Energy-efficient Load-balanced Heterogeneous Mobile Cloud

Chen, Chien-An; Stoleru, Radu; Xie, Geoffrey G.

IEEE

Chen, Chien-An, Radu Stoleru, and Geoffery G. Xie. "Energy-Efficient Load-Balanced Heterogeneous Mobile Cloud." Computer Communication and Networks (ICCCN), 2017 26th International Conference on. IEEE, 2017.

https://hdl.handle.net/10945/61024

Energy-efficient Load-balanced Heterogeneous Mobile Cloud

Chien-An Chen, Radu Stoleru[†], Geoffrey G Xie[‡]
Accenture Technology Lab
[†]Department of Computer Science and Engineering, Texas A&M University
[‡]Department of Computer Science, Naval Postgraduate School
jay.chen@accenture.com, stoleru@cse.tamu.edu, xie@nps.edu

***Abstract*—Today's integration of mobile technologies and traditional cloud computing exploits the abundant computation and storage resources in the cloud, to enhance the capabilities of end-user mobile devices. The designs that rely on remote cloud services, however, sometimes overlook the abundant resources (e.g., storage, communication, and computation) on mobile devices. In particular, when the remote cloud services are unavailable (due to service downtime or network issues), these smart devices can no longer function. We propose a Heterogeneous Mobile Cloud (HMC) computing design that efficiently utilizes the communication and computation resources to support data storage and data processing services in a group of mobile devices. Each mobile device may have different energy, communication and computation capabilities, but our Mobile Storage & Processing System (*MSPS*) ensures that: i) the communication and computation tasks are executed in an energy-efficient manner, ii) task allocation considers device heterogeneity and achieves system-wide load balancing, and iii) the stored data are fault-tolerant. Through extensive simulations and real hardware implementations on Android devices, we demonstrate the performance and feasibility of deploying *MSPS* in a real heterogeneous mobile environment.**

I. Introduction

Due to the popularity of smart devices and the rapid advance in mobile technologies, Mobile Cloud Computing (MCC) has attracted enormous interest from industry, academia, and even military. Mobile cloud is defined as an integration of cloud computing with mobile devices that makes mobile devices more resourceful in terms of energy, computation and context awareness. Depending on the application, environment, and requirement, the scale of a mobile cloud can range from several mobile devices in a local area network to millions of mobile devices in the internet. In this paper, we consider a mobile cloud deployed in restricted areas where the internet is unavailable and the cloud is self-organized by a collection of mobile devices. Nodes may have various hardware specifications and their communication and computation capabilities may also be different. We refer to this type of cloud as a *Heterogeneous Mobile Cloud (HMC)*. HMC is of particular interest for military tactical cloud systems [1] and civilian use cases such as disaster relief or mining operations in remote areas [2] where the network is dynamic and the internet is unavailable.

Many existing works have considered data storage and data processing in mobile clouds. The *Remote Cloud Server* architecture offloads data and computation to remote data centers, e.g, Google Drive and iCloud; these cloud services rely on stable and constantly available internet connection. The *Virtual Resource Cloud* architecture creates a computing platform on a collection of mobile devices [3] [4] [5] [6] [7]; these solutions use mobile devices for job execution or data storage; in particular, [8] and [9] optimize energy and reliability for data storage and data processing in an ad-hoc mobile cloud. However, most existing virtual resource cloud solutions can neither scale up to larger network nor adapt in HMC environments. In a heterogeneous setting where nodes have various hardware specifications, i.e., different energy capacity, processing speed, and communication interfaces, it is difficult to efficiently utilize and allocate resources considering the various capabilities of each node. In this paper, we study an *energy-efficient, load-balanced, and fault-tolerant* distributed data storage and data processing middleware for HMC. We propose a distributed algorithm that allocates *data, computation, and communication* resources adaptively in a heterogeneous network. Energy efficiency ensures that the middleware minimizes the system-wide energy consumption, fault tolerance ensures that the stored data are resilient to node failures, and load balancing ensures that each node is allocated proper workload according to its available resources. We refer to this Mobile Storage & Processing System as *MSPS*.

MSPS greedily minimizes the *standardized energy* consumption for executing each operation 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* on each node. *Consumed energy* is the total energy used for wireless communication or CPU processing since *MSPS* starts; *energy capacity* is the amount of energy that the mobile device can store. When measuring the energy that a node has contributed to *MSPS*, standardized energy quantify the contribution based on the capability of each node. As an example, when a car powered tablet and a battery powered tablet both process a same task, although they consume the same amount of absolute energy, *MSPS* considers that the car powered tablet consumes less standardized energy due to its high energy capacity. *MSPS* tries to dissipate energy of each node at approximately the same rate such that no node runs out of energy much earlier than others and the system maintains the maximum number of active nodes at any time.

The *load* of a node is defined as the standardized energy consumption within a period of time, and the *load imbalance* of the network measures how uneven the load of each node is. Any communication or computation task executing on a node directly affects the node's load. *MSPS* heuristically reduces the *load imbalance* to prolong the system operational time. The contribution of this paper is as follows: **i)** we propose a scalable distributed algorithm for data storage & processing in heterogeneous mobile clouds; **ii)** the load balancing algorithm effectively and efficiently reduces system-wide *load imbalance* and prolongs the system operational time; **iii)** the agent-based search algorithm efficiently explores and discovers computation and storage resources in the network; **iv)** the proposed algorithm is evaluated extensively in a network simulator and in a real-world hardware implementation on Android devices.

II. Background and Related Work

Several distributed data processing frameworks for mobile cloud have been developed. In 2009, Marinelli introduced a Hadoop based platform Hyrax [10] for distributed data processing on smartphones. In particular, the Hadoop TaskTracker and DataNode processes were ported to Android phones. Later, Huerta-Canepa and Lee proposed a virtual cloud computing framework [11] targeting an ad-hoc network 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. Meanwhile, *MobiCloud* [12] and the work of Huang et al. [7] focused on security issues unique to ad-hoc mobile cloud, such as trust and risk management, private data isolation, and secure routing. Finally, *Scavenger* [13] 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. Different from most of the existing works, we study the energy efficiency, load balancing, and fault tolerance in an integrated manner in mobile clouds.

Huchton et al. [14] were the first to introduce the concept of k-out-of-n reliability to a mobile cloud setting while aiming primarily for military operations. Chen et al. [8] [15] later proposed several generalizations to the concept and a new resource allocation scheme to improve energy efficiency. They formulated an optimal resource allocation problem and solved the problem in a centralized manner. Several assumptions such as file request pattern and homogeneous network were made. [16] proposed a caching framework for k-out-of-n data storage that selects data fragments and caching locations based on each file's popularity. None of these prior efforts, however, targets large scale heterogeneous networks. Table I compares the pros and cons of the centralized solutions in [8] [15] [16] and the distributed solution proposed in this work. Overall, our distributed solution is scalable to larger network, has lower time and space complexity, and adapts in heterogeneous network. The centralized solution, under certain assumptions (e.g., homogeneous network and file request pattern), can obtain better solution, but it can hardly scale up to larger network or heterogeneous network due to the complexity of the solver.

	Scalability	**Complexity**	**Quality**	**Hetero**
Centralized	Low	High	High	No
Distributed	High	Low	Medium	Yes

TABLE I
Centralized vs Distributed Mobile Cloud Solutions

III. System Architecture and Problem Formulation

We consider a heterogeneous mobile cloud where each node may have different hardware specification or energy capacity. Nodes can 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 the file using *Reed-Solomon code* in which a file is encoded into n data fragments, and any subset of k data fragments together can recover the original file; data fragments are then sent to a set of storage nodes. When another node needs to read the file, it searches and retrieves k of the n data fragments from the network to recover the original file. (k, n) is referred as *storage parameter*. This coding scheme ensures the stored data is fault-tolerant. Any node can submit a job to process a subset of stored files. A processing job consists of multiple independent tasks where each task corresponds to processing a single file on a selected processor node.

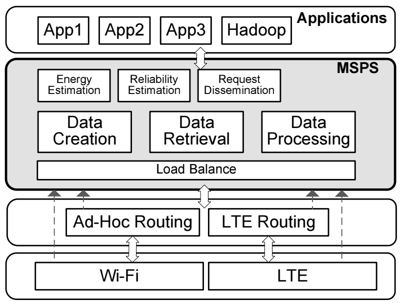


Fig. 1. *MSPS* System Architecture

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

Fig. 1 illustrates the system architecture. *MSPS* serves as a middleware that provides applications data storage and data processing services. *MSPS* accesses routing information and link quality information from network layer and MAC layer. The three major data operations, *data creation*, *data retrieval*, and *data processing*, access this cross-layer information (e.g., routing table and link quality) when allocating communication and processing tasks. As a proof of concept, Wi-Fi and LTE are the only two communication interfaces considered in our experiment. The *Energy Estimation* component estimates the

$$X_{opt} = \underset{X \subset \hat{X}}{\arg\min} \sum_{\forall x_i \in X} E^{std}_{x_i}(task, t) \quad (1)$$

Subject to:

$$\frac{L_{x_i}(t)}{L_{\bar{\mu}}(t)} - 1 \leq S^{LI} \; \forall x_i \in X \quad (2)$$

$$U^{com_j}_{x_i}(t) \leq S^{com} \; \forall x_i \in X \quad (3)$$

$$U^{cpu}_{x_i}(t) \leq S^{cpu} \; \forall x_i \in X \quad (4)$$

energy for transferring data or processing tasks based on the information such as file size, available wireless interfaces, routing table, link quality, and device profile. The *Reliability Estimation* component estimates the reliability of a node based on its remaining energy, connectivity, and application-dependent factors. The *reliability* of a node is the probability that the node remains accessible by all other nodes from the current time t to time $t + T_{rel}$ where T_{rel} is he period in which the reliability is concerned. The reliability estimation procedure can be found in [8]. The *Load balancing* component monitors the utilization rate of communication, CPU, and energy resources on each node and tries to maintain a low system-wide load imbalance. The *Request Dissemination* component uses agent-based search algorithm to discover storage nodes, file fragments, or processor nodes in the network when performing a data operation. The detail of each component will be explained in the next section.

We now formulate *MSPS* as an abstract optimization problem in Eqs. 1–4. The *energy load* $L_{x_i}(t)$ of node x_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 Eq.5 where $L_{\overline{15}}(t))$ is the mean load of top 15% of the nodes, and $L_{\bar{\mu}}(t)$ is the mean load of all nodes. The objective is to minimize the standardized energy consumption constrained on the utilization rate of energy, communication, and CPU resources. This optimization problem is solved for each data operation.

$$LI(t) = \frac{L_{\overline{15}}(t)}{L_{\bar{\mu}}(t)} - 1 \quad (5)$$

$\hat{X}$ is the set of nodes discovered by the request dissemination procedure. Given a data operation, the objective function (Eq. 1) finds a subset of nodes $X = \{x_1, x_2, ...\}$ in $\hat{X}$ to perform tasks of this operation such that the cumulative standardized energy $\sum_{\forall x_i \in X} E^{std}_{x_i}(task, t)$ for executing the operation is minimized. Using the data creation task shown in Fig. 2 as an example, node 11 selects nodes 2, 3, 9, 10, and 11 to execute the data creation operation. The task for each selected node is to transfer data or store data fragment. Because *MSPS* does not assume global information, the optimization problem is considered in a *local sense*, meaning that the subset X_{opt} is selected only from the nodes discovered by the *request dissemination* procedure.

The tasks of an operation may require the participating 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 utilization rates. Eq. 2 indicates that a selected node should not have an energy load higher than the network average by ratio of S^{LI}. $U^{com_j}_{x_i}(t)$ in Eq. 3 is a value in $[0, 1]$ representing the average utilization rate of communication interface j on node x_i during time $[t - T_{LB}, t)$. The index j may indicate Wi-Fi, LTE, or other possible wireless interfaces. Similar to $U^{com_j}_{x_i}(t)$, $U^{cpu}_{x_i}(t)$ represents the CPU average utilization rate on node x_i during time $[t - T_{LB}, t)$. Eqs. 3 and 4 ensure that a node should not be assigned more communication or processing tasks that may overload its available bandwidth.

IV. MSPS Design

This section presents the details of each data operation (i.e., data creation, data retrieval, and data processing) followed by the load balancing algorithm. For simplicity, data operations are first described without considering load balancing. Once the load balancing algorithm is explained, we then show how the load balancing component is integrated into each data operation.

A. Data Creation

A node starts a *Data Creation* operation when it needs to store or share a file in *MSPS*. We refer to this node as *File Creator*. Based on the storage parameter (k, n) and the reliability requirement r_{req} specified by the application, a file is encoded into n data fragments and each fragment is sent to one of the storage node. These storage nodes together ensure that the reliability of the file is at least r_{req}. The file creator first uses the *request dissemination* component to discover nodes that can store data fragments. A node receiving the request replies its routing table, reliability, and current energy profile. The details of request dissemination procedure will be explained in the next section.

Data Reliability of a file can be estimated from the reliability of its storage nodes. We consider the probability that k or more selected storage nodes remain functional. Suppose X is a set of n selected storage nodes and $X \subset \hat{X}$ where $\hat{X}$ is the nodes that the request dissemination component discovers. c is the subset of functional nodes in X such that $k \leq |c| \leq n$ and $\bar{c} = X \setminus c$ is the subset of malfunctioning nodes in X. For each size of $|c|$, there are $\binom{n}{|c|}$ combinations that need to be considered. Eq. 6 calculates the probability of exactly i functioning nodes and $(n - i)$ malfunctioning nodes in X; R_l is the reliability of the l^{th} node in c; Q_m is the failure probability of the m^{th} node in $\bar{c}$. Eq. 7 calculates the probability of k or more functioning nodes in X, i.e., the data reliability of a file stored in X. Note that the time parameter t is omitted for clarity.

$$\mathbf{S_i} = \sum_{j=1}^{\binom{n}{i}} \prod_{l \in c} R_l \prod_{m \in \bar{c}} Q_m \quad (6)$$

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

$$R^{(k,n)}(X) = \sum_{i=k}^{n} S_i \quad (7)$$

Using the information collected from request dissemination, file creator can estimate the required standardized energy for delivering a fragment to a potential storage node. When a node is reachable through multiple wireless interfaces, the interface that consumes the least standardized energy for transmitting data is chosen. Suppose $E^{std}_{creator}(x_i)$ gives the minimal standardized energy for sending a data fragment from the file creator to node x_i, Eq. 8 describes a simplified optimization problem for data creation without considering the load balancing constraints (Eqs. 2–4). The additional constraint in Eq. 9 ensures that the reliability of the selected storage nodes meet the reliability requirement r_{req}.

$$X_{opt} = \arg\min_{X \subset \hat{X}} \sum_{\forall x_i \in X} E^{std}_{creator}(x_i) \quad (8)$$

$$\text{s.t.:} \quad R^{(k,n)}(X) \geq r_{req}, \ \|X\| = n \quad (9)$$

Solving this problem, however, is computation-intensive due to the complexity of reliability estimation $R^{(k,n)}(X)$. The combination term in Eq. 6 makes its time complexity $O(n!)$. In order to quickly find a feasible solution, we design a heuristic solver to approximate X_{opt}. Instead of exhaustively searching for all possible subset $X \subset \hat{X}$ and evaluating its data reliability $R^{(k,n)}(X)$, we consider only the nodes with reliability at least r_{min} such that any subset of n nodes guarantees to satisfy the reliability constraint (Eq. 9). r_{min} enforces a minimal reliability to each selected storage node and effectively eliminates the reliability constraint.

To determine r_{min}, we solve the equation $R^{(k,n)}(X) = r_{req}$. In $R^{(k,n)}(X)$, the reliability of each node $R_l(\cdot)$ is replaced by r_{min}, and the failure probability of each node $Q_m(\cdot)$ is replaced by (1-r_{min}). r_{min} thus becomes the only unknown in the polynomial equation and it 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_{min}, any subset X with all nodes' reliability no less than r_{min} must have $R^{(k,n)}(X) \geq r_{req}$. The data creation procedure can now be simplified to these three steps: 1) Solve $R^{(k,n)}(X) = r_{req}$. 2) Find all x_i such that $R_{x_i} \geq r_{min}$. 3) Sort all nodes found in step 2 in increasing order with respect to $E^{std}_{creator}(x_i)$. The first n nodes are then selected as the storage nodes.

B. 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 information 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 standardized energy for retrieving data fragments from each discovered storage node. Each participating node may use any available wireless interface to transfer data, but the file requester chooses the interface with the minimal standardized energy. $E^{std}_{request}(x_i)$ gives the minimal standardized energy for the file requester to download a data fragment from node x_i. $\hat{F} = \{f_1, f_2, ...\}$ represents a set of data fragments that the request dissemination component discovers, and $x_{sto}(f)$ gives the storage node of fragment f. Eq. 10 describes a simplified data retrieval optimization problem without the load balancing constraints. The objective here is to find a set of k fragments F that minimizes the standardized data retrieval energy. The problem can be solved by selecting the k fragments with the lowest estimated retrieval energy. Once F_{opt} is found, the nodes that carry these fragments are then selected to execute the operation, i.e., X_{opt}=$\bigcup_{\forall f \in F_{opt}} x_{sto}(f)$.

$$F_{opt} = \arg\min_{F \subset \hat{F}} \sum_{\forall f \in F} E^{std}_{request}(x_{sto}(f)) \quad (10)$$

$$\text{s.t.:} \quad \|F\| = k$$

C. Data Processing

Any node can submit a data processing job to process/analyze a set of files. $T = \{\tau_1, \tau_2, ..., \tau_M\}$ represents a job of M tasks where each task corresponds to a file to be processed. A processor node assigned to process a task needs to retrieve, recover, and then process the file. Similar to other data operations, the *job creator* uses *request dissemination* component to announce the job and discover the available processor nodes. A node receiving the job request estimates its standardized energy for retrieving and processing each task and sends this information back to the job requester. If a node has retrieved any file in this job before and it still has the complete file, the estimated task retrieval energy for retrieving this task is zero. Let $E^{std}_{proc}(x_i, \tau)$ represent the minimal standardized energy for node x_i to retrieve and process task τ. Eq. 11 describes a simplified data processing optimization problem without the load balancing constraints. The objective here is to find a set of processor nodes X and assign each task to one of the processor node such that the total standardized task processing energy is minimized. The problem can be solved by first assigning each task τ to the node $x_{pro}(\tau)$ that has the minimal standardized processing energy. The selected processor nodes can be described as X_{opt}=$\bigcup_{\forall \tau \in T} x_{pro}(\tau)$.

$$X_{opt} = \arg\min_{X \subset \hat{X}} \sum_{i=1}^{M} E^{std}_{proc}(x \in X, \tau_i) \quad (11)$$

D. Load Balancing

To avoid overloading a small number of nodes or causing performance bottleneck, *MSPS* considers load balancing when performing data operations. The system-wide load imbalance $LI(t)$ is formally defined in Eq. 5. $LI(t) = 0$ if the system is perfectly balanced, and $LI(t)$ increases positively as the system becomes more imbalanced. Our goal is to keep the load of each node $L_{x_i}(t)$ as close to the system-wide mean load $L_{\bar{\mu}}(t)$ 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-wide load imbalance.

The system-wide mean load $L_{\bar{\mu}}(t)$, however, cannot be calculated exactly because *MSPS* does not assume global information. Instead, a *sample mean load* $\widehat{L_{\bar{\mu}}}(t)$ estimated from local information is used to approximate the population mean $L_{\bar{\mu}}(t)$. $\widehat{L_{\bar{\mu}}}(t)$ is calculated for each data operation using the information the search agents collected. Because nodes

$$X_{opt} = \arg\min_{X \subset \hat{X}} \sum_{\forall x_i \in X} E_{x_i}^{std}(task, t) \times (1 + g(i,t)) \tag{12}$$

$$\begin{aligned} g(i,t) = {} & (\tilde{L}) e^{\frac{\alpha}{|1-\tilde{L}|}} (H(\tilde{L} - S^{LI}) - H(\tilde{L} - 1)) \\ & + (\tilde{U_1} - S^{com}) e^{\frac{\alpha}{|1-\tilde{U_1}|}} (H(\tilde{U_1} - S^{com}) - H(\tilde{U_1} - 1)) \\ & + (\tilde{U_2} - S^{cpu}) e^{\frac{\alpha}{|1-\tilde{U_2}|}} (H(\tilde{U_2} - S^{cpu}) - H(\tilde{U_2} - 1)) \end{aligned} \tag{13}$$

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

can move freely and data operations can be initiated from any region of the network, $\widehat{L_{\bar{\mu}}}(t)$ estimation is not biased to any subset of nodes.

To solve the complete optimization problem with load balancing constraints (Eqs. 1–4), we first transform the constrained optimization problem into an unconstrained optimization problem using *penalty method*. The constraints are combined into a penalty function $g(i,t)$ shown in Eq. 13. It is multiplied to the original objective function (Eq. 1) to construct the new unconstrained optimization problem (Eq. 12). The characteristic of the penalty function is that it is zero if all the constraints in Eqs. 2–4 are satisfied, and grows exponentially if any constraint is violated. 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 $g(i,t)$. $S^{LI}, S^{com},$ and S^{cpu} define the tolerance for load imbalance and over-utilization rate; α is a positive real value that controls how fast the penalty function grows; $H(\cdot)$ is Heaviside step function that limits the domain of interest. As an example, suppose the communication utilization and the CPU utilization constraints are satisfied, but the load on node x_i is slightly higher than the mean load ($L_{x_i}(t) = 0.6$, $L_{\bar{\mu}}(t) = 0.5$, $S^{LI} = 1$, $\alpha = 1$). The objective function will be penalized by a factor of $1 + \frac{1}{5}e^{\frac{5}{4}}$.

Another benefit of this transformation is that it relaxes the hard constraints and allows a solution to violate slightly if the advantages (energy saving) of violation is considerably higher than the disadvantage (load imbalance or over-utilization) of violation. The tradeoff between energy conservation and load imbalance are reflected in the objective function. When describing the data operations, we focused only on minimizing the standardized energy and neglected the load balancing constraints for simplicity. To integrate the load balancing components into the data operations, we simply multiply each objective function in Eqs. 8 – 11 by the penalty function $g(i,t)$. The new solution avoids assigning tasks to nodes with overloaded utilization (energy, communication, or processing), and gradually alleviates the load of nodes with $L_{x_i}(t) > L_{\bar{\mu}}(t)$.

E. 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 [17]. 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 processing energy, the CPU of each device is profiled in advance so that the energy consumption for processing a file can be estimated based on the processing function and file size.

V. Request Dissemination

All three 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 to process a set of files. The simplest solution is to broadcast the request through the LTE network, but it affects all nodes in the network and is too costly. Another naive solution is to flood 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 messages traverse too many hops, or may fail to find sufficient resources (storage, fragments, or processors) if messages are not relayed 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.

A. 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 nearby nodes. *Search Initiator* is the node that starts the search task; it dispatches one or multiple *agents* that explore 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 child agents that collaboratively share the parent agent's responsibility, i.e., parent's target resource is distributed to the child agents. Each child agent is then dispatched to different regions to accomplish the parent agent's search 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 the target resource has not been reached and there is still unexplored neighbors), it forks into one or multiple child agent and divides the remaining target resource to each child agent. The number of the child agents created is determined by the number of unexplored neighbor nodes. The agent-based search procedure is described in Algorithm 1.

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

Algorithm 1: Agent-based Search

```
agent arrives node :
/* agent exchanges information with visited node      */
agent.collectInfo(node)
node.collectInfo(agent)
if node.resource $\geq$ 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 = $\frac{agent.targetRsc}{unexploredNeib.size()}$
    for n $\in$ node.unexploredNeib do
        send(agent, n)
    end
end
```

sparse or the resources are not distributed uniformly, some agents may terminate without finding enough target resource. 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. It is worth noting that it is possible that a data operation can not be performed because there is insufficient resource in the network. The procedure is outlined in the following four steps: 1) Search initiator defines a target resource and starts an agent-based search. 2) At each node, a search agent either forks into multiple child agents or terminates and replies. 3) The search procedure stops if the search agents have found enough resources. 4) If the initiator's target resource is not reached and there is still unexplored nodes, then restart step 1 at boundary nodes.

Using the storage request as an example, Fig. 2 shows how the agent-based search is performed. Assume all nodes are valid 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 that one unit of the target resource has been found. The update function than forks the agent into 4 child agents destined to nodes 3, 4, 9 and 10 respectively; each child agent's target resource is set to $6/4$. When the child 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 child agents are sent to node 1 and node 13. Child agents at node 3 and 10 proceed to node 2 and 8 respectively with their updated target resource $1/2$. The agent at node 4 terminates

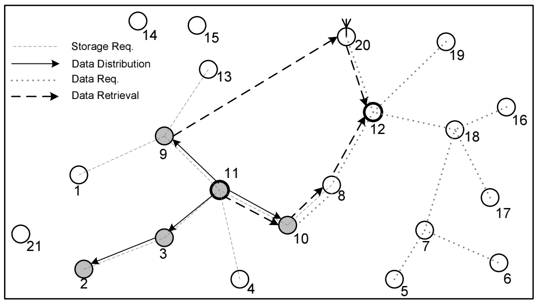


Fig. 2. **Data Creation & Data Retrieval**. Node 11 creates a file with (k,n)=(3,5). After request dissemination completes, nodes 2, 3, 9, 10, 11 are selected as storage nodes. Node 12 later requests to read the file. After request dissemination completes, 2 fragments are retrieved through Wi-Fi network from node 10 and 11, and 1 fragment is retrieved through LTE network from node 9.

because it has no more new node to explore. When the child 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 can terminate. Therefore, the search initiator successfully finds 9 resources (node 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 performed by *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 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 request. 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 child 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 node carries the requested file's fragment 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 unique 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 one processor node for each task. A processor node may process one or multiple files depending on its capability and current load. The target resource is M, and a node can provide a processing resource if it can retrieve and process a task without overloading itself. Essentially, a processor discovery tries to find the data fragments of M files simultaneously. Each visited node tells the agent the fragments it carries and the estimated energy for processing each file.

VI. Performance Evaluation

We evaluate *MSPS* through extensive simulations on Jist/Swans [18] network simulator and real-world implementation on Android devices. In simulations, we are interested

Simulator		
Phy. Interfaces:WiFi, LTE	Comm. Range: WiFi($\leq$ 160m), LTE($\leq$ 1500m)	
Mobility: Rnd Waypoint	Moving Speed: 0-4m / sec.	Size: 75 nodes
Routing: AODV	File size: 2MB, (k,n)=(3,9)	Field: 800m^2
Hardware Implementation		
Devices: Nexus 5, Nexus 7, HTC One, Note 2, Galaxy S3...		Size: $\leq$12 nodes
Image file $\leq$ 3MB, Video file $\leq$ 30MB		Field: 300m^2

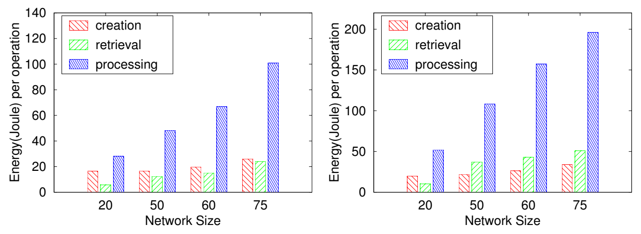
TABLE II
PERFORMANCE EVALUATION SETTINGS

in the *energy efficiency, system-wide load imbalance, and the system lifetime* under various heterogeneous networks; in hardware-based evaluation, we want to understand the feasibility of our algorithm and how it performs on modern smart devices. The default settings of the simulator and our hardware are summarized in Table II.

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 *Random* allocation that selects storage nodes or processor nodes in a random manner. We then show how *MSPS* saves communication energy utilizing multiple wireless interfaces (Wi-Fi and LTE). The load balancing algorithm is evaluated 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 agents successfully explore and the number of packets the search algorithm exchanges during the 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 (images and videos) on a group of smart devices. We evaluated and collected data of this application during 2015 Summer Institute on Flooding exercise [19].

A. Energy consumption of data operations

We first look at the energy consumption of each data operation under 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) that have the largest battery capacity (10,000 mAh), CPU power (2 Watt), and processing throughput (1 MB/sec); the low performance nodes (LPC) that 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) that have all the hardware capabilities in-between HPC and LPC. In Fig. 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 in order to recover a file. Each processing job processes 5 randomly selected files stored in *MSPS*.



(a) (b)

Fig. 3. (a) Data operations in *MSPS*. (b) Data operation with random allocation.

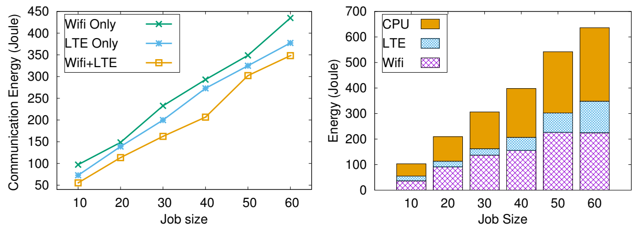
The energy consumption of data operations increases with the network size due to the energy overhead from the additional nodes. When the network density increases, the radio interference also causes lower throughput and thus higher communication energy consumption. In general, data creation consumes 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 much smaller than factor of 3. The energy consumption of data processing is expected to be the highest because each processing job involves retrieving and processing 5 files. 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* (Fig. 3a) with a random allocation scheme (Fig. 3b) in which storage nodes and processor nodes are selected in a random manner. *MSPS* outperforms the random scheme by at least 40%.

B. Effects of communication interfaces

In this section, we evaluate the energy savings gained by intelligently using multiple communication interfaces. Fig. 4 shows the communication energy of various job sizes when using different communication interfaces. LTE consumes approximately $4 - 6$ times higher power than Wi-Fi, but LTE takes the 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. Fig. 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. Fig. 4b shows the breakdown of energy consumption for processing a job. The fact that each individual energy consumption increases almost linearly with the job size shows that *MSPS* highly scalable.

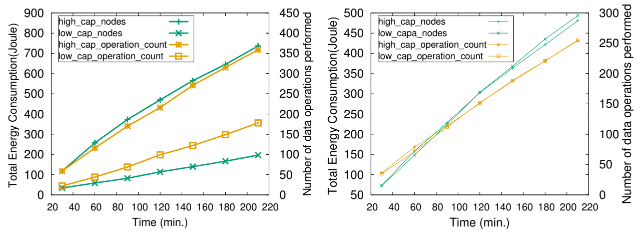
C. Performance of Load balancing algorithm

Fig. 5 shows how *MSPS* allocates communication and processing tasks considering the energy capacities of each



(a) (b)

Fig. 4. (a) Comparison of using Wi-Fi only, LTE only, or both for data processing.(b) Energy consumption of different components.



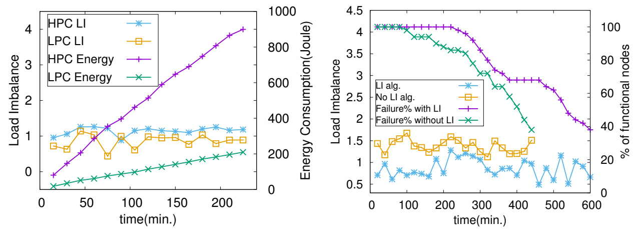
(a) (b)

Fig. 5. (a) Energy consumption and tasks assigned to nodes with different energy capacities. (b) Same as (a), but without using standardized energy.

node. 50 nodes of three different types HPC, MPC, and LPC are deployed. Each node is assigned one of the three types with equal probability. Fig. 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 capacity or processing resources. Fig. 5b shows the result of the same experiment without using the *standardized energy*. This way, *MSPS* neglects the differences of energy capacities and evenly allocates tasks to each node based on the absolute energy consumption. The load balancing algorithm ensures that each node receives approximately the same workload regardless of the remaining energy. This causes low energy capacity nodes to die much earlier and impacts the overall performance, as shown in Fig. 6.

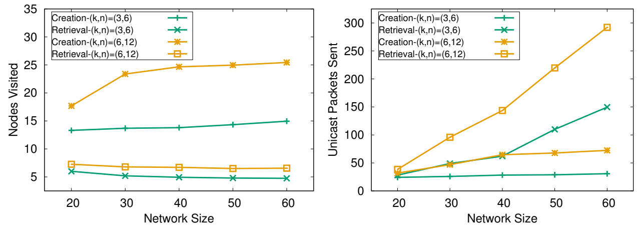
Fig. 6 evaluates the performance of the load balancing algorithm. When allocating a communication or processing task, *MSPS* considers the energy load (Eq. 2), communication utilization (Eq. 3), and CPU utilization (Eq. 4). A node should not receive more tasks than it can handle, which causes system bottleneck and high delay; neither should a node be much busier than other nodes, which causes a network hotspot and harms the system lifetime. Fig. 6a shows the load imbalance of high performance nodes (HPC) and low performance nodes (LPC) at different times. The thresholds 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 Fig. 6b, we compare the load imbalance and the system lifetime between enabling or disabling the load balancing algorithm (set $g(i,t)=0$ in Eq. 13). We declare a system



(a) (b)

Fig. 6. (a) Energy consumption and load imbalance of different types of nodes. (b) Load Imbalance and percentage of functional nodes.



(a) (b)

Fig. 7. (a) Number of resources discovered by search agents. (b) Number of packets sent during a search procedure.

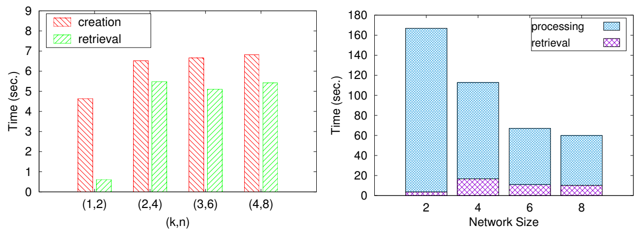
failed when more than 50% nodes have failed due to depleted energy. The figure shows that our load balancing algorithm not only reduces the system-wide load imbalance by 30-50%, but it also extends the system lifetime by 30%.

D. Performance of agent-base search algorithm

Agent-based search procedure disseminates data operation requests and explores 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 efficiently search for desired resources with minimal network traffic and energy overhead. 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 Fig. 7a shows the number of resources that search agents found at the end of a searching procedure. For data creation, the number of the discovered resources is about two times of the n value because we set the target resource to $2n$. This setting allows the creator to choose more reliable storage nodes from a larger group of candidate nodes. Fig. 7b shows the total number of unicast packets sent during a search procedure, 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. 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-base search algorithm is scalable to larger or denser network.

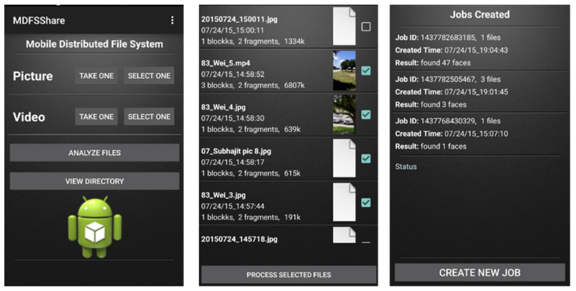
E. Hardware Implementation

To understand the feasibility and performance of our algorithm in real hardware, we implemented *MSPS* on Android devices and created the *MediaShare* application that shares and processes multimedia files stored on mobile devices. The processing function of *MediaShare* extracts image frames from



(a) (b)

Fig. 8. (a) Running time of data operations under different (k,n) settings. (b) Data processing time in different network sizes.



20150724_150011.jpg
07/24/15_15:00:11
1 blockks, 2 fragments, 1334k

83_Wei_5.mp4
07/24/15_14:58:52
3 blockks, 2 fragments, 6807k

83_Wei_4.jpg
07/24/15_14:58:30
1 blockks, 2 fragments, 639k

07_Subhajit pic 8.jpg
07/24/15_14:58:17
1 blockks, 2 fragments, 615k

83_Wei_3.jpg
07/24/15_14:57:44
1 blockks, 2 fragments, 191k

20150724_145718.jpg

Job ID: 1437782683185, 1 files
Created Time: 07/24/15_19:04:43
Result: found 47 faces

Job ID: 1437782505467, 3 files
Created Time: 07/24/15_19:01:45
Result: found 3 faces

Job ID: 1437768430329, 1 files
Created Time: 07/24/15_15:07:10
Result: found 1 faces

Fig. 9. Screenshots of *MediaShare* Application. From left to right are Home screen, Directory, and Job List pages

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. The facial recognition library can process one image frame in 0.5–1.5 seconds depending on the processor speed and image complexity. The MSPS middleware and MediaShare application each contains about 9,000 and 2,000 lines of Java code. The application is developed on Android SDK 4.2.2. Some user interface of *MediaShare* is shown in Fig. 9.

During Summer Institute on Flooding exercise [19], 10 participants used *MediaShare* App to share and process media files in a controlled disaster environment. The App was installed on at least 5 different types of Android devices as listed in Table II. Fig. 8a shows the average time for creating and retrieving a 3MB file using different (k, n) settings. As expected, data creation takes longer than data retrieval because a file creator needs to distribute more data fragments than a file requester needs to retrieve. The resource discovery time of agent-based search is small ($<1\%$) compared to the actual data transferring time and thus is not shown. Fig. 8b shows the data retrieval time and CPU processing time for analyzing eleven 30 seconds 30MB 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 more processing resources. From the first responders' positive feedback and the performance results, we are confident that *MSPS* is efficient and feasible on real hardware.

VII. Conclusions

This paper presents the design of *MSPS* - a distributed storage and processing system for heterogeneous mobile clouds. Envisioning the pervasiveness and diversity of mobile devices in the near future, we study how a collection of mobile devices with different hardware specifications can work together in an energy-efficient and load-balanced manner. Our algorithm considers the diverse characteristics of the communication interfaces, processing capabilities, and energy capacities when allocating resources. In particular, our distributed solution adapts well in networks of heterogeneous device types and is scalable to larger networks. From extensive simulations and a real-world implementation, we show that *MSPS* achieves our expected objectives and is practical on real hardware.

References

[1] B. McGarry, "Army set to introduce smartphones into combat," http://www.military.com/, March 2013.
[2] S. M. George et. al., "Distressnet: a wireless ad hoc and sensor network architecture for situation management in disaster response," *Communications Magazine, IEEE*, vol. 48, 2010.
[3] D. Neumann, C. Bodenstein, O. F. Rana, and R. Krishnaswamy, "STACEE: enhancing storage clouds using edge devices," in *WACE*, 2011.
[4] P. Stuedi, I. Mohomed, and D. Terry, "WhereStore: location-based data storage for mobile devices interacting with the cloud," in *MCS*, 2010.
[5] R. K. Panta, R. Jana, F. Cheng, and Y.-F. R. Chen, "Phoenix: Storage using an autonomous mobile infrastructure," *TPDS*, 2013.
[6] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: enabling remote computing among intermittently connected mobile devices," in *MobiHoc*, 2012.
[7] D. Huang, Z. Zhou, L. Xu, T. Xing, and Y. Zhong, "Secure data processing framework for mobile cloud computing," in *INFOCOM WKSHPS*, 2011.
[8] C. Chen, M. Won, R. Stoleru, and G. Xie, "Energy-efficient fault-tolerant data storage and processing in dynamic network," in *MobiHoc*, 2013.
[9] J. George, C.-A. Chen, R. Stoleru, G. G. Xie, T. Sookoor, and D. Bruno, "Hadoop mapreduce for tactical clouds," in *CloudNet*. IEEE, 2014.
[10] E. E. Marinelli, "Hyrax: cloud computing on mobile devices using mapreduce," CMU DTIC Document, Tech. Rep., 2009.
[11] G. Huerta-Canepa and D. Lee, "A virtual cloud computing provider for mobile devices," in *Proc. of the Workshop on MCS*, 2010.
[12] D. Huang, X. Zhang, M. Kang, and J. Luo, "MobiCloud: Building secure cloud framework for mobile computing and communication," in *SOSE*, 2010.
[13] M. D. Kristensen, "Scavenger: Transparent development of efficient cyber foraging applications," in *PerCom*, 2010.
[14] S. Huchton, G. Xie, and R. Beverly, "Building and evaluating a k-resilient mobile distributed file system resistant to device compromise," in *MILCOM*, 2011.
[15] C. Chen, M. Won, R. Stoleru, and G. Xie, "Resource allocation for energy efficient k-out-of-n system in mobile ad hoc networks," in *Proc. of ICCCN*, 2013.
[16] Y. Feng, R. Stoleru, C.-A. Chen, and G. G. Xie, "Resource allocation for energy efficient k-out-of-n system in mobile ad hoc networks," in *Proc. of ICCCN*, 2014.
[17] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. 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*. ACM, 2012, pp. 225–238.
[18] R. Barr, Z. J. Haas, and R. van Renesse, "Jist: An efficient approach to simulation using virtual machines," *Software: Practice and Experience*, 2005.
[19] CRASAR, "2015 summer institute on flooding," hidden for blind review, July 2015.