A Catalogue of Architectural Tactics for Cyber-Foraging

Grace A. Lewis and Patricia Lago
VU University Amsterdam
{g.a.lewis, p.lago}@vu.nl

Technical Report 2014-12.001

December 18, 2014

Contents

List of Figures

List of Tables

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 and 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 to enable 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 reports presents a catalogue of architectural tactics for cyber-foraging that was derived from the results of a systematic literature review (SLR) 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.

Chapter 1

Introduction

Mobile Cloud Computing (MCC) refers to the combination of mobile devices and cloud computing in which cloud resources perform computing-intensive tasks and store massive amounts of data. Increased mobile device capabilities, combined with better network coverage and speeds, have enabled MCC such that mobile devices have become for many the preferred form for interacting with the Internet, social media, and the enterprise. However, mobile devices still offer less computational power than conventional desktop or server computers, and limited battery life remains a problem especially for computation- and communication-intensive applications.

Cyber-foraging is an area of work within MCC 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.

One of the main challenges of building cyber-foraging systems is the dynamic nature of the environments that they operate in. For example, the connection to an external resource may not be available when needed or may become unavailable during a computation offload or data staging operation. As another example, multiple external resources may be available for a cyber-foraging system but not all have the required capabilities. Adding capabilities to deal with the dynamicity of the environment has to be balanced against resource consumption on the mobile device so as to not defeat the benefits of cyber-foraging. Being able to reason about the behavior of a cyber-foraging system in light of this uncertainty is key to meeting all its desired quality attributes, which is why software architectures are especially important for cyber-foraging systems.

Given the potential complexity of cyber-foraging systems, it would be of great value for software architects to have a set of reusable software architectures and design decisions that can guide the development of these types of systems, but also the rationale that went to making these decisions, as well as the external context/environment in which they were made; this is called *architectural knowledge* [KLvV06][LA06]. One way to capture architectural knowledge is in the form of *architectural tactics*. We define architectural tactics as

design decisions that influence the achievement of a quality attribute response [BCK12].

This report contains a catalogue of architectural tactics for cyber-foraging that was derived from the results of an SLR on architectures for cyber-foraging systems. The details of the SLR can be found at `http://www.cs.vu.nl/~patricia/Patricia_Lago/Shared_files/SLR-ArchCyberForaging.pdf`. A set of 57 primary studies was identified.[1] Table 1.1 shows the computation offload systems found in the primary studies and Table 1.2 shows the data staging systems. The columns in the table correspond to decisions on where, when and what to offload from the perspective of the mobile device.

- 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 offload or is there a runtime decision on whether or not to offload?
- 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 preliminary analysis of these studies has been published in [LLP14].

Table 1.1: Computation Offload Systems in Primary Studies

System	Where			When		What													
						Granularity					Payload								
	Prox. Disconnected	Prox. Connected	Remote	Runtime Decision	Always Offload	Process	Function	Component	Service	Application	Computation	Partitioning Algo.	Parameters	Application State	Device Context	Source Location	Setup Instructions	Continuous Data
mHealthMon [AP13]			X	X					X				X					
Mobile Agents [AB13]			X	X				X			X		X					
Clone-to-Clone (C2C) [ACH12]	X			X			X						X					
Chroma [BGSH07]	X			X			X						X					
Collaborative Applications [CH11]	X			X			X						X	X				
Computation and Compilation Offload [CKK+04]	X			X			X						X					
Cloud Media Services [CP13]			X		X			X							X			
Roam [CSW+04]		X	X	X				X					X	X		X		
CloneCloud [CM09]	X				X	X								X				
MAUI [Cue12]	X		X	X			X						X					
Kahawai [Cue12]	X				X					X								X
HPC-as-a-Service [Dug11]	X		X		X				X				X					
OpenCL-Enabled Kernels [EW11]	X		X	X				X			X		X					
Real Options Analysis [EML11]	X		X	X			X						X					

Continued on next page

[1] The total of primary studies is 57 but the total of systems analyzed is 52 for computation offload and 8 for data staging for a total of 60 systems because two of the computation offload studies present two different systems and one study presents systems for both computation offload and data staging.

Table 1.1 – *Continued from previous page*

System	Where			When		What													
						Granularity					Payload								
	Prox. Disconnected	Prox. Connected	Remote	Runtime Decision	Always Offload	Process	Function	Component	Service	Application	Computation	Partitioning Algo.	Parameters	Application State	Device Context	Source Location	Setup Instructions	Continuous Data
3DMA [FMD05]			X	X				X					X					
Spectra [FPS02]	X			X			X						X					
AlfredO [GRJ+09]			X	X				X				X	X					
Collective Surrogates [Goy11]		X	X		X					X			X				X	
Grid-Enhanced Mobile Devices [Gua08]		X	X		X			X					X					
Cloudlets [HLSS11]	X				X					X	X		X					
Virtual Phone [HSL11]			X		X			X						X				
Single-Server Offloading [Ima12]	X			X			X						X					
Cloud Operating System [Ima12]		X			X			X			X		X					
Android Extensions [I+12]			X		X			X					X					
ThinAV [JBA12]		X	X		X				X				X					
Cuckoo [KPKB12]	X		X	X				X			X		X					
ThinkAir [KAH+12]			X	X			X				X		X					
MACS [KK12]	X		X	X				X			X		X					
Scavenger [Kri10]	X			X				X			X		X					
AMCO [KT13]	X		X	X				X					X	X				
MCo [Lee12]			X		X			X			X		X					
PowerSense [MCF+11]	X			X					X				X					
AIDE [MGB+02]	X		X	X				X					X					
Application Virtualization [ML13]	X				X					X	X		X					
PARM [MV03]	X			X				X					X					
Resource Furnishing System [OSP07]		X	X		X					X								X
Cloud Personal Assistant [OG13]		X	X		X				X				X					
SOME [PCCY12]			X		X		X						X					
SmartVirtCloud [PXJZ13]		X		X			X				X		X					
Odessa [RSM+11]	X		X	X				X					X					
Smartphone-Based Social Sensing [Rac12]	X		X	X				X					X					
MAPCloud [RVMV12]		X	X	X					X				X				X	
VM-Based Cloudlets [SBCD09]	X				X					X	X		X					
IC-Cloud [SPN+13]	X		X	X			X						X					
SPADE [SVF08]			X		X			X					X					
Slingshot [SF05]		X			X					X			X					
Offloading Toolkit and Service [YOC08]	X		X	X				X			X		X					
Mobile Data Stream Application Framework [YCY+13]			X	X				X				X	X					
Heterogeneous Auto-Offloading Framework [ZGHC09]			X	X				X					X					
Weblets [ZKJG11]			X	X				X					X					
DPartner [ZHZ+12]	X		X	X				X					X					
Elastic HTML5 [ZJGK12]	X		X	X				X					X			X		

Chapter 2 presents the architectural tactics for cyber-foraging that were codified from the primary studies, grouped into functional and non-functional tactics. Chapter 3 presents related work. Chapter 4 concludes the report and outlines the next steps in our research.

Table 1.2: Data Staging Systems in Primary Studies

System	Where			When		What							
						Data Type				Data Operations			
	Prox. Disconnected	Prox. Connected	Remote	Runtime Decision	Always Offload	Data Updates	Application Data	Data Files	Field-Collected Data	Pre-Fetching	In-Bound Processing	Out-Bound Processing	Storage
Edge Proxy [ATAdL06]		X			X	X					X		
Mobile Information Access Architecture for Occasionally-Connected Computing [BWYH06]	X				X		X			X			
Trusted and Unmanaged Data Staging Surrogates [FSTS03]	X				X			X		X			
Android Extensions [I+12]			X		X		X						X
Telemedik [KMM+07]	X		X		X		X				X		
Feel the World [PEPD13]	X	X	X		X				X				X
Large-Scale Mobile Crowdsensing [XSP+13]		X			X				X			X	
Sonora [YQC+12]	X	X	X		X				X			X	

Chapter 2

Architectural Tactics for Cyber-Foraging

The tactics that are presented in this chapter 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. We codified common design decisions into architectural tactics and grouped these tactics into functional and non-functional tactics.

Figure 2.1 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 is described using the following template:

- Motivation: rationale behind the tactic
- Description: components introduced by the tactic and explanation of their roles
- Constraints: necessary conditions for applying the tactic in an existing software architecture
- Example: previous application of the tactic in one or more systems; the example(s) map back to the elements of the architecture diagram presented in the description
- Dependencies: whether the tactic requires other tactics to be applied
- Variations (Optional): slight variations of the tactic.

The tactics are divided 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. Non-functional tactics have to be used in conjunction with functional tactics.

The tactics described in this chapter will include a nearby surrogate as the offload target to take advantage of the lower latency and battery consumption

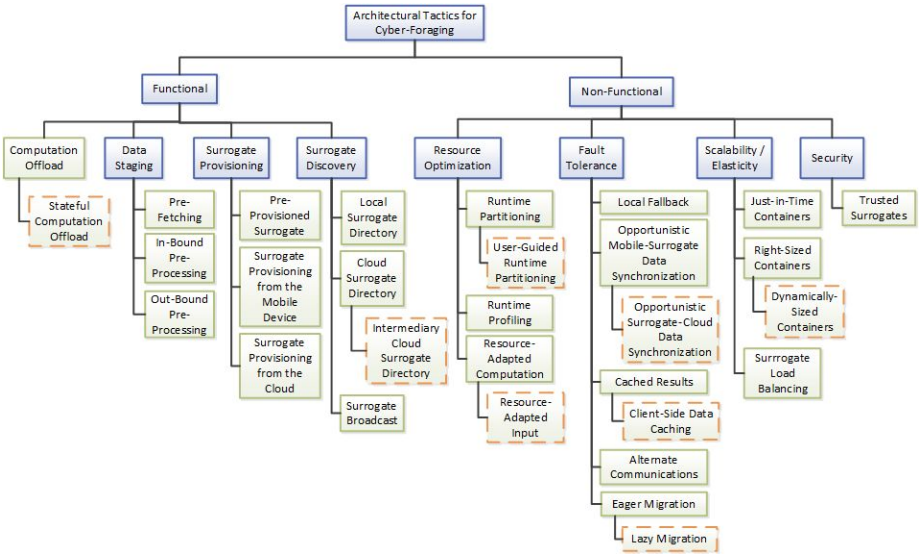

Figure 2.1: Architectural Tactics for Cyber-Foraging

that come from using WiFi or short-range radio instead of broadband wireless (e.g., 3G/4G) [BBV09][LM02]. The notion is that the elements of the tactic that apply to the surrogate will also apply to a remote cloud server as the offload target.

2.1 Functional Architectural Tactics for Cyber-Foraging

2.1.1 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 to a nearby surrogate with minimal disruption to the mobile device user.

The Computation Offload tactic can be found in the computation offload systems shown in Table 1.1 for which *What to Offload - Granularity* is Component, Service, or Application. It can also be mapped to the data staging systems in Table 1.2 for which *What to Offload - Data Operations* corresponds to Storage because even though these systems are using the surrogate for extended storage they are indeed offloading the data management computation.

The Computation Offload tactic needs to be combined with a Surrogate Provisioning tactic (Section 2.1.3) that prepares the surrogate for computation offload. It also needs to be combined with a Surrogate Discovery tactic (Section 2.1.4) to discover surrogates in the environment. This tactic is also often combined with non-functional tactics to achieve desired system qualities. For example, it is often combined with Resource Optimization tactics (Section 2.2.1)

to make better decisions on resource usage and with Fault Tolerance tactics (Section 2.2.2) to attempt to provide continued operations.

Motivation. Mobile devices still do not have the computing power and battery life that will allow them to perform effectively over long periods of time or for executing applications that require extensive communication or computation. 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 [BBV09][LM02] and improves the user experience especially for highly-interactive applications.

Description. Figure 2.2 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 2.2 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 because the *Offload Server* becomes a bottleneck. However, some systems that implement Fault Tolerance tactics (Section 2.2.2) 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.

Constraints. The tactic as described assumes that (1) offloaded computation already exists on the surrogate (provisioned via the application of a Surrogate Provisioning tactic (Section 2.1.3)), (2) computation that is marked for offload is always offloaded, and (3) the surrogate is always available.

Example. An example of how to apply the Computation Offload tactic is the Mobile Agents system [AB13] shown in Figure 2.3. In the Mobile Agents system 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 *Execution Manager* determines if the agent marked as offloadable should be offloaded based on a comparison of local and remote execution times (Section 2.2.1.1 contains details on runtime partitioning). If so, the *Execution Manager* sends the *Mobile Agent* (which carries its input parameters) to the *Agent Management System* so that it can migrate the *Mobile Agent* to the *JVM Container* in the *Cloud Host*. After migration, the offloaded component starts executing and communicates directly with the *Mobile App*.

Dependencies. The Computation Offload tactic requires a Surrogate Provisioning tactic (Section 2.1.3) that prepares the surrogate for computation of-

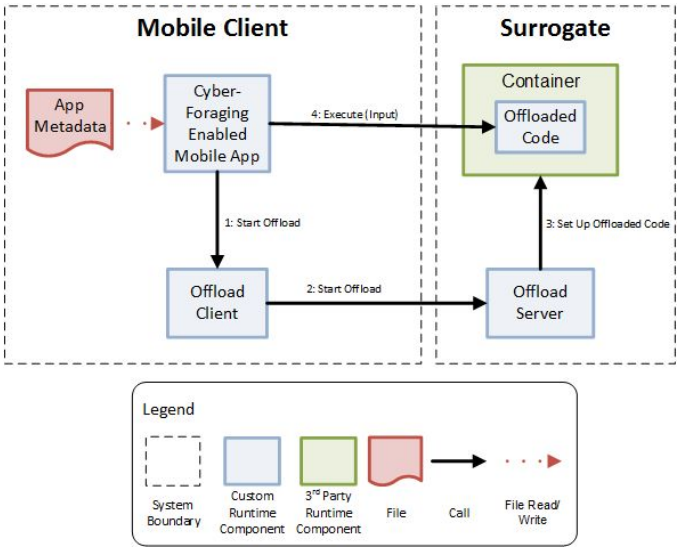

Figure 2.2: Computation Offload Tactic

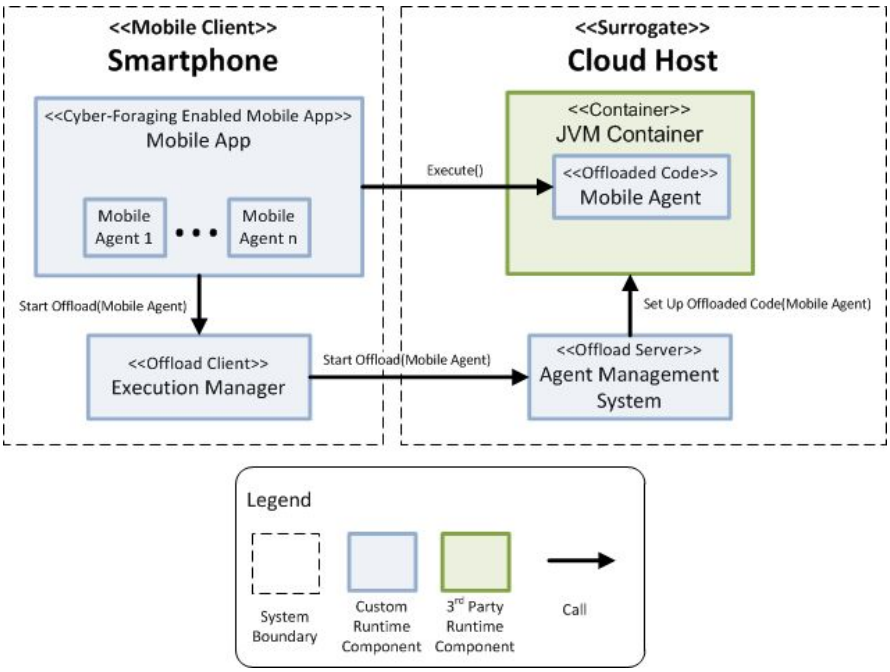

Figure 2.3: Mobile Agents as an Example of the Computation Offload Tactic

fload.

Variation: Stateful Computation Offload. The tactic as described 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 the 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 in a *State Manager* component that is invoked either periodically or on-demand has to execute before the offloaded code is executed to guarantee that the state is equivalent on both sides. This stateful variation of the tactic can be mapped to the computation offload systems in Table 1.1 for which *What to Offload - Granularity* corresponds to Process or Function.

An example of how to apply the Stateful Computation Offload tactic is the CloneCloud system [CM09] shown in Figure 2.4, marked with numbers that indicate the sequence of operations. In CloneCloud, the *Container* on the *Surrogate* is a *Clone Application VM* of the *Application VM* that is executing on the mobile device. At runtime, when a computation block of the *Instrumented Mobile App* is marked for offload, a *Migrator* component running in the VM is invoked that puts the running process into a sleep state and transfers this state to the *Clone Application VM* via the pair of *Node Managers* running on both the mobile device and the surrogate. The *Migrator* in the *Clone Application VM* creates a new process with the received state and marks it as runnable so it executes. The cloned process executes from the beginning of the computation block until it reaches the end of the computation block. The *Migrator* on the cloned VM then transfers the new process state back to the mobile device. The *Migrator* on the mobile device receives the new process state, merges it with the sleeping process, and then wakes up the sleeping process to continue its execution.

2.1.2 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.

The Data Staging tactics require a configuration such as the one shown in Part(c) of Figure **??** 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.

The Data Staging tactics need to be combined with a Surrogate Provisioning tactic (Section 2.1.3) that prepares the surrogate for data staging. This tactic is also often combined with other functional and non-functional tactics to achieve

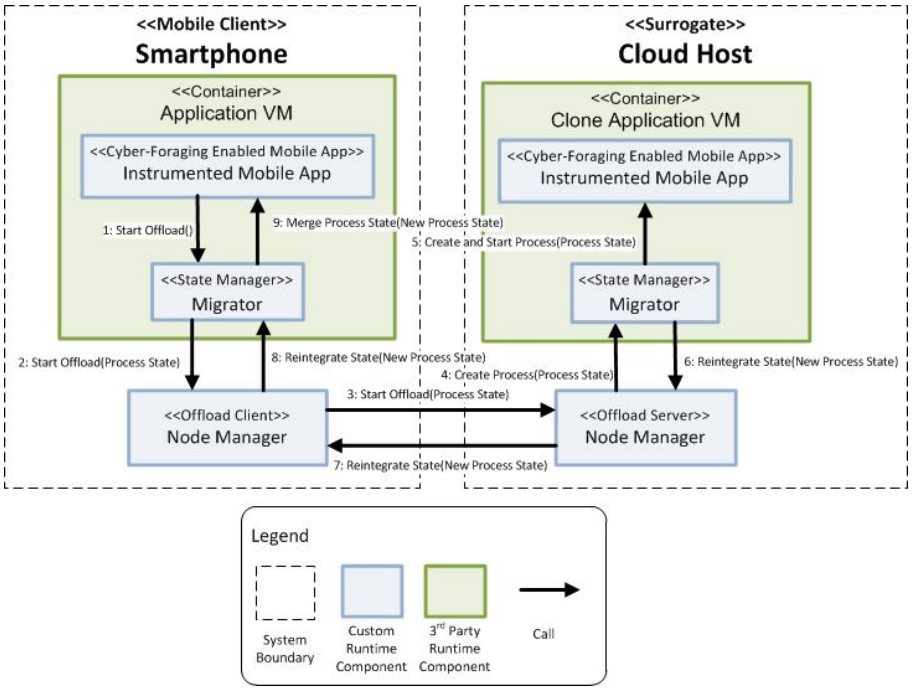

Figure 2.4: CloneCloud as an Example of the the Stateful Computation Offload Tactic

desired system qualities. It is typically combined with a Surrogate Discovery tactic (Section 2.1.4) to discover surrogates in the environment. It is also often combined with Fault Tolerance tactics (Section 2.2.2) to attempt to provide continued operations.

2.1.2.1 Pre-Fetching

The Pre-Fetching tactic can be found in the data staging systems shown in Table 1.2 for which *What to Offload - Data Operations* is Pre-Fetching.

Motivation. 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.

Description. Figure 2.5 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* (or the equivalent of a master data repository), 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, that *Data Staging Manager* will sync the *Cache* with the *Cloud Data Repository* to ensure that data is consistent locally and remotely.

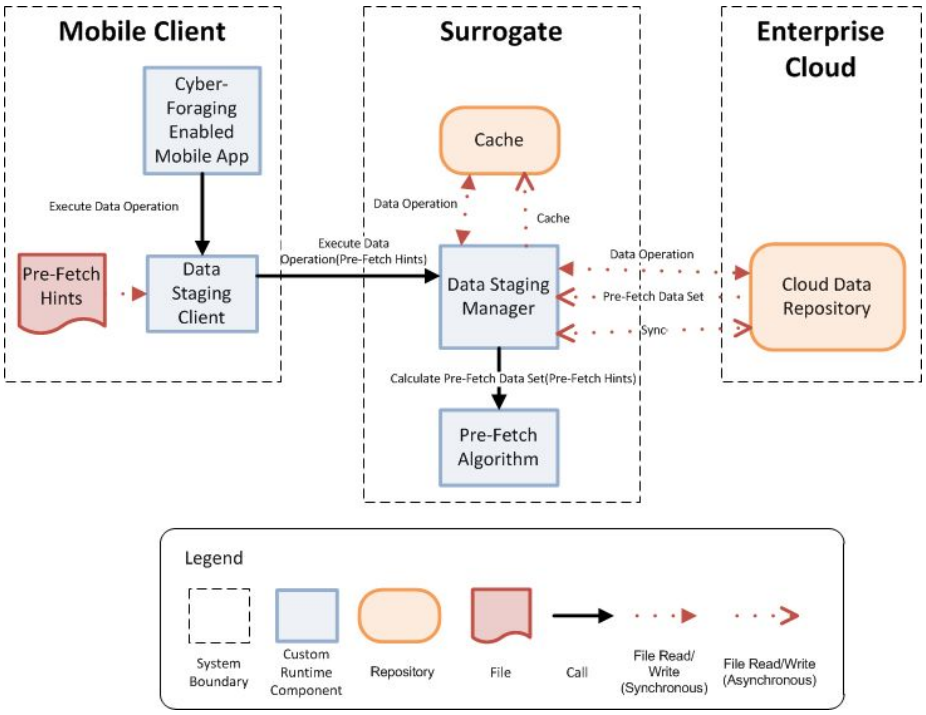

Figure 2.5: Pre-Fetching Tactic

Constraints. The tactic as presented requires connectivity between the mobile device and the surrogate for access to any data that is being staged, and eventual connectivity between the surrogate and the enterprise cloud to serve cache misses and synchronize data. The tactic also assumes that there is a mechanism

in place, either manual or automatic, to resolve any synchronization conflicts between the Cache and the Cloud Data Repository, especially if cached data is not read-only.

Example. An example of how to apply the Pre-Fetching tactic is the Trusted and Unmanaged Data Staging Surrogates system [FSTS03] shown in Figure 2.6. Data is staged on a *Staging Server* in the *Surrogate*. A *Client Proxy* running on the *Wimpy Client* intercepts all data operations. If it detects high latency it sends the data operation to the *Surrogate*, which then uses a pre-defined *User Role* to determine the initial set of files that the user is going to need based on the this role. The *User Role* basically establishes the set of files that are commonly used together. The *Staging Server* obtains the set of files from the *File Server* and caches them on the surrogate.[1] After the *Cache* has been loaded with the 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.

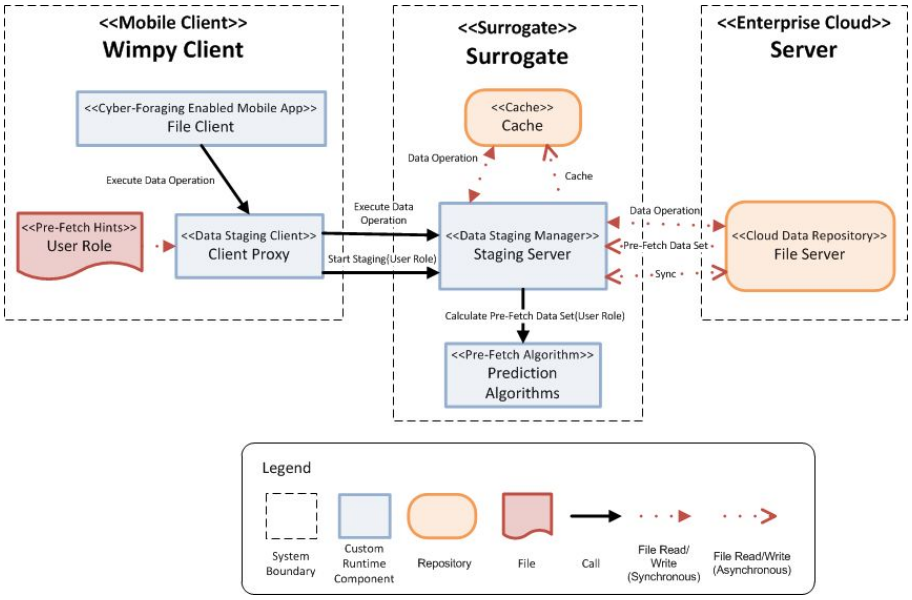

Figure 2.6: Trusted and Unmanaged Data Staging Surrogates as an Example of the Pre-Fetching Tactic

Dependencies. The Pre-Fetching tactic requires a Surrogate Provisioning tactic (Section 2.1.3) that prepares the surrogate for data staging.

2.1.2.2 In-Bound Pre-Processing

The In-Bound Pre-Processing tactic can be found in the data staging systems shown in Table 1.2 for which *What to Offload - Data Operations* is In-Bound

[1] For simplicity, the desktop and its trusted authority role are not included in the discussion of this tactic but are addressed in Section 2.2.4.1.

Processing.

Motivation. Data-intensive mobile apps often rely on data that resides in the cloud. However, access to this data is likely over a lower-bandwidth and multi-hop connection, that in addition consumes more energy than the single-hop connection that exists between a mobile device and a surrogate. 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.

Description. Figure 2.7 shows the main elements of the In-Bound Processing tactic. This tactic requires a *Communications Manager* that runs on the *Mobile Client* and handles all communication with the *Data Processor* on the *Surrogate*. The *Mobile Client* can request data on demand or periodically (synchronous) or can register with the surrogate for data of interest (asynchronous). In the case of synchronous requests, as shown by the S# operations in Figure 2.7, the *Cyber-Foraging-Enabled Mobile App* requests data via the *Communications Manager*. The *Data Processor* retrieves the data from the *Cloud Data Repository* and pre-processes it according to defined algorithms/rules before sending the data to the mobile app. The *Data Processor* may store data in its local *Cache* for additional processing, to serve additional requests based on the same data, or if the algorithm/rules involve partitioning or priorization of data such that it is sent incrementally upon request. In case of asynchronous requests, as shown by the A# operations in Figure 2.7, the *Cyber-Foraging-Enabled Mobile App* registers for data of interest via the *Communications Manager*. The *Data Processor* periodically polls the *Enterprise Cloud* for the data of interest (e.g., new data, updated data, data conditions satisfied) and when conditions are met it sends the data asynchronously back to the mobile app using some form of callback mechanism.

Constraints. The tactic as presented requires (1) connectivity between the mobile device and the surrogate for access to any data that is being staged, and (2) connectivity between the surrogate and the enterprise cloud to receive data as required.

Example. An example of how to apply the In-Bound Pre-Processing tactic is the Edge Proxy system [ATAdL06] shown in Figure 2.8. The Edge Proxy system uses a surrogate called an *Edge Server* to monitor changes in web pages on behalf of a *Web Browser* running on the *Mobile Device*. The user marks areas of interest on a web page (e.g., stock prices, temperature, news) and sends them to an *Edge Proxy* running on the *Edge Server* via the *Mobile Proxy*. The *Edge Proxy* saves the current state of the web page along with the areas of interest in its *Cache*. The *Edge Proxy* then 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. Instead of sending separate messages for the web page and its embedded objects, the *Edge Proxy* bundles the web page with all its embedded objects in a single batch update message, further reducing the amount of communication between the *Mobile Device* and the *Surrogate*.

Dependencies. The In-Bound Pre-Processing tactic requires a Surrogate Provisioning tactic (Section 2.1.3) that prepares the surrogate for data staging.

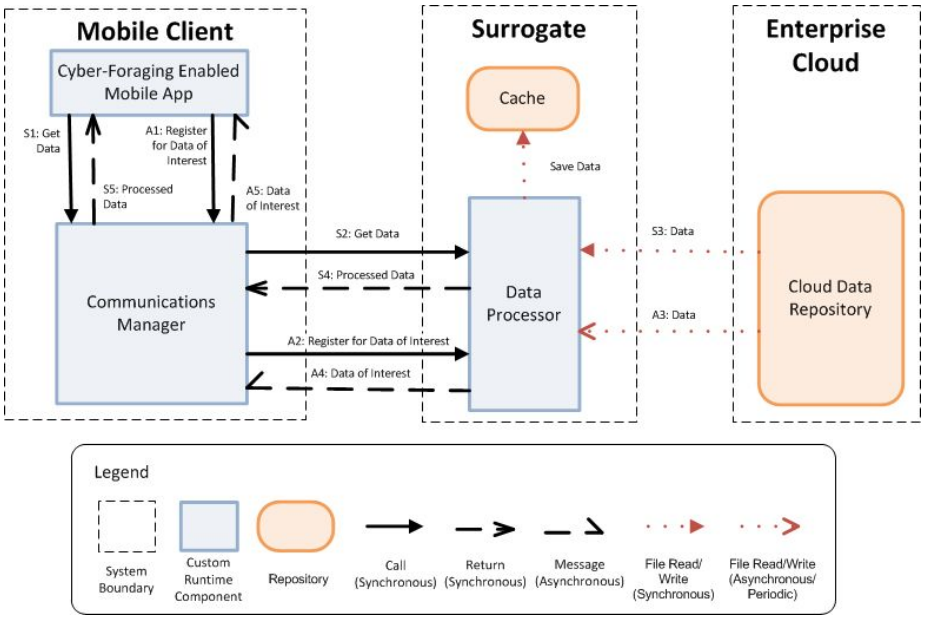

Figure 2.7: In-Bound Pre-Processing Tactic

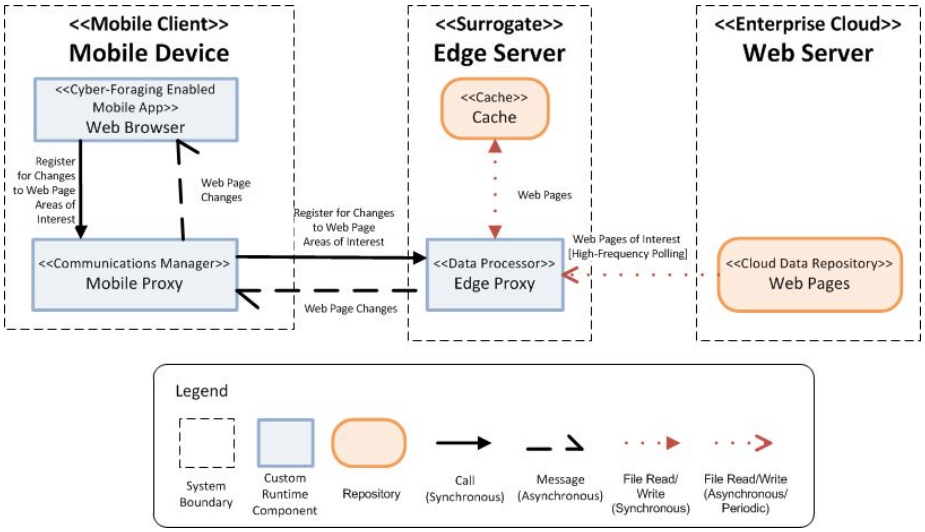

Figure 2.8: Edge Proxy an Example of the In-Bound Pre-Processing Tactic

2.1.2.3 Out-Bound Pre-Processing

The Out-Bound Pre-Processing tactic can be found in the data staging systems shown in Table 1.2 for which *What to Offload - Data Operations* is Out-Bound Processing.

Motivation. 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. Complete data from the mobile device and/or the surrogate can be uploaded to the cloud when network connectivity is available.

Description. Figure 2.9 shows the main components of the Out-Bound Pre-Processing tactic. This tactic requires a *Mobile Sensing App* that uses a *Communications Manager* on the mobile device to buffer data to send to its counterpart on the *Surrogate*. The *Communications Manager* can also batch data according to user or application preferences to conserve the energy spent on turning the radio on and off for communication. The *Communications Manager* on the *Surrogate* receives the data and stores it in a local *Cache*. One or more *Data Processing Applications* on the *Surrogate* can either subscribe to data coming in from the *Mobile Device*, perform continuous processing and forwarding of the data as it is coming in, or provide on-demand capabilities to other mobile devices being served by the same surrogate or cloud applications.

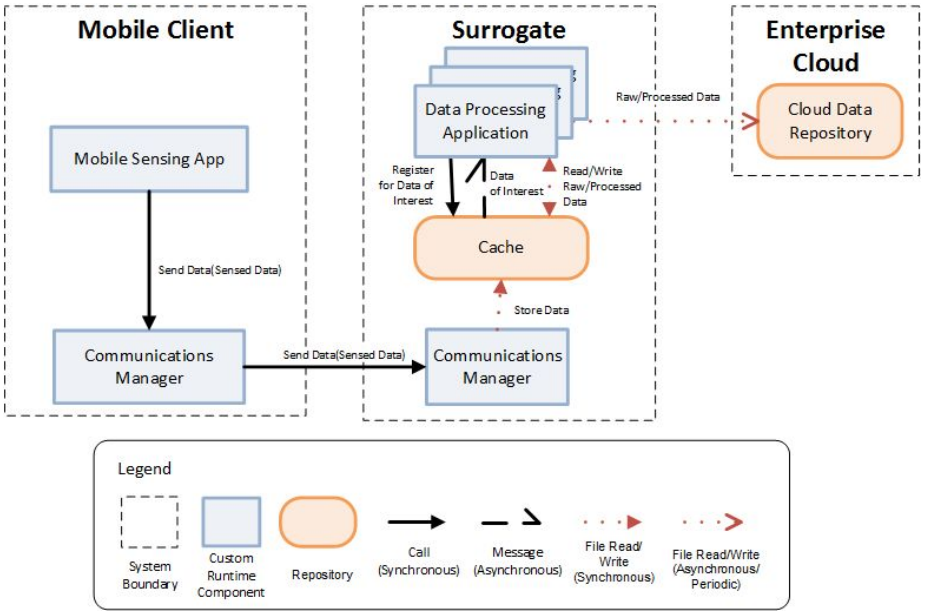

Figure 2.9: Out-Bound Pre-Processing Tactic

Constraints. The tactic as presented requires (1) eventual connectivity between the mobile device and the surrogate to offload data captured in the field and (2) eventual connectivity between the surrogate and the enterprise cloud to offload data that is staged on the surrogate.

Example. An example of how to apply to Out-Bound Processing tactic is the Large-Scale Mobile Crowdsensing system [XSP+13]. Crowdsensing refers to individuals using mobile devices with sensors that share information about an event or task of interest such as environmental monitoring, public safety, traffic monitoring, or collaborative searches. As shown in Figure 2.10, the Large-Scale Mobile Crowdsensing system relies on a single *Crowdsensing Participation App* to gather data from one or more sensors on the *Mobile Device* and create a *Data Sensing Stream* that is sent to a *Proxy VM* on a surrogate called a *Cloudlet*. The *Proxy VM* serves the role of both *Communications Manager* and *Cache* and is essentially a proxy of the mobile device that handles all requests for sensor data on behalf of the mobile device. A *Cloudlet* can run one or more *Proxy VMs* that each corresponds to a mobile device that is participating in a crowdsensing task. In addition, the *Proxy VM* can perform processing on the data sensing stream to, for example, enforce privacy settings. One or more *Crowdsensing Application VMs* that also run on the surrogate access the *Proxy VM* to obtain the sensed data to process locally or to format and send the data to applications running in the cloud on an *Application Server*.

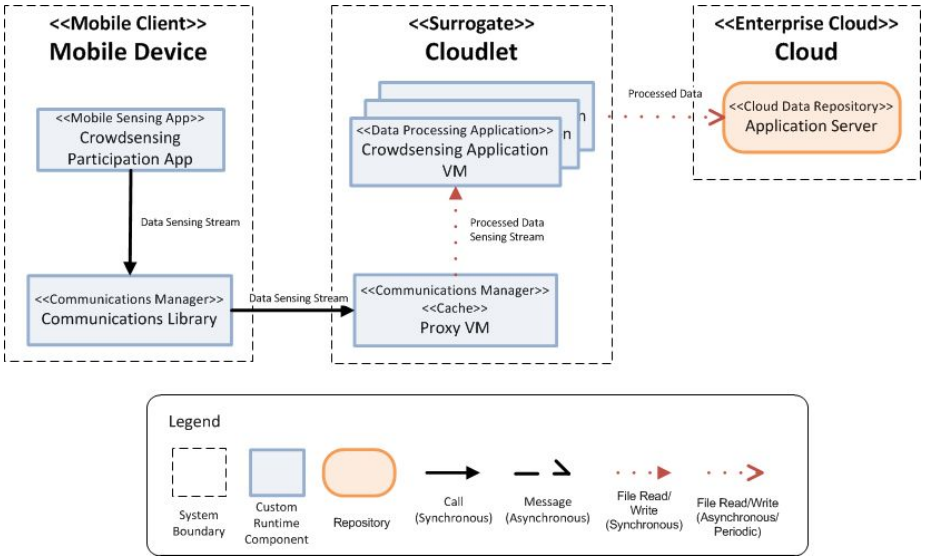

Figure 2.10: Large-Scale Mobile Crowdsensing as an Example of the Out-Bound Pre-Processing Tactic

Dependencies. The Out-Bound Pre-Processing tactic requires a Surrogate Provisioning tactic (Section 2.1.3) that prepares the surrogate for data staging.

2.1.3 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.

The Surrogate Provisioning tactics are a pre-requisite for Data Staging (Section 2.1.2) and Computation Offload (Section 2.1.1) tactics. Surrogate Provisioning tactics need to be matched with a Surrogate Discovery tactic (Section 2.1.4) that can enable the mobile device to locate and use the surrogate for cyber-foraging.

2.1.3.1 Pre-Provisioned Surrogate

Many of the systems described in the primary studies assume that the offloaded computation and/or data staging elements are already installed (pre-provisioned) on the surrogate at deployment time. The computation offload systems shown in Table 1.1 that make this assumption are those for which *What to Offload -Payload* is (1) Parameters but not Computation, Source Location nor Setup Instructions, (2) Application State, (3) Device Context, or (4) Continuous Data. It is also true of all the data staging systems shown in Table 1.2. However, for these systems, there is no detail of how the surrogates were provisioned with the necessary offloaded computation and or data staging elements. This observation relates to one of the findings from the SLR that states that most systems tend to focus on the algorithms and implementation details for enabling cyber-foraging and not on system-level concerns such as ease of distribution and installation that have to be considered when moving from experimental prototypes to operational systems. Indeed, a cyber-foraging system could be implemented with a static, hard-coded connection between the mobile device and the offloaded computation or data staging elements in the surrogate. However, this static link between mobile device and surrogate does not enable the flexibility that is implied by cyber-foraging as the opportunistic leverage of resource-rich surrogates.

Motivation. 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 (offloaded computation and data staging computing elements) and appropriate execution containers, and (2) maintain a list of these capabilities (similar to a service registry).

Description. Figure 2.11 shows the main components of the Pre-Provisioned Surrogate tactic. This tactic requires a *Surrogate Manager* that acts as a management component for the *Surrogate*. The *Surrogate Manager* is accessed by a system administrator from a *Local User Interface* running on the *Surrogate* or a *Remote User Interface* that resides on an external *Admin Client* (e.g., laptop, desktop, mobile device). When a system administrator uses the *Surrogate Manager* to install a new offload or data staging capability on the *Surrogate*,

the capability it is stored in a *Capabilities Repository* such as a file system or database. The *Capabilities Repository* contains the set of capabilities that are either started when the *Surrogate* is started, or started on demand when the *Offload Server* (from the Computation Offload tactic (Section 2.1.1)) or the *Data Staging Manager* (from the Data Staging tactics (Section 2.1.2)) receive a request from a mobile device. In the latter case, the *Capability Metadata* contains metadata that enables to set up these capabilities on-demand, such as resource requirements, installation scripts, and configuration data. Installed capabilities are then registered in a *Capability Registry* that is used by Surrogate Discovery (from the Surrogate Discovery tactics (Section 2.1.4)) for advertising capabilities to mobile cyber-foraging clients.

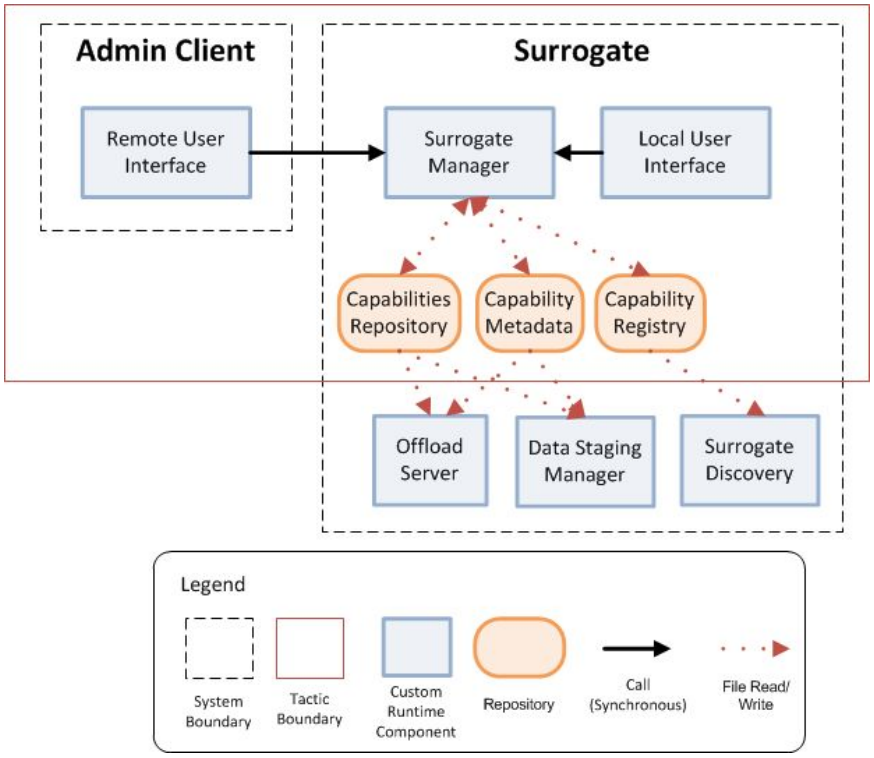

Figure 2.11: Pre-Provisioned Surrogate Tactic

Examples. This tactic is not present in any of the systems, but could be integrated into any of the cyber-foraging systems in the primary studies that assumes that offloaded computation and/or data staging elements are already available on the surrogate at runtime. What would vary between pre-provisioned systems that implement this tactic is the form of the capabilities that are stored in the repository and capability metadata. The form of the capabilities and their metadata depends on the *What to Offload - Granularity* architecture decision.

- For systems that offload at the process level, such as CloneCloud [CM09] shown in Figure 2.4 the capabilities take the form of a container to which the process and its state can migrate. For CloneCloud this is an Application VM.

- For systems that offload at the Method, Function or Operation level the capabilities take the form of the larger programming construct that these are a part of (i.e., class, module or program). As an example, if the MAUI system [Cue12] would implement this tactic the capabilities would take the form of .NET components that are stored in the Capabilities Repository and at runtime would be deployed inside a .NET CLR environment (i.e., execution container).

- For systems that offload at the Class, Module, Component, Task, Service, Application, Program or Server level the capabilities take this exact form because they are self contained. As an example, if the AIDE system [MGB+02] implemented this tactic the capabilities would take the form of Java classes that at runtime would be deployed inside a JVM.

In addition, something that would also vary across these systems is whether the offloaded computation is started once and always running, as in the mHealthMon system [AP13], or if it is started upon offload request as in the Grid-Enhanced Mobile Devices system [Gua08]. In mHealthMon the services that correspond to offloaded computation are running and waiting for requests from mobile clients. Even though it is not explicitly stated in the study, starting up the system would involve starting all the services. If mHealthMon implemented this tactic, a startup process would start all the services in the Capabilities Repository. In Grid-Enhanced Mobile Devices, upon an offload request an object called a deputy object is created on the surrogate to manage all the mobile device's requests and then destroyed when the mobile device terminates the connection. This latter approach also promotes scalability and elasticity, as shown in the Just-In Time Containers tactic (Section 2.2.3.1).

2.1.3.2 Surrogate Provisioning from the Mobile Device

The Surrogate Provisioning from the Mobile Device tactic can be found in the computation offload systems shown in Table 1.1 for which *What to Offload - Payload* is Computation.

Motivation. In Pre-Provisioned Surrogates (Section 2.1.3.1) 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.

Description. Figure 2.12 shows the main elements of the Surrogate Provisioning from the Mobile Device tactic with numbers that indicate the sequence of operations. In this tactic each *Cyber-Foraging-Enabled Mobile App* has one or more files that correspond to *Offloaded Code for Cyber-Foraging-Enabled Mobile App*, such as a class, module or application. The *Cyber-Foraging-Enabled Mobile App* starts the offload process. The *Offload Client* sends the *Offloaded Code for Cyber-Foraging-Enabled Mobile App* to the *Offload Server* on the *Surrogate*. The *Offload Server* installs the offloaded code in an execution *Container* and notifies the mobile app that it is ready for execution. At this point the *Cyber-Foraging-Enabled Mobile App* starts the execution of the offloaded code.

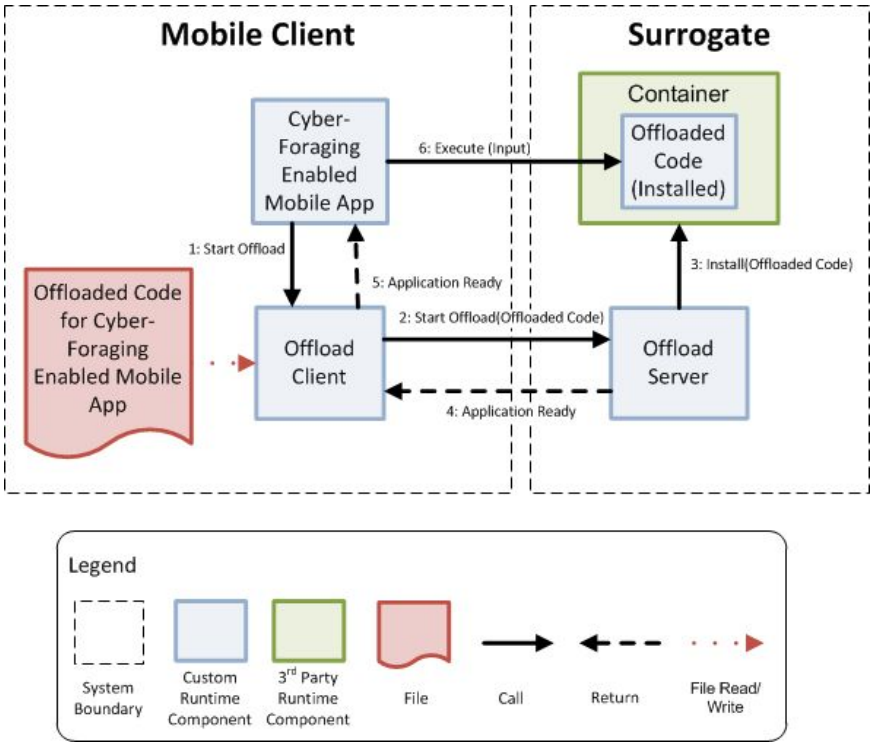

Figure 2.12: Surrogate Provisioning from the Mobile Device Tactic

Constraints. The tactic as presented requires a pre-established agreement between mobile devices and surrogates 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 (i.e., payload), the tactic may require additional components on the mobile device and surrogate to manage and provide reliable communications during the transmission of the offloaded code.

Example. An example of how to apply the Surrogate Provisioning from the Mobile Devices tactic is the VM-Based Cloudlets system [SBCD09] shown in Figure 2.13. In this system, an *Application Overlay* is created for each *Cyber-Foraging Enabled Mobile App* by starting a *Base VM* (a minimally configured VM with a guest (OS) installed), installing the application in the *Base VM*, and then suspending the VM. The binary difference is calculated between the resulting VM image file and the *Base VM*, and saved as an *Application Overlay*. At runtime the *Application Overlay* is sent by the *KCM Client* to the *KCM Server*. The *KCM Server* performs VM Synthesis by taking the same *Base VM* from which the *Application Overlay* was created and applying the overlay to it in order to recreate the VM with the installed application.The resulting VM is called a *Launch VM* and is started within a *VM Manager* (in this system it is VirtualBox[2]). Once the *Launch VM* is started and ready, the *KCM Client* is notified that the application is ready for execution. The user then interacts

[2] `https://www.virtualbox.org/`

with the application via a *VNC Client*.[3]

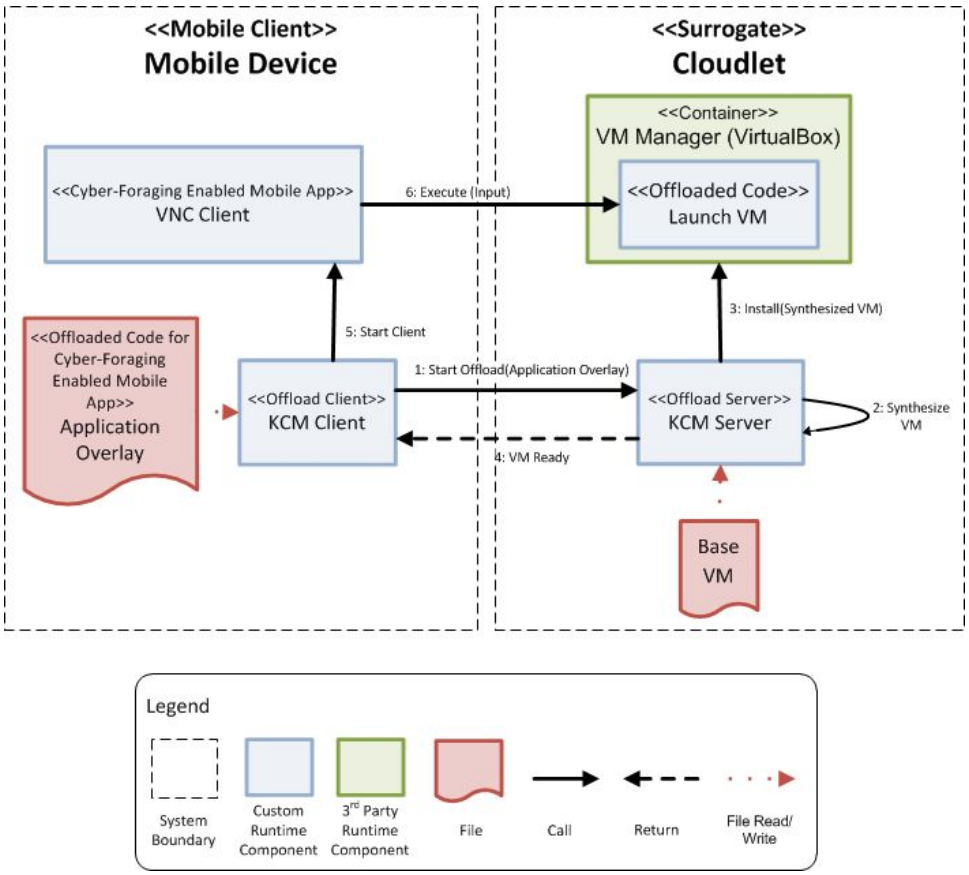

Figure 2.13: VM-Based Cloudlets as an Example of the Surrogate Provisioning from the Mobile Device Tactic

2.1.3.3 Surrogate Provisioning from the Cloud

The Surrogate Provisioning from the Cloud tactic can be found in the computation offload systems shown in Table 1.1 for which *What to Offload - Payload* is Source Location, which are the Roam [CSW+04] and the Elastic HTML5 [ZJGK12] systems. For these two systems the payload is the URL of the location of the offloaded computation. It can also be found in the Collective Surrogates [Goy11] and MAPCloud [RVMV12] systems for which *What to Offload - Payload* is Setup Instructions. In the first system the payload is a script that obtains the offloaded computation from the cloud; in the second system it is an application request that is modeled as a workflow of tasks to be located in the cloud.

[3]VNC stands for Virtual Network Computing and is a protocol for remote access to graphical user interfaces.

Motivation. Provisioning surrogates from the mobile device has the advantage of enabling the execution of a greater number of applications (Section 2.1.3.2) compared to pre-provisioned surrogates (Section 2.1.3.1). However, the size of the computation that is sent to the surrogate at runtime can be significant. In the examples for the MAUI system [Cue12], 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 [SBCD09], 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.
Description. Figure 2.14 shows the main elements of the Surrogate Provisioning from the Cloud tactic with numbers that indicate the sequence of operations. In this tactic the *Cyber-Foraging-Enabled Mobile App* contains the URL that indicates the location from which the offloaded code has to be downloaded. The *Cyber-Foraging-Enabled Mobile App* starts the offload process by sending the URL to the *Offload Client*, which in turn sends it to the *Offload Server* on the *Surrogate*. The *Offload Server* downloads the offload code from an *Offload Code Repository* at the URL, installs it in an execution *Container*, and notifies the mobile app that it is ready for execution. At this point the *Cyber-Foraging-Enabled Mobile App* starts the execution of the *Offloaded Code*.
Constraints. The tactic as presented requires connectivity between the surrogate and the cloud and potentially additional components on the surrogate and cloud server to manage and provide reliable communications during the transmission of the offloaded code. Also, the computation has to exist at the indicated location. In addition, it requires a pre-established agreement between surrogates and the cloud servers on the format of the offloaded code (e.g., Java class, Python script, Windows application).
Example. An example of how to apply the Surrogate Provisioning from the Cloud tactic is the Collective Surrogates system [Goy11]. As shown in Figure 2.15, at runtime once a *Participating Node* is assigned to an offload operation, the *Offload Client* sends a shell script to a *Daemon* running on the *Participating Node* which executes the script on behalf of the client. The script downloads the application that corresponds to the offloaded code from an *Application Repository* on an *Internet Server*, installs the application and starts it. Once the *Application* is started and ready, the *Offload Client* is notified that the application is ready for execution. The user then interacts with the application via a *Client Interface*.

2.1.4 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 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

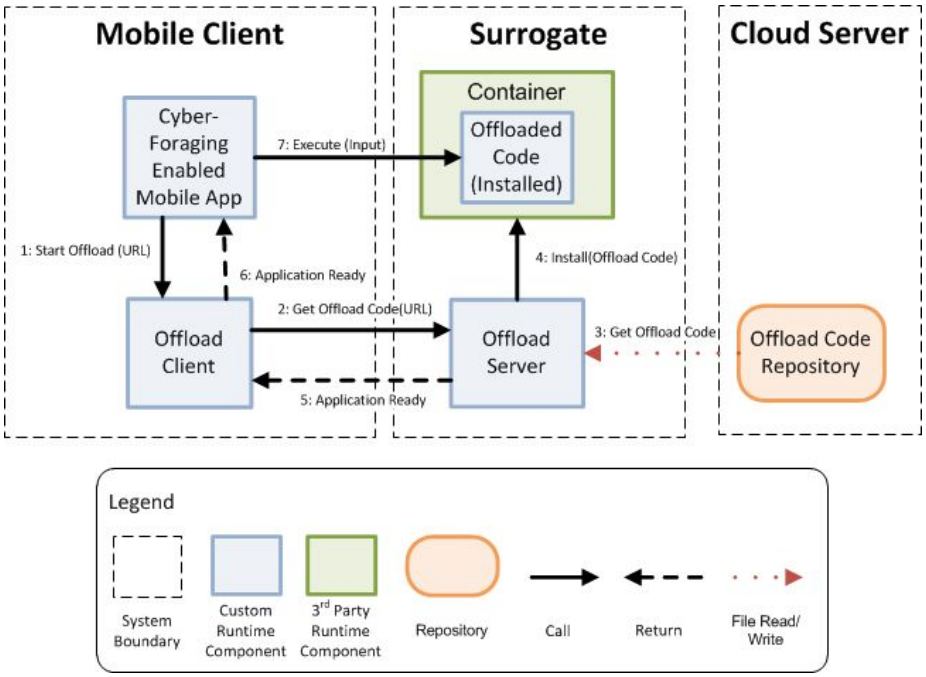

Figure 2.14: Surrogate Provisioning from the Cloud

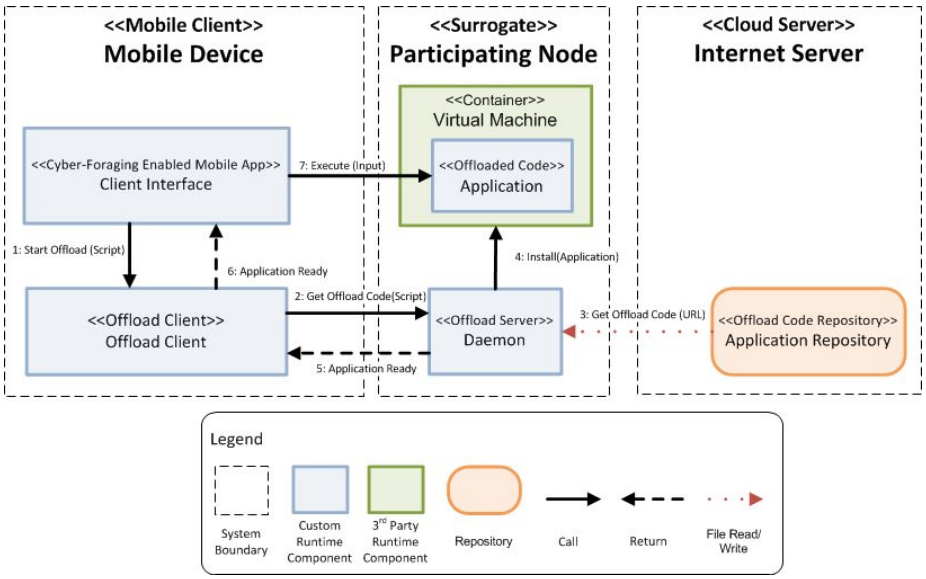

Figure 2.15: Collective Surrogates as an Example of the Surrogate Provisioning from the Cloud Tactic

selects the surrogate that is the best match for the offloaded task.

The Surrogate Discovery tactics are a pre-requisite for Data Staging (Section 2.1.2) and Computation Offload (Section 2.1.1) tactics. The surrogate discovery protocol becomes the initial part of the offload process. Surrogate Discovery tactics need to be matched with a Surrogate Provisioning tactic (Section 2.1.3) that prepares the surrogate for cyber-foraging.

2.1.4.1 Local Surrogate Directory

The Local Surrogate Directory tactic can be found in six systems that maintain a list of potential surrogates on which to offload computation or stage data: Roam [CSW+04]. Spectra [FPS02], Cuckoo [KPKB12], SPADE [SVF08], Offloading Toolkit and Service [YOC08], and Heterogeneous Auto-Offloading Framework for Mobile Web Browsers [ZGHC09].

Motivation. 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 with their network addresses or URLs, in addition to any information that can help the mobile device to select the best offload target 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.

Description. The Local Surrogate Directory Tactic has two parts. One part involves the *Surrogate Directory UI* which populates and maintains the *Surrogate Directory*. The other part involves the components that interact during the offload process as shown in Figure 2.16 with numbers that indicate the sequence of operations. At runtime, the *Cyber-Foraging Mobile App* calls the *Offload Client* to start the offload process. The *Offload Client* obtains that list of potential surrogates from the *Surrogate Directory* and pings each *Surrogate* to see if it is available for offload. The *Offload Server* of each available *Surrogate* responds to the *Offload Client* with any *Surrogate Metadata* data required by the discovery protocol, such as current load or available capabilities. Based on this information and any network information available, the *Offload Client* selects the best surrogate for offload and starts the actual offload operation with the selected *Surrogate*. Optionally, the *Offload Client* may update the *Surrogate Directory* based on the availability and performance of the selected surrogate.

Constraints. The tactic as presented 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.

Examples. The systems that implement the Local Surrogate Directory tactic maintain a list of potential surrogates for offload. What varies between systems is how the list is populated and whether or not the list is updated based on network conditions or offload execution data.

- Roam [CSW+04] maintains a list of servers that can accept offloadable components, along with their characteristics. These characteristics are used at runtime to determine an appropriate offload target.

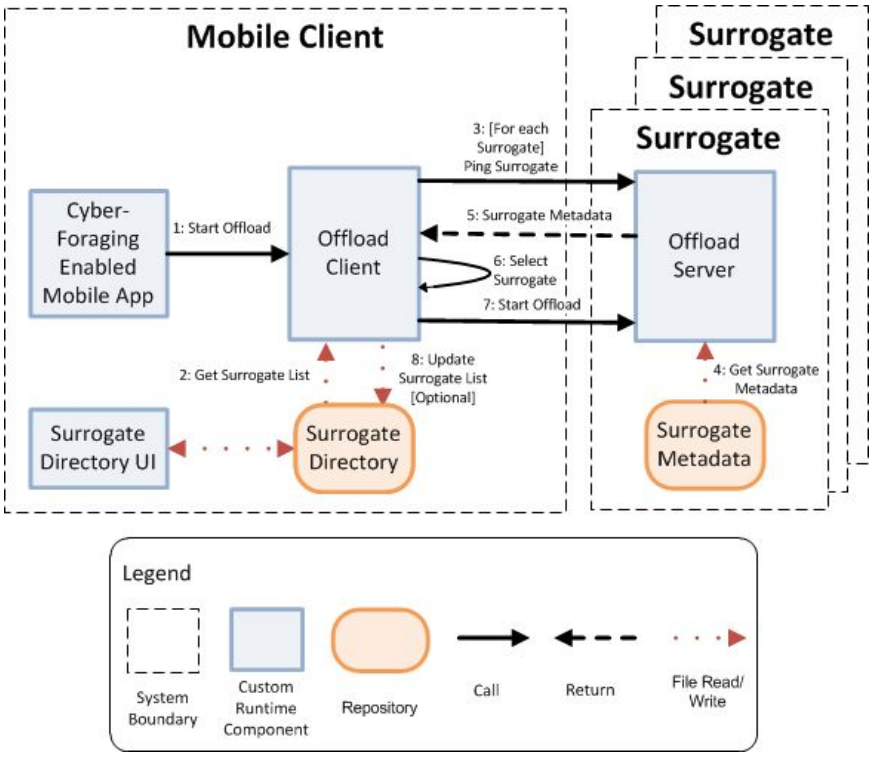

Figure 2.16: Local Surrogate Directory

- Spectra [FPS02] keeps a list of surrogates that are willing to host computation in a configuration file. As the system executes, the status of each surrogate is updated (e.g., availability, CPU load, file cache state).

- Cuckoo [KPKB12] has a component called a *Resource Manager* that maintains a list of surrogates. If the surrogate has a visual display, upon loading it shows a QR code[4] that is read by the mobile device and then added to the list of resources (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.

- SPADE [SVF08] users have to associate remote computers called *Cycle Providers* to specific tasks that are part of a job. At runtime, the mobile device uses this list to locate cycle providers based on each of the tasks that it needs to execute. An interesting aspect of this system is that surrogates have functionality to discover other surrogates on the same network and can provide this list back to the mobile device. However, the mobile device does not have capabilities to discover surrogates on its own. Details of this system are shown as an example in Figure 2.17. A single *User Interface* acts as the UI for maintaining the *Cycle Provider List* and for starting an

[4] A QR code, or Quick Response Code, is a machine-readable code consisting of an array of black and white squares that typically contains URLs or other information that can be read by the camera on a smartphone (`http://www.qrcode.com/en/`).

offload job. The *Job Manager* selects a *Cycle Provider* for each task and starts the offload for each in a separate process so that tasks can execute in parallel.

- Offloading Toolkit and Service [YOC08] maintains a list of surrogates (service providers) that are queried at runtime for desired capabilities. Each surrogate maintains its own service registry.
- Heterogeneous Auto-Offloading Framework for Mobile Web Browsers [ZGHC09] queries all potential surrogates on its list for matching required capabilities. Each matching surrogate sends back quality information (e.g., server capability and network bandwidth) and the client decides whether to offload the computation to a matching surrogate or execute locally.

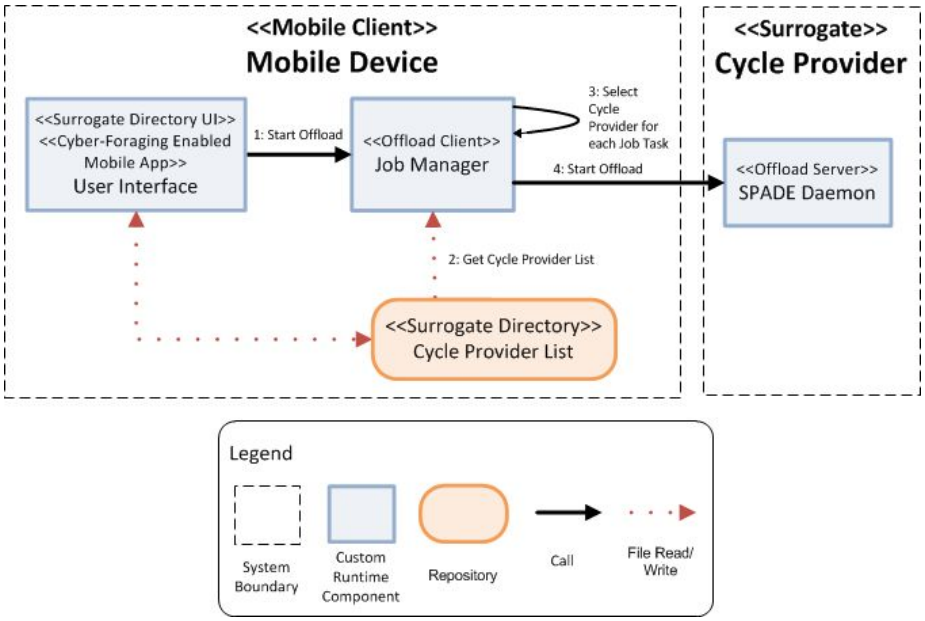

Figure 2.17: SPADE as an Example of the Local Surrogate Directory Tactic

2.1.4.2 Cloud Surrogate Directory

The Cloud Surrogate Directory tactic can be found in 12 systems in which the mobile device contacts a cloud server that maintains a list of potential surrogates on which to offload computation or stage data: Mobile Agents [AB13], HPC-as-a-Service [Dug11], Collective Surrogates [Goy11], Grid-Enhanced Mobile Devices [Gua08], ThinAV [JBA12], MCo [Lee12], Resource Furnishing System [OSP07], Cloud Personal Assistant (CPA) [OG13], MAPCloud [RVMV12], Large-Scale Mobile Crowdsensing [XSP+13], Mobile Data Stream Application Framework [YCY+13], and Weblets [ZKJG11].

Motivation. In the Local Surrogate Directory tactic (Section 2.1.4.1) 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. All surrogate metadata is populated and updated in this central repository. 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 2.2.4.1).

Description. In the Cloud Surrogate Directory tactic the *Surrogate Directory* is located in a *Cloud Server*. Figure 2.18 shows the main elements of the tactic with numbers that indicate the sequence of operations. The *Cyber-Foraging-Enabled Mobile App* starts the offload process by querying the *Surrogate Directory* via the *Surrogate Directory Interface*. This is the same interface that would be used by any program that populates and maintains the *Surrogate Directory* or by *Surrogates* that provide live data. The *Surrogate Directory Interface* selects the optimal surrogate from the directory based on data such as mobile device characteristics, type of offload request, surrogate availability, surrogate load, or any other data that is available in the directory or was provided by the mobile device as query parameters. The *Surrogate Directory Interface* then sends the *Offload Client* the data for the selected surrogate which includes the surrogate address. The *Offload Client* contacts the *Offload Server* of the selected *Surrogate* to continue the offload process.

Constraints. The tactic as presented 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. Cloud servers that perform service discovery instead of simply maintaining a surrogate directory suffer from the traditional challenges of service discovery in service-oriented computing [PTDL08].

Examples. The systems that implement the Cloud Surrogate Directory tactic maintain a list of potential surrogates on a centralized cloud server. What varies between systems is (1) the parameters that are used for surrogate selection, (2) whether the surrogate selection algorithm runs on the cloud server or the mobile device, (3) whether the surrogate directory maintains lists of surrogates or a list of services that are hosted on each surrogate, and (4) whether the cloud server returns a surrogate address or forwards the offload request to the surrogate therefore acting as an intermediary.

- Mobile Agents [AB13]: As shown in Figure 2.19, the *Execution Manager* on 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.

- HPC-as-a-Service [Dug11]: The mobile device queries a centralized repository of HPC (high-performance computing) services to locate a service with given characteristics.

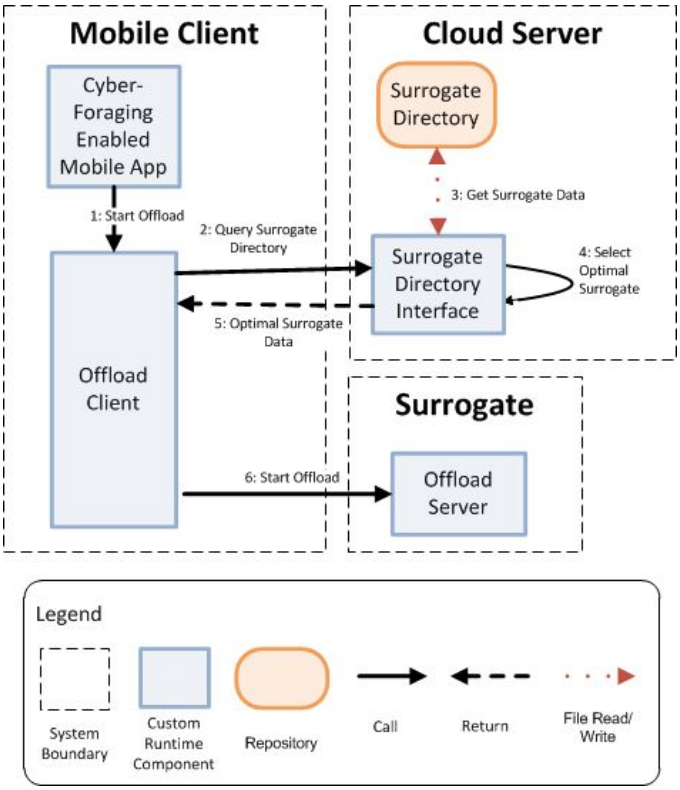

Figure 2.18: Cloud Surrogate Directory

- Collective Surrogates [Goy11]: The mobile device contacts a *Collective Manager* that manages a set of surrogates (participating nodes) and uses profile and historic information to determine the specific surrogate on which the computation will be offloaded.
- Grid-Enhanced Mobile Devices [Gua08]: Mobile devices contact the *Grid Gateway* which locates Grid services available on surrogates and then forwards the offload request, acting as as intermediary.
- ThinAV [JBA12]: The cloud server (ThinAV Server) submits received offload requests to surrogates and returns results to mobile clients when available. The *ThinAV Server* acts as an intermediary.
- MCo [Lee12]: Upon receipt of a computation offload request from a mobile device, the *Master Node* (Cloud Server) searches its list of *Worker Nodes* (Surrogates) on which computation can be offloaded. Once a *Worker Node* is selected the offload request is forwarded. The *Master Node* acts as an intermediary.
- Resource Furnishing System [OSP07]: A *Dispatching Surrogate* maintains the software list of known surrogates (application servers), and selects an application server based on contents of the request packet and application server load.

- Cloud Personal Assistant (CPA) [OG13]: CPA receives a set of tasks to execute from a mobile device, discovers the necessary cloud services, invokes them and then delivers the results back to the mobile device, acting as an intermediary.
- MAPCloud [RVMV12]: For each offload request (modeled as a workflow of tasks) from a mobile device, the *Broker* consults the registry of available surrogates and services and returns the addresses of services that can execute each task.
- Large-Scale Mobile Crowdsensing [XSP+13]: A cloud server (Application Server) consults a global registry for a list of suirrogates (Cloudlets) that are located in a certain area.
- Mobile Data Stream Application Framework [YCY+13]: Mobile devices send offload requests to a cloud server (Resource Manager) which then assigns a surrogate (Application Master) to handle the request.
- Weblets [ZKJG11]: A *Cloud Elasticity Service (CES)* allocates surrogates to offload requests based on usage information (e.g., compute power, bandwidth and storage).

Variation: Intermediary Cloud Surrogate Directory. The tactic as described returns the address of the selected surrogate to the mobile device, which then contacts the surrogate directly. In Grid-Enhanced Mobile Devices [Gua08], ThinAV [JBA12], MCo [Lee12], Cloud Personal Assistant (CPA) [OG13], and Large-Scale Mobile Crowdsensing [XSP+13] 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*.

2.1.4.3 Surrogate Broadcast

The Surrogate Broadcast tactic can be found in 5 systems in which surrogates broadcast or advertise their presence to mobile devices: Scavenger [Kri10], Real Options Analysis [EML11], Application Virtualization on Cloudlets [ML13], VM-Based Cloudlets [SBCD09], and Slingshot [SF05].
Motivation. The Local Surrogate Directory (Section 2.1.4.1) and Cloud Surrogate Directory (Section 2.1.4.2) 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.
Description. As shown in Figure 2.20, in the Surrogate Broadcast tactic all *Surrogates* broadcast selected metadata using a *Broadcast Component*. The numbers in the figure indicate the sequence of operations, starting with the broadcast operation as 0 to mean that it occurs in advance of the offload request. The *Cyber-Foraging-Enabled Mobile App* initiates the offload request.

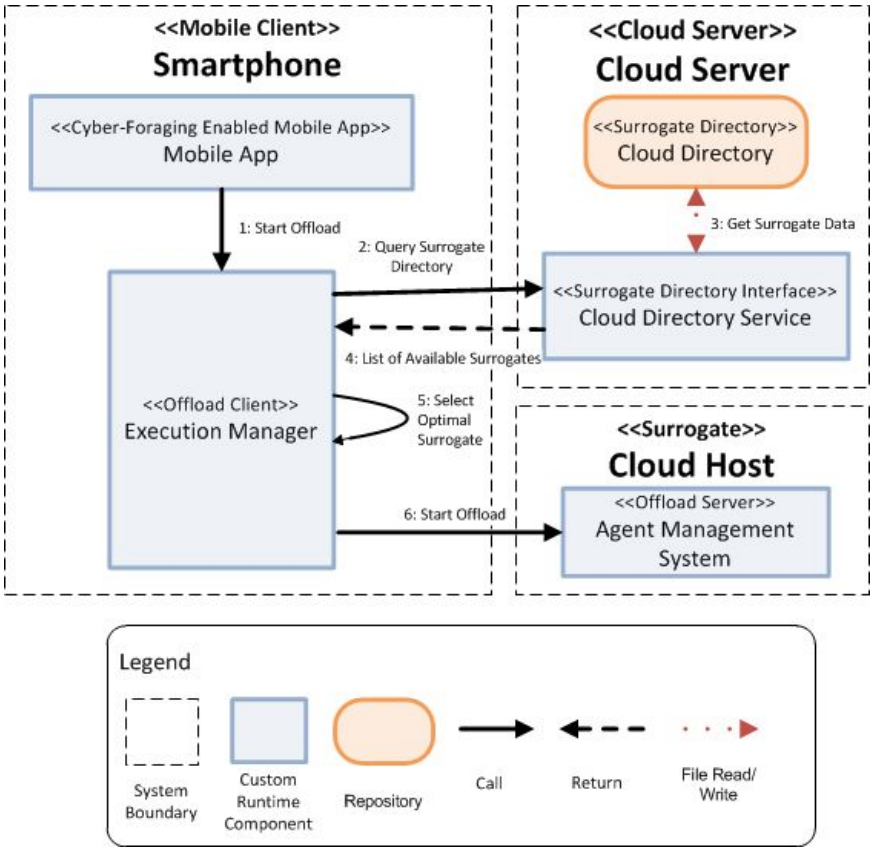

Figure 2.19: Mobile Agents as an Example of the Cloud Surrogate Directory Tactic

The *Offload Client* finds available surrogates by analyzing broadcast information which will include at least the surrogate address. The *Offload Client* then selects the optimal surrogate and starts the offload process by contacting the *Offload Server* of the selected surrogate. In addition to basic surrogate metadata such as surrogate address, the surrogate can also broadcast data retrieved from a Capability Metadata repository as described in the Pre-Provisioning tactic (Section 2.1.3.1).

Constraints. The tactic as described 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 2.2.4.1).

Examples. The surrogates in the systems that implement the Surrogate Broadcast tactic broadcast their availability and selected metadata to mobile devices for discovery. What varies between systems is the broadcast mechanism and the information or metadata that they broadcast.

- Scavenger [Kri10]: Surrogates periodically broadcast their service descriptions using UDP broadcast.[5] As shown in Figure 2.21, a *Presence Daemon*

[5] UDP stands for User Datagram Protocol and is one of the core protocols of the IP suite.

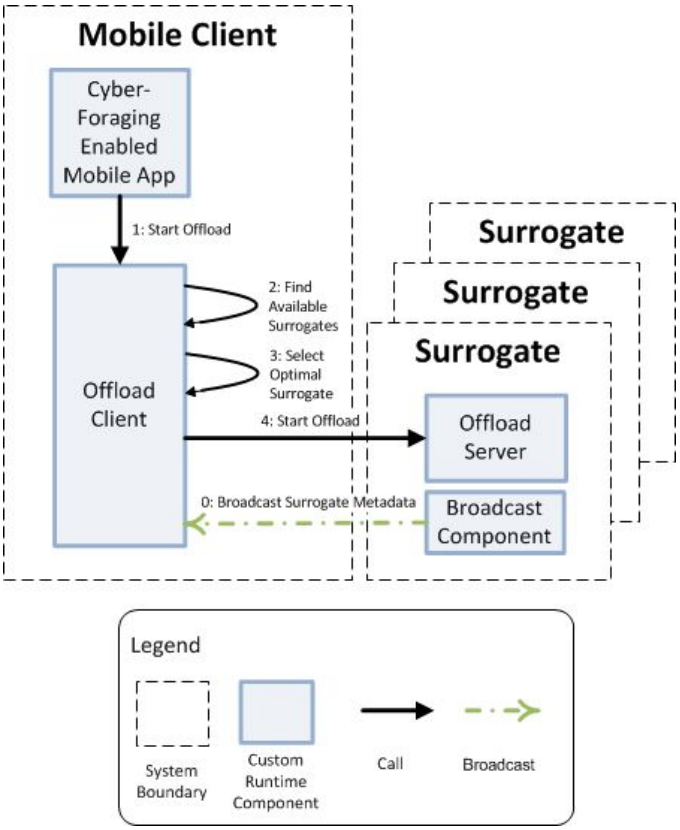

Figure 2.20: Surrogate Broadcast

running on each *Surrogate* periodically packs all its service descriptions into a single UDP packet and broadcasts it onto the local subnet. An *Application* running on a mobile device uses the *Scavenger Library* to find available surrogates, select the optimal surrogate on which to offload, and finally contact the *Scavenger Front-End* of the selected surrogate.

- Real Options Analysis [EML11]: As surrogates come online, they broadcast their availability and address over a broadcast channel.
- Application Virtualization on Cloudlets [ML13] and VM-Based Cloudlets [SBCD09]: Surrogate information that includes surrogate address is broadcast using an implementaton of Zeroconf.[6]
- Slingshot [SF05]: This system uses UPnP[7] to discover new surrogates in its surrounding network environment.

UDP broadcast is the broadcasting of UDP packets to an entire subnet.

[6]Zerconf stands for Zero Configuration Networking and is a set technologies that enables automated network configuration of devices and services without the use of central services such as DNS or DHCP (`www.zeroconf.org`).

[7]UPnP stands for Universal Plug and Play and is a set of networking protocols that enable networked devices to seamlessly discover each other's presence on the network and establish functional network services (`www.upnp.org`).

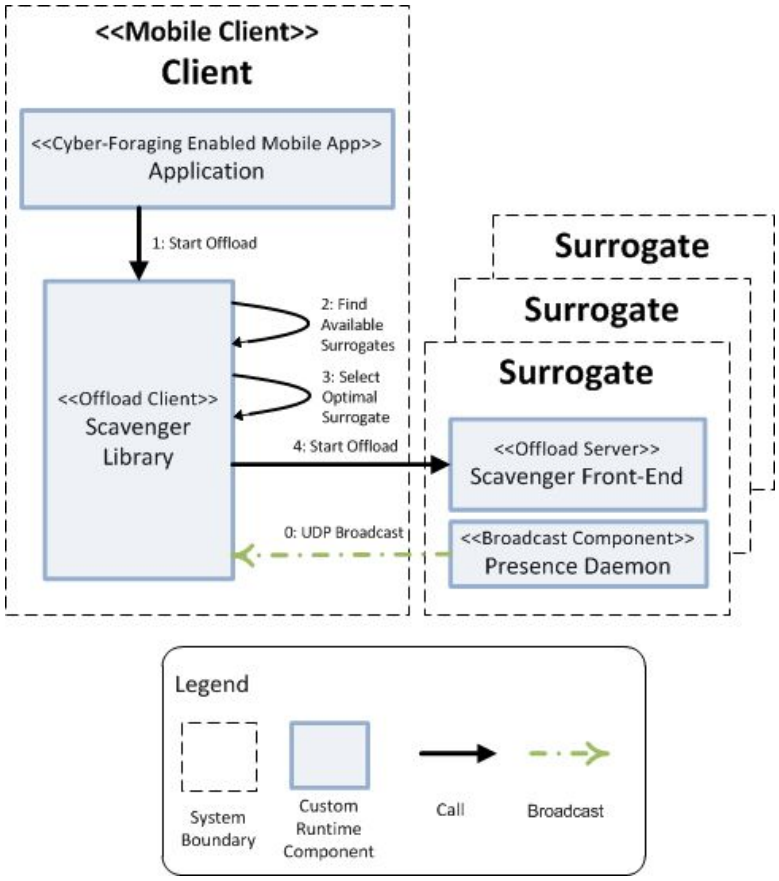

Figure 2.21: Scavenger as an Example of the Surrogate Broadcast Tactic

2.2 Non-Functional Architectural Tactics for Cyber-Foraging

2.2.1 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 the network conditions between the mobile device and the surrogate are not ideal, the computation is executed locally instead of offloaded to the surrogate.

The Runtime Optimization tactics need to be combined with the Computation Offload tactic (Section 2.1.1) to enable the computation offload process.

2.2.1.1 Runtime Partitioning

The Runtime Partitioning tactic can be found in the computation offload systems shown in Table 1.1 for which *When to Offload* is Runtime Decision.

Motivation. In general, offloading is beneficial when large amounts of computation are needed with relatively small amounts of communication [KL10]. 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 based on payload size and network conditions, the communication time based 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).

Description. Figure 2.22 shows the main components of the Runtime Partitioning tactic with numbers to indicate the sequence of operations. 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 *Offload Decision Engine* uses *App Metadata* such as required compute cycles, payload size based on input and output parameters, and required energy for execution and communication. Even though the *App Metadata* is depicted in Figure 2.22 as an external file, this data can also reside within the code as annotations. Upon a request for execution of a computational element that is marked for offload, the *Cyber-Foraging Enabled Mobile App* invokes the *Offload Decision Engine*, passing it the necessary metadata for the *Offloadable Element*. In addition, although optional, the *Offload Decision Engine* can also make use of *Environment Monitors* to obtain runtime environment data such as network conditions or load of the mobile device and surrogate if these are required by the defined optimization function. It can also make use of *Cost Models* (e.g., an energy model for the mobile device) as input to the optimization function. Based on the results of the optimization function, the *Cyber-Foraging-Enabled Mobile App* invokes the local copy of the *Offloadable Element* or invokes the *Offload Client* in order to invoke the remote copy of the *Offloadable Element* running on the *Surrogate*.

Constraints. The Runtime Partitioning tactic assumes 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.

Example. An example of how to apply the Runtime Partitioning tactic is the MACS system [KK12], as shown in Figure 2.23. In MACS, *Cyber-Foraging Enabled Mobile Apps* contain offloadable elements defined as *Services*. Each service has *Service Metadata* related to memory size, code size, and input/output parameter size. When the mobile app is going to execute a service, the *Performance and Context Monitor* is invoked to determine the feasibility of remote execution as well as to compare the cost of local execution of the service against the cost of remote execution. The *Performance and Context Monitor* uses a *Mo-*

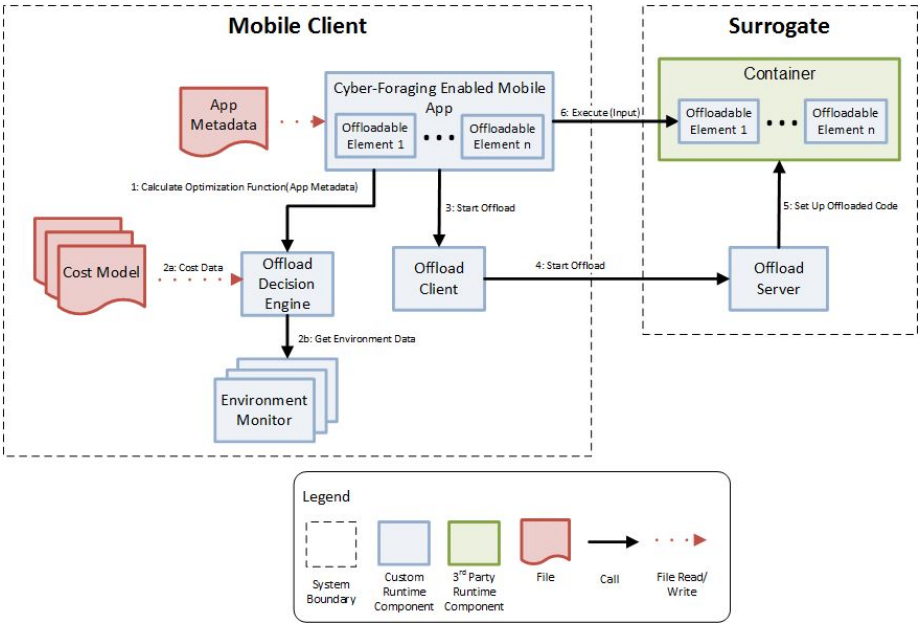

Figure 2.22: Runtime Partitioning Tactic

bile Device Monitor implemented as calls to the Android API to obtain available memory information, CPU load and remaining battery. It also uses a *Network Monitor* to obtain connectivity and bandwidth information. In addition, based on a pre-built *Energy Model* it calculates the energy cost of local vs remote execution using the service metadata. If the decision is to offload, the *Offload Manager* and *Remote Execution Manager* coordinate to set up the offloaded service for remote execution.

Dependencies. The Runtime Partitioning tactic requires the Computation Offload tactic (Section 2.1.1) as the infrastructure for computation offload.

Variation: User-Guided Runtime Partitioning. The tactic as described assumes a static optimization function. However, in some systems what to optimize is determined based on user preferences or input. In the PowerSense system [MCF+11] the user can select a *Time Saver* option to minimize processing time or an *Energy Saver* option to minimize energy consumption. The ThinkAir system [KAH+12] offers four optimization options (profiles) to users: execution time; energy consumption; execution time and energy consumption; execution time, energy consumption and cost of cloud services. These systems have a user interface on the mobile device to set these preferences.

2.2.1.2 Runtime Profiling

The Runtime Profiling tactic can be found in nine systems: MAUI [Cue12], Real Options Analysis[EML11], Single-Server Offloading [Ima12], ThinkAir [KAH+12], AMCO [KT13], SmartVC [PXJZ13], Odessa [RSM+11], IC-Cloud [SPN+13], and Mobile Data Stream Application Framework [YCY+13].

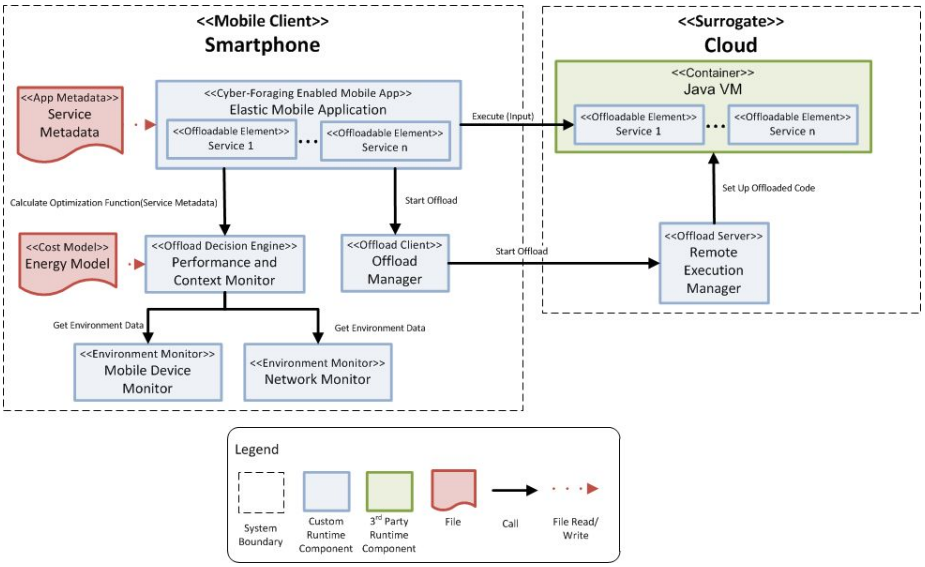

Figure 2.23: MACS as an Example of the Runtime Partitioning Tactic

Motivation. Systems that implement the Runtime Partitioning tactic (Section 2.2.1.1) 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 [DZ11]. 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.

Description. Figure 2.24 shows the main components of the Runtime Profiling tactic. The difference between the Runtime Profiling tactic and the Runtime Partitioning tactic (Section 2.2.1.1) is the data that is used in the offload decision and what happens after the offloading process ends. The *Cyber-Foraging Enabled Mobile App* invokes the *Offload Decision Engine*, passing it the necessary metadata for the *Offloadable Element*. In addition to runtime data obtained from *Environment Monitors* and *Cost Models*, the *Offload Decision Engine* uses *Historical Execution Data* as input to the optimization function. Large differences between estimated and historic cost data might trigger the *Offload Decision Engine* to adjust the *Cost Models*. Based on the results of the optimization function, the *Cyber-Foraging-Enabled Mobile App* invokes the local copy of the *Offloadable Element* or invokes the *Offload Client* in order to invoke the remote copy of the *Offloadable Element* running on the *Surrogate*. After the offload process is completed, the *Offload Client* saves current execution data for the offloadable element such as timestamp, input parameters, energy con-

sumption, network quality, and execution time in the *Historical Execution Data* repository. In addition, although optional, the *Environment Monitors* may store environment data periodically in the *Historical Execution Data* repository.

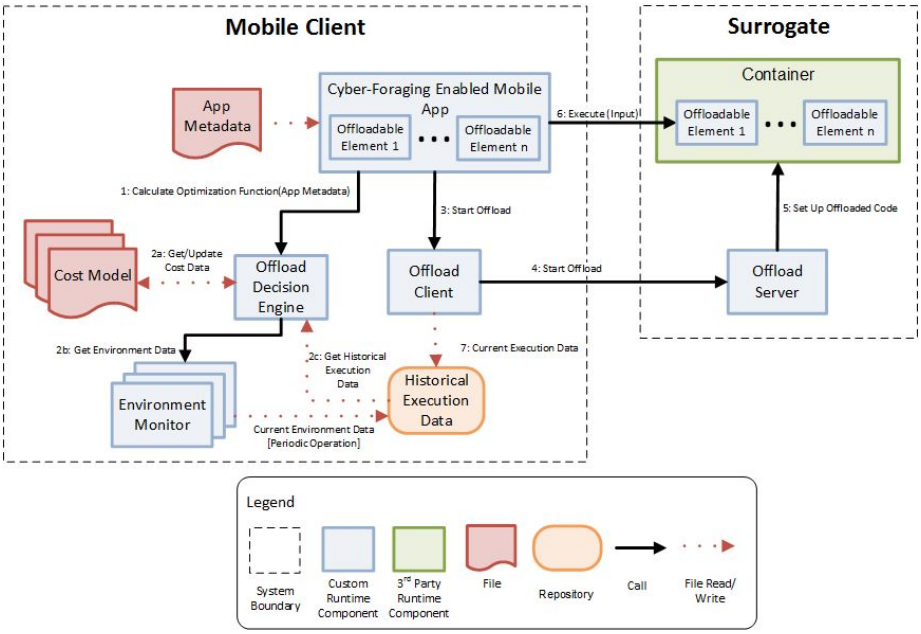

Figure 2.24: Runtime Profiling Tactic

Constraints. As in the Runtime Partitioning tactic (Section 2.2.1.1), the Runtime Profiling tactic assumes that there is equivalent code for the offloaded computation on both the mobile device and the surrogate. In addition, the cost of profiling is not negligible and can impact overall application performance [Cue12]. System designers need to consider the type and frequency of data to capture at runtime.

Examples. The systems that implement the Runtime Profiling tactic update the data that is used by the optimization function based on current execution data and environmental conditions. What varies between systems is the type of data that is captured and and the frequency of data capture.

- MAUI [Cue12]: As shown in Figure 2.25, the *Solver+Profiler* 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, the *Client Proxy* updates the *Past Program Execution and Network Data* to better predict whether future invocations of the method should be offloaded.

- Real Options Analysis[EML11]: The system maintains a list of accessible servers and estimates the network delay to each of them using the default routing. Once offload completes, the network traffic model is updated.

- Single-Server Offloading [Ima12]: Remote execution time is calculated for the first execution as communication time plus remote computation time. The latter is sent back from the surrogate as part of the results. From the second execution on, the model predicts local and remote execution time and offloads only if remote execution time is less than local execution time. The system updates the execution time parameters from actual computation results only if the difference between predicted and actual execution times (local and remote) is greater than an established threshold.
- ThinkAir [KAH+12]: When a method is encountered for the first time,the decision to offload is based only on environmental parameters such as network quality. From that point on, the profilers start collecting execution and energy consumption data for that method. If the method is invoked again, the decision to offload is based on the method's past execution times and energy consumed.
- AMCO [KT13]: Based on a feedback-loop mechanism, energy consumption data is updated after the execution of code portions marked as "energy hotspots" and used in the calculation of future energy consumption which drives offload decisions.
- SmartVC [PXJZ13]: The system records the execution time and power consumption for each method as historical data to better inform future offloading decisions.
- Odessa [RSM+11]: The system's decision engine uses the recent history of network measurements to determine if offloading or increasing the level of parallelism will improve performance.
- IC-Cloud [SPN+13]: The system uses signal strength and historical information of network states to obtain a coarse-grained estimation of network access quality that influences the offload decision.
- Mobile Data Stream Application Framework [YCY+13]: The profiler on the mobile device measures the device's characteristics at startup and continuously monitors its CPU workload and wireless network bandwidth. If any of the parameters varies by a value exceeding an established threshold, a new partitioning is generated for the application.

Dependencies. The Runtime Profiling tactic requires the Runtime Partitioning tactic (Section 2.2.1.1) to enable the system to make a runtime decision on whether or not to offload computation. It also requires the Computation Offload tactic (Section 2.1.1) to establish the infrastructure for computation offload.

2.2.1.3 Resource-Adapted Computation

The Resource-Adapted Computation tactic can be found in the Cuckoo system [KPKB12]. The system has elements that enable it to use different versions of offloadable elements to match the resource characteristics of mobile devices and surrogates, depending on whether code executes locally or remotely.
Motivation. In the Runtime Partitioning tactic (Section 2.2.1.1) 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

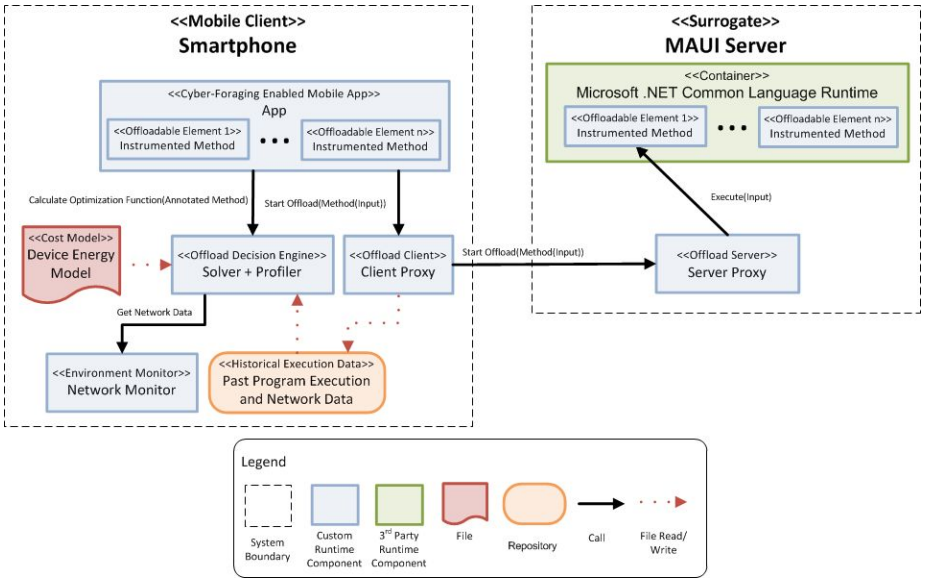

Figure 2.25: MAUI as an Example of the Runtime Profiling Tactic

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) [Sat01]. 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.

Description. Figure 2.26 shows a simplified representation of the Runtime Partitioning tactic (Section 2.2.1.1) with additional elements that describe the Resource-Adapted Computation tactic. At runtime, the *Offload Decision Engine* calculates the optimization function for the *Offloadable Element*. If the decision is to execute locally, the *Cyber-Foraging Enabled Mobile App* executes the *Offloadable Element (Mobile Version)* that is adapted to the resource characteristics of the mobile device. However, if the decision is to execute remotely, the *Offloadable Element (Surrogate Version)* is executed to take advantage the more powerful resources of the *Surrogate*.

Constraints. The Resource-Adapted Computation tactic requires developing, profiling and maintaining different versions of offloadable elements.

Example. Cuckoo [KPKB12] is an example of a system that implements the Resource-Adapted Computation tactic. 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

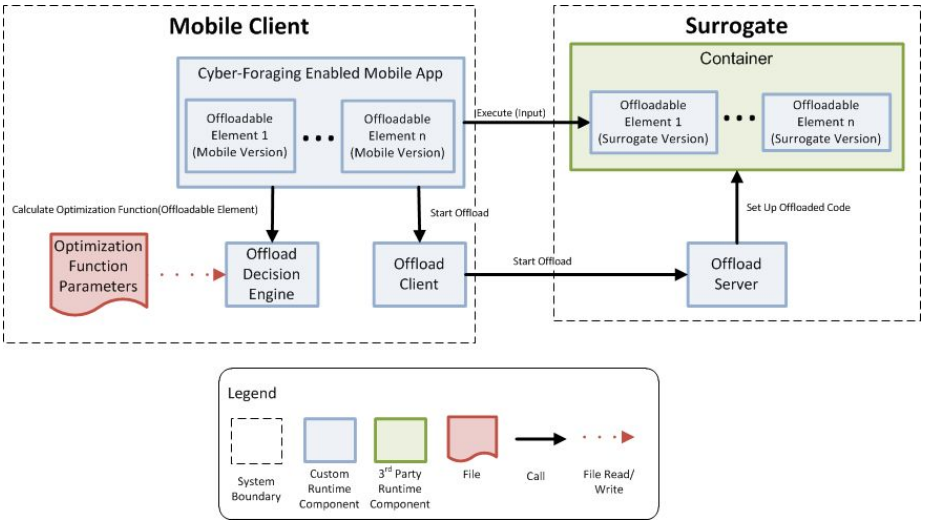

Figure 2.26: Resource-Adapted Computation

remote implementation can run a different algorithm, use a different library, or take advantage of parallelization on the more powerful surrogate. Figure 2.27 shows the Cuckoo system at runtime with numbers to indicate the sequence of operations. The *Cuckoo Framework* intercepts all service calls. It then uses the *Cuckoo Resource Manager* to decide whether to execute the local or the remote implementation of the service. In the current implementation it will execute the remote implementation if a surrogate is available (details of how it locates surrogates are in Section 2.1.4.1). If a surrogate (*Cuckoo Server*) is not available, the *Local Service Implementation* is executed. If a surrogate is available, it uses the *Ibis Middleware* to invoke the *Remote Service Implementation*.

Dependencies. The Resource-Adapted Computation tactic requires the Runtime Partitioning tactic (Section 2.2.1.1) to enable the system to make a runtime decision on whether or not to offload computation. It also requires the Computation Offload tactic (Section 2.1.1) to establish the infrastructure for computation offload.

Variation: Resource-Adapted Input. A variation of this tactic is for the *Offloadable Element (Mobile Version)* and the *Offloadable Element (Surrogate Version)* to be identical, but what varies is the input parameters. The enabler is that different input parameters will lead to different resource consumption. PowerSense [MCF+11] is an image processing system for dengue detection that implements this variation of the tactic. PowerSense uses the same algorithm (implementation) locally and remotely for image processing, but uses images of lower resolution if processed locally and higher resolution images if processed remotely because processing these higher quality images requires greater computing power.

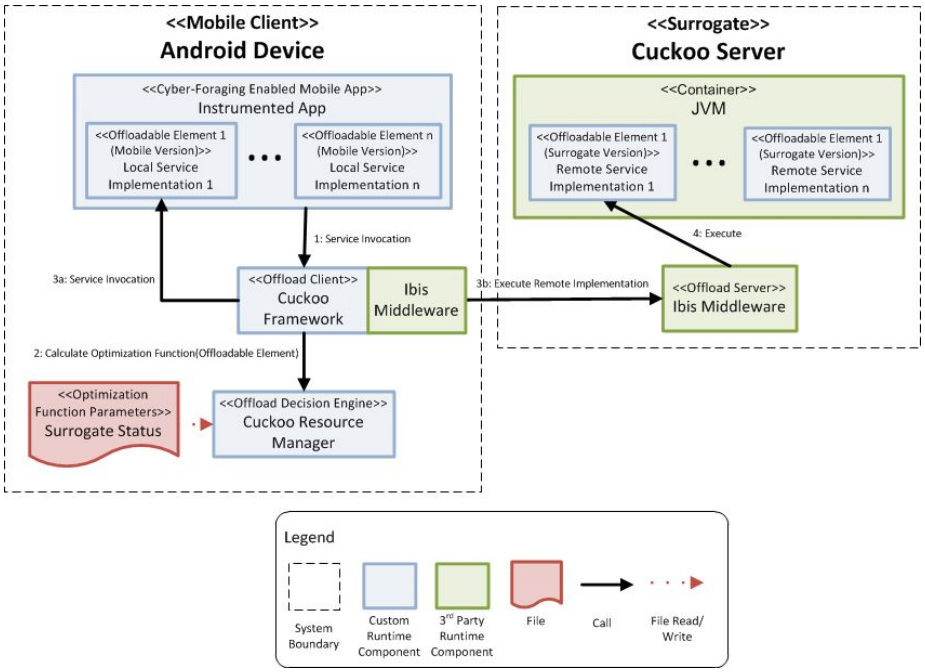

Figure 2.27: Cuckoo as an Example of the Resource-Adapted Computation Tactic

2.2.2 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.

The Fault Tolerance tactics need to be combined with a Surrogate Provisioning tactic (Section 2.1.3) to prepare the surrogate for computation offload or data staging, and a Computation Offload tactic (Section 2.1.1) or Data Staging tactic (Section 2.1.2) to enable the computation offload or data staging process.

2.2.2.1 Local Fallback

The Local Fallback tactic can be found in the MAUI [Cue12] and ThinkAir [KAH+12] systems. These systems have elements that enable them to use the local copy of the offloadable computation in case the connectivity to the surrogate is lost.

Motivation. 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.

Description. Figure 2.28 is an extension of the Computation Offload tactic (Section 2.1.1) marked with numbers that indicate the sequence of operations that trigger the local fallback. The *Cyber-Foraging Enabled Mobile App* starts the computation offload process by contacting the *Offload Client* which in turn contacts the *Offload Server* that sets up the *Offloaded Code* on the *Surrogate*. Upon completion of the setup process the *Cyber-Foraging Enabled Mobile App* starts execution of the *Offloaded Code* on the *Surrogate*. During execution the *Cyber-Foraging Enabled Mobile App* detects a timeout in the communication with the *Surrogate* (or a network monitor detects loss of connectivity). At this point the *Cyber-Foraging Enabled Mobile App* executes the local version of the offloaded code.

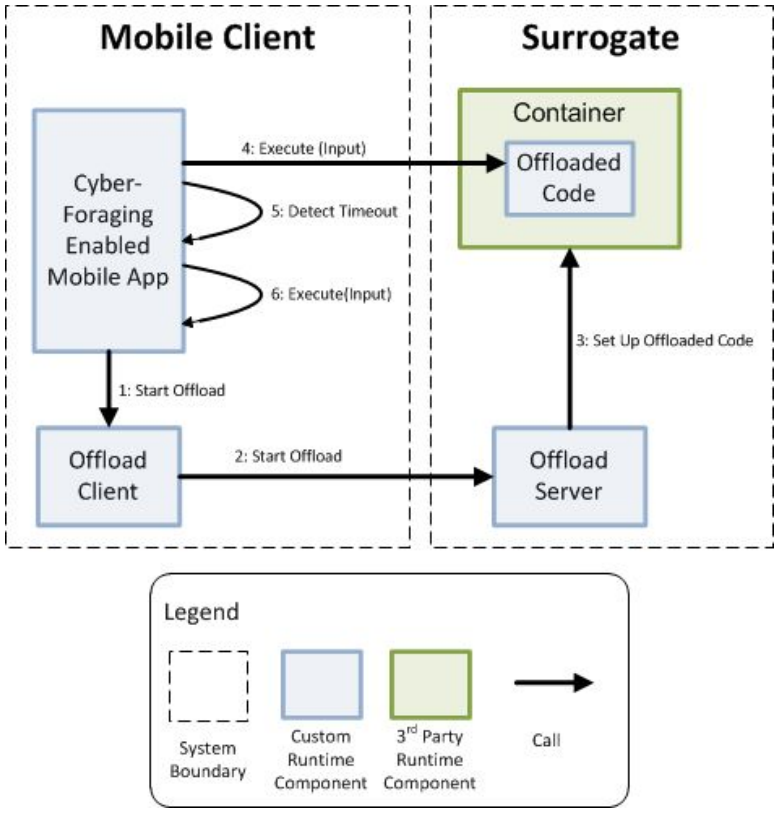

Figure 2.28: Local Fallback

Constraints. 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. For stateful operations, program state has to be synchronized between the local and remote versions of the computation. In cases of data staging, results would need to be cached locally until connectivity is available and would have to use local data that can potentially be out-of-date. For systems that implement the Just-In-Time Containers tactic (Section 2.2.3.1)

with the Local Fall back tactic, these systems 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.

Examples. The following two examples illustrate the Local Fallback tactic:

- MAUI [Cue12]: MAUI detects failures using a simple timeout feature that returns control back to the mobile device. If a disconnect occurs, MAUI 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. Figure 2.29 is based on Figure 2.25 to reflect what occurs in the MAUI system after the remote execution decision has been made. The *App* starts the offload process by invoking the *Client Proxy* which invokes the *Server Proxy* that invokes the remote method. When the *Client Proxy* detects a timeout, it invokes the local method.

- ThinkAir [KAH+12]: If the connection fails for any reason during remote execution, the framework falls back to local execution, discarding any data collected by the profiler. There is no need to synchronize state because an offload request includes the computation itself along with its state and parameters.

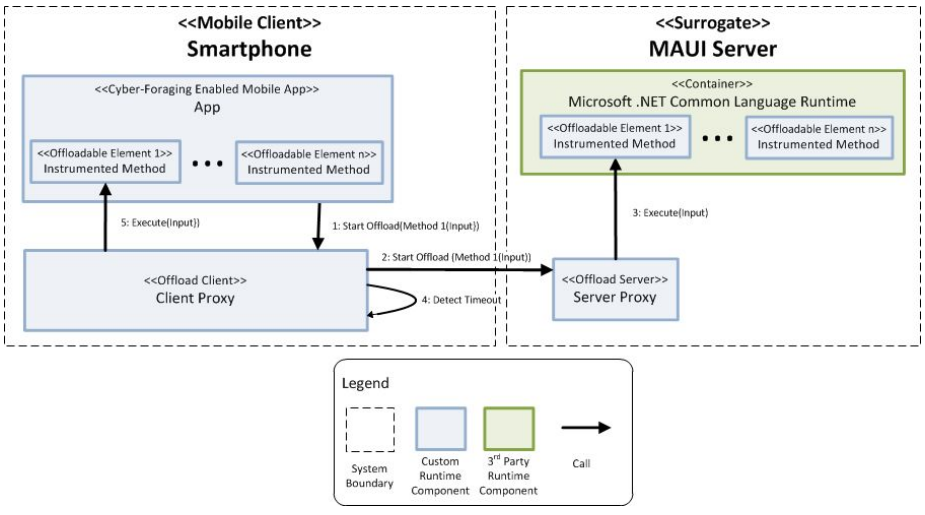

Figure 2.29: MAUI as an Example of the Local Fallback Tactic

Dependencies. The Local Fallback tactic requires a Surrogate Provisioning tactic (Section 2.1.3) to prepare the surrogate for computation offload or data staging, and a Computation Offload tactic (Section 2.1.1) or Data Staging tactic (Section 2.1.2) to enable the actual computation offload or data staging process.

2.2.2.2 Opportunistic Mobile-Surrogate Data Synchronization

The Opportunistic Mobile-Surrogate Data Synchronization tactic for fault tolerance is not present in any of the cyber-foraging systems in the primary studies. However, elements present in the Collaborative Applications [CH11] and Virtual Phone [HSL11] systems could be used in a system that implements the tactic.

Motivation. Data-reliant cyber-foraging systems, as their name indicates, rely on stored data to fulfill their operations. As in the Local Fallback tactic (Section 2.2.2.1), the mobile device may lose connectivity to the surrogate during the computation offload or data staging process. 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.

Description. Figure 2.30 shows the main elements of the tactic. The data synchronization process can be triggered by the *Cyber-Foraging Enabled Mobile App* right before computation offload by synchronously invoking the *Data Synchronization Client* that ensures that *App Data* is synchronized. It can also be started by the *Data Synchronization Client* asynchronously according to pre-defined *Data Synchronization Policies* that determine an optimal time for synchronization such as periodic synchronization, optimal bandwidth, or detection of re-connection.

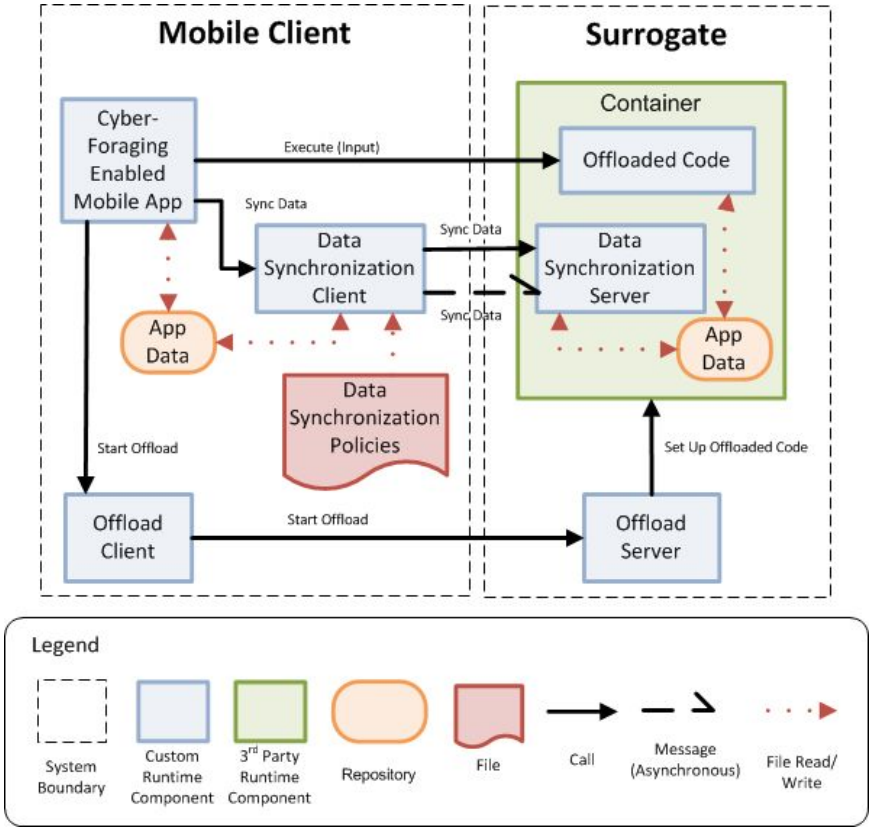

Figure 2.30: Opportunistic Mobile-Surrogate Data Synchronization

Constraints. Systems that implement this 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. Finally, like in any distributed data system, conflict resolution between systems that update data simultaneously is challenging.
Examples. As mentioned earlier, there are no systems in the primary studies that implement the Opportunistic Mobile-Surrogate Data Synchronization tactic for fault tolerance as described, but the principle of using distributed storage is the same: to opportunistically keep data/state synchronized without placing the responsibility on the actual applications. The Collaborative Applications [CH11] and Virtual Phone [HSL11] are computation offload systems that use FUSE[8] for state synchronization between the mobile device and the surrogate to guarantee fidelity of results, meaning that the local and remote computation produce identical results because they are operating on the same state.
Dependencies. The Opportunistic Mobile-Surrogate Data Synchronization tactic requires a Surrogate Provisioning tactic (Section 2.1.3) to prepare the surrogate for computation offload or data staging, and a Computation Offload tactic (Section 2.1.1) or Data Staging tactic (Section 2.1.2) to enable the computation offload or data staging process.
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, the *Data Synchronization Client* runs on the *Surrogate* and the *Data Synchronization Server* runs in the cloud. The Trusted and Unmanaged Data Staging Surrogates [FSTS03] is a data staging system that implements this tactic. It uses a distributed filesystem based on Coda[9] 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. In Figure 2.6 the *Staging Server* includes a *Coda Client* and the *File Server* includes a *Coda Server*.

2.2.2.3 Cached Results

The Cached Results tactic can be found in the Mobile Agents [AB13], 3DMA [FMD05], Grid-Enhanced Mobile Devices [Gua08], CPA [OG13], and Sonora [YQC+12] systems. These systems contain elements that enable them to cache results on the surrogate that can be delivered to, or retrieved by a mobile device after a disconnection.
Motivation. 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

[8] FUSE stands for Filesystem in Userspace; a mechanism that enables a user to create a filesystem without editing kernel code
[9] Coda is a an advanced networked filesystem that supports disconnected operations. More information is available at `http://www.coda.cs.cmu.edu/`

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.

Description. Figure 2.31 shows the main elements of the Cached Results tactic with numbers that indicate the sequence of operations. Steps 1 through 4 describe the basic computation offload process. Starting at Step 5, the *Offloaded Code* on the *Surrogate* executes the offloaded operation and tries to send the results back to the *Cyber-Foraging Enabled Mobile App*. However, it detects that the mobile device is disconnected and therefore saves the results in the *Results Cache* along with information that associates the results with a particular mobile client/user. When the *Mobile Client* reconnects to the *Offloaded Code* on the *Surrogate*, the *Offloaded Code* retrieves the results from the *Results Cache* and sends them back to the *Cyber-Foraging Enabled Mobile App*. Detecting disconnection could be implemented using assured delivery mechanisms that require receipt acknowledgment, or an external component that detects when a mobile device has been disconnected. In systems that always go through the *Offload Client* and the *Offload Server* for interaction, the disconnection detection mechanism and the interaction with the *Results Cache* would be the responsibility of the *Offload Server*. As another option, using message-oriented middleware for communications would enable the results to be delivered automatically to the *Mobile Client* upon reconnection without requiring a *Results Cache*.

Constraints. The tactic as described is best fit for asynchronous interactions between mobile devices and surrogates or applications that are not time-sensitive or require immediate results. In addition, the tactic requires a mechanism for detecting disconnection from mobile devices.

Examples. The following systems implement the Cached Results tactic:

- Grid-Enhanced Mobile Devices [Gua08]: An example of how this system implements the tactic is shown in Figure 2.32 with numbers to indicate sequence of operations. The *User Interface* starts the offload process by invoking the *Connection Manager* with the task to be offloaded. The *Connection Manager* contacts the *Grid Gateway Adapter* on the *Surrogate* which locates a *Grid Service* that can execute the task. Periodically, the *Connection Manager* sends a keep-alive message to the *Grid Gateway Adapter*. If the mobile device fails to send a keep-alive message after a certain period the *Grid Gateway Adapter* assumes that the mobile device has disconnected, whether voluntarily or involuntarily, and informs the *Device Monitor* to mark the device status as disconnected. When the results from the *Grid Service* come back, the *Grid Gateway Adapter* first checks the device status. If it is disconnected, it saves the results in the *Cache*. When the mobile device is re-connected, the *Grid Gateway Adapter* gets the results from the *Cache* and sends them back to the mobile device.

- Mobile Agents [AB13]: Offloadable elements in the form of autonomous mobile agents are migrated from a mobile device to a surrogate for asynchronous execution. The mobile agent platform (JADE) handles the migration back to the client once execution is completed and the mobile device is available.

- 3DMA [FMD05]: The middleware used in the 3DMA system uses the

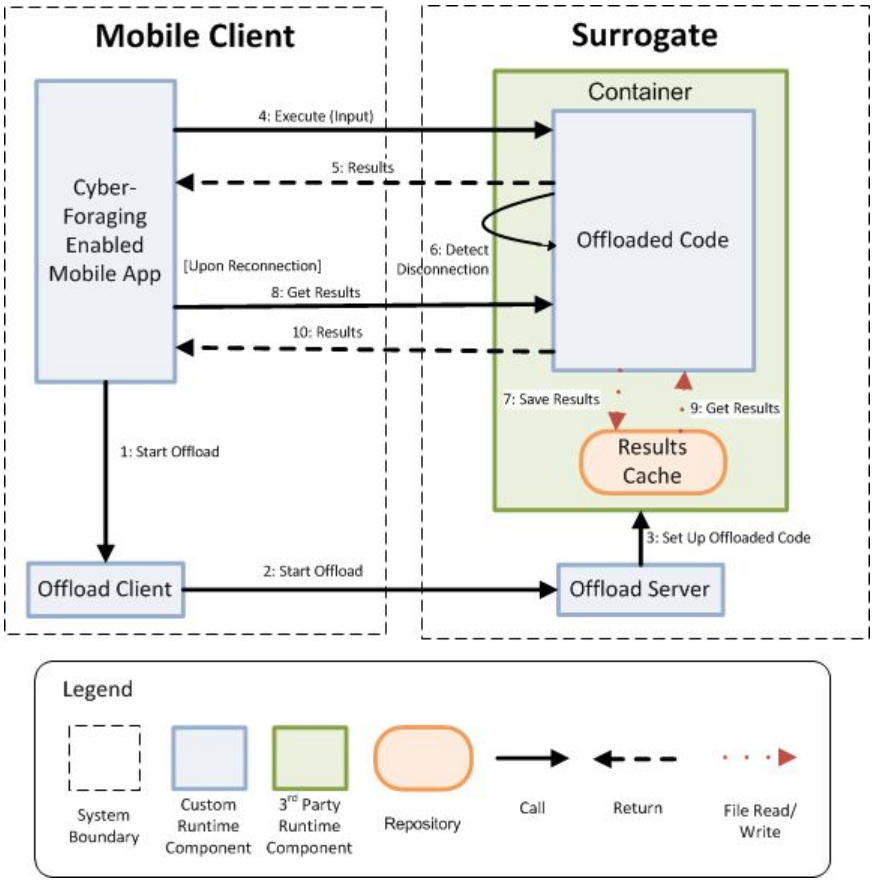

Figure 2.31: Cached Results

concept of spaces to enable asynchronous communication and message buffering. Offload requests from mobile devices are placed in a space, are processed on the surrogate, and results are placed in the same space. When a device becomes disconnected, it waits until a connection is restored, and then reads all available messages (results) from the space.

- CPA [OG13]: Offload requests are sent to the Cloud Personal Assistant component on the surrogate. The request is added as a user task, the task executes, and the status and result data are added as task information. If the mobile device is disconnected, the user can later log in to the system to check task status and results.

- Sonora [YQC+12]: Sonora uses a construct called a sync stream that buffers data during disconnections and resumes normal operation upon reconnection. Connectivity interruptions can either be handled transparently or a mobile app may decide to be notified when disconnections occur.

Dependencies. The Cached Results tactic requires a Surrogate Provisioning tactic (Section 2.1.3) to prepare the surrogate for computation offload or data

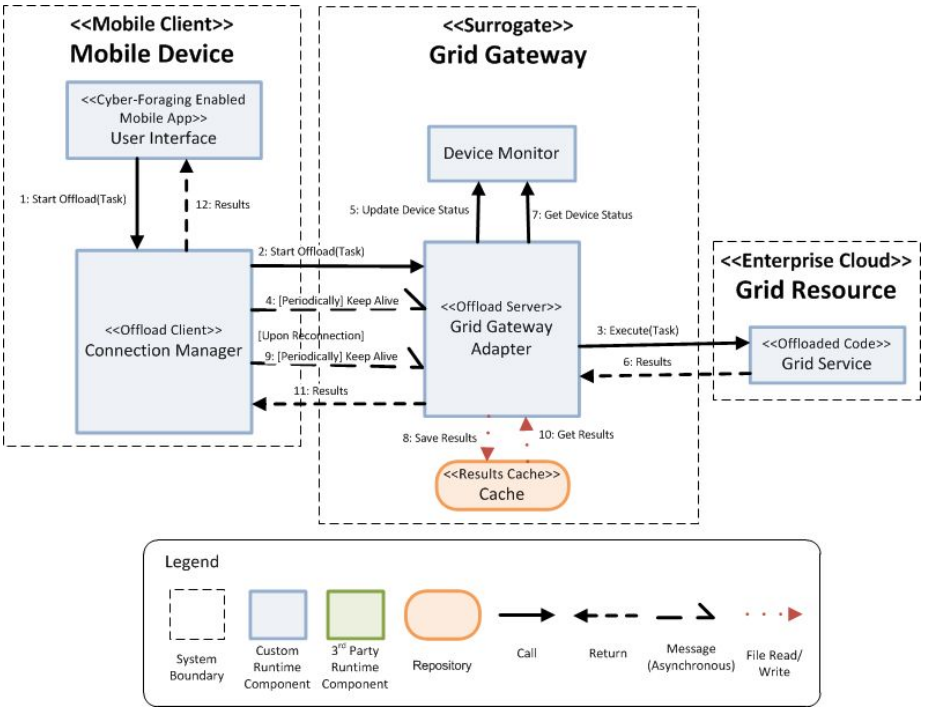

Figure 2.32: Grid-Enhanced Mobile Devices as an Example of the Cached Results Tactic

staging, and a Computation Offload tactic (Section 2.1.1) or Data Staging tactic (Section 2.1.2) to enable the computation offload or data staging process.

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 (Section 2.1.2.2) is to cache collected data on the mobile device and send it to the surrogate upon reconnection. The Feel the World system [PEPD13] is an example of this variation that collects sensor data that can be aggregated and/or transformed locally on the mobile client and uploaded to the surrogate in real-time if the connection is available, or at a later moment if it is unavailable.

2.2.2.4 Alternate Communications

The Alternate Communications tactic is present in the Edge Proxy system [ATAdL06]. This system has elements that enable the surrogate to use an alternate communications mechanism when the mobile device becomes disconnected.

Motivation. 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 [BBV09]. 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 due less amount of information or less timely responses).

Description. Figure 2.33 shows the main elements of the Alternate Communications tactic with numbers to indicate the sequence of operations. Steps 1 to 11 correspond to the basic offload process using the *Default Communications Manager*. In this tactic the interaction between the *Cyber-Foraging Enabled Mobile App* and the *Offloaded Code* happens through the *Offload Client* and the *Offload Server*. When the *Offload Server* is ready to send the results back to the mobile device it detects that it is disconnected. Therefore, the results are delivered to the mobile device using the *Alternate Communications Manager*.

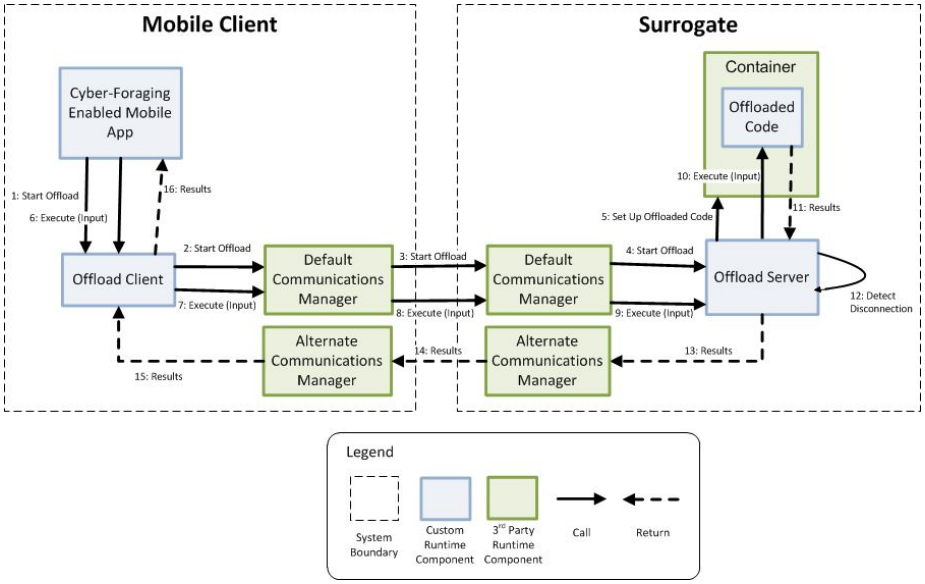

Figure 2.33: Alternate Communications

Constraints. The Alternate Communications tactic as decribed 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.

Example. Edge Proxy [ATAdL06] is a data staging system that implements the Alternate Communications tactic. The system enables a user to be notified when web pages of interest change (Section 2.1.2.2 contains system details). Steps 1 to 4 in Figure 2.34 show the registration process using the *WiFi Manager*. When the *Edge Proxy* 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. It creates a single SMS message with two parts and sends it using the *SMS Manager*. The first part contains control information which includes the number of updates and the size of the download. The second part is an update summary that includes a list of the pages that have changed, and if particular values were being monitored, the changes that occurred. The *Mobile Proxy* intercepts the SMS message, extracts the control information, and passes the update summary back to the *SMS Manager* for delivery to the user via an *SMS Client*. The *Mobile Proxy* uses the control information to make a decision on how to acquire the updates. Because the user receives an update summary, it may be the case that the information of interest is already there and therefore there is no immediate need to reconnect.

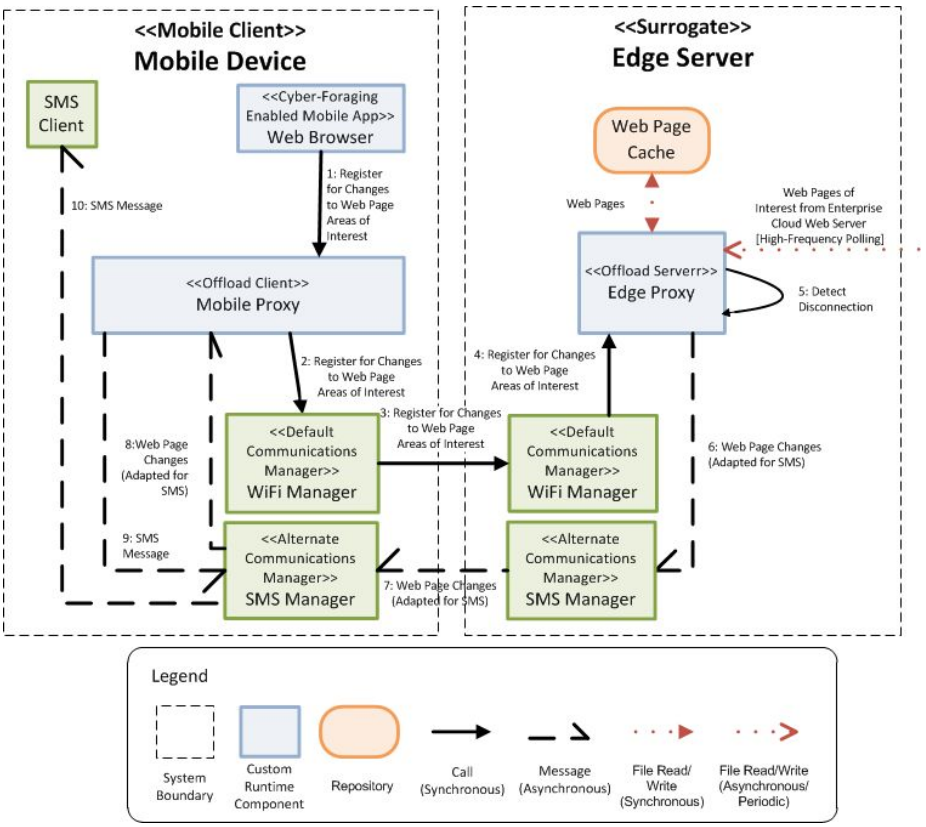

Figure 2.34: Edge Proxy as an Example of the Alternate Communications Tactic

Dependencies. The Alternate Communications tactic requires a Surrogate Provisioning tactic (Section 2.1.3) to prepare the surrogate for computation offload or data staging, and a Computation Offload tactic (Section 2.1.1) or Data Staging tactic (Section 2.1.2) to enable the computation offload or data staging process.

2.2.2.5 Eager Migration

The Eager Migration tactic is present in the Offloading Toolkit and Service system [YOC08]. This system has elements that enable the surrogate to migrate the offloaded computation to a another connected surrogate when it detects that it might not be able to continue serving the mobile device that generated the offload request.

Motivation. 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 2.2.2.1), Cached Results (Section 2.2.2.3), and Alternate Communications (Section 2.2.2.4) 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 supporting the computation offload or data staging operations.

Description. Figure 2.35 shows the main elements of the Eager Migration tactic with numbers to indicate the sequence of operations. Steps 1 to 4 are part of the basic offload process from the *Mobile Client* to the *Source Surrogate*. Periodically, the *Offload Client* sends connection information to the *Offload Server* that it uses to determine if there is a potential for disconnection. This information could be location, signal strength, or available bandwidth. An alternative is for the *Offload Server* to obtain this information periodically using a network monitor. Once the *Offload Server* determines that there is a potential for disconnection, it starts the migration process by contacting the *Offload Server* of the *Target Surrogate* to migrate the offloaded code. It may be the case that there is more the one *Target Surrogate* available, in which case the *Offload Server* would have to select one based on a defined optimization function such as connection bandwidth, load, or available resources on the target. Depending on the granularity of the offloaded code and whether state needs to be transfered or not, the migration process can range from changing the endpoint for communication, to migrating just the offloaded code, to migrating the full container. Once the migration is complete, the *Offload Server* informs the *Offload Client* to connect to the *Target Surrogate*. Optionally, the *Offload Server* may need to clean up the offload process by for example stopping running instances, deleting state files, or terminating VMs. The *Target Surrogate* takes over the execution entirely. The interaction between the *Cyber-Foraging Mobile App* and the *Source Surrogate* finishes. The results from invoking the *Offloaded Code* will come from the *Target Surrogate* and any new interactions will be with the *Target Surrogate*.

Constraints. The tactic as described 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 be able 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.

Example. The Offloading Toolkit and Service [YOC08] system implements the Eager Migration technique as shown in Figure 2.36. If the communication between the *Source Surrogate* and the *Mobile Handheld* deteriorates based on

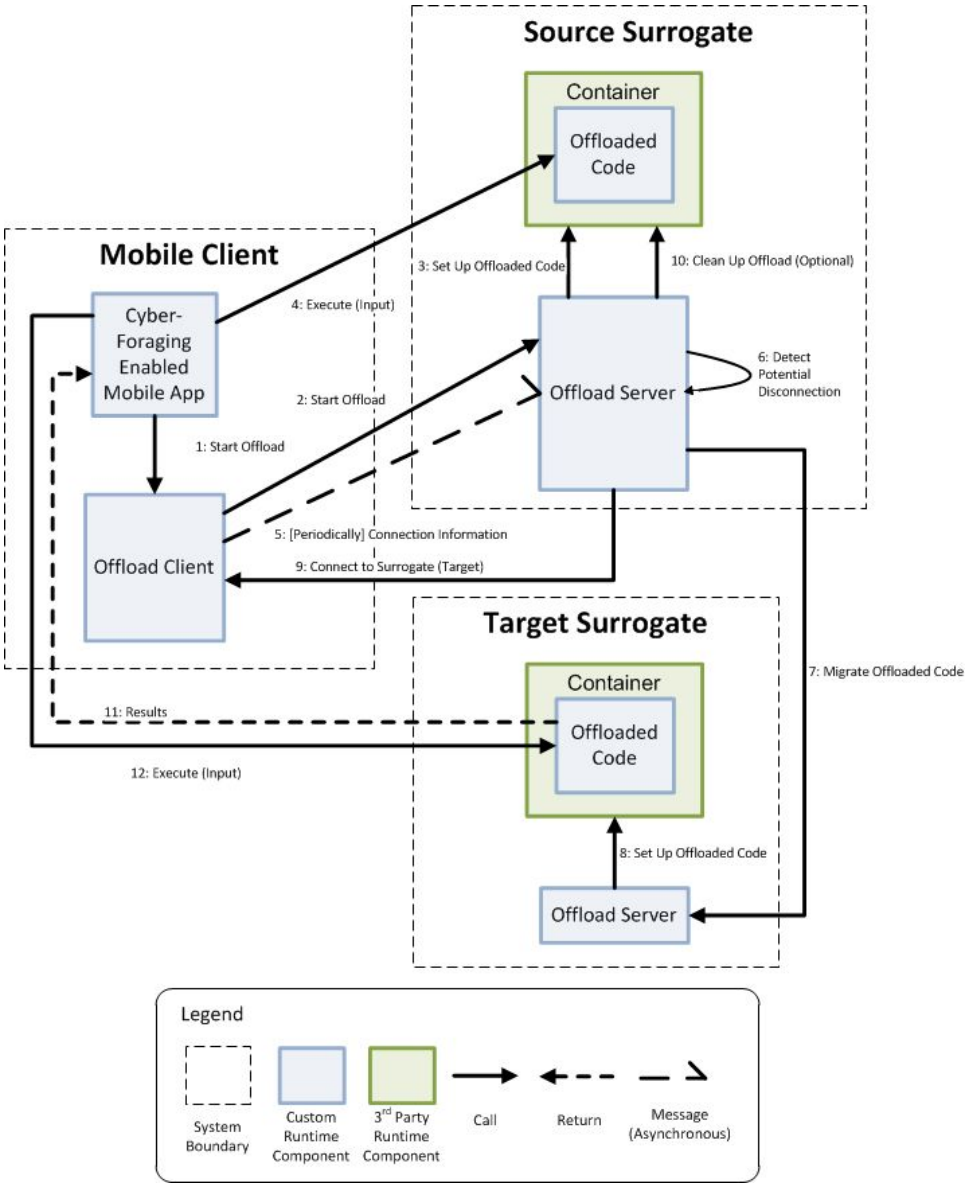

Figure 2.35: Eager Migration

reaching an established threshold for connection quality, the execution of the offloaded *Classes* is terminated on the *Source Surrogate* and migrated from the *Source Surrogate* to a *Connected Target Surrogate*. The migration consists of serializing and sending the *Classes* from the JVM on the *Source Surrogate* to the JVM on the *Connected Target Surrogate* where they are deserialized and loaded.

Dependencies. The Eager Migration tactic requires a Surrogate Provisioning tactic (Section 2.1.3) to prepare the surrogate for computation offload or data

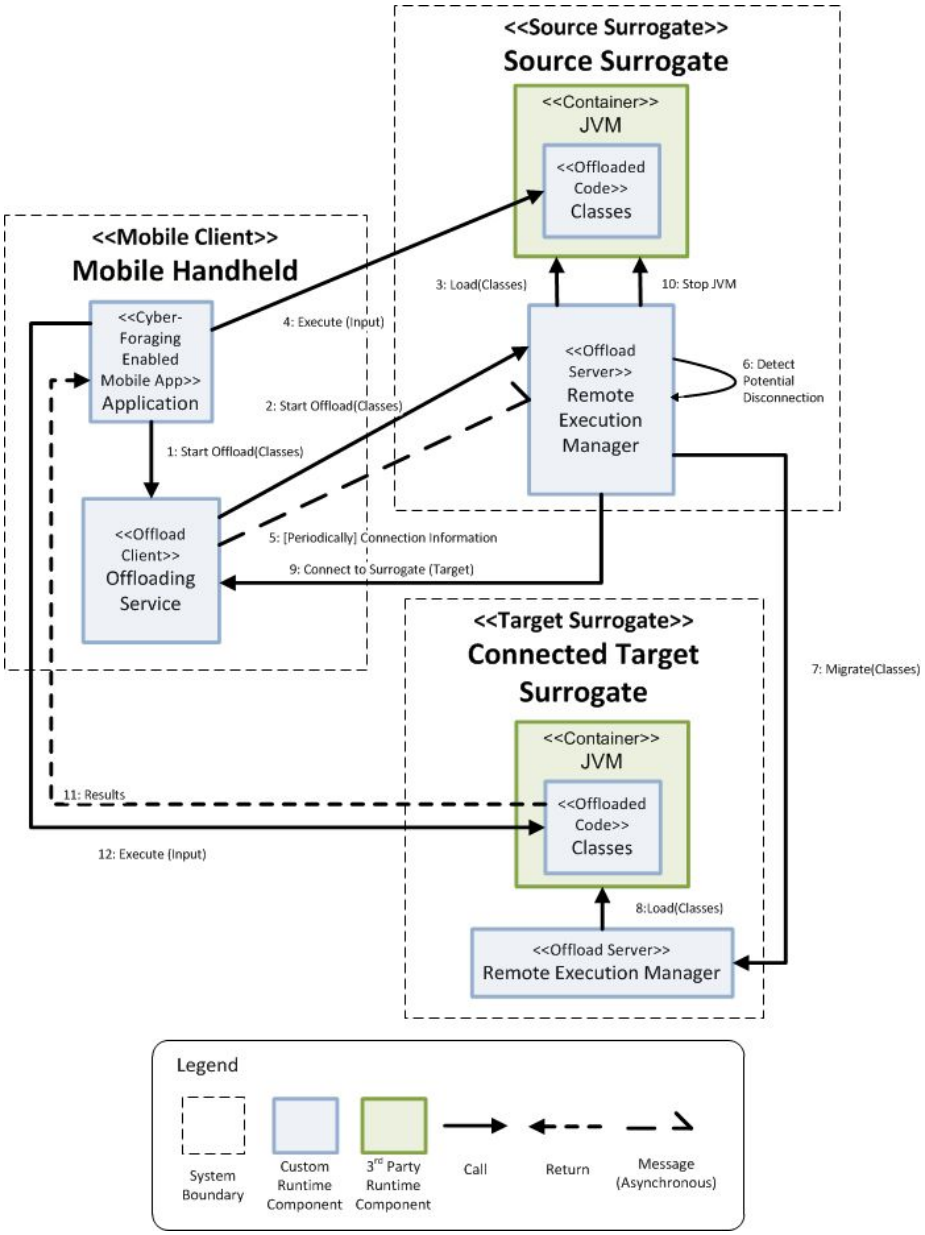

Figure 2.36: Offloading Toolkit and Service as an Example of the Eager Migration Tactic

staging, and a Computation Offload tactic (Section 2.1.1) or Data Staging tactic (Section 2.1.2) to enable the computation offload or data staging process.

Variation: Lazy Migration. In Eager Migration the offloaded computation fully moves from a the *Source Surrogate* to a *Target Surrogate* and the *Mobile Client* 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 Client* is handed off to the *Target Surrogate*. This means that all interaction between the *Mobile Client* 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 [YOC08] system. It was not selected because of the high bandwidth between surrogates that enabled the system to perform a fast full migration.

2.2.3 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.

The Scalability/Elasticity tactics need to be combined with a Surrogate Provisioning tactic (Section 2.1.3) to prepare the surrogate for computation offload or data staging and a Computation Offload tactic (Section 2.1.1) or Data Staging tactic (Section 2.1.2) to enable the computation offload or data staging process.

2.2.3.1 Just-in-Time Containers

The Just-In-Time Containers tactic is present in the Grid-Enhanced Mobile Devices [Gua08] and VM-Based Cloudlets [SBCD09] systems.

Motivation. 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.

Description. Figure 2.37 contains the main elements of the Just-In-Time Containers tactic with numbers to indicate the sequence of operations. The *Cyber-Foraging Enabled Mobile App* starts the offload process by invoking the *Offload Client*. When the *Offload Server* on the *Surrogate* receives the offload request, it creates and starts an instance of the *Offloaded Code* inside the *Container*. The *Cyber-Foraging Enabled Mobile App* interacts with the *Offloaded Code* until it finishes the offload request or closes. At this time the *Cyber-Foraging Enabled Mobile App* ends the offload process by invoking the *Offload Client*. When the *Offload Server* receives the request to end the offload process it destroys the instance of the *Offloaded Code*, thereby releasing the resources that were allocated to it.

Constraints. The tactic as described has a greater startup time than a tactic in which the offloaded code is already running because it has to set up the container, which is the execution environment for the offloaded code.

Examples. In the Grid-Enhanced Mobile Devices [Gua08] system 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

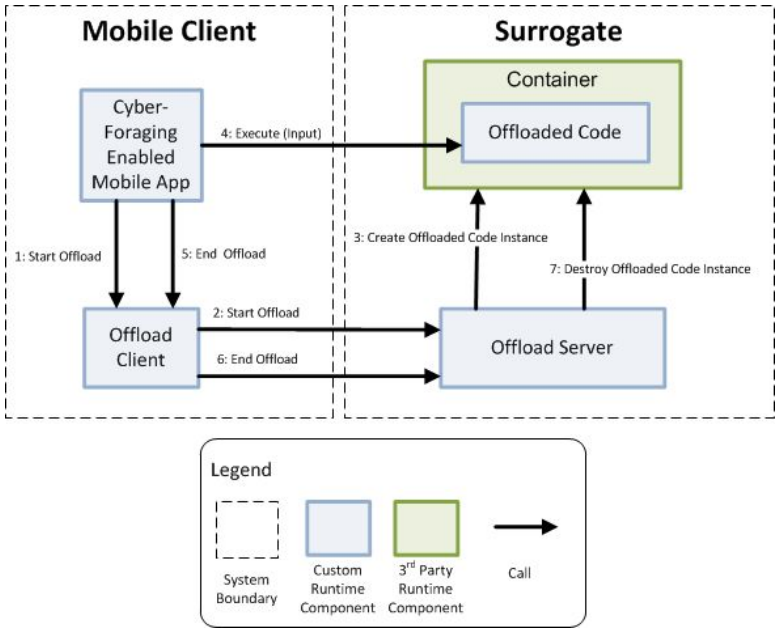

Figure 2.37: Just-In-Time Containers

the connection to the Grid Gateway, resources on the surrogate are released and the Deputy Object is destroyed. The Grid Gateway has a *gateway capacity* that measures its load. Offload requests are granted by the Grid Gateway only if load values are below the gateway capacity. If not, offload requests have to wait until resources are released. In the VM-Based Cloudlets [SBCD09] system shown in Figure 2.38 offloaded computation is prepared for execution on a *Cloudlet* using a technique called *VM Synthesis* (details are provided in Section 2.1.3.2). The *KCM Client* starts the offload process. The *KCM Server* creates and installs the synthesized VM inside the *VM Manager* and informs the *KCM Client* that the VM is ready for execution. The *KCM Client* starts a *VNC Client* that is used to interact with the *Launch VM*. When the *VNC Client* closes, the *KCM Client* ends the offload process by invoking the *KCM Server*, which terminates the *Launch VM*. The term used by the authors to describe the approach is *transient customization of cloudlet infrastructure using hardware VM technology.*

Dependencies. The Just-In Time Containers tactic requires a Surrogate Provisioning tactic (Section 2.1.3) to prepare the surrogate for computation offload or data staging, and a Computation Offload tactic (Section 2.1.1) or Data Staging tactic (Section 2.1.2) to enable the computation offload or data staging process.

2.2.3.2 Right-Sized Containers

The Right-Sized Containers tactic is present in theThinkAir system [KAH+12]. This system has elements that create execution containers that are of thhe appropriate size for the offloaded computation in order to optimize resource usage on the surrogate.

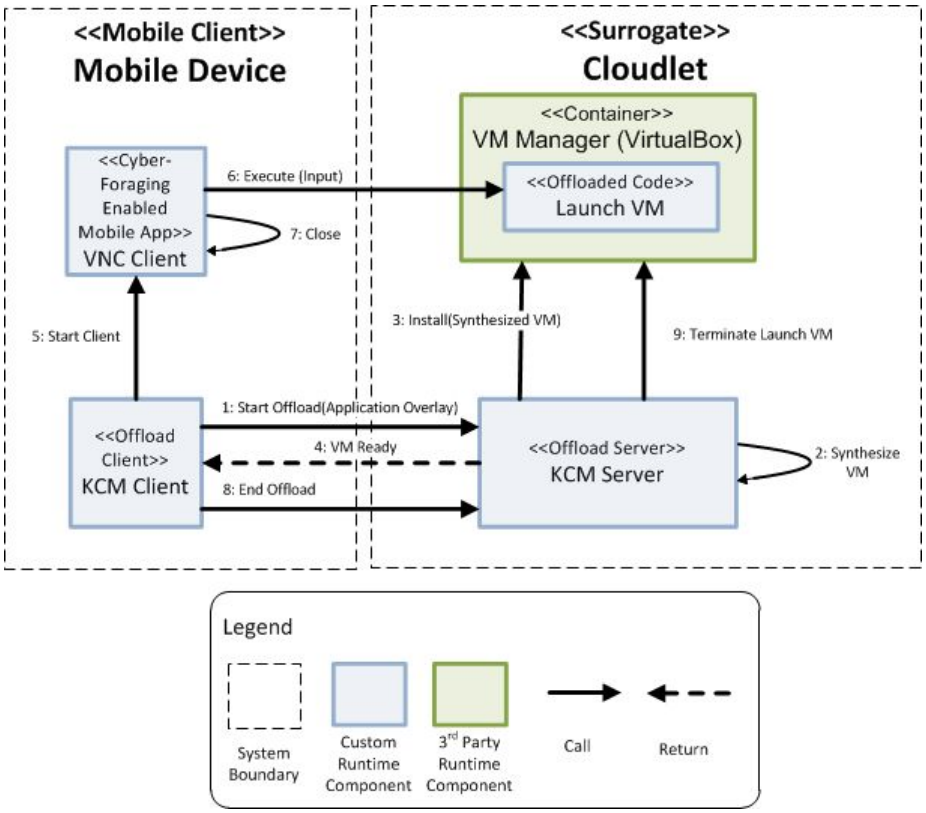

Figure 2.38: VM-Based Cloudlets as an Example of the Just-In-Time Containers Tactic

Motivation. 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.

Description. Figure 2.39 shows the main elements of the Right-Sized Containers tactic. The *Cyber-Foraging Enabled Mobile App* starts the offload process by invoking the *Offload Client* with *Offloaded Code Metadata* that indicates the computing requirements for the *Offloaded Code*. In the case of pre-provisioned surrogates (Section 2.1.3.1) the *Offloaded Code Metadata* could reside on the *Surrogate*. Based on the metadata received from the *Offload Client*, the *Offload Server* obtains a container from the *Container Repository* that best matches the metadata, meaning that the resources that are required from the *Surrogate*

are sufficient to execute the *Offloaded Code*. The *Offload Server* then starts the container and sets up the *Offloaded Code* so that it is ready for execution from the *Cyber-Foraging Enabled Mobile App*.

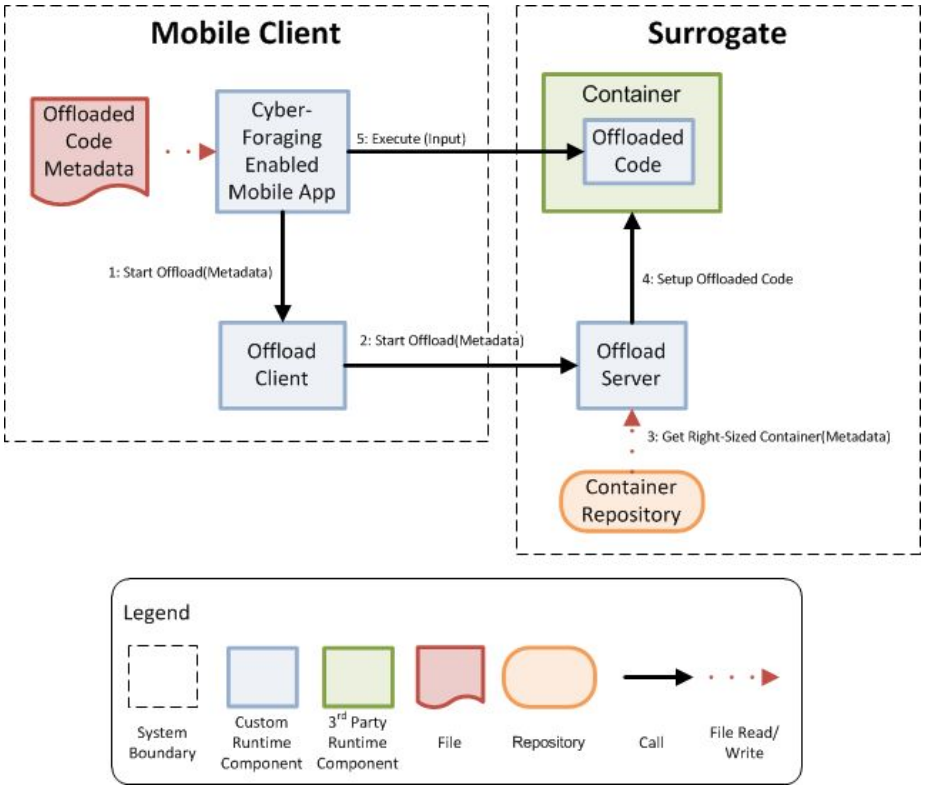

Figure 2.39: Right-Sized Containers

Constraints. The tactic as described requires a surrogate to maintain different container configurations. In addition, similar to the Just-In-Time Containers tactic (Section 2.2.3.1), it has a greater startup time than a tactic in which the offloaded code is already running because it has to set up the right container as the execution environment for the offloaded code.

Example. The ThinkAir system [KAH+12] implements the Right-Sized Containers tactic, as shown in Figure 2.40. When a surrogate (*Application Server*) receives an offload request, the *ThinkAir Framework* on the *Application Server* 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 configuration which differ in terms of CPU and memory). The *ThinkAir Framework* starts the selected VM configuration and sets up the offloaded code (*Code and Data*) in the VM.

Dependencies. The Right-Sized Containers tactic requires a Surrogate Provisioning tactic (Section 2.1.3) to prepare the surrogate for computation offload or data staging, and a Computation Offload tactic (Section 2.1.1) or Data Staging tactic (Section 2.1.2) to enable the computation offload or data staging process.

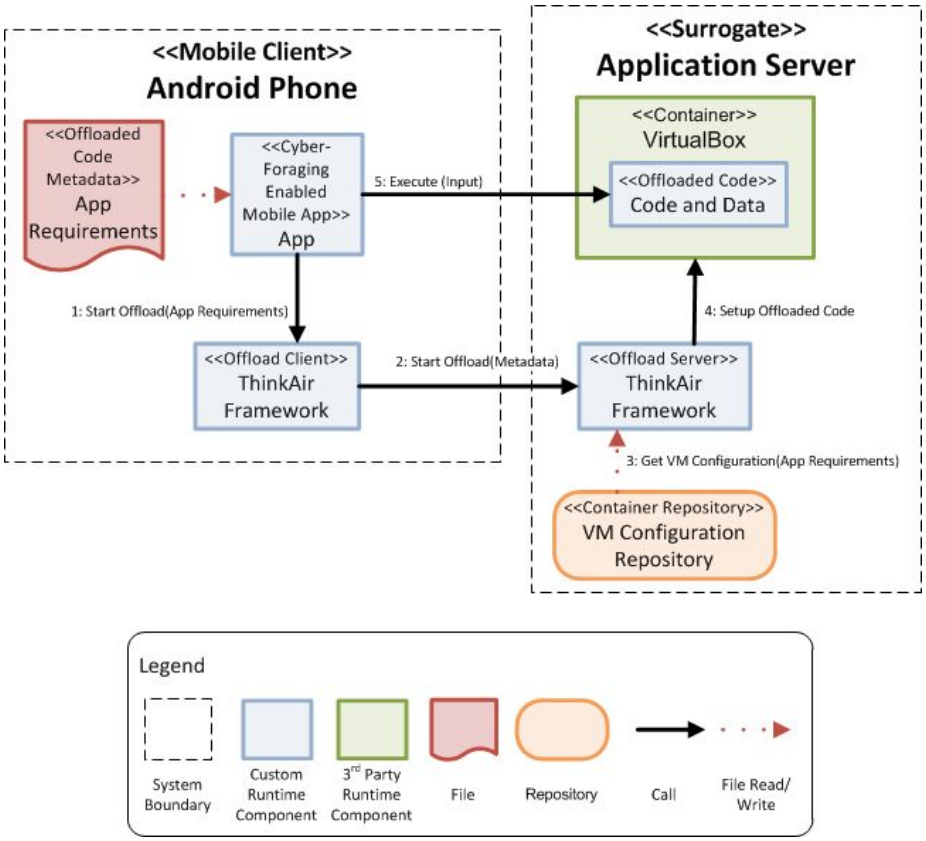

Figure 2.40: ThinkAir as an Example of the Right-Sized Containers Tactic

Variation: Dynamically-Sized Containers. The ThinkAir system [KAH+12] 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 Client Handler starts a more powerful VM and moves the offload request to the newly started VM.

2.2.3.3 Surrogate Load Balancing

The Surrogate Load Balancing tactic is present in the The Cloud Operating System to Support Multi-Server Offloading [Ima12]. The system has elements that enable it to leverage connected surrogates for load balancing.

Motivation. 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.

Description. The Surrogate Load Balancing tactic uses the same computation migration techniques as the Eager Migration tactic (Section 2.2.2.5) but for a

different purpose (scalability/elasticity instead of fault tolerance). Figure 2.41 shows the main elements of the tactic with numbers that indicate the sequence of operations. Steps 1 to 4 are part of the basic offload process from the *Mobile Client* to the *Source Surrogate*. During the execution of the *Offloaded Code*, the *Load Monitor* informs the *Offload Server* that the *Surrogate* has reached its load threshold. The *Offload Server* then migrates one or more instances of *Offloaded Code* to a *Target Surrogate*. It may be the case that there is more than one connected *Target Surrogate* available, in which case the *Offload Server* would have to select one based on a defined optimization function which should balance the load among all connected surrogates, but may also include connection bandwidth or available resources on the *Target Surrogate*. Depending on the granularity of the offloaded code and whether state needs to be transfered or not, the migration process can range from changing the endpoint for communication, to migrating just the offloaded code (application-level migration), to migrating the full container (container-level migration). Once the migration is complete, the *Offload Server* informs the *Offload Client* to connect to the *Target Surrogate*. The *Offload Server* terminates the instance of the *Offloaded Code* by stopping running instances, deleting state files, or terminating VMs in order to reduce the load on the *Source Surrogate*. The *Target Surrogate* takes over the execution entirely. The interaction between the *Cyber-Foraging Mobile App* and the *Source Surrogate* finishes. The results from invoking the *Offloaded Code* will come from the *Target Surrogate* and any new interactions will be with the *Target Surrogate*.

Constraints. The tactic as described 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 in order to migrate computation to the less-loaded one and keep the load on all the surrogates balanced.

Example. The Cloud Operating System to Support Multi-Server Offloading (COS) system [Ima12] implements this tactic. *Surrogates* in COS are not connected to the enterprise but to other surrogates to load balance. As shown in Figure 2.42, application modules are implemented as *SALSA Actors* that are self-contained and therefore can easily migrate between a *Source Node* and a *Target Node* (application-level migration). The *Target Node* is selected based on resource availability, communication cost with other actors, and the cost for migration. Because migrating actors is similar to performing a split (removing an actor from a VM on a node) and a merge (adding an actor to a VM on a node), COS refers to this aspect of the system as *VM malleability*. The system also has a *COS Manager* that is connected to all *Node Managers* and is contacted during the *Identify Target Surrogate* operation (Step 6 in the figure). The *COS Manager* can run on any COS node or in a separate node. When the *Source Node* reaches a load threshold, the *Node Manager* informs the *COS Manager*, which determines the optimal *Target Node* and then prepares the *Target Node* for migration.

Dependencies. Even though the Surrogate Load Balancing tactic does not require any other tactic in order to be implemented, it only makes sense if combined with a Surrogate Provisioning tactic (Section 2.1.3) to prepare the surrogate for computation offload or data staging, and a Computation Offload tactic (Section 2.1.1) or Data Staging tactic (Section 2.1.2) to enable the com-

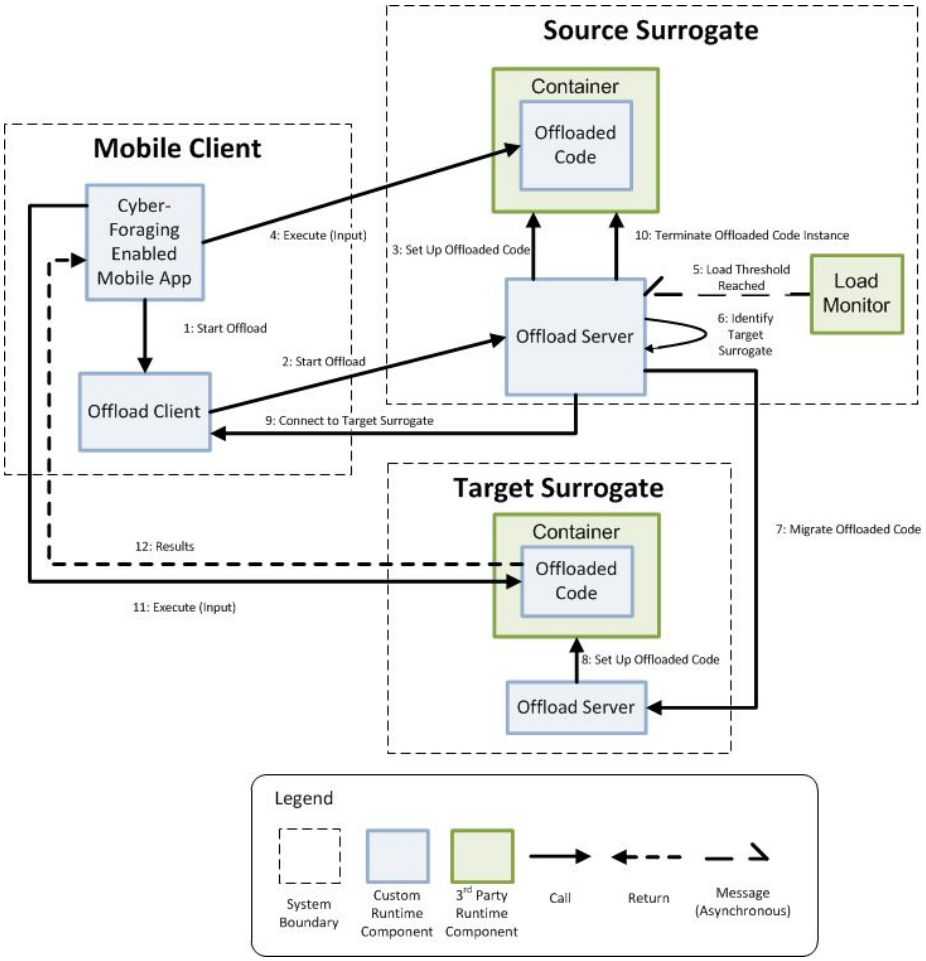

Figure 2.41: Surrogate Load Balancing

putation offload or data staging process. The Surrogate Load Balancing tactic then provides scalability to a computation offload or data staging system.

2.2.4 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.

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.

Even though a Security tactic does not require any other tactic in order to be

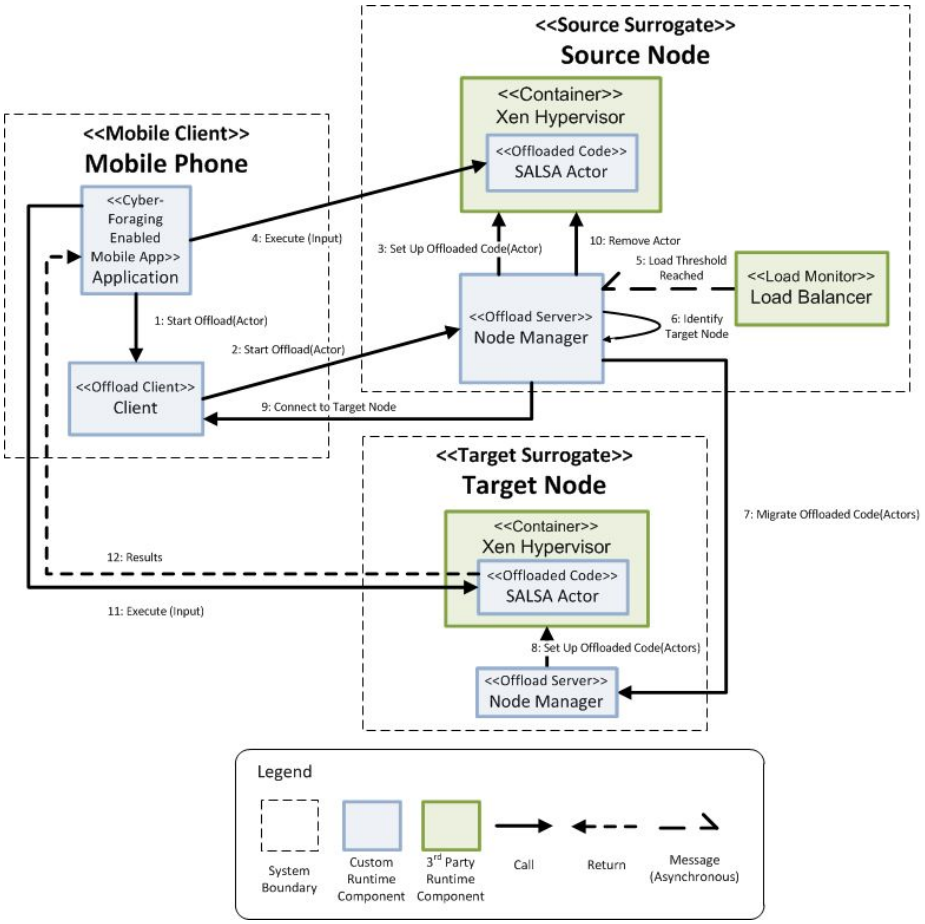

Figure 2.42: Cloud Operating System to Support Multi-Server Offloading as an Example of the Surrogate Load Balancing Tactic

implemented, in the context of cyber-foraging it only makes sense if combined with a Surrogate Provisioning tactic (Section 2.1.3) to enable the surrogate for computation offload or data staging, and a Computation Offload tactic (Section 2.1.1) or Data Staging tactic (Section 2.1.2) to enable the computation offload or data staging process. The Security tactic then provides assurance to both the mobile device and the surrogate that they will not be compromised or used to compromise other systems.

2.2.4.1 Trusted Surrogates

Motivation. 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.

Description. As mentioned earlier, there is not much discussion about security or trust in the primary studies. An approach that is shown in some of the primary studies is to own the surrogate. Roam [CSW+04] assumes that a user would only offload applications among his/her personal devices such as cell phones, PDAs, and home PCs. Collaborative Applications [CH11] and SPADE [SVF08] offload only to personal trusted servers such as a home server. The Grid-Enhanced Mobile Devices system [Gua08] assumes a pre-existing trust relation between mobile devices, the *Grid Gateway* that serves as an intermediary between the mobile device and the surrogates, and the surrogates (*Grid Service Providers*). In the proposed implementation, the mobile user uses his own desktop as the Grid Gateway and all Grid Service Providers are owned by the user's organization.

Another hardware-based approach that is suggested for establishing trust, but not implemented in any of the primary studies, is to use an on-board secure hardware component such as Trusted Platform Module (TPM). TPM is a device/chip that has a unique and secret RSA key that is burned into it when it is produced.[10] Collaborative Applications [CH11], Virtual Phone [HSL11] and VM-Based Cloudlets [SBCD09] suggest the use of TPM for providing stronger levels of trust.

In the Collective Surrogates system [Goy11] only the trusted *Collective Manager* that serves as the broker between mobile devices and surrogates has direct access to the VM running on a surrogate (*Participating Node*). This system exploits the isolation provided by VM technology for safely running arbitrary code provided by mobile devices. However, the system assumes a trust relationship between mobile device and the Collective Manager.

While a password- or hardware-based approach are useful for some scenarios, it is not appropriate in more dynamic scenarios in which mobile devices discover nearby surrogates that are not owned by the owner of the mobile device (Section 2.1.4.3). These scenarios require more dynamic ways of establishing trust between mobile devices and surrogates, such as a third-party, online trusted authority that validates credentials or a certificate authority that provides certificates and keys for authentication, to determine if data or code has been tampered with, or even encryption (as an example, the Virtual Phone system [HSL11] has a fully-encrypted filesystem on the surrogate to ensure that data is not accessible by surrogate owners or other virtual machines running on the surrogate).

Constraints. Each of the approaches listed above has constraints related to 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 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.

Example. The only system that implements a trust solution that uses a third-

[10] The ISO/IEC 11889 specification for TPM is available at `http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html`

party trusted authority is the Trusted and Unmanaged Data Staging Surrogates system [FSTS03]. This system was used as an example for the Pre-Fetching tactic in Figure 2.6. A subset of this figure with additional detail related to the trust components is presented in Figure 2.43 with numbers to indicate the sequence of operations. The user's idle *Desktop* serves as the trusted third party that sits in between the *Server* and the *Surrogate*. When the *File Client* requests a file, the *Client Proxy* 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 in the *Cache*. It then sends the *Client Proxy* 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.

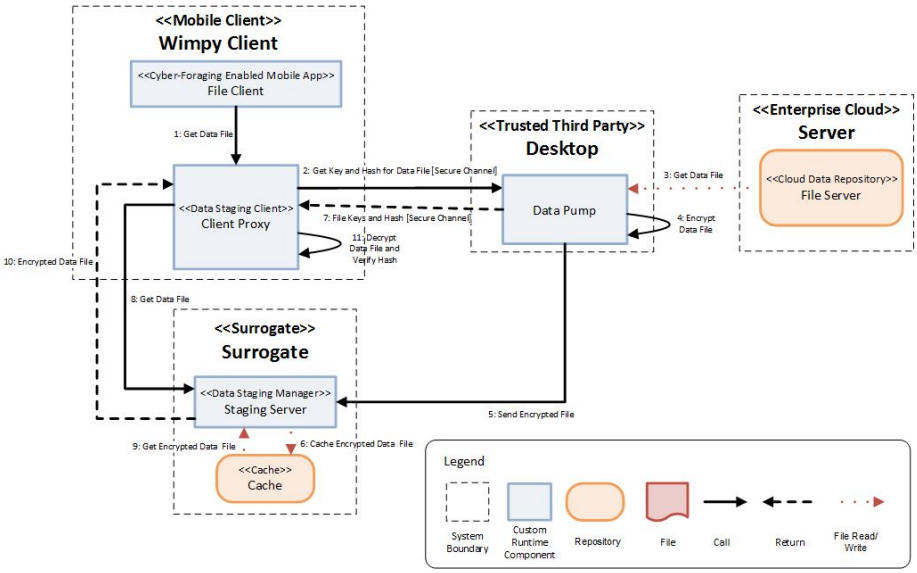

Figure 2.43: Trusted and Unmanaged Data Staging Surrogates as an Example of the Trusted Surrogates Tactic

Dependencies. Even though the Trusted Surrogate tactic does not require any other tactic in order to be implemented, it only makes sense if combined with a Surrogate Provisioning tactic (Section 2.1.3) to prepare the surrogate for computation offload or data staging, and a Computation Offload tactic (Section 2.1.1) or Data Staging tactic (Section 2.1.2) to enable the computation offload or data staging process. The Trusted Surrogate tactic then provides a trusted environment for computation offload or data staging.

Chapter 3

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 systematic literature reviews and do not have an architecture focus. Abolfazli et al [ASA+14] present a survey of cloud-based mobile augmentation (CMA) approaches, one of which is cyber-foraging. One of the challenges stated by this work is the lack of a reference architecture for CMA. Dinh at al [DLNW11] present a survey on mobile cloud computing (MCC). Computation offload is discussed as a technique for extending battery lifetime of mobile devices and listed as one of the challenges for MCC. Fernando et al [FLR12] present a more complete survey on mobile cloud computing. Some of the research that addresses efficient computation offload and distribution to the cloud and how it differs from traditional distributed systems is discussed in this paper. Kumar et al [KLLB13] present a survey on computation offloading but focus primarily on the algorithms used to partition and offload programs in order to improve performance or save energy. Finally, Yu et al [YMCL12] present a survey on seamless application mobility, which is the continuous or uninterrupted computing experience as a user moves across devices. Code offloading is mentioned as a future direction for seamless application mobility.

The work that is most similar to ours is by Flinn et al [Fli12] 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, to the best of our knowledge this is the first attempt to codify design decisions in software architectures for cyber-foraging systems into a set of tactics.

Chapter 4

Conclusions and Next Steps

We presented a set of architectural tactics for cyber-foraging that were obtained from the results of a systematic literature review 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. Non-functional tactics provide the basic cyber-foraging operations and non-functional tactics are combined with the functional tactics to support required system qualities.

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. The case studies will be analyzed to identify tactics that are common across case studies and codify them into architectural patterns, based on the definition that an architectural pattern is a widely recognized and reused solution to a recurring design problem in the field of software architectures [BMR+96].

Bibliography

[AB13] Pelin Angin and Bharat 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.

[ACH12] Andrius Aucinas, Jon Crowcroft, and Pan Hui. Energy efficient mobile M2M communications. In *Proceedings of ExtremeCom '12*, 2012.

[AP13] JongHoon Ahnn and Miodrag Potkonjak. mhealthmon: Toward energy-efficient and distributed mobile health monitoring using parallel offloading. *Journal of Medical Systems*, 37(5):1–11, 2013.

[ASA+14] Saeid Abolfazli, Zohreh Sanaei, Ejaz Ahmed, Abdullah Gani, and Rajkumar Buyya. Cloud-based augmentation for mobile devices: Motivation, taxonomies, and open challenges. *IEEE Communications Surveys Tutorials*, 16(1):337–368, 2014.

[ATAdL06] Trevor Armstrong, Olivier Trescases, Cristiana Amza, and Eyal 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.

[BBV09] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun 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.

[BCK12] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley, 3rd edition, 2012.

[BGSH07] Rajesh Krishna Balan, Darren Gergle, Mahadev Satyanarayanan, and James Herbsleb. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, MobiSys '07, pages 272–285, New York, NY, USA, 2007. ACM.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture: A*

System of Patterns. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[BWYH06] Ali Bahrami, Changzhou Wang, Jun Yuan, and Anne Hunt. The workflow based architecture for mobile information access in occasionally connected computing. In *Services Computing, 2006. SCC'06. IEEE International Conference on*, pages 406–413. IEEE, 2006.

[CH11] Yu-Shuo Chang and Shih-Hao Hung. Developing collaborative applications with mobile cloud-a case study of speech recognition. *Journal of Internet Services and Information Security (JISIS)*, 1(1):18–36, 2011.

[CKK+04] Guangyu Chen, B-T Kang, Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Rajarathnam Chandramouli. Studying energy trade offs in offloading computation/compilation in java-enabled mobile devices. *Parallel and Distributed Systems, IEEE Transactions on*, 15(9):795–809, 2004.

[CM09] Byung-Gon Chun and Petros 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.

[CP13] Bin Cheng and Michael Probst. HBB-NEXT I D4.4.1: Intermediate middleware software components for cloud service offloading. Technical report, HBB-NEXT Consortium 2013, 2013.

[CSW+04] Hao-hua Chu, Henry Song, Candy Wong, Shoji Kurakake, and Masaji Katagiri. Roam, a seamless application framework. *Journal of Systems and Software*, 69(3):209–226, 2004.

[Cue12] Eduardo Cuervo. *Enhancing Mobile Devices through Code Offload*. PhD thesis, Duke University, 2012.

[DLNW11] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, 13:1587–1611, 2011.

[Dug11] Naod Duga. Optimality analysis and middleware design for heterogeneous cloud HPC in mobile devices. Master's thesis, Addis Ababa University, 2011.

[DZ11] Mian Dong and Lin 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.

[EML11] Rodolfo G Esteves, Michael D McCool, and Christiane Lemieux. Real options for mobile communication management. In *GLOBECOM Workshops (GC Wkshps), 2011 IEEE*, pages 1241–1246. IEEE, 2011.

[EW11] Holger Endt and Kay Weckemann. Remote utilization of opencl for flexible computation offloading using embedded ECUs, CE devices and cloud servers. In *Volume 22: Applications, Tools and Techniques on the Road to Exascale Computing*, volume 22 of *Advances in Parallel Computing*, pages 133–140. IOS Press EBooks, 2011. This paper was not available online. I got it from the library.

[Fli12] Jason Flinn. Cyber foraging: Bridging mobile and cloud computing. In Mahadev Satyanarayanan, editor, *Synthesis Lectures on Mobile and Pervasive Computing*. Morgan & Claypool Publishers, 2012.

[FLR12] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu. Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29:84ï¿½106, 2012.

[FMD05] Tore Fjellheim, Stephen Milliner, and Marlon Dumas. Middleware support for mobile applications. *International Journal of Pervasive Computing and Communications*, 1(2):75–88, 2005.

[FPS02] Jason Flinn, Soyoung Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *In Proceedings of the 22nd International Conference on. Distributed Computing Systems*, pages 217–226, 2002.

[FSTS03] Jason Flinn, Shafeeq Sinnamohideen, Niraj Tolia, and Mahadev 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.

[Goy11] Sachin Goyal. *A Collective Approach to Harness Idle Resources of End Nodes*. PhD thesis, School of Computing, University of Utah, 2011.

[GRJ+09] Ioana Giurgiu, Oriana Riva, Dejan Juric, Ivan Krivulev, and Gustavo Alonso. Calling the cloud: Enabling mobile phones as interfaces to cloud applications. In JeanM. Bacon and BrianF. Cooper, editors, *Middleware 2009*, volume 5896 of *Lecture Notes in Computer Science*, pages 83–102. Springer Berlin Heidelberg, 2009.

[Gua08] Tao Guan. *A System Architecture to Provide Enhanced Grid Access for Mobile Devices*. PhD thesis, University of Southampton, 2008.

[HLSS11] Kiryong Ha, Grace Lewis, Soumya Simanta, and Mahadev Satyanarayanan. Cloud offload in hostile environments. Technical report, Carnegie Mellon University, 2011.

[HSL11] Shih-Hao Hung, Jeng-Peng Shieh, and Chen-Pang Lee. Migrating android applications to the cloud. *International Journal of Grid and High Performance Computing (IJGHPC)*, 3(2):14–28, 2011.

[I+12] Anantharaman Narayana Iyer et al. Extending android application programming framework for seamless cloud integration. In *Mobile Services (MS), 2012 IEEE First International Conference on*, pages 96–104. IEEE, 2012.

[Ima12] Shigeru Imai. Task offloading between smartphones and distributed computational resources. Master's thesis, Rensselaer Polytechnic Institute, 2012.

[JBA12] Chris Jarabek, David Barrera, and John Aycock. ThinAV: truly lightweight mobile cloud-based anti-malware. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 209–218. ACM, 2012.

[KAH+12] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen 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.

[KK12] Dejan Kovachev and Ralf Klamma. Framework for computation offloading in mobile cloud computing. *International Journal of Interactive Multimedia and Artificial Intelligence*, 1(7):6–15, 2012.

[KL10] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, April 2010.

[KLLB13] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140, February 2013.

[KLvV06] Philippe Kruchten, Patricia Lago, and Hans van Vliet. Building up and reasoning about architectural knowledge. In Christine Hofmeister, Ivica Crnkovic, and Ralf Reussner, editors, *Quality of Software Architectures*, volume 4214 of *Lecture Notes in Computer Science*, pages 43–58. Springer Berlin Heidelberg, 2006.

[KMM+07] Suman Kundu, Jayanta Mukherjee, Arun K Majumdar, Bandana Majumdar, and Sirsendu Sekhar Ray. Algorithms and heuristics for efficient medical information display in PDA. *Computers in Biology and Medicine*, 37(9):1272–1282, 2007.

[KPKB12] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a computation offloading framework for smartphones. In *Mobile Computing, Applications, and Services*, pages 59–79. Springer, 2012.

[Kri10] Mads Daro Kristensen. *Empowering Mobile Devices Through Cyber Foraging.* PhD thesis, Aarhus University, 2010.

[KT13] Young-Woo Kwon and Eli Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM 2013)*, 2013.

[LA06] Patricia Lago and Paris Avgeriou. First workshop on sharing and reusing architectural knowledge. *SIGSOFT Softw. Eng. Notes*, 31(5):32–36, September 2006.

[Lee12] Byoung-Dai Lee. A framework for seamless execution of mobile applications in the cloud. In *Recent Advances in Computer Science and Information Engineering*, pages 145–153. Springer, 2012.

[LLP14] Grace Lewis, Patricia Lago, and Giuseppe Procaccianti. Architecture strategies for cyber-foraging: Preliminary results from a systematic literature review. In Paris Avgeriou and Uwe Zdun, editors, *Proceedings of the 8th European Conference on Software Architecture (ECSA 2014)*, volume 8627 of *Lecture Notes in Computer Science*, pages 154–169. Springer International Publishing, 2014.

[LM02] William Lehr and Lee W. McKnight. Wireless internet access: 3G vs. WiFi? Technical Report 166, MIT Sloan School of Management, 2002.

[MCF+11] Jerrid Matthews, Max Chang, ZhiNan Feng, Ravi Srinivas, and Mario 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.

[MGB+02] Alan Messer, Ira Greenberg, Philippe Bernadat, Dejan Milojicic, Deqing Chen, TJ Giuli, and Xiaohui Gu. Towards a distributed platform for resource-constrained devices. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 43–51. IEEE, 2002.

[ML13] Dominik Messinger and Grace A Lewis. Application virtualizaton as a strategy for cyber foraging in resource-constrained environments. Technical report, Carnegie Mellon Software Engineering Institute, 2013.

[MV03] Shivajit Mohapatra and Nalini Venkatasubramanian. Optimizing power using a reconfigurable middleware. Technical report, UC Irvine, 2003.

[OG13] Michael J O'Sullivan and Dan Grigoras. The cloud personal assistant for providing services to mobile clients. In *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*, pages 478–485, 2013.

[OSP07] MinHwan Ok, Ja-Won Seo, and Myong-soon Park. A distributed resource furnishing to offload resource-constrained devices in cyber foraging toward pervasive computing. In *Network-Based Information Systems*, pages 416–425. Springer, 2007.

[PCCY12] Sehoon Park, Youngil Choi, Qichen Chen, and H.Y. Yeom. SOME: Selective offloading for a mobile computing environment. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 588–591, 2012.

[PEPD13] Theophilos Phokas, Hariton Efstathiades, George Pallis, and MariosD. Dikaiakos. Feel the world: A mobile framework for participatory sensing. In Florian Daniel, GeorgeA. Papadopoulos, and

Philippe Thiran, editors, *Mobile Web Information Systems*, volume 8093 of *Lecture Notes in Computer Science*, pages 143–156. Springer Berlin Heidelberg, 2013.

[PTDL08] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: A research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, 2008.

[PXJZ13] Lingjun Pu, Jingdong Xu, Xing Jin, and Jianzhong Zhang. SmartVirtCloud: virtual cloud assisted application offloading execution at mobile devices' discretion. In *2013 IEEE Wireless Communications and Networking Conference (WCNC): SERVICES and APPLICATIONS*, 2013.

[Rac12] Kiran K Rachuri. *Smartphones based Social Sensing: Adaptive Sampling, Sensing and Computation Offloading.* PhD thesis, University of Cambridge, 2012.

[RSM+11] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 43–56, New York, NY, USA, 2011. ACM.

[RVMV12] M Reza Rahimi, Nalini Venkatasubramanian, Sharad Mehrotra, and Athanasios V Vasilakos. MAPCloud: mobile applications on an elastic and scalable 2-tier cloud architecture. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, pages 83–90. IEEE Computer Society, 2012.

[Sat01] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 8(4):10–17, Aug 2001.

[SBCD09] Mahadev 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.

[SF05] Ya-Yunn Su and Jason Flinn. Slingshot: deploying stateful services in wireless hotspots. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, MobiSys '05, pages 79–92, New York, NY, USA, 2005. ACM.

[SPN+13] Cong Shi, Pranesh Pandurangan, Kangqi Ni, Juyuan Yang, Mostafa Ammar, Mayur Naik, and Ellen Zegura. IC-Cloud: Computation offloading to an intermittently-connected cloud. Technical report, Georgia Institute of Technology, 2013.

[SVF08] Joao Nuno Silva, Luis Veiga, and Paulo Ferreira. SPADE: scheduler for parallel and distributed execution from mobile devices. In *Proceedings of the 6th international workshop on Middleware for pervasive and ad-hoc computing*, pages 25–30. ACM, 2008.

[XSP+13] Yu Xiao, Pieter Simoens, Padmanabhan Pillai, Kiryong Ha, and Mahadev Satyanarayanan. Lowering the barriers to large-scale mobile crowdsensing. In *Mobile Computing Systems and Applications*, 2013.

[YCY+13] Lei Yang, Jiannong Cao, Yin Yuan, Tao Li, Andy Han, and Alvin Chan. A framework for partitioning and execution of data stream applications in mobile cloud computing. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):23–32, 2013.

[YMCL12] Ping Yu, Xiaoxing Ma, Jiannong Cao, and Jian Lu. Application mobility in pervasive computing: A survey. *Pervasive and Mobile Computing*, 9:2–17, 2012.

[YOC08] Kun Yang, Shumao Ou, and Hsiao-Hwa Chen. On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications. *Communications Magazine, IEEE*, 46(1):56–63, 2008.

[YQC+12] Fan Yang, Zhengping Qian, Xiuwei Chen, Ivan Beschastnikh, Li Zhuang, Lidong Zhou, and Jacky Shen. Sonora: A platform for continuous mobile-cloud computing. Technical report, Technical Report. Microsoft Research Asia, 2012.

[ZGHC09] Yang Zhang, Xue-tao Guan, Tao Huang, and Xu Cheng. A heterogeneous auto-offloading framework based on web browser for resource-constrained devices. In *Internet and Web Applications and Services, 2009. ICIW'09. Fourth International Conference on*, pages 193–199. IEEE, 2009.

[ZHZ+12] Ying Zhang, Gang Huang, Wei Zhang, Xuanzhe Liu, and Hong Mei. Towards module-based automatic partitioning of java applications. *Frontiers of Computer Science*, 6(6):725–740, 2012.

[ZJGK12] Xinwen Zhang, Won Jeon, Simon Gibbs, and Anugeetha Kunjithapatham. Elastic HTML5: Workload offloading using cloud-based web workers and storages for mobile devices. In *Mobile Computing, Applications, and Services*, pages 373–381. Springer, 2012.

[ZKJG11] Xinwen Zhang, Anugeetha Kunjithapatham, Sangoh Jeong, and Simon Gibbs. Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing. *Mobile Networks and Applications*, 16(3):270–284, 2011.