A Catalog of Architectural Tactics for Cyber-Foraging

Grace Lewis[†‡]
[†]Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
glewis@sei.cmu.edu, g.a.lewis@vu.nl

Patricia Lago[‡]
[‡]VU University Amsterdam
Amsterdam, The Netherlands
p.lago@vu.nl

ABSTRACT

Mobile devices have become for many the preferred way of interacting with the Internet, social media and the enterprise. However, mobile devices still do not have the computing power or battery life that will allow them to perform effectively over long periods of time or for executing applications that require extensive communication or computation, or low latency. Cyber-foraging is a technique enabling mobile devices to extend their computing power and storage by offloading computation or data to more powerful servers located in the cloud or in single-hop proximity. This paper presents a catalog of architectural tactics for cyber-foraging that was derived from the results of a systematic literature review on architectures for cyber-foraging systems. Elements of the architectures identified in the primary studies were codified in the form of *Architectural Tactics for Cyber-Foraging.* These tactics will help architects extend their design reasoning towards cyber-foraging as a way to support the mobile applications of the present and the future.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture—*domain-specific architectures*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*distributed systems*

General Terms

Design

Keywords

cyber-foraging, software architecture, architectural tactics, mobile cloud computing, mobile computing

1. INTRODUCTION

Cyber-foraging is an area of work within mobile cloud computing that leverages external resources (i.e., cloud servers, or local servers called surrogates) to augment the computation and storage capabilities of resource-limited mobile devices while extending their battery life. There are two main forms of cyber-foraging. One is computation offload, which is the offload of expensive computation in order to extend battery life and increase computational capability. The second is data staging to improve data transfers between mobile devices and the cloud by temporarily staging data in transit.

This paper presents a catalog of architectural tactics for cyber-foraging derived from the results of a Systematic Literature Review (SLR) on architectures for cyber-foraging systems. A set of 57 primary studies was identified (their preliminary analysis is available in [19]). The studies were categorized based on decisions regarding the following concerns:

- Where to offload? Is computation and/or data offloaded to proximate (single-hop) resources or remote (multi-hop) resources?
- When to offload? With optimization in mind, when does it make sense to offload? Is computation always offloades or is whether or not to offload a runtime decision?
- What to offload? What is the granularity of the computation that is offloaded? What is the size of the payload to use the computation? What type of data is offloaded? What data operations are offloaded?

The next section introduces the architectural tactics for cyber-foraging, grouped into functional tactics (Section 3) and non-functional tactics (Section 4). Section 5 presents related work. Section 6 concludes the paper and outlines the next steps in our research.

2. ARCHITECTURAL TACTICS FOR CYBER-FORAGING

The tactics were extracted from the literature based on (1) common components found in the studies, (2) quality attributes explicitly stated in the studies, and (3) quality attributes inferred from system and component descriptions. Common design decisions were codified into architectural tactics and grouped into functional and non-functional tactics. *Functional tactics* are broad and basic in nature and correspond to the architectural elements that are necessary to meet cyber-foraging functional requirements. *Non-functional tactics* are more specific and correspond to architecture decisions made to achieve certain quality attributes.

QoSA'15, May 4–8, 2015, Montréal, QC, Canada.
Copyright © 2015 ACM 978-1-4503-3470-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2737182.2737188.

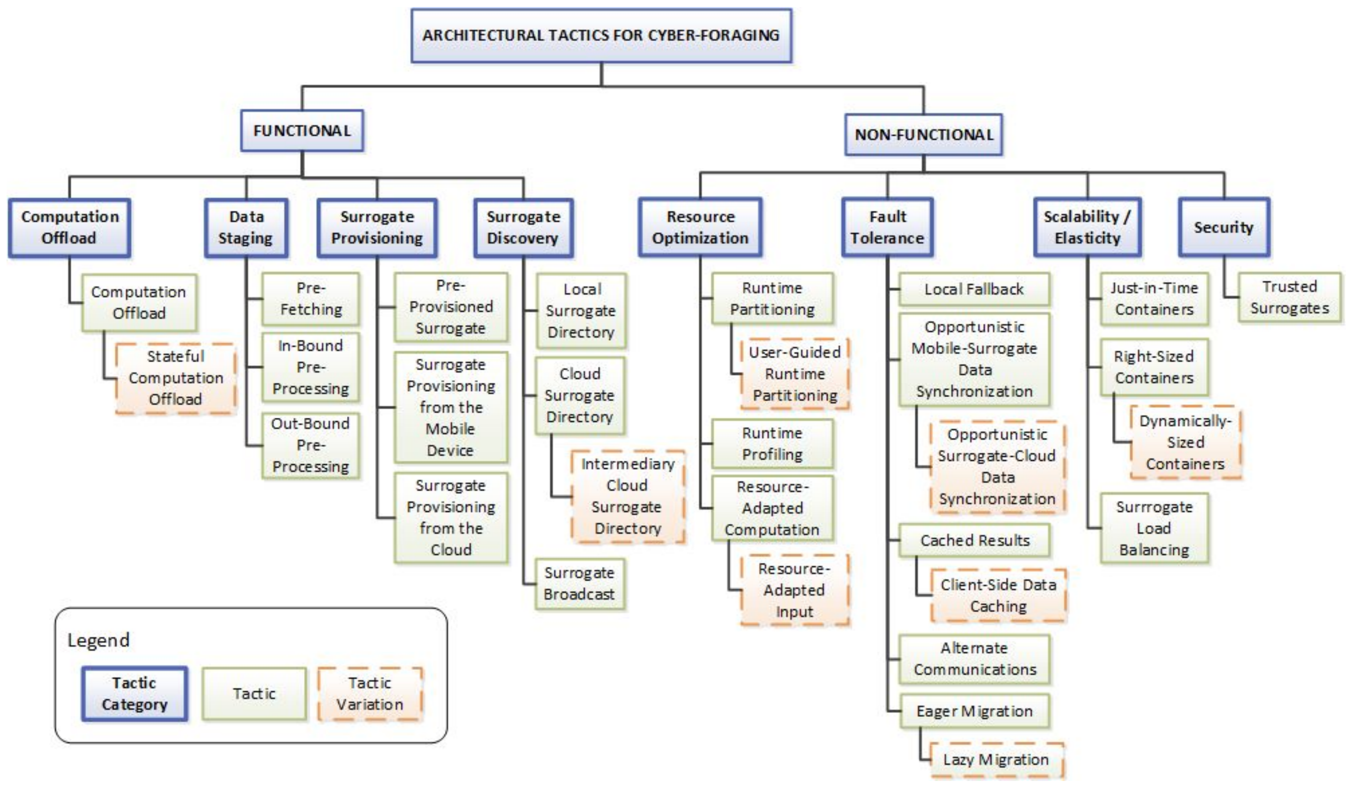

Figure 1: Architectural Tactics for Cyber-Foraging

Non-functional tactics have to be used in conjunction with functional tactics. Figure 1 presents the set of identified tactics. The top levels of the figure are the tactic categories. The boxes with solid lines under each category are the tactics. A box with a dashed line under a tactic is a variation of that tactic.

Each tactic category presented in the paper will include a **scenario**, a **short description** of each tactic in the category including an **example system**, and **general observations** about constraints and tradeoffs. The Computation Offload tactic and the Pre-Fetching tactic for Data Staging will be explained in greater detail because many of the other tactics build on these two basic tactics. The full catalog of tactics is available as a technical report [18]. The goal of the catalog is to serve as a **reference** for architects designing cyber-foraging systems.

3. FUNCTIONAL TACTICS

3.1 Computation Offload

A scenario for Computation Offload from a mobile device to a surrogate is the following: The user of a mobile device executes a cyber-foraging-enabled mobile application. The application offloads the computation-intensive portions of the application to a nearby surrogate, with minimal disruption to the mobile device user. Computation Offload extends battery life by offloading computation-intensive portions of an application to nearby surrogates with greater computation power. In addition, the single-hop proximity of surrogates combined with the use of WiFi or short-range radio instead of broadband wireless (e.g., 3G/4G) also decreases latency [3] and improves the user experience especially for highly-interactive applications.

Figure 2 shows the main components of this tactic with numbers that indicate the sequence of operations. The Computation Offload tactic requires an *Offload Client* running on the *Mobile Device* and an *Offload Server* running on the *Surrogate*. This pair of components communicates to coordinate the offload operation. The *Cyber-Foraging Enabled Mobile App* invokes the *Offload Client* when it encounters a portion of code that has been identified as offloadable computation and passes it any *App Metadata* that is required to set up the *Offloaded Code*. The *Offload Client* then coordinates with the *Offload Server* to set up the *Offloaded Code* so that it can be invoked by the *Cyber-Foraging Enabled Mobile App*. The *Offloaded Code* runs inside a *Container* on the *Surrogate*. Examples of a *Container* are a virtual machine, application server, web server, or the operating system. Figure 2 shows the *Cyber-Foraging Enabled Mobile App* communicating directly with the *Offloaded Code*. An alternative is for the *Cyber-Foraging Enabled Mobile App* to always communicate through the *Offload Client*. This latter alternative has the potential for performance problems as the number of mobile clients using the surrogate increases. This is because the *Offload Server* becomes a bottleneck as all communication between mobile devices and the surrogate would go through this component. However, some systems that implement Fault Tolerance tactics (Section 4.2) place the responsibility of detecting and managing disconnections in the *Offload Client* and *Offload Server* which therefore benefits from the single point of communication of the latter alternative.

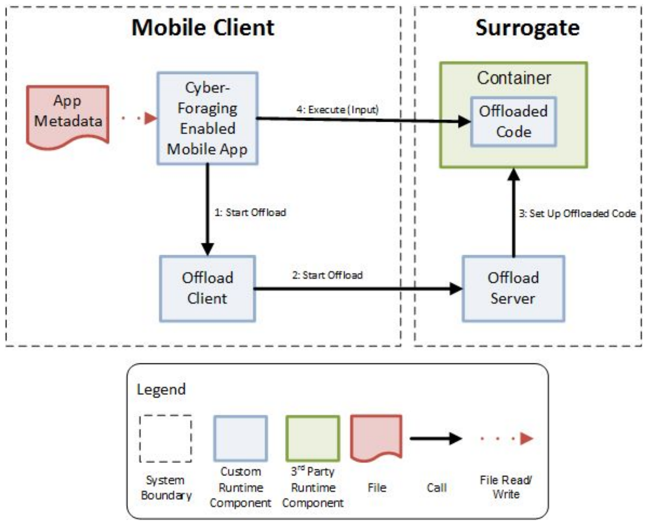

Figure 2: Computation Offload Tactic

An example of the Computation Offload tactic is in the Mobile Agents system [1] in which applications are manually partitioned into components that have to be executed locally and components that can be offloaded. These offloadable components are set up as *Mobile Agents* using the Java Agent Development Environment (JADE). At runtime, the system determines if the agent marked as offloadable should be offloaded based on a comparison of local and remote execution times.

Variation: Stateful Computation Offload. The Computation Offload tactic assumes that the offload operation is stateless. This means that no mobile app state needs to be transferred between the *Offload Client* and the *Offload Server* during the offload operation. This is what happens when the granularity of the offload operation is a module or class, a service, or a complete application (or server portion of an application). When the granularity of the offload operation is at the process or at the method level, the state of the program or object that contains the process or method being offloaded has to be transferred to the equivalent program or object on the surrogate. In this case, a state synchronization operation is invoked either periodically or on-demand before the offloaded code is executed to guarantee that the state is equivalent on both sides. An example of the Stateful Computation Offload tactic is in the CloneCloud system [5].

3.1.1 Observations

The Computation Offload tactics assume that offloaded computation already exists on the surrogate (loaded on the surrogate via a Surrogate Provisioning tactic at deployment or run time (Section 3.3)) and that computation that is marked for offload is always offloaded. Combining Computation Offload tactics with Resource Optimization tactics (Section 4.1) enables the system to determine when it is optimal to offload and when not. The tactics also assume that the surrogate is always available for offload. Combining the Computation Offload tactics with Fault Tolerance tactics (Section 4.2) enables the system to deal with unavailable surrogates.

3.2 Data Staging

A scenario for Data Staging is the following: A mobile application is being used by multiple users to collect data in the field. Upon detection that it is close to a surrogate, the mobile application offloads the collected data. When the operation is complete, the mobile device deletes the transmitted data to free up storage space. In addition, when the surrogate establishes connectivity to the main data center in the cloud, it forwards the data that was collected by the multiple users, where it is integrated into the enterprise data repository. An additional capability of the application is to provide data visualizations pertaining to the data collected by the user, the data collected in the region that is served by the surrogate, and the data collected by the entire set of users. Therefore, data is pushed from the enterprise data center to the surrogate either on-demand or periodically so that the data is closer to the user and accessible even if the surrogate is disconnected from the enterprise.

3.2.1 Pre-Fetching

Data-intensive mobile apps often rely on data located in the cloud. However, access to this data is likely over a lower-bandwidth and multi-hop connection, compared to the higher-bandwidth, single-hop connection that exists between a mobile device and a surrogate. Pre-fetching anticipates data needs in order to minimize communication to the cloud and reduce latency. The surrogate, according to a defined pre-fetch algorithm, retrieves data from the cloud and stores it locally so that it is available to the mobile device when it needs it. Access to the cloud is therefore only necessary when the data is not already available on the surrogate.

Figure 3 presents the main components of this tactic. The Pre-Fetching tactic requires a *Data Staging Client* that runs on the *Mobile Client* and a *Data Staging Manager* that runs on the *Surrogate*. The *Data Staging Client* handles all data operations on behalf of a *Cyber-Foraging-Enabled Mobile App*. Before sending the data operation to the *Data Staging Manager*, the *Data Staging Client* captures and also sends along any *Pre-Fetch Hints* that are used by the *Pre-Fetch Algorithm* to determine and anticipate data needs. Examples of pre-fetching hints include mobile device location, user profile and preferences, and the user's schedule. The *Data Staging Manager* first executes the data operation against the local *Cache*. If the operation is successful it returns the results of the data operation. If the operation is not successful the *Data Staging Manager* obtains the data from the *Cloud Data Repository* in the *Enterprise Cloud*, stores it in the local *Cache*, and returns the results of the data operation to the *Mobile Client*. Asynchronously, either periodically or triggered by certain conditions, the *Data Staging Manager* will use the *Pre-Fetch Hints* from the *Mobile Client* and any local data such as the user's access history as parameters to a *Pre-Fetch Algorithm* that will calculate the data set that is likely to be needed next by the *Cyber-Foraging Enabled Mobile App*. It will then retrieve this data set from the *Cloud Data Repository* and store it in the local *Cache* so that it is available when it is needed by the *Cyber-Foraging Enabled Mobile App*. Similarly, either periodically or in response to certain conditions, the *Data Staging Manager* will sync the *Cache* with the *Cloud Data Repository* to ensure that data is consistent locally and remotely.

In the Trusted and Unmanaged Data Staging Surrogates system [9] data is staged on a *Staging Server* in the *Surro-*

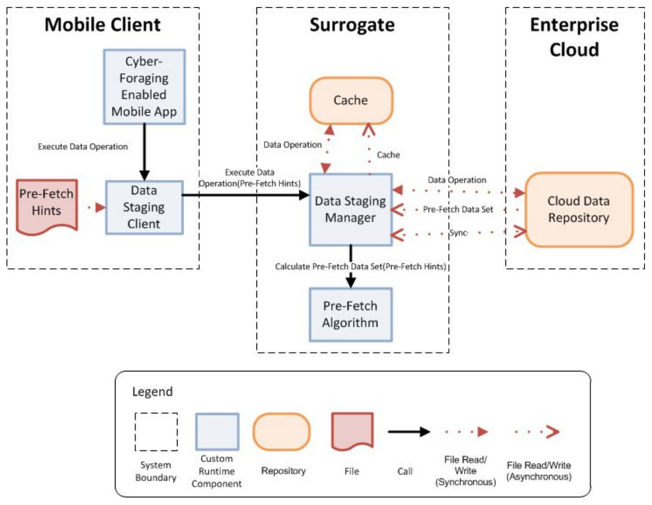

Figure 3: Pre-Fetching Tactic

gate. After the *Cache* has been loaded with an initial data set, all data operations are routed to the *Staging Server*. If the requested file exists in the *Cache* then the data operation takes place locally on the *Surrogate*. If the file is not available in the *Cache* it obtains the file from the *File Server* and stores it in the *Cache*, along with any other files that are predicted to be required based on the request.

3.2.2 In-Bound Pre-Processing

In order to reduce the amount of data received by the mobile device, avoid direct communication to the cloud for every data operation, and avoid the computation costs of processing this data for visualization on mobile devices, the surrogate pre-processes the data that is retrieved or pushed from the enterprise cloud. The mobile device receives data that is ready to be consumed, or filtered such that it only receives data of interest or relevance. The mobile device can request data on demand or periodically (synchronous) or can register with the surrogate for data of interest (asynchronous). The Edge Proxy system [2] uses a surrogate called an *Edge Server* to monitor changes in user-defined areas of interest in web pages. It does high-frequency polling of the web page on the web server and notifies the mobile device if it detects a change in the areas of interest compared to the cached web page.

3.2.3 Out-Bound Pre-Processing

Data-intensive mobile apps are often used to collect data in the field, where Internet connectivity might not be available to mobile devices or might be costly. In addition, although the field-collected data is valuable, it might be overwhelming for a device to transmit all data collected to the enterprise, especially if Internet connectivity is a scarce resource. In these cases, a surrogate can pre-process – clean, filter, summarize, or merge – the data that is received from the mobile devices that it serves such that the data that is sent on to the enterprise cloud is ready for consumption and serves an immediate need. The mobile device can also batch data according to user or application preferences to conserve the energy spent on turning the radio on and off for communication. Data collected on the surrogate can be uploaded to the cloud when network connectivity is available. In the Large-Scale Mobile Crowdsensing system [24] crowdsensing participation apps gather data from one or more sensors on the mobile device and send them to a surrogate. Applications running on the surrogate process the data streams coming from mobile devices locally and/or format and send the data to applications in the cloud.

3.2.4 Observations

All the Data Staging tactics require a Surrogate Provisioning tactic (Section 3.3) to prepare the surrogate for data staging. In addition, they require a configuration in which the mobile device is connected to a surrogate and the surrogate is connected to the enterprise or cloud data center, even if connectivity is intermittent or periodic. Combining Data Staging tactics with Fault Tolerance tactics (Section 4.2) enables implementations in disconnected or intermittent environments. Finally, if data is being modified on the mobile device, as in potentially the Pre-fetching tactic, there needs to be a mechanism in place, either manual or automatic, to resolve any synchronization conflicts between surrogate caches and cloud repositories.

3.3 Surrogate Provisioning

To be able to use a surrogate for cyber-foraging, it has to be provisioned with the offloaded computation and/or the computational elements that enable data staging. A scenario for surrogate provisioning is as follows: a mobile device needs to execute a computation-intensive task. Instead of executing the task locally, it locates a surrogate and sends it a request to execute the computation on its behalf. The surrogate first checks if it already has the computation to support the task. Because it does not, it sees if it can locate the computation in a cloud repository. Because the surrogate is not able to locate the capability in the cloud, the mobile device sends the computation to the surrogate for installation. Once the surrogate installs and starts the computation it notifies the mobile device that it is ready, executes the computation, and sends back the results of the computation.

3.3.1 Pre-Provisioned Surrogate

Pre-provisioned surrogates have the advantage of shorter response time to offload requests from mobile devices because the offloaded computation or data staging elements already reside on the surrogate. In an operational setting in which surrogates support multiple clients, a surrogate should have minimal management capabilities that (1) help surrogate administrators to install capabilities and appropriate execution containers, and (2) maintain a list of these capabilities (similar to a service registry). This tactic requires a *Surrogate Manager* component to manage a *Capabilities Repository*, *Capability Metadata* to enable setup of capabilities on demand, and a *Capability Registry* that is used by Surrogate Discovery tactics (Section 3.4) for advertising capabilities to mobile cyber-foraging clients. This tactic is not present in any of the systems, but could be integrated into any of the cyber-foraging systems that assumes that offloaded computation and/or data staging elements are already available on the surrogate at runtime.

3.3.2 *Surrogate Provisioning from the Mobile Device*

In Pre-Provisioned Surrogates (Section 3.3.1) a mobile device can only execute applications that already exist on the surrogate. Provisioning the surrogate from the mobile device has the advantage of enabling the execution of a greater number of applications because surrogates are provisioned at runtime. The mobile device sends the offloaded computation to the surrogate at runtime. The surrogate installs the computation inside an execution container and starts the application on behalf of the mobile device. In the VM-Based Cloudlets system [23] application overlays are sent at runtime to the surrogate (cloudlet) and combined with a base VM to produce a VM with the running application.

3.3.3 *Surrogate Provisioning from the Cloud*

Provisioning surrogates from the mobile device has the advantage of enabling the execution of a greater number of applications (Section 3.3.2) compared to pre-provisioned surrogates (Section 3.3.1). However, the size of the computation that is sent to the surrogate at runtime can be significant. In the examples for the MAUI system [6], the size of the .NET components transmitted at runtime is between 0.2 MB and 13.8 MB. In the examples for the VM-Based Cloudlets system [23], the size of an application overlay is between 63 MB and 196 MB. An alternative is to send the location of the computation in the form of a URL for the surrogate to download and install. The payload in this case is almost insignificant but the time to provision may be longer due to potentially higher and unpredictable latency between the cloud and the surrogate. However, the mobile device is not consuming battery due to high transmission costs. In addition, because the computation exists in a defined place in the cloud it is easier to update because it does not have to be sent to each mobile device after patches or upgrades. In the Collective Surrogates system [10] a shell script is sent to the surrogate at runtime that downloads the application that corresponds to the offloaded code from an application repository on an Internet server, installs the application and starts it.

3.3.4 *Observations*

The Pre-Provisioned Surrogate tactic requires a management component that provisions the surrogate with capabilities before surrogate deployment. The Surrogate Provisioning from the Mobile Device and Surrogate Provisioning from the Cloud tactics requires a pre-established agreement on the format of the offloaded code (e.g., Java class, Python script, Windows application). In addition, depending on the size of the offloaded code the tactics may require additional components to provide reliable communications. Finally, in the Surrogate Provisioning from the Cloud tactic the computation has to exist at the indicated location.

3.4 Surrogate Discovery

In order to leverage cyber-foraging, mobile devices need to be able to locate available surrogates on which to offload computation or stage data. A scenario for surrogate discovery is as follows: a mobile device needs to execute a computation-intensive task and has already decided that it will offload the task to a surrogate. The mobile device is able to locate all nearby surrogates and selects the surrogate that is the best match for the offloaded task.

3.4.1 *Local Surrogate Directory*

For mobile devices to leverage nearby surrogates they need to know where the surrogates are located; that is, they need to know their network address (i.e., surrogate IP address or URL). A simple solution is for mobile devices to maintain a list of potential surrogates including any information that can help the mobile device to select the best surrogate in case more than one is available. The list can be static, or updated based on network conditions or offload execution data. An advantage of a local list is that it will potentially include only surrogates that are trusted by the mobile device. This tactic has two parts: one part involves a user interface to populate and maintain the *Surrogate Directory*; the other part involves the components that interact during the offload process to obtain the list of surrogates from the directory and ping each to see if it is available for offload. The Cuckoo system [14] has a component that maintains a list of surrogates. If the surrogate has a visual display, upon loading it shows a QR code that is read by the mobile device and then added to the list of surrogates it can use for offload. If it does not have a visual display, the resource description file for the surrogate has to be copied to the mobile device so that it can be added to the list.

3.4.2 *Cloud Surrogate Directory*

In the Local Surrogate Directory tactic (Section 3.4.1) the mobile device is responsible for populating and maintaining the list of surrogates on which it can offload computation. This is a rather static solution because as more surrogates become available in the environment there is no automated way of discovering these new surrogates or updating their metadata as changes occur. Maintaining the surrogate directory in the cloud has the advantage of a centralized location for surrogate registration and metadata. All the mobile device needs to know is the network address of the cloud server that manages the surrogate directory. In addition, optimal surrogate selection algorithms can run in the cloud, which is an additional offload operation that can lead to battery savings on the mobile device. Regarding trust, in this tactic the mobile device only needs to trust the cloud surrogate directory server assuming that the directory only contains trusted surrogates (Section 4.4.1). In the Mobile Agents system [1] the mobile device contacts a *Cloud Directory Service* to get a list of available surrogates and selects the one with the highest communication link speed with the mobile device as well as the highest computing power.

Variation: Intermediary Cloud Surrogate Directory. The Cloud Surrogate Directory tactic returns the address of the selected surrogate to the mobile device, which then contacts the surrogate directly. In systems such as Large-Scale Mobile Crowdsensing [24] the cloud server does not return the surrogate address to the mobile device, but rather forwards the offload request to the selected surrogate and then returns the results to the mobile device. In this variation the cloud server acts as an intermediary between the mobile device and the surrogate.

3.4.3 *Surrogate Broadcast*

The Local Surrogate Directory (Section 3.4.1) and Cloud Surrogate Directory (Section 3.4.2) tactics require a directory of potential surrogates to be maintained either on the mobile device or on a cloud server, respectively. Having surrogates broadcast their availability and metadata to mobile devices removes the burden of having to maintain surrogate directories up to date. It creates a much more dynamic environment in which mobile devices can discover nearby surrogates without needing to know their addresses in advance or retrieving the addresses from a cloud server that could potentially not be available when needed. In the VM-Based Cloudlets system [23] surrogate information that includes surrogate address is broadcast using an implementation of Zeroconf (`http://www.zeroconf.org/`).

3.4.4 Observations

The Local Surrogate Directory tactic places the responsibility of surrogate identification on the mobile device user. If surrogate metadata changes or new surrogates are made available, a cyber-foraging system will not have an automated way of updating the surrogate directory. The Cloud Surrogate Directory tactic requires the mobile device to know the address of the cloud server that holds the surrogate directory. The cloud server can become a single-point-of-failure if it becomes unavailable to mobile devices. In the cases that the cloud server acts as an intermediary it also becomes a potential bottleneck. The Surrogate Broadcast tactic offers the most flexibility but requires an agreement between mobile devices and surrogates on the broadcast protocol. Regarding trust, mobile devices will require additional components to determine whether broadcast information is coming from a valid, trusted surrogate (Section 4.4.1).

4. NON-FUNCTIONAL TACTICS

4.1 Resource Optimization

A scenario for Runtime Optimization is the following: A mobile app is enabled for cyber-foraging. Upon request for execution of computation that has been targeted for offload, the mobile app first checks if it is better from a performance and latency perspective to execute the computation locally or remotely. Given that the network conditions between the mobile device and the surrogate are not ideal, the computation is executed locally instead of offloaded to the surrogate.

4.1.1 Runtime Partitioning

In general, offloading is beneficial when large amounts of computation are needed with relatively small amounts of communication [17]. Runtime Partitioning enables mobile devices to make runtime decisions regarding the benefits of offloading. Computation is offloaded only if remote execution is better than local execution according to a defined optimization function (often called a utility function). Local execution cost typically takes into consideration the energy consumed by local execution as well as the local execution time. Remote execution cost typically considers the energy consumed by communication, the communication time based on payload size and network conditions, and remote execution time. If local execution cost is lower than remote execution cost then the computation is executed locally; if not, it is executed remotely (i.e., offloaded). In addition to the components required by the Computation Offload tactic, the Runtime Partitioning tactic requires an *Offload Decision Engine* component that compares predicted local execution cost against predicted remote execution cost. The MACS system [16] uses service metadata related to memory size, code size, and input/output parameter size; available memory information, CPU load and remaining battery on the mobile device; network connectivity and bandwidth information; and a pre-built energy model to make an offload decision.

Variation: User-Guided Runtime Partitioning. The Runtime Partitioning tactic assumes a static optimization function. However, in some systems what to optimize is determined based on user preferences or input. In the PowerSense system [20] the user can select a *Time Saver* option to minimize processing time or an *Energy Saver* option to minimize energy consumption. The system has a user interface on the mobile device to set these preferences.

4.1.2 Runtime Profiling

Systems that implement the Runtime Partitioning tactic (Section 4.1.1) require developer input or static profiling to obtain the values or models that are used in the calculation of the optimization function that determines whether code should run locally or remotely. However, models tend to be inaccurate because (1) applications are not deterministic; (2) smartphones scale the CPU's voltage dynamically to save energy (i.e., dynamic voltage scaling); (3) energy models highly depend on hardware configuration, usage, and even the battery model of a mobile device; and (4) network quality is highly variable and often unpredictable [7]. To account for this variability and take into consideration current conditions, once the offload operation ends, or periodically, the system updates the profiling data and models that are used by the optimization functions. In the MAUI system [6] a *Solver+Profiler* component uses data from the annotated method (inputs, outputs and CPU cycles), the *Device Energy Model*, network data obtained via a *Network Monitor*, and *Past Program Execution and Network Data* to compute an energy-efficient program partition. Once an offloaded method terminates, it updates the *Past Program Execution and Network Data* to better predict whether future invocations of the method should be offloaded.

4.1.3 Resource-Adapted Computation

In the Runtime Partitioning tactic (Section 4.1.1) a decision is made at runtime to execute code locally or remotely depending on an optimization function. In this tactic the local and remote code are identical. Even though this makes development and versioning easier, computation ends up being limited to what can execute on the mobile device, which will always lag behind static elements such as surrogates in terms of compute resources (power, CPU, memory, storage) [22]. Resource-Adapted Computation enables cyber-foraging systems to fully take advantage of the computing power of surrogates by adapting the computation to the resource on which it will be executing. In an image processing scenario, the object recognition algorithm that runs on the surrogate can be much more computation-intensive than the one that runs on the mobile device and can therefore deliver a much more precise result. In the Cuckoo system [14] the *Cuckoo Framework* generates an implementation of the same interface for a local and a remote service. Initially, the remote implementation will contain dummy method implementations, which the developer has to replace with real method implementations that can be executed at the remote location. The real methods can be identical to the local service implementation, but may also be completely different, because the remote implementation can run a different algorithm, use different libraries, or take advantage of parallelization on the more powerful surrogate.

Variation: Resource-Adapted Input. A variation of this tactic is for the mobile and surrogate versions of the offloaded code to be identical, but what varies is the input parameters. The enabler is that different input parameters will lead to different resource consumption. PowerSense [20] is an image processing system for dengue detection that uses the same algorithm locally and remotely for image processing, but uses lower resolution images if processed locally and higher resolution images if processed remotely because processing these higher quality images requires greater computing power.

4.1.4 Observations

The Runtime Partitioning and Runtime Profiling tactics assume that there is equivalent code for the offloaded computation on both the mobile device and the surrogate. This aspect limits the direct reusability of legacy code because a version would have to be written for the mobile device or surrogate depending on the original platform of the legacy code. In addition, the optimization function should not be a computation-intensive task because it would then cancel the benefits of cyber-foraging. Finally, data collection of app metadata to be used as optimization function parameters has to be gathered in advance using techniques such as static profiling. For the Runtime Profiling tactic, the cost of profiling is not negligible and can impact overall application performance [6]. System designers need to consider the type and frequency of data to capture at runtime. Finally, the Resource-Adapted Computation tactic requires developing, profiling and maintaining different versions of offloadable elements.

4.2 Fault Tolerance

A scenario for Fault Tolerance is the following: A mobile app is enabled for cyber-foraging and is leveraging a surrogate for computation offload. During the execution of the remote computation the mobile device loses connectivity to the surrogate. The mobile device detects the situation and executes the local copy of the computation instead with minimal effect on user experience.

4.2.1 Local Fallback

Due to movement of a mobile device to an area with no connectivity to the surrogate, problems with network quality, or service disruption, the mobile device may lose connectivity to the surrogate during the computation offload or data staging process. The *Local Fallback* tactic enables the cyber-foraging enabled mobile app to detect loss of connectivity and revert to local execution of the offloaded element. The MAUI system [6] detects failures using a simple timeout feature that returns control back to the mobile device if a disconnect occurs and resumes running the method on the local smartphone. After every offload operation, MAUI returns program state as part of the results, which is applied to the local computation so that state is synchronized between the local and remote computation.

4.2.2 Opportunistic Mobile-Surrogate Data Synchronization

Data-reliant cyber-foraging systems, as their name indicates, rely on stored data to fulfill their operations. The Opportunistic Mobile-Surrogate Data Synchronization tactic keeps data synchronized during periods of connection such that the system can continue operating in periods of disconnection. There are no systems in the primary studies that implement this tactic for fault tolerance as described, but the principle of using distributed storage in the Virtual Phone system [12], for example, is the same: to opportunistically keep data/state synchronized without placing the responsibility on the actual applications.

Variation: Opportunistic Surrogate-Cloud Data Synchronization. The principles of the Opportunistic Mobile-Surrogate Data Synchronization tactic can also be applied to handle disconnection between the surrogate and the cloud, especially for data staging systems. Opportunistic Surrogate-Cloud Data Synchronization enables a system to continue operating in the event of disconnection between the surrogate and the cloud and to synchronize data when reconnection occurs. To support this tactic, a *Data Synchronization Client* runs on the *Surrogate* and a *Data Synchronization Server* runs in the cloud. Trusted and Unmanaged Data Staging Surrogates [9] is a data staging system that uses a distributed filesystem based on Coda (`http://www.coda.cs.cmu.edu/`) between the surrogate and the cloud that supports disconnected operations to maintain data opportunistically synchronized such that it is available on the surrogate when needed.

4.2.3 Cached Results

Offload requests from mobile devices are not always as simple as request-response interactions. Some requests may take a long time to execute or may rely on data that has been gathered and maintained over time. In the case of disconnection between a mobile device and a surrogate during an offload operation, restarting the offload request or losing data is not desired. The Cached Results tactic enables a system to cache results and state on a surrogate until the mobile device is able to reconnect. In the Grid-Enhanced Mobile Devices system [11] the mobile device periodically sends a keep-alive message to the surrogate to inform that it is still connected. Before sending the results back to the mobile device, the surrogate checks the device status and if disconnected saves the results in a cache. When the mobile device reconnects, the surrogate gets the results from the cache and sends them back to the mobile device.

Variation: Client-Side Data Caching. The tactic as described caches results on the surrogate and sends them to mobile clients upon request or reconnection. A variation of this tactic that is useful for data staging systems that implement the In-Bound-Pre-Processing tactic (Section 3.2.2) is to cache collected data on the mobile device and send it to the surrogate upon reconnection. The Feel the World system [21] collects sensor data that can be aggregated and/or transformed locally on the mobile device and uploaded to the surrogate in real-time if the connection is available, or at a later moment if it is unavailable.

4.2.4 Alternate Communications

Cyber-foraging systems typically leverage single-hop, higher bandwidth communications mechanisms such as WiFi or short-range radio instead of broadband wireless (e.g., 3G/4G) because of the potential for energy savings and faster response time [3]. However, these mechanisms require the mobile device to be in proximity of the surrogate. The Alternate Communications tactic enables the system to switch to an alternate, potentially less energy-efficient communications mechanism, to continue serving the mobile user in spite of disconnection (even if in a degraded mode). Edge Proxy [2] is a data staging system that enables a user to be notified when web pages of interest change. By default it communicates using WiFi but when the surrogate is ready to send web page changes to the mobile device and detects that it is disconnected, it leverages the existing Short Message Service (SMS) infrastructure that most wireless carriers provide.

4.2.5 Eager Migration

Due to mobile device mobility or decrease in the quality of the communications channel between the mobile device and the surrogate, the mobile device might lose connectivity to the surrogate. The Local Fallback (Section 4.2.1), Cached Results (Section 4.2.3), and Alternate Communications (Section 4.2.4) tactics for fault tolerance are reactive; that is, they perform a corrective action after the disconnection is detected. The Eager Migration tactic takes a more proactive approach and migrates the offloaded computation to a connected surrogate before it becomes disconnected from the mobile device so that it can continue operating. In the Offloading Toolkit and Service system [25], if the communication between the surrogate and the mobile device deteriorates based on reaching an established threshold for connection quality, the execution of the offloaded code is terminated on the current surrogate and migrated to a connected target surrogate.

Variation: Lazy Migration. In Eager Migration the offloaded computation fully moves from a source surrogate to a target surrogate and the mobile device continues its interaction with the target surrogate. In Lazy Migration, the execution of the offloaded computation remains on the source surrogate but the interaction with the mobile device is handed off to the target surrogate. This means that all interaction between the mobile device and the source surrogate goes through the target surrogate that acts as an intermediary. This tactic is not present in any of the systems but was considered as an alternative for the Offloading Toolkit and Service system [25]. It was not selected because of the high bandwidth that already existed between surrogates to enable a fast full migration.

4.2.6 Observations

The Local Fallback tactic assumes that there is equivalent code for the offloaded computation on both the mobile device and the surrogate. Because disconnection may happen at any point in the offload process, this tactic is best fit for stateless request-response operations that can be restarted on the mobile device if the operation fails. Systems that implement the Just-In-Time Containers tactic (Section 4.3.1) with the Local Fallback tactic would require a component or a periodic clean-up process that destroys containers that are not being used in order to reduce the load on the surrogate. Systems that implement the Opportunistic Mobile-Surrogate Data Synchronization tactic need to be aware of the energy consumption on the mobile device for keeping data synchronized. Also, while disconnected, it is possible that data may not be up-to-date, which may lead to incorrect results for applications that operate on time-sensitive data. The Cached Results tactic is best fit for asynchronous interactions between mobile devices and surrogates or applications that are not time-sensitive or require immediate results. In addition, it requires a mechanism for detecting disconnection from mobile devices. The Alternate Communications tactic assumes that the mobile device is enabled to use the alternate communication mechanism. In addition, depending on the type of interaction between the surrogate and the mobile device (i.e., responding to a single offload request or sending data periodically to the mobile device), the surrogate would require a mechanism to determine when connectivity has been restored so it can go back to the default communications mechanism. Finally, the Eager Migration tactic requires the source and target surrogates to be connected. The impact on the user experience will highly depend on the bandwidth between surrogates. In addition, the system has to obtain any parameters for the algorithm that determines potential disconnection, such as the distance and communications quality between the mobile device and both the source and target surrogate.

4.3 Scalability/Elasticity

A scenario for Scalability/Elasticity is the following: A mobile app is enabled for cyber-foraging and is leveraging a surrogate for computation offload that is also being leveraged by other mobile apps on other mobile devices. The surrogate is able to optimize computing resources either locally or by leveraging other connected surrogates so that multiple mobile devices can be supported with the goal of minimal effect on user experience due to surrogate load.

4.3.1 Just-in-Time Containers

In an operational cyber-foraging scenario a single surrogate may support multiple mobile users. To decrease the load on a surrogate, and therefore support a greater number of offload requests, the Just-in-Time Containers tactic creates a container and/or an instance of the offloaded code upon receipt of an offload request and then destroys the instance of the offloaded code when the offload request is completed. In the Grid-Enhanced Mobile Devices system [11] a *Deputy Object* is created for each offload request (task) from a mobile device in the *Grid Gateway*. When the task is completed and the mobile device terminates the connection to the Grid Gateway, resources on the surrogate are released and the Deputy Object is destroyed.

4.3.2 Right-Sized Containers

In an operational cyber-foraging scenario a single surrogate may support multiple mobile users. However, not all mobile users are offloading the same computation. Some users may be executing a small task that does not require a large quantity of surrogate resources while others may be executing very computation-intensive tasks that require much more resources. To optimize resources on a surrogate, and therefore support a greater number of offload requests, the Right-Sized Containers tactic creates a container for the offloaded code that is of the smallest size possible in order to run the offloaded computation, based on computation requirements metadata related to the offloaded code. In the ThinkAir system [15], when a surrogate receives an offload request, the *ThinkAir Framework* on the surrogate determines the configuration of the VM (or VMs) to allocate for the task based on app requirements in the offload request that indicate the need for extra computing power (the system has six VM configurations which differ in terms of CPU and memory).

Variation: Dynamically-Sized Containers. The ThinkAir system [15] also implements this tactic. If an error occurs at runtime that would indicate that the VM does not have the necessary computing power for the task, such as an *OutOfMemoryError* error, the system starts a more powerful VM and moves the offload request to the newly started VM.

4.3.3 Surrogate Load Balancing

In an operational cyber-foraging scenario the relationship between mobile devices and surrogates may be many-to-many, meaning that multiple mobile devices may be leveraging multiple surrogates for computation offload and data staging. The Surrogate Load Balancing tactic enables surrogates to send offloaded computation or data to other less-loaded, connected surrogates in order to provide a better user experience to all mobile devices. In the Cloud Operating System to Support Multi-Server Offloading (COS) system [13] application modules are implemented as *SALSA Actors* that are self-contained and therefore can easily migrate between a source surrogate and a target surrogate. When the source surrogate reaches a load threshold, it informs the *COS Manager*, which determines the optimal target surrogate based on resource availability, communication cost with other actors, and the cost for migration, and prepares it for migration.

4.3.4 Observations

The Just-in-Time Containers and Right-Sized Containers tactics have a greater startup time than tactics in which the offloaded code is already running because they have to set up the container, which is the execution environment for the offloaded code. In addition, the Right-Sized Containers tactic requires a surrogate to maintain different container configurations. The Surrogate Load Balancing tactic requires the source and target surrogates to be connected. The impact on the user experience will highly depend on the on the bandwidth between surrogates. The source surrogate requires a mechanism to access the load level of all connected surrogates (or an external manager that maintains this information) in order to migrate computation to the less-loaded surrogate and keep the load on all the surrogates balanced.

4.4 Security

One of the main findings from the primary studies is that there is very little discussion of system-level concerns that have to be addressed when moving from experimental prototypes to operational systems. One of these system-level concerns is security [19]. A scenario for Security is the following: A mobile app is enabled for cyber-foraging and is in the process of discovering a surrogate for computation offload. User and surrogate credentials are exchanged and validated before the offload process so that the mobile app and surrogate can interact according to agreed security policies.

4.4.1 Trusted Surrogates

When a mobile device discovers a surrogate it expects a trustworthy surrogate execution environment, meaning that once an offload operation starts, code and data are not maliciously modified or stolen and that it provides trustful services. In the same way, a surrogate expects that a mobile device is a valid client and that it will not offload malicious code or use it as a vehicle to other code and data offloaded by other mobile devices. The Trusted Surrogate tactic adds this trust element to the interaction between a mobile device and a surrogate. The only system that implements a trust solution that uses a third-party trusted authority is the Trusted and Unmanaged Data Staging Surrogates system [9]. The user's idle desktop serves as the trusted third party that sits in between the file server and the surrogate. When the mobile client requests a file, it communicates with the *Data Pump* that runs on the desktop to obtain the key and hash for the requested data file. The Data Pump retrieves the data file from the file server and encrypts it before sending it to the surrogate for staging. It then sends the mobile client the key and hash for the file so it can be compared it to the hash of the file that is retrieved from the surrogate to determine if the file has been tampered with.

4.4.2 Observations

Any trust mechanism will be constrained by how the trust relationship is established. Password-based approaches such as those employed by systems in which surrogates are owned by the mobile device user require users to be registered on the surrogate. Hardware-based approaches such as TPM (Trusted Platform Module) require surrogates to have TPM chips on them. Systems that rely on third parties have to be connected to online authorities or require certificates and keys to be obtained from a central certificate authority.

5. RELATED WORK

There are several studies that survey the field of mobile cloud computing and identify cyber-foraging as a research area and challenge, but are not SLRs and do not have an architecture focus. The work that is most similar to ours is by Flinn et al [8] that presents a discussion of representative cyber-foraging systems and their characteristics. However, it is limited to a small number of systems and does not follow a systematic process. To the best of our knowledge, ours is the first systematic literature review related to architectures for cyber-foraging.

As far as architectural tactics for cyber-foraging, this is the first attempt to codify design decisions in software architectures for cyber-foraging systems into a set of tactics.

6. CONCLUSIONS AND NEXT STEPS

We presented a set of architectural tactics for cyber-foraging that were obtained from the results of an SLR in architectures for cyber-foraging systems. Common design decisions present in the cyber-foraging systems were codified into architectural tactics for cyber-foraging and then grouped into functional and non-functional tactics. Functional tactics provide the basic cyber-foraging operations and non-functional tactics are combined with the functional tactics to support required system qualities. Each tactic has trade-offs that were briefly presented as observations in each tactic category.

The next steps in our research are to create case studies that validate these tactics in real systems to demonstrate that they satisfy the functional and non-functional quality attribute responses that they are intended to promote. Because tactics are not used in isolation, but rather combined to satisfy system requirements, the case studies will be analyzed to identify tactics that are commonly used together and codify them into architectural patterns [4] for cyber-foraging systems.

7. ACKNOWLEDGMENTS

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution (DM-0002040).

8. REFERENCES

[1] P. Angin and B. Bhargava. An agent-based optimization framework for mobile-cloud computing. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 4:1–17, 2013.

[2] T. Armstrong, O. Trescases, C. Amza, and E. de Lara. Efficient and transparent dynamic content updates for mobile clients. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 56–68. ACM, 2006.

[3] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 280–293, New York, NY, USA, 2009. ACM.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[5] B.-G. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *Proceedings of the 12th conference on Hot topics in operating systems*, pages 8–8. USENIX Association, 2009.

[6] E. Cuervo. *Enhancing Mobile Devices through Code Offload*. PhD thesis, Duke University, 2012.

[7] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 335–348, New York, NY, USA, 2011. ACM.

[8] J. Flinn. Cyber foraging: Bridging mobile and cloud computing. In M. Satyanarayanan, editor, *Synthesis Lectures on Mobile and Pervasive Computing*. Morgan & Claypool Publishers, 2012.

[9] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan. Data staging on untrusted surrogates. In *Proceedings 2nd USENIX Conference on File and Storage Technologies (FAST03), Mar 31-Apr 2, 2003, San Francisco, CA.*, 2003.

[10] S. Goyal. *A Collective Approach to Harness Idle Resources of End Nodes*. PhD thesis, School of Computing, University of Utah, 2011.

[11] T. Guan. *A System Architecture to Provide Enhanced Grid Access for Mobile Devices*. PhD thesis, University of Southampton, 2008.

[12] S.-H. Hung, J.-P. Shieh, and C.-P. Lee. Migrating android applications to the cloud. *International Journal of Grid and High Performance Computing (IJGHPC)*, 3(2):14–28, 2011.

[13] S. Imai. Task offloading between smartphones and distributed computational resources. Master's thesis, Rensselaer Polytechnic Institute, 2012.

[14] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: a computation offloading framework for smartphones. In *Mobile Computing, Applications, and Services*, pages 59–79. Springer, 2012.

[15] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[16] D. Kovachev and R. Klamma. Framework for computation offloading in mobile cloud computing. *International Journal of Interactive Multimedia and Artificial Intelligence*, 1(7):6–15, 2012.

[17] K. Kumar and Y.-H. Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, Apr. 2010.

[18] G. Lewis and P. Lago. A catalogue of architectural tactics for cyber-foraging. Technical Report 2014-12.001, VU University Amsterdam, Dec. 2014. `http://www.cs.vu.nl/~patricia/Patricia_Lago/Shared_files/report-tactics-cyber-foraging.pdf`.

[19] G. Lewis, P. Lago, and G. Procaccianti. Architecture strategies for cyber-foraging: Preliminary results from a systematic literature review. In *Proceedings of the 8th European Conference on Software Architecture (ECSA 2014)*, volume 8627 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2014.

[20] J. Matthews, M. Chang, Z. Feng, R. Srinivas, and M. Gerla. PowerSense: power aware dengue diagnosis on mobile phones. In *Proceedings of the First ACM Workshop on Mobile Systems, Applications, and Services for Healthcare*, page 6. ACM, 2011.

[21] T. Phokas, H. Efstathiades, G. Pallis, and M. Dikaiakos. Feel the world: A mobile framework for participatory sensing. In *Mobile Web Information Systems*, volume 8093 of *Lecture Notes in Computer Science*, pages 143–156. Springer, 2013.

[22] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 8(4):10–17, Aug 2001.

[23] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for VM-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, 2009.

[24] Y. Xiao, P. Simoens, P. Pillai, K. Ha, and M. Satyanarayanan. Lowering the barriers to large-scale mobile crowdsensing. In *Mobile Computing Systems and Applications*, 2013.

[25] K. Yang, S. Ou, and H.-H. Chen. On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications. *Communications Magazine, IEEE*, 46(1):56–63, 2008.