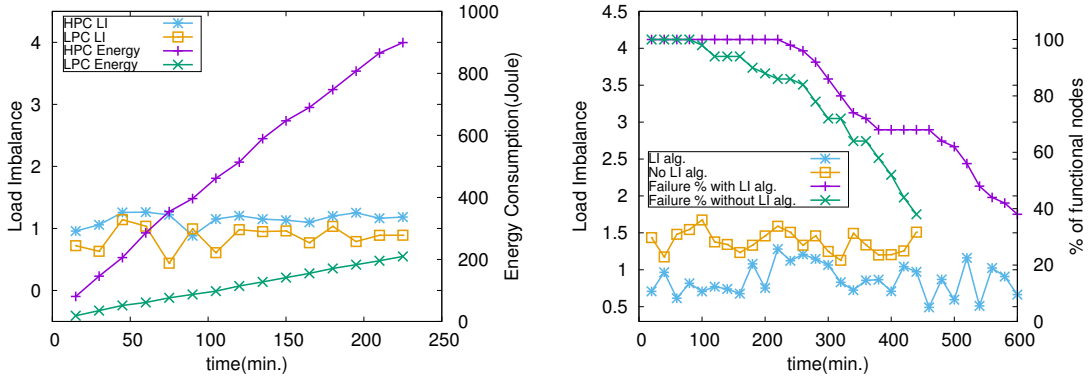
Figure 6.6 evaluates performance of the load-balancing algorithm. When allo-

cating a communication or processing task, *MSPS* considers the energy load (Equation 4.2), communication utilization (Equation 4.3), and CPU utilization (Equation 4.4). A node should not receive more tasks than it can handle, which causes system bottleneck and high delay; neither should a node be much busier than other nodes, which causes a network hotspot and harms the system lifetime. Figure 6.6a shows the load imbalance of high performance nodes (HPC) and low performance nodes (LPC) at different times. The thresholds values S^{LI}, S^{com}, and S^{cpu} values are all set to 0.5, meaning that *MSPS* tries to keep the communication utilization and CPU utilization around 50%. The load imbalance values of both HPC and LPC nodes stay around 1 most of the times.

In Figure 6.6b, we compare the load imbalance and the system lifetime between turning on and turning off the load-imbalance algorithm (set $g(i,t) = 0$ in Equation 6.12). We declare a system failed when more than 50% nodes have failed due to depleted energy. The figure shows that our load-imbalance algorithm not only reduces the system-wide load imbalance by 30-50%, but that it also extends the system lifetime by 30%.

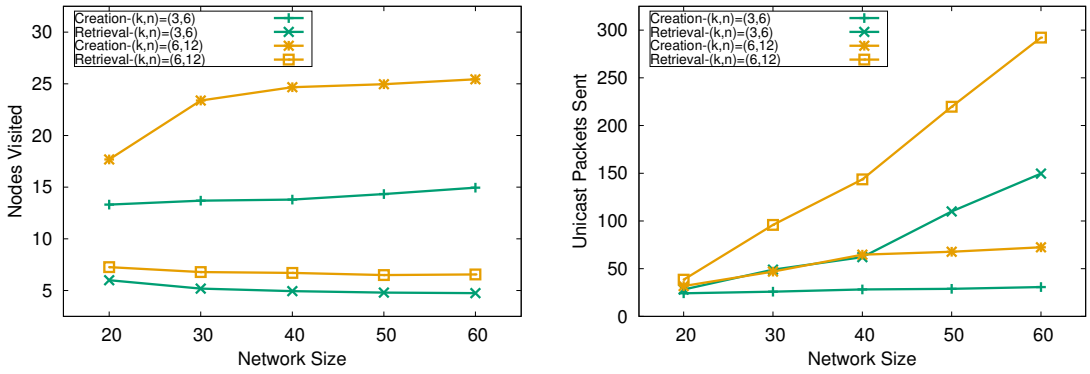
6.4.4 Performance of the Agent-Based Search Algorithm

Agent-based search is used for disseminating data operation requests and exploring the storage nodes, data fragments, and processor nodes. Although LTE broadcast can immediately reach all nodes in the field, the purpose of the agent-based search is to limit the scale of search requests and reduces the impact on network traffic and communication energy. When $(k, n) = (3, 6)$, a *storage discovery* searches for at least 6 storages node and a *file discovery* needs to find at least 3 data fragments. The Y-axis of Figure 6.7a shows the number of resources that a search agent found at the end of a searching procedure. For data creation, the number of discovered resources



(a) (b)

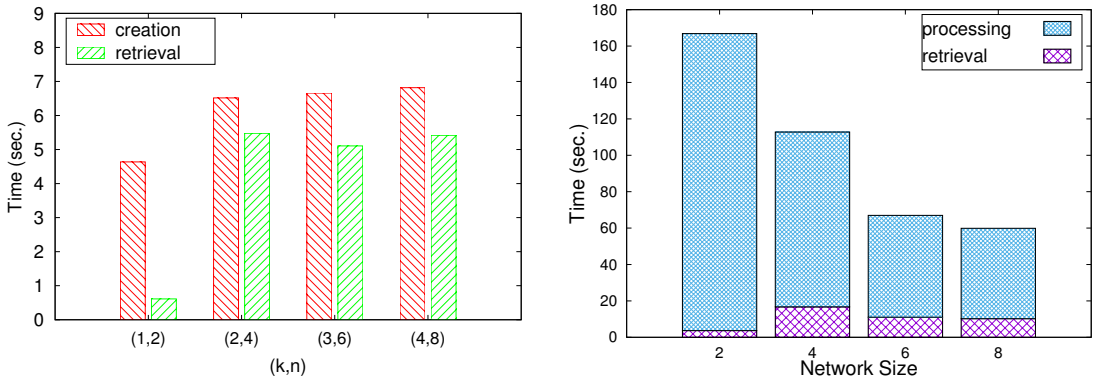
Figure 6.6: (a) Energy consumption and Load Imbalance on different types of nodes. (b) Load Imbalance and percentage of functional nodes.



(a) (b)

Figure 6.7: (a) Number of resources discovered by a search agent. (b) Packets sent during a searching procedure.

is about two times that of the n value because we increase the target resource by a scale of two so that the creator can choose more reliable storage from a larger group of nodes. Figure 6.7b shows the total number of unicast packets sent during a search request, i.e., the number of search agents dispatched from all nodes. Data creation induces much less traffic because it simply discovers the nodes around the file creator.



(a) (b)

Figure 6.8: (a) Running time of data operations under different (k,n) settings. (b) Data processing time in different network size.

Data retrieval agent, however, needs to explore further in order to find the desired fragments. The result shows that the number of packets sent is approximately linear to the network size, which indicates the agent-based search algorithm is scalable to larger or denser network.

6.4.5 Hardware Implementation

To understand the feasibility and performance of our algorithm in real hardware, we implemented *MSPS* on Android smartphones and created a *MediaShare* application that uses the *MSPS* framework to share and process multimedia files. The processing function of *MediaShare* extracts image frames from the stored videos and pictures that contain human faces. The processing function needs to decode video, extracts video frames, and performs facial recognition on each frame. Each video is sampled at 2Hz (extract 2 frames per second). The facial recognition library can process one image frame in 0.5–1.5 seconds depending on the processor speed. The MSPS framework and MediaShare application each contains about 9,000 and 2,000 lines of Java code, respectively. Both projects were developed on Android SDK 4.2.2

.

During 2015 Summer Institute on Flooding [19], 10 participants used *MediaShare* App to share and process media files in a simulated disaster environment. The App was installed on at least 5 different Android smartphones as listed in Table 6.1. Figure 6.8a shows the average time for creating and retrieving a 3MB file using different (k, n) settings. The value n here also indicates the network size. As expected, data creation takes longer than data retrieval because a file creator needs to distribute more data fragments that a file requester needs to retrieve. Figure 6.8b shows the data retrieval time and CPU processing time for analyzing eleven 30 seconds 3MB video files. Note that the data retrieval time is extremely low in 2 nodes network because each node can simply recover the files directly from its local stored fragment when $(k, n) = (1, 2)$. The overall data processing time reduces as more nodes join and provide processing resources. From the user feedback and these performance results, we are confident that *MSPS* is efficient and practical on real hardware.

7. CONCLUSIONS

In this section, we present the conclusions of this dissertation and propose ideas for future work.

7.1 Conclusions

This research investigates the challenges and solutions for deploying a cloud computing platform consisting of entirely mobile devices. Different from the traditional cloud computing in which clients access services, software, or infrastructure from remote servers, this type of cloud (i.e., Mobile Cloud) does not rely on the network infrastructure (Internet) or remote servers. The infrastructureless and autonomous characteristics make this type of mobile cloud attractive to applications operating in environments where the network infrastructure is unavailable, e.g., disaster relief, battlefield, or crowded event. Specifically, we are interested in bringing big data services (data storage and data processing) to mobile clouds to enable data sharing applications, video/image processing applications, or map-reduce type applications. Potentially, existing data storage and data processing applications can be quickly migrated to mobile environments using this mobile cloud computing (MCC) framework.

Although mobile technologies are advancing in every aspect, there are still many challenges in realizing big data services in such mobile cloud. In particular, limited energy, low reliability, and heterogeneity are the challenges that we need to overcome. This research proposes the k-out-of-n MCC framework that provides distributed data storage and data processing services to applications in an energy-efficient, fault-tolerant, and load-balanced manner. By intelligently allocating data and scheduling the processing tasks, the algorithm minimizes the system-wide communication

energy and processing energy while meeting the fault-tolerant and load-balanced requirements. Through extensive simulations and real-world implementations, we show that the k-out-of-n MCC framework effectively meet all the expectations and is feasible on real hardware.

7.2 Future Work

In this section, we present several potential research directions, as follows:

Stream data processing. This research has focused on the batch data processing. There is, however, a trend and need for real-time stream data processing, e.g., Apache Storm [97], Apache Spark [106]. A stream data processing framework allows continuous data source such as video, audio, or sensor readings to be processed in real-time.

Secure communication and processing. The k-out-of-n MCC framework assumes all the participating nodes are trustworthy and allows any node to retrieve and process any data block. Because the data needs to be decoded and decrypted before processing, it poses security concern if malicious nodes intentionally temper or steal the data. A security mechanism that allows only authorized applications/users to decode the data while still allowing unauthorized users to contribute the computation/storage resources is necessary.

Delay-tolerant network. All the data operations in the k-out-of-n MCC framework fail if the required data fragments or processors are not immediately available through the network. As a result, unstable wireless links are the major causes of the unsuccessful data operations. Making the data operations and the underlying routing protocol delay-tolerant can further improve the energy-efficiency and reliability of the framework.

Mutable k-out-of-n storage. The current k-out-of-n storage supports only immutable data, i.e., once a file is created, it can no longer be modified. A mutable data storage framework enables a new class of applications such as document collaboration tools. One major challenge is to ensure data consistency across all nodes in the mobile cloud. Each data block has multiple fragments and caches are stored in different locations. Consequently, any update from a client needs to be propagated to all these fragments.

Cooperative communication. In a heterogeneous mobile cloud, nodes may have different cost or throughput on a specific wireless interface. For example, suppose node A has high download throughput but low upload throughput on cellular provider `X` while node B has low download throughput but high upload throughput on cellular provider `Y`. If node A and node B also have stable Wi-Fi or Bluetooth communication channel, they can cooperate to improve each individual's performance. An algorithm that considers the link quality and queuing effect of the communication tasks is necessary.

Mobile code task offloading. The current k-out-of-n data processing relies on remote procedure call (RPC) or remote method invocation (RMI). It is inconvenient because processing functions need to be pre-installed on the processor nodes, which makes the framework less flexible. We can exploit the mobile code technique to allow clients to send processors executable code at runtime. In such manner, processor nodes do not need to keep all the processing functions and any node can provide data processing services to any mobile application.

REFERENCES

[1] M.K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *Proc. of DSN*, 2005.

[2] M. Alicherry and T.V. Lakshman. Network aware resource allocation in distributed clouds. In *Proc. of INFOCOM*, 2012.

[3] Salah Aly, Zhenning Kong, Emina Soljanin, et al. Raptor codes based distributed storage algorithms for wireless sensor networks. In *Information Theory, 2008. ISIT 2008. IEEE International Symposium on*, pages 2051–2055. IEEE, 2008.

[4] Frank-Uwe Andersen, Hermann de Meer, Ivan Dedinski, Cornelia Kappler, Andreas Mäder, Jens O Oberender, and Kurt Tutschku. An architecture concept for mobile p2p file sharing services. *GI Jahrestagung (2)*, 51:229–233, 2004.

[5] UIMA Apache. Apache software foundation. *URL http://java. apache. org*, 2011.

[6] Rajesh Krishna Balan, Darren Gergle, Mahadev Satyanarayanan, and James Herbsleb. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 272–285. ACM, 2007.

[7] Rimon Barr, Zygmunt J Haas, and Robbert van Renesse. Jist: An efficient approach to simulation using virtual machines. *Software: Practice and Experience*, 2005.

[8] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud

computing. *Future Generation Computer Systems*, 28:755 – 768, 2012.

[9] William J Bolosky, John R Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *ACM SIGMETRICS Performance Evaluation Review*, volume 28, pages 34–43. ACM, 2000.

[10] Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Recent advances in checkpoint/recovery systems. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006.

[11] Francesco Calabrese, Francisco C Pereira, Giusy Di Lorenzo, Liang Liu, and Carlo Ratti. The geography of taste: analyzing cell-phone mobility and social events. In *Pervasive computing*, pages 22–37. Springer, 2010.

[12] 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. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[13] Chien An Chen, Myounggyu Won, R. Stoleru, and G.G. Xie. Resource allocation for energy efficient k-out-of-n system in mobile ad hoc networks. In *Proc. ICCCN*, 2013.

[14] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proc. of EuroSys*, 2011.

[15] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual

machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[16] David W. Coit and Jiachen Liu. System reliability optimization with k-out-of-n subsystems. *International Journal of Reliability, Quality and Safety Engineering*, 7(2):129–142, 2000.

[17] Maria João Cortinhal and Maria Eugenia Captivo. Upper and lower bounds for the single source capacitated location problem. *European Journal of Operational Research*, 151, 2003.

[18] Douglas S. J. De Couto. *High-Throughput Routing for Multi-Hop Wireless Networks*. PhD dissertation, MIT, 2004.

[19] CRASAR. 2015 summer institute on flooding. hidden for blind review, July 2015.

[20] Manuel Crotti, Diego Ferri, Francesco Gringoli, Manuel Peli, and Luca Salgarelli. Pp2db: A privacy-preserving, p2p-based scalable storage system for mobile networks. In *Security and Privacy in Communication Networks*, pages 533–542. Springer, 2012.

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

[22] Douglas SJ De Couto, Daniel Aguayo, John Bicket, and Robert Morris. A high-throughput path metric for multi-hop wireless routing. *Wrless Netws.*, 11, 2005.

[23] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Su. A survey on network codes for distributed storage. *Proc. of the IEEE*, 99(3):476–489, 2010.

[24] A.G. Dimakis, V. Prabhakaran, and K. Ramchandran. Decentralized erasure codes for distributed networked storage. *Information Theory, IEEE Transactions on*, june 2006.

[25] A.G. Dimakis and K. Ramchandran. Network coding for distributed storage in wireless networks. In *Networked Sensing Information and Control*, pages 115–136. Springer, New York, 2008.

[26] PeterR. Elespuru, Sagun Shakya, and Shivakant Mishra. Mapreduce system over heterogeneous mobile devices. In *Software Technologies for Embedded and Ubiquitous Systems*. Springer, 2009.

[27] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[28] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu. Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(1):84–106, 2013.

[29] Apache Software Foundation. Apache hadoop. http://hadoop.apache.org/.

[30] Apache Software Foundation. Spark streaming programming guide. Accessed: 09/24/2015.

[31] Johnu George, Chien-An Chen, Radu Stoleru, Geoffrey G Xie, Tamim Sookoor, and David Bruno. Hadoop mapreduce for tactical clouds. In *Proc. of CloudNet*, 2014.

[32] S.M. George, Wei Zhou, H. Chenji, Myounggyu Won, Yong Oh Lee, A. Pazarloglou, R. Stoleru, and P. Barooah. Distressnet: a wireless ad hoc and sensor

network architecture for situation management in disaster response. *Communications Magazine, IEEE*, 2010.

[33] Stephen M George et. al. Distressnet: a wireless ad hoc and sensor network architecture for situation management in disaster response. *Communications Magazine, IEEE*, 2010.

[34] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proc of SOSP*, 2003.

[35] Diptesh Ghosh. Neighborhood search heuristics for the uncapacitated facility location problem. *EJOR*, 150, 2003.

[36] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13, 1986. Tabu Search first paper.

[37] Robert S. Gray et. al. CRAWDAD data set dartmouth/outdoor. Downloaded from http://crawdad.org/dartmouth/outdoor/, November 2006.

[38] JIN Hai, Weizhong QIANG, and ZOU Deqing. Dric: dependable grid computing framework. *IEICE transactions on information and systems*, 89(2):612–623, 2006.

[39] Oliver Heckmann and Axel Bock. The edonkey 2000 protocol. *Rapport technique, Multimedia Communications Lab, Darmstadt University of Technology*, 13, 2002.

[40] N Hemming. Kazaa. *Web Site-www. kazaa. com*, 2001.

[41] John H Howard, Michael L Kazar, Sherri G Menees, David A Nichols, Mahadev Satyanarayanan, Robert N Sidebotham, and Michael J West. Scale and perfor-

mance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[42] Dijiang Huang, Xinwen Zhang, Myong Kang, and Jim Luo. MobiCloud: Building secure cloud framework for mobile computing andcommunication. In *Proc. of SOSE*, 2010.

[43] Dijiang Huang, Zhibin Zhou, Le Xu, Tianyi Xing, and Yunji Zhong. Secure data processing framework for mobile cloud computing. In *INFOCOM WKSHPS*, 2011.

[44] Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 225–238. ACM, 2012.

[45] Jean-Pierre Hubaux, Levente Buttyán, and Srdan Capkun. The quest for security in mobile ad hoc networks. In *Proc. of MobiHoc*, 2001.

[46] Scott Huchton, Geoffrey Xie, and Robert Beverly. Building and evaluating a k-resilient mobile distributed file system resistant to device compromise. In *Proc. of MILCOM*, 2011.

[47] Eduardo Huedo, Ruben S Montero, and Ignacio M Llorente. A framework for adaptive execution in grids. *Software-Practice and Experience*, 34(7):631–652, 2004.

[48] Eduardo Huedo, Rubén S Montero, and Ignacio M Llorente. Evaluating the reliability of computational grids from the end users point of view. *Journal of Systems Architecture*, 52(12):727–736, 2006.

[49] Gonzalo Huerta-Canepa and Dongman Lee. A virtual cloud computing provider for mobile devices. In *Proc. of the Workshop on MCS*, 2010.

[50] Soonwook Hwang and Carl Kesselman. A flexible framework for fault tolerance in the grid. *Journal of Grid Computing*, 1(3):251–272, 2003.

[51] Jang-uk In, Paul Avery, Richard Cavanaugh, Laukik Chitnis, Mandar Kulkarni, and Sanjay Ranka. Sphinx: A fault-tolerant system for scheduling in dynamic grid environments. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 12b–12b. IEEE, 2005.

[52] Shudong Jin and Limin Wang. Content and service replication strategies in multi-hop wireless mesh networks. In *Proceedings of the 8th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 79–86, 2005.

[53] Hideyuki Jitsumoto, Toshio Endo, and Satoshi Matsuoka. Abaris: An adaptable fault detection/recovery component framework for mpis. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[54] T. Kakantousis, I. Boutsis, V. Kalogeraki, D. Gunopulos, G. Gasparis, and A. Dou. Misco: A system for data analysis applications on networks of smartphones using mapreduce. In *Proc. Mobile Data Management*, 2012.

[55] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a computation offloading framework for smartphones. In *Mobile Computing, Applications, and Services*, pages 59–79. Springer, 2012.

[56] Ab Rouf Khan, Marini Othman, Sajjad Ahmad Madani, and Samee U Khan. A survey of mobile cloud computing application models. *Communications*

Surveys & Tutorials, IEEE, 16(1):393–413, 2014.

[57] Kyu-Han Kim, Sung-Ju Lee, and Paul Congdon. On cloud-centric network architecture for multi-dimensional mobility. *ACM SIGCOMM Computer Communication Review*, 42(4):509–514, 2012.

[58] S. Kosta, A. Aucinas, Pan Hui, R. Mortier, and Xinwen Zhang. ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proc. of INFOCOM*, 2012.

[59] Sokol Kosta, Vasile Claudiu Perta, Julinda Stefa, Pan Hui, and Alessandro Mei. Clone2clone (c2c): Peer-to-peer networking of smartphones on the cloud. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud13)*, 2013.

[60] Emmanouil Koukoumidis, Dimitrios Lymberopoulos, Karin Strauss, Jie Liu, and Doug Burger. Pocket cloudlets. *ACM SIGARCH Computer Architecture News*, 39(1):171–184, 2011.

[61] Mads Darø Kristensen. Scavenger: Transparent development of efficient cyber foraging applications. In *PerCom*, 2010.

[62] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[63] Ming Lei, Susan V Vrbsky, and Q Zijie. Online grid replication optimizers to improve system reliability. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[64] A. Leon-Garcia. *Probability, Statistics, and Random Processes for Electrical Engineering*. Prentice Hall, 2008.

[65] D. Leong, A. G. Dimakis, and T. Ho. Distributed storage allocation for high reliability. In *Proc. of ICC*, 2010.

[66] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys (CSUR)*, 22(4):321–374, 1990.

[67] Antonios Litke, Dimitrios Skoutas, Konstantinos Tserpes, and Theodora Varvarigou. Efficient task replication and management for adaptive fault tolerance in mobile grid environments. *Future Generation Computer Systems*, 23(2):163–178, 2007.

[68] Cong Liu, Xiao Qin, S. Kulkarni, Chengjun Wang, Shuang Li, A. Manzanares, and S. Baskiyar. Distributed energy-efficient scheduling for data-intensive applications with deadline constraints on data grids. In *Proc. of IPCCC*, 2008.

[69] Yadi Ma, Thyaga Nandagopal, Krishna PN Puttaswamy, and Suman Banerjee. An ensemble of replication and erasure codes for cloud file systems. In *Proceedings of the IEEE International Conference on Computer Communications*, pages 1276–1284, 2013.

[70] Balakrishan S Manoj and Alexandra Hubenko Baker. Communication challenges in emergency response. *Communications of the ACM*, 50(3):51–53, 2007.

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

[72] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. P2p-mapreduce: Parallel data processing in dynamic cloud environments. *J. Comput. Syst. Sci.*, 2012.

[73] Brendan McGarry. Army set to introduce smartphones into combat. http://www.military.com/, March 2013.

[74] Alex F Mills and James H Anderson. A stochastic framework for multiprocessor soft real-time scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 311–320. IEEE, 2010.

[75] Abderrahmen Mtibaa, Afnan Fahim, Khaled A Harras, and Mostafa H Ammar. Towards resource sharing in mobile device clouds: Power balancing across mobile devices. In *Proc. of ACM SIGCOMM workshop*, 2013.

[76] David Nagle, Denis Serenyi, and Abbie Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53. IEEE Computer Society, 2004.

[77] LLC Napster. Napster. *URL: http://www. napster. com*, 2001.

[78] Dirk Neumann, Christian Bodenstein, Omer F Rana, and Ruby Krishnaswamy. STACEE: enhancing storage clouds using edge devices. In *WACE*, 2011.

[79] JamesB. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78, 1997.

[80] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: A system for migrating computing environments. *ACM SIGOPS Operating Systems Review*, 36(SI):361–376, 2002.

[81] Rajesh Krishna Panta, Rittwik Jana, Fan Cheng, Yih-Farn Robin Chen, and Vinay A Vaishampayan. Phoenix: Storage using an autonomous mobile infrastructure. *TPDS*, 2013.

[82] Rafael Pereira, Marcello Azambuja, Karin Breitman, and Markus Endler. An architecture for distributed high performance video processing in the cloud. In *Proc. of CLOUD*, 2010.

[83] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *Peer-to-Peer Systems IV*, pages 205–216. Springer, 2005.

[84] Korlakai Vinayak Rashmi, Nihar B Shah, and P Vijay Kumar. Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction. *Information Theory, IEEE Transactions on*, 57(8):5227–5239, 2011.

[85] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 99–100. IEEE, 2001.

[86] S Siva Sathya, S Kuppuswami, and K Syam Babu. Fault tolerance by check-pointing mechanisms in grid computing. In *Proceedings of the International Conference on Global Software Development, Coimbatore*, pages 26–28, 2007.

[87] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for VM-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8:14–23, 2009.

[88] Mahadev Satyanarayanan, James J Kistler, Puneet Kumar, Maria E Okasaki, Ellen H Siegel, and David C Steere. Coda: A highly available file system for a distributed workstation environment. *Computers, IEEE Transactions on*, 39(4):447–459, 1990.

[89] Texas A&M Engineering Extension Service. Disaster preparedness and response. Accessed: 26/10/2012.

[90] Muhammad Zubair Shafiq, Lusheng Ji, Alex X Liu, Jeffrey Pang, Shobha Venkataraman, and Jia Wang. A first look at cellular network performance during crowded events. In *ACM SIGMETRICS Performance Evaluation Review*, volume 41, pages 17–28. ACM, 2013.

[91] Cong Shi, Vasileios Lakafosis, Mostafa H. Ammar, and Ellen W. Zegura. Serendipity: enabling remote computing among intermittently connected mobile devices. In *Proc. of MobiHoc*, 2012.

[92] D Shires, B Henz, S Park, and J Clarke. Cloudlet seeding: Spatial deployment for high performance tactical clouds. In *Proc. of WorldComp*, 2012.

[93] Clay Shirky, Kelly Truelove, Rael Dornfest, L Gonze, and D Dougherty. P2p networking overview. *The Emergent P2P Platform of Presence, Identity, and Edge Resources. O'Reilly & Associates*, 2001.

[94] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proc of MSST*, 2010.

[95] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[96] Patrick Stuedi, Iqbal Mohomed, and Doug Terry. WhereStore: location-based data storage for mobile devices interacting with the cloud. In *MCS*, 2010.

[97] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIG-*

MOD international conference on Management of data, pages 147–156. ACM, 2014.

[98] Paul Townend and Jie Xu. Replication-based fault tolerance in a grid environment. In *UK e-Science 3rd All-Hands Meeting*, 2004.

[99] Luca Valcarenghi and Piero Castoldi. Qos-aware connection resilience for network-aware grid computing fault tolerance. In *Transparent Optical Networks, 2005, Proceedings of 2005 7th International Conference*, volume 1, pages 417–422. IEEE, 2005.

[100] Hakim Weatherspoon and John D Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Peer-to-Peer Systems*, pages 328–337. Springer, 2002.

[101] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[102] Ye Wen, Rich Wolski, and Chandra Krintz. Online prediction of battery lifetime for embedded and mobile devices. In *Power-Aware Computer Systems*. Springer Berlin Heidelberg, 2005.

[103] Gosia Wrzesińska, Rob V Van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E Bal. Fault-tolerant scheduling of fine-grained tasks in grid environments. *International Journal of High Performance Computing Applications*, 20(1):103–114, 2006.

[104] Hao Yang, Haiyun Luo, Fan Ye, Songwu Lu, and Lixia Zhang. Security in mobile ad hoc networks: challenges and solutions. *Wireless Communications,*

IEEE, 2004.

[105] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. *ACM SIGOPS Operating Systems Review*, 37, 2003.

[106] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[107] Xianan Zhang, Flavio Junqueira, Keith Marzullo, Richard D Schlichting, and Richard D Schlichting. Replicating nondeterministic services on grid environments. In *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 105–116. IEEE, 2006.

[108] Zehua Zhang and Xuejie Zhang. Realization of open cloud computing federation based on mobile agent. In *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, volume 3, pages 642–646. IEEE, 2009.