Future Generation Computer Systems 96 (2019) 119–147

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A tale of three systems: Case studies on the application of architectural tactics for cyber-foraging

Grace Lewis [a], Patricia Lago [a,*], Sebastián Echeverría [b], Pieter Simoens [c]

[a] Vrije Universiteit Amsterdam, The Netherlands
[b] Universidad de los Andes, Chile
[c] Ghent University - imec, Belgium

HIGHLIGHTS

- Cyber-foraging is a technique to extend computing power of mobile devices.
- Two case studies to discover tactics for cyber-foraging in existing systems.
- Third case study uses the tactics in the development of a new system.
- Results are an initial demonstration of the validity of the tactics.
- Potential for taking a tactics-driven for development of cyber-foraging systems.

ARTICLE INFO

Article history:
Received 11 June 2018
Received in revised form 19 November 2018
Accepted 24 January 2019
Available online 6 February 2019

Keywords:
Mobile cloud computing
Cyber-foraging
Computation offload
Data staging
Architectural tactics
Software architecture
Software engineering
Case study

ABSTRACT

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. In previous work, we developed a set of reusable architectural tactics for cyber-foraging systems. We define architectural tactics as design decisions that influence the achievement of a system quality. In this article we present the results of three case studies to validate the application of the tactics to promote their intended functional and non-functional requirements. The first two case studies focus on the identification of architectural tactics in existing cyber-foraging systems. The third case study focuses on the development of a new cyber-foraging system using the architectural tactics. The results of the case studies are an initial demonstration of the validity of the tactics, and the potential for taking a tactics-driven approach to fulfill functional and non-functional requirements for cyber-foraging systems.

1. Introduction

Cyber-foraging is a mechanism that leverages cloud servers, or local servers called surrogates, to augment the computation and storage capabilities of resource-limited mobile devices while extending their battery life [1]. 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 power. The second is data staging to improve data transfers between mobile devices and the cloud by temporarily staging data in transit on intermediate, proximate nodes. While cyber-foraging can take place between mobile devices and cloud resources, our focus is on systems that use intermediate, proximate surrogates.

In previous work we conducted a systematic literature review (SLR) on architectures for cyber-foraging systems [2,3]. The common design decisions present in the cyber-foraging systems identified in the SLR were codified into functional and non-functional architectural tactics [3,4]. We define *architectural tactics* as design decisions that influence the achievement of a system quality (i.e., quality attribute) [5]. However, these tactics needed to be validated in real cyber-foraging systems. We therefore developed case studies for three different cyber-foraging systems to validate the application of the tactics to promote particular functional and non-functional requirements. A summary of the architectural tactics for cyber-foraging is presented in Appendix as a reference. 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 promote certain quality attributes.

The goal of the first case study is to discover the architectural design decisions in the existing implementation of the Tactical

* Corresponding author.
E-mail addresses: g.a.lewis@vu.nl (G. Lewis), p.lago@vu.nl (P. Lago), sebastian.echeverria@gmail.com (S. Echeverría), pieter.simoens@ugent.be (P. Simoens).

https://doi.org/10.1016/j.future.2019.01.052
0167-739X/

Cloudlets system developed by the Carnegie Mellon Software Engineering Institute to support computation offload [6], and then verify the mapping of the architectural design decisions to the architectural tactics for cyber-foraging. Similarly, the goal of second case study is to discover the architectural design decisions in the existing implementation of the GigaSight system developed by Ghent University, Aalto University, Intel Labs, and Carnegie Mellon University to support data staging of crowd-sourced video [7]. Finally, the goal of the third case study is to identify tactics that could be used in the development of the AgroTempus system, targeted at agricultural knowledge exchange in resource-challenged regions, and then to validate if the implementation of each of the tactics met its intended functional and non-functional requirements. Tactical Cloudlets and GigaSight are academic systems developed in the context of cyber-foraging research. As cyber-foraging is still an emerging concept, we were not successful in finding industry systems (or additional academic systems) that were full systems and had source code available for analysis (mostly simulations, algorithms, and experimentation code). AgroTempus was developed as part of a Masters Thesis at the Vrije Universiteit Amsterdam.

The rest of the paper is structured as follows. Section 2 presents the design of the case studies. Sections 3, 4, and 5 show the results of the Tactical Cloudlets, GigaSight, and AgroTempus case studies, respectively. Section 6 presents the threats to validity of the results. Section 7 presents related work. Finally, Section 8 concludes the article and outlines next steps.

2. Case study design

For the three case studies we followed the guidelines for conducting case studies from [8] and [9].

2.1. Research questions

For the first two case studies, given the goal to discover architectural design decisions in an existing system (Tactical Cloudlets and GigaSight), we defined the following research questions to be answered in the execution of the case study.

RQ1: Which of the architectural tactics for cyber-foraging can be identified in the system?
RQ2: How do the implemented tactics support their intended functional and non-functional requirements?

For the third study, given the goal to determine if the identified architectural tactics for cyber-foraging can be used in the development of the AgroTempus system, we defined the following research questions to be answered in the execution of the case study. These questions are slightly different from the previous two case studies as the context is the use of the tactics in new development, as opposed to the analysis of an existing system to identify tactics.

RQ3: Which of the architectural tactics for cyber-foraging can be used in the development of the system to fulfill its functional and non-functional requirements?
RQ4: How do the selected tactics support their intended functional and non-functional requirements?

2.2. Data collection procedure

Data collection involves identifying the data to be collected, defining a data collection plan, and defining how the data will be stored [8]. Given that the goal of the first two case studies is to discover the architectural design decisions in an existing system implementation, and both the system artifacts and system developers are available, the data collection is executed with an independent analysis of work artifacts (third degree data collection method) combined with developer interviews for validation (first degree data collection method) [9]. We therefore define the following steps to collect data about the design and implementation of the Tactical Cloudlets system that will enable us to answer the case study research questions:

1. Understand system requirements: System requirements are gathered from the project Wiki, system documentation, and publications. The identified requirements are documented and confirmed by members of the development team.
2. Recover software architecture: The software architecture is recovered from the project Wiki, system documentation, and publications. The as-designed architecture is compared to the as-is architecture through code inspection of the code available at https://github.com/SEI-AMS/pycloud and verification with the development team.
3. Map architectural design decisions to system requirements: Architectural design decisions are mapped to system requirements in order to fully understand how each requirement was met.

Given that the research questions identified for the GigaSight system are the same as for the Tactical Cloudlets system, the data collection procedure is the same. The main difference between systems is that Tactical Cloudlets is targeted at computation offload while the GigaSight system is targeted at data staging. The code for the GigaSight system that is analyzed as part of the data collection procedure is available at https://github.com/cmusatyalab/GigaSight.

For the AgroTempus system, given that the goal of the case study is to determine if the architectural tactics for cyber-foraging can be used in the development of a new system, the data collection is executed with direct observation of the development process (first degree data collection method) combined with developer and project stakeholder interviews for validation (first degree data collection method) [9]. We therefore define the following steps to collect data about the use of architectural tactics for cyber-foraging in the development of the AgroTempus system that will enable us to answer the case study research questions:

1. Gather system requirements: System requirements are gathered by the system developer from the main project stakeholder. The identified requirements are documented and confirmed by the main stakeholder.
2. Map system requirements to architectural tactics for cyber-foraging: The system developer maps system requirements to functional and non-functional tactics for cyber-foraging that could be used in the realization of the requirements.
3. Develop system architecture: The system developer designs the software architecture for the AgroTempus system using components derived from the identified architectural tactics, combined with components to address requirements that are outside the scope of the architectural tactics for cyber-foraging. The system architecture is documented as a component-and-connector diagram.
4. Map architecture components to system requirements: The system developer maps architecture components to system requirements to ensure that all system requirements are assigned to components of the architecture.
5. Map architecture components to identified architectural tactics: The system developer maps architecture components and design decisions to elements of the identified tactics.
6. Implement system based on system architecture: The system developer implements the system according to specifications provided by the system architecture.

2.3. Analysis procedure

Once the system requirements and architectural design decisions are fully understood we perform two activities as part of the analysis.

1. Map architectural design decisions to architectural tactics: The identified architectural design decisions are mapped to elements of the tactics. We do this by (1) selecting tactics that could meet systems requirements based on the description of the tactic, and (2) mapping components of the tactics to component(s) in the architecture that perform each component role. Both matches and gaps are identified in order to determine completeness of the tactics, as well as variations of the tactics implemented in the system to fulfill specific requirements.
2. Qualitatively and quantitatively (if feasible) determine if the implementation of the tactics meets the corresponding system requirements: Through system testing, data collected (and published) by system developers, as well as discussions with the system developers, we determine if the implementations of the tactics meet their intended requirements.

3. Case study 1: Tactical Cloudlets

3.1. System context

Tactical environments, such as those in which first responders and military personnel operate, are characterized by dynamic context, limited computing resources, disconnected-intermittent-limited (DIL) network connectivity, and high levels of stress. Forward-deployed, discoverable, virtual-machine-based surrogates called tactical cloudlets can be hosted on vehicles or other platforms to (1) provide infrastructure to offload computation, (2) provide forward data staging for a mission, (3) perform data filtering to remove unnecessary data from streams intended for mobile users, and (4) serve as collection points for data heading for enterprise repositories.

The forward-deployed, single-hop proximity to mobile devices promotes energy efficiency as well as lower latency (faster response times). Given the uncertainty and dynamicity of tactical environments, one of the main drivers for the Tactical Cloudlets system is survivability, defined as the capability of a system to continue functioning in spite of adversity [6]. In particular, because a mobile device might lose connectivity to a cloudlet given the highly dynamic environment, a cloudlet should to be able to provide the mobile device access to needed computation and data in the shortest time possible, before the mobile device loses connectivity.

3.2. System requirements

The requirements of the Tactical Cloudlets system can be divided into functional and non-functional requirements.

3.2.1. Functional requirements

- **FR1: Offload of Computation-Intensive Operations:** Applications that are useful to first responders and military personnel include speech and image recognition, natural language processing, and situational awareness. These are all computation-intensive tasks that take a heavy toll on the device's battery power and computing resources and should therefore be offloaded to proximate, more powerful cloudlets.
- **FR2: Cloudlet Discovery:** Due to the dynamic nature and potential mobility of cloudlets in tactical environments (e.g., vehicle-hosted cloudlets), mobile devices need to be able to discover nearby cloudlets.
- **FR3: Disconnected Operations:** In tactical environments it is not possible to guarantee connectivity between cloudlets in the field and the cloud. Therefore, offloaded capabilities should be self-contained and pre-loaded so they do not require connectivity to the cloud in order to operate.
- **FR4: Support for Separate Deployment of Mobile Devices and Cloudlets:** Cloudlets should be able to be used by mobile devices already deployed or available in the field. Therefore the cloudlet should enable mobile devices to be provisioned with the required apps to use its capabilities.
- **FR5: Optimal Cloudlet Selection:** If more than one cloudlet is available, the mobile device should offload computation to the cloudlet that is likely to return a response in the shortest amount of time, before the mobile device loses connectivity to the cloudlet (relationship to the survivability driver).
- **FR6: Cloudlet Management:** In addition to being able to provision the cloudlet with capabilities for use by mobile devices, the cloudlet administrator should be able to see what capabilities have been started from mobile devices as well as start capabilities and stop capabilities as needed.
- **FR7: Cloudlet Migration:** Due to the potential mobility of cloudlets in tactical environments, offloaded capabilities should be able to migrate between cloudlets when requested.

3.2.2. Non-functional requirements

- **NFR1: Energy Efficiency**: Energy consumption on the mobile device when offloading computation-intensive operations (request, execution, and response) should be less than energy consumed by local execution.
- **NFR2: Scalability and Elasticity**: Tactical cloudlets cannot be servers with huge computing power due to power availability and size limitations of what can be carried into a tactical environment to support a mission. Tactical Cloudlets therefore should only run capabilities when they are actively being used by mobile devices.
- **NFR3: Ease of Deployment and Re-Deployment**: First responders and military personnel executing a mission cannot rely on the availability of IT personnel in the field to help with cloudlet setup. Therefore, tactical cloudlets should be easy to set up by non-IT personnel.

3.3. System architecture and design

The Tactical Cloudlets system contains 7.7 KLOC of Java and 4.5 KLOC of Python. It had five non-full-time developers over three years.. The as-is architecture for the system is shown in Fig. 1. The main elements of the architecture are:

- Client: Mobile device running Android 4.x that hosts three main components:
  - Cloudlet-Ready App(s): Mobile apps that are set up to offload computation to a cloudlet.
  - Cloudlet Client GUI: Mobile app that is used to access the app store capability.
  - Cloudlet Client Library: Library that is used by the two components above to discover cloudlets, retrieve cloudlet metadata, select cloudlets, and offload computation. It interacts with the Cloudlet Host using HTTP.
- Cloudlet Host: Linux server that runs a tactical cloudlet. The main components are:
  - PyCloud Library: Python component that implements the core cloudlet functionality.
  - Cloudlet API: Python component that is used by the Cloudlet Client Library to start Services as Service VMs.

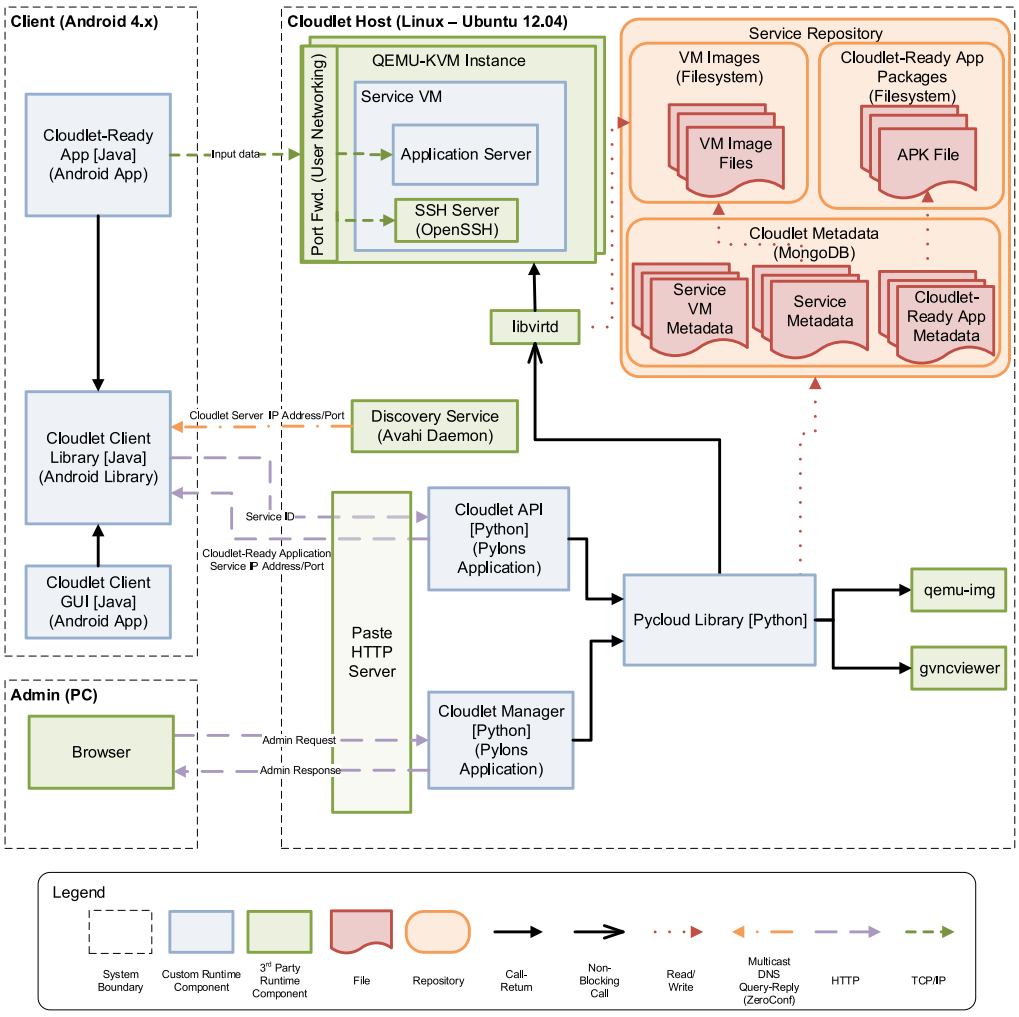


Fig. 1. High-level architecture of the tactical Cloudlets system.

- Cloudlet Manager: Python web application that is used by an administrator to manage Services (along with their VM Images) and Cloudlet-Ready Apps.
- Service Repository: Each capability that is made available to mobile apps is considered a service. A running service is called a Service VM. Each service has associated metadata (Service Metadata), the actual capabilities packaged as VM disk and memory images (VM Images), and one or more Cloudlet-Ready Apps that can use the capability (Cloudlet-Ready App Packages). In addition, the repository stores metadata related to running services (Service VM Metadata) and the available Cloudlet-Ready Apps (Cloudlet-Ready App Metadata)
- QEMU-KVM Instance: Each Service VM runs inside a QEMU-KVM virtual machine instance.

- Admin (PC): Browser that is used to access the Cloudlet Manager web application.

3.4. Mapping of architectural design decisions to architectural tactics

The following subsections describe the tactics that were identified in the Tactical Cloudlets system, how they were implemented, and how they map to system requirements.

3.4.1. Computation Offload

The Computation Offload tactic enables mobile clients to offload expensive computation to surrogates, as shown in Fig. 2(a). This tactic can be identified in the Tactical Cloudlets architecture as shown in Fig. 2(b), with numbers to indicate the sequence of operations. The component names in Fig. 2(a) are used as stereotypes for the components in Fig. 2(b) to indicate the mapping between components. Only the components that are relevant to the tactic are included. This convention is followed for all the implementation diagrams in this article. The computation offload operation takes place as follows:

1–4. The Cloudlet-Ready App requests to offload service *Service ID*.
5. The Pycloud Library retrieves Service Metadata and VM Image Files for *Service ID*.
6. The Pycloud Library starts the Service VM as a QEMU-KVM Instance.
7. The Pycloud Library saves Service VM Metadata in the Service Repository.
8–11. The Pycloud Library returns the IP address and port on which the Service VM is listening.
12. The Cloudlet-Ready App opens a socket to the given IP address and port and starts interacting with the Service VM.

The Computation Offload tactic supports the requirement to offload expensive computation to nearby surrogates (FR1) as well as the energy efficiency requirement (NFR1). The developers of the tactical cloudlets system split applications into a very thin client

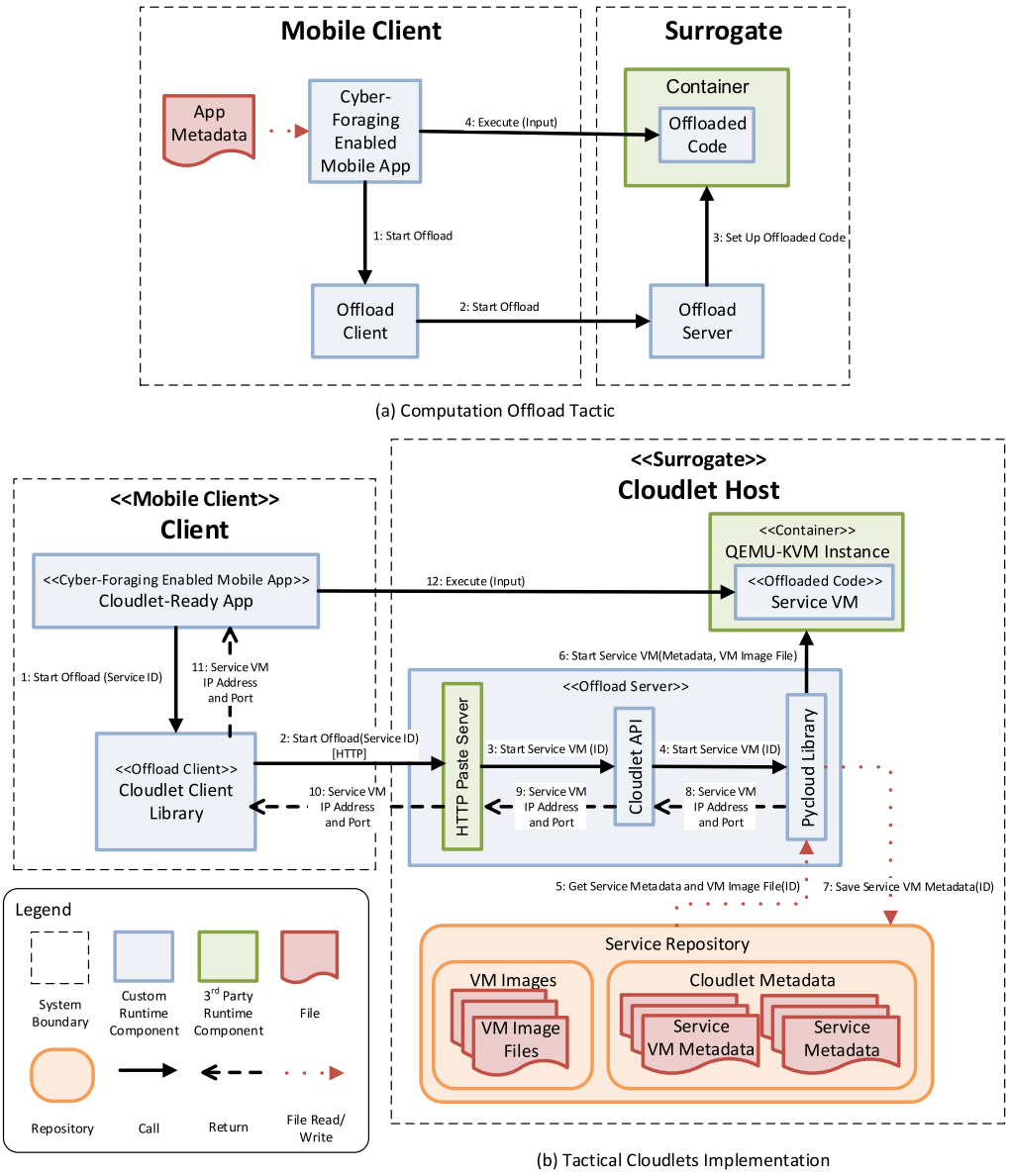


Fig. 2. Tactical Cloudlets implementation of the Computation Offload tactic.

(Cloudlet-Ready App) and a very computation-intensive server (Service VM) such that energy efficiency is reached on the mobile device. The mapping between the tactic and the Tactical Cloudlet implementation in Fig. 2 shows two differences:

1. The Tactical Cloudlets system does not use an external *App Metadata* file in the offload process. This is because the only metadata that is required is the *Service ID* which is hard-coded in the *Cloudlet-Ready App*. An improvement for a future version of the tactic is to mark the *App Metadata* component as optional.
2. The Tactical Cloudlets system has an additional *Service Repository* component from which offloaded code is fetched and then started as a *Service VM*. This additional step would be required of any system that implements the Computation Offload tactic together with the Pre-Provisioned Surrogate tactic, as is the case of the Tactical Cloudlets system (Section 3.4.2). An additional improvement for the catalog would be to include variations of the Computation Offload tactic when used with the different surrogate provisioning tactics.

3.4.2. Pre-Provisioned Surrogate

In the Pre-Provisioned Surrogate tactic surrogates are provisioned before their deployment with the capabilities that are offloaded by mobile clients, as shown in Fig. 3(a). This tactic can be identified in the Tactical Cloudlets architecture as shown in Fig. 3(b). Provisioning a cloudlet with a service capability takes place as follows:

1–3. The Admin Client requests to add a new service *Service ID* to a cloudlet.
4. In order to provide a faster startup time for when service capabilities are requested, the Pycloud Library first starts the Service VM from the given VM Image Disk File.
5. The Pycloud Library then suspends the Service VM, which generates a VM Image Memory File. The faster startup time is because the Service VM will be started from a suspended state instead of a cold state.
6. Both the VM image disk and memory file are saved as VM Image Files in the Service Repository along with Service Metadata.

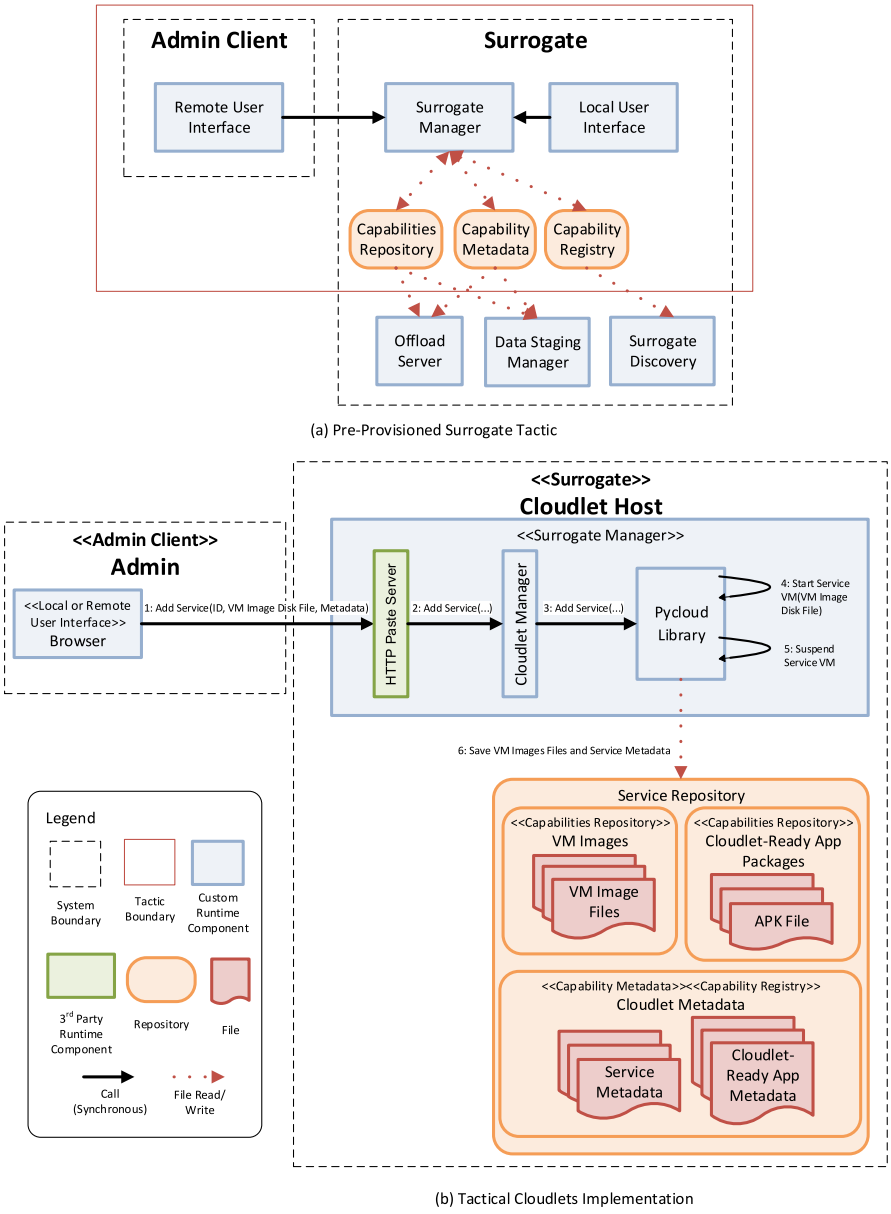


Fig. 3. Tactical Cloudlets implementation of the Pre-Provisioned Surrogate tactic.

This same general process is followed when adding a Cloudlet-Ready App to the Service Repository. Cloudlet-Ready Apps are linked to services by *Service ID*. At runtime, the Cloudlet-Ready App uses the *Service ID* provided in steps 1–3 to start the computation offload process.

The Pre-Provisioned Surrogate tactic supports the requirement for disconnected operations (FR3) because cloudlets are pre-provisioned with capabilities that are needed for a mission. In addition, because cloudlets are also pre-provisioned with the apps to use the capabilities, the tactic also supports the requirement to enable mobile devices to be provisioned in the field (FR4). The mapping between the tactic and the Tactical Cloudlet implementation is complete, as shown in Fig. 3.

3.4.3. Surrogate Broadcast

In the Surrogate Broadcast tactic surrogates advertise their availability and selected metadata to mobile devices for discovery, as shown in Fig. 4(a). This tactic can be identified in the Tactical Cloudlets architecture as shown in Fig. 4(b). Cloudlet discovery is based on the Avahi daemon[1] that implements Zeroconf (Zero Configuration Networking).[2] Avahi uses DNS Service Discovery (DNS-SD) along with Multicast DNS to enable a client to request a service without knowing the IP address of the server that provide the service. Cloudlet discovery by cloudlet-ready apps takes place as follows:

0. When the cloudlet starts, its Discovery Service joins a specific Cloudlet Multicast IP Address as a listener.
1. The Cloudlet-Ready App requests to offload service *Service ID*.
2. The Cloudlet Client Library sends a DNS-SD Query for cloudlet services (defined as a *_cloudlet._tcp* service) through Multicast DNS to the Cloudlet Multicast IP Address. The query reaches the Discovery Service of any cloudlets in the network through Multicast DNS. The Discovery Service

[1] http://avahi.org.
[2] http://zeroconf.org.

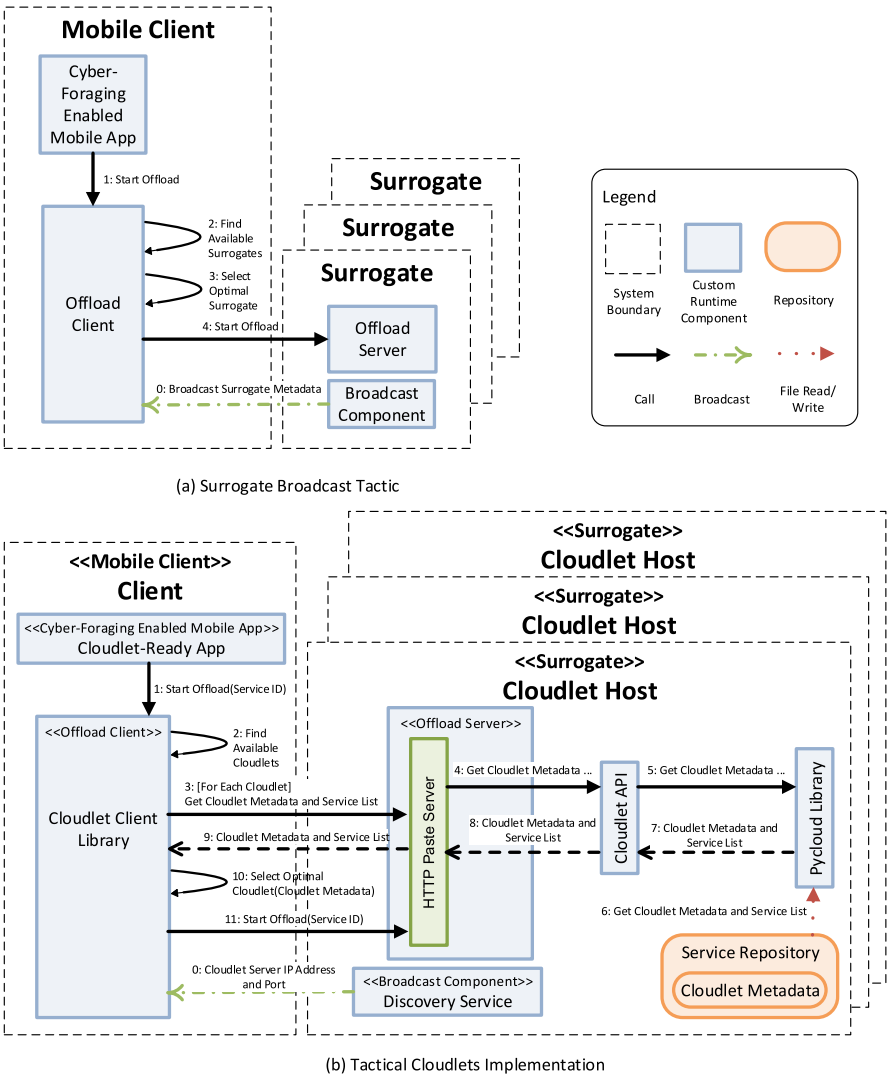


Fig. 4. Tactical Cloudlets implementation of the Surrogate Broadcast tactic.

replies with a DNS-SD Response indicating the IP address and port of the cloudlet server.

3–9. The Cloudlet Client Library sends a request for cloudlet metadata and the list of available services to each cloudlet that replied.

10. The Cloudlet Client Library selects the cloudlet that contains the service *Service ID* and has the lowest load, based on the assumption that it will have the fastest processing and response time. The architecture enables other algorithms to be plugged in.

11. The Cloudlet Client Library starts the computation offload process (Section 3.4.1) with the selected cloudlet.

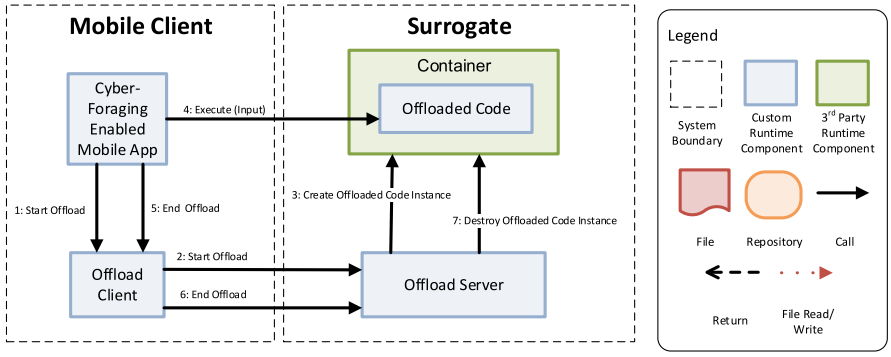
The Surrogate Broadcast tactic supports the requirement for cloudlet discovery (FR2) as well as the requirement for optimal cloudlet selection when more than one cloudlet is available (FR5). The mapping between the tactic and the Tactical Cloudlet implementation in Fig. 4 shows two differences:

1. The cloudlet selection process is a two-step process in which the *Cloudlet Server IP Address and Port* broadcast by the *Broadcast Component* (Step 0) is used to query each cloudlet for capabilities (Step 3). The reason for this is that the Zero-conf protocol used by the Tactical Cloudlets implementation has a size limitation for broadcast information. While not a gap in the tactic itself, what this shows is that technology selection can introduce variations in the implementation of a tactic. An improvement for the catalog would be to include variations of the Surrogate Broadcast tactic when used with different technologies (or known limitations of technologies).
2. For this same reason, the *Surrogate Repository* is added to the implementation of the tactic. The cloudlet metadata and service list is obtained from the repository when the cloudlet is queried for its capabilities. This component would be part of the variation introduced by the broadcast protocol size limitation.

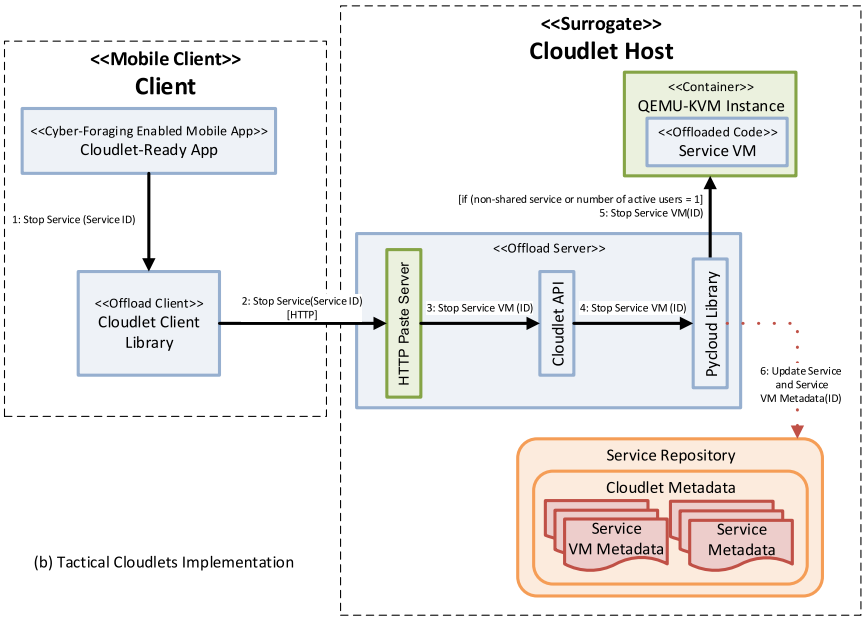
3.4.4. *Just-In-Time Containers*

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, as shown in Fig. 5(a). In the Tactical Cloudlets system, as shown in Fig. 2(b), the computation offload process presented in Section 3.4.1, a QEMU-KVM Instance for a Service VM is only created upon an offload request.

In addition, to promote greater scalability and elasticity, when adding a service to a cloudlet (Section 3.4.2), one of the elements of the Service Metadata is whether the service will be shared or non-shared. A non-shared service will start a separate instance with



(a) Just-in-Time Containers Tactic



(b) Tactical Cloudlets Implementation

Fig. 5. Tactical Cloudlets implementation of the Just-in-Time Containers tactic.

every request. However, a shared service will only start an instance for its first request. All other requests will share the same instance. A counter of active users for the service is maintained as Service Metadata. This means that Step 6 in Fig. 2 only takes place if the service is non-shared, or if it is the first request for a shared service.

The final step in the computation offload process presented in Section 3.4.1 is that the Cloudlet-Ready App starts the interaction with the Service VM. To implement the Just-in-Time Containers tactic, when the Cloudlet-Ready App is closed, the operations shown in Fig. 5(b) take place, namely:

1–4. The Cloudlet-Ready App requests to stop service *Service ID*.
5. If the service is non-shared or the number of active users for the service is one (i.e., last active user), the Pycloud Library stops the instance of the service *Service ID*.
6. Service Metadata and Service VM Metadata are updated to indicate that the service has stopped and/or the number of active users for the shared service is one less.

The Just-in-Time Containers tactic supports the requirement for capabilities to only be running when they are being used in order to promote scalability and elasticity (NFR2). The mapping between the tactic and the Tactical Cloudlet implementation in Fig. 5 shows two differences:

1. The Tactical Cloudlets system introduces the concept of shared and non-shared capabilities, which is not specified in the original tactic. This is why the container is destroyed only if is it is a non-shared capability or the number of active users is one (i.e., only active user of the capability). An improvement for the catalog would be to include a variation of the Just-in-Time Containers tactic to support shared and non-shared capabilities.
2. For the same reason, the *Surrogate Repository* is added to the implementation of the tactic. *Service Metadata* and *Service VM Metadata* needs to be updated based on the results of the request to end the offload request. This component would be part of the variation introduced by the support for shared/non-shared capabilities.

Although not stated as a benefit of the tactic, and not stated as a requirement for the system in Section 3.2, the Just-in-Time Containers tactic also supports energy efficiency on the cloudlet, which is critical in tactical environments where access to power might not always be available.

3.5. Analysis

3.5.1. Mapping between tactics and requirements

The mapping between the identified tactics and the Tactical Cloudlets functional and non-functional requirements is shown in Table 1.

The requirement to provide a form of management console for a cloudlet admin to use (FR6) does not map to any of the tactics,

Table 1
Mapping of architectural tactics to functional and non-functional requirements.

Tactic	FR1: Offload	FR2: Discovery	FR3: Disconnected Ops.	FR4: Separate Deployment	FR5: Selection	FR6: Management	FR7: Migration	NFR1: Energy Efficiency	NFR2: Scalability	NFR3: Ease of Deployment
Computation Offload	X							X		
Pre-Provisioned Surrogate			X	X						
Surrogate Broadcast		X			X					
Just-in-Time Containers								X	X	

as shown in Table 1. This fact is expected as it relates to one of the findings from the SLR that states a lack of focus on system-level concerns that is required when moving from experimental prototypes to operational systems. One of these concerns is management of deployed capabilities. Related to this fact, there is not a tactic in the catalog that maps to ease of deployment and redeployment (NFR3). However, in the Tactical Cloudlets system, the Admin component that implements the Admin Client in the Pre-Provisioned Surrogates tactic (Fig. 3(b)) is a lightweight, web-based interface that enables cloudlet management and easy deployment and redeployment of capabilities (FR6 and NFR3). The extension of the catalog with tactics for ease of deployment and management would be useful for moving from experimental prototypes to operational cyber-foraging systems.

The requirement to be able to migrate capabilities between cloudlets when requested (FR7) does not map directly to any of the tactics either. However, the functionality in the Pycloud Library that enables this migration is very similar to that explained in the Eager Migration tactic once the monitoring component detects that the connection between the mobile device and the cloudlet is deteriorating. The Admin component of the tactical cloudlets system that implements the Admin Client in the Pre-Provisioned Surrogates tactic (Fig. 3(b)) also contains functionality to manually migrate a Service VM Instance to another connected cloudlet. An improvement for the catalog would be to include a variation of the Eager Migration tactic to support manual migration.

To determine if the tactics meet their intended functional and non-functional requirements, the developers conducted extensive system testing and collected data to support their design and implementation decisions. In addition to successful test results, data collected included cloudlet provisioning time, energy consumption on the mobile device, payload size and response time. All implementation details and supporting data are available in several publications [2,6,10,11].

3.5.2. Findings

The analysis of the Tactical Cloudlets system identified four architectural tactics for cyber-foraging. There were some gaps in the identified tactics (Section 3.5.1) that create opportunities for improvement of the tactics catalog:

1. Tactics should differentiate between core and optional components and interactions. Each optional component/interaction should contain rationale for when it is necessary to include in the implementation of the tactic.
2. As tactics are implemented in operational cyber-foraging systems it is likely that variations will arise. The Tactical Cloudlets system introduced several potential tactic variations: (1) variations of the Computation Offload tactic based on the surrogate provisioning tactic selected for the system, (2) a variation of the Just-in-Time Containers tactic to support shared and non-shared capabilities, and (3) a variation of the Eager Migration tactic to support manual migration.
3. Technology selection can also lead to tactic variations. As tactics are implemented and evaluated in cyber-foraging systems, technology limitations and constraints may require the implementation of additional components or interactions between components. The Tactical Cloudlets system introduced a variation of the Surrogate Broadcast tactic due to limitations in broadcast message size.
4. There is great potential for extending the catalog with tactics to support system qualities necessary for moving from experimental prototypes to operational systems. The Tactical Cloudlets system showed the need for tactics to support Ease of Deployment and Manageability.
5. Even if tactics are targeted at promoting a particular system quality, the tactics may have an effect on other system qualities. As an example, the Just-in-Time Containers tactic is a tactic for scalability/elasticity but also promotes energy efficiency on the surrogate. Even though the secondary effect of the tactic is positive, it could also have been a negative effect.
6. Related to the previous point, energy efficiency in cyber-foraging systems is mainly targeted at energy savings on mobile devices because of battery limitations. However, the Tactical Cloudlets system showed the need for tactics to support energy efficiency on surrogates, especially if deployed in areas with power limitations.

The utility of the tactics was supported by the main developer for the Tactical Cloudlets system in the following statement: "Having a set of architectural tactics for cyber-foraging systems would help considerably when starting the design of a new system. Cyber-foraging software has very particular requirements, and it is not easy to know how to create the architecture for the overall system to properly satisfy the appropriate quality attributes. A set of tactics would be an invaluable guide to make decisions at this stage".

4. Case study 2: GigaSight

4.1. System context

GigaSight is a cyber-foraging system targeted at continuous collection of crowd-sourced video from mobile devices and wearables [7]. Given the potentially-sensitive nature of video, GigaSight collects video on surrogates called cloudlets where privacy-sensitive information is automatically removed from the video based on user-defined privacy settings related to time, location, and content — a process called denaturing. Denatured video is then indexed and resulting tags and metadata are uploaded to a cloud catalog where users can perform content-based searches on the total catalog of denatured videos.

Use cases for crowd-sourced video systems such as GigaSight include marketing and advertising; location of missing people, pets and things; creation of family vacation albums; public safety; and fraud detection [12].

4.2. System requirements

The requirements of the GigaSight system can be divided into functional and non-functional requirements.

4.2.1. Functional requirements

- **FR1: Video capture**: The mobile device captures and stores video.

- **FR2: User-specified privacy settings**: Users are able to specify privacy settings based on location, time, and image content. These settings are used by the denaturing process to automatically remove privacy-sensitive content from videos.
- **FR3: Video upload to cloudlets**: When a cloudlet becomes available, the mobile device uploads captured video and privacy settings.
- **FR4: Offload of video denaturing and indexing processes**: The highly computation-intensive denaturing and indexing operations are executed on the cloudlet according to user-specified privacy settings.
- **FR5: Index upload to cloud catalog**: Video metadata and tags generated by the indexing process are uploaded from the cloudlet to a cloud catalog that can be queried by users.
- **FR6: User requests for denatured videos**: A user of the cloud catalog can request denatured videos from cloudlets.

4.2.2. Non-functional requirements

- **NFR1: Energy Efficiency**: Energy consumption on the mobile device when offloading the computation-intensive denaturing and indexing operations should be less than energy consumed by executing them locally.
- **NFR2: Scalability**: One cloudlet should be able to process and store video from multiple users.
- **NFR3: Fault Tolerance**: If a cloudlet is not available for upload, the mobile device should be able to cache video until a cloudlet becomes available.
- **NFR4: Privacy**: Privacy-sensitive information should not be made available to users of the cloud catalog.

4.3. System architecture and design

The GigsSight system contains 3.3 KLOC of Python and 8.4 KLOC of Java. It had four non-full-time developers over one year.The as-is architecture for the system is shown in Fig. 6. The main elements of the architecture are:

- Mobile Device: The mobile device is an Android 4.0.4 device. It leverages the device's built-in camera for video capture.
  - GigaSight App: Performs all the user and privacy setting management. User settings include IP address and port of its Personal VM. Privacy settings include time filters, location filters and object-based filters. The object-based filters are currently limited to the faces present in the training set of the face recognition algorithms.
  - File Uploader: Connects to the user's Personal VM and uploads video files and metadata. Once files are successfully uploaded, these are removed from the mobile device to make space for more video content.
- Cloudlet: Cloudlets are data staging points for denatured video data en route to the cloud. Cloudlets in GigaSight are implemented as servers running Linux 3.2.0.
  - Personal VM: Each mobile device user is associated to a Personal VM that performs the customized denaturing for that user according to the user-defined privacy settings. The Denaturing Process that executes inside this VM is implemented using C++ and OpenCV 2.4.2[3] as a multi-step pipeline: video decoding, early-discard of frames based on metadata and sampling rate, content-based blurring, and video encoding. The output of the denaturing process is a low-frame-rate denatured video file. For additional privacy, an encrypted version of the original video is also created during the upload process. Both files are stored in the Data Management VM so that they are accessible to other VMs on the cloudlet.
  - Data Management VM: The Data Manager inside this VM handles all video and metadata storage and retrieval in the Storage and Metadata Database. It notifies the Indexer when new denatured video is available for indexing. In addition, each time the Indexer adds tags to the database, these are automatically synchronized with the Global Catalog running in the Cloud.
  - Video Content Indexer VM: The Indexer inside this VM is a background activity that extracts metadata about denatured videos (e.g., owner (anonymized), location of capture, start and end time of capture, cloudlet address where stored, and tags) and sends it to the Data Manager which in turn pushes this information to the Global Catalog in the Cloud. The metadata is also stored locally for use by search algorithms that could be implemented inside the Personal VM for personal use.
  - Diamond Search Module: The Diamond Search Module is a third-party component for interactive search of non-indexed data.[4]
- Cloud: Cloud-based data center that aggregates video metadata from a set of associated cloudlets.
  - Global Catalog: The Global Catalog is a web application implemented using Django[5] that stores and manages the metadata from denatured videos available on cloudlets. The front end to the application enables users to browse through the metadata and select videos of interest for viewing.
  - Diamond Client: Once a user selects videos of interest, the Diamond Client contacts the Diamond Server of each cloudlet that contains a video of interest to initiate content-based search.

4.4. Mapping of architectural design decisions to architectural tactics

The following subsections describe the tactics that were identified in the GigaSight system, how they were implemented, and how they map to system requirements. Implementation diagrams for tactics that have already been presented will not be included due to space limitations. The full set of implementation diagrams can be found in [13].

4.4.1. Out-Bound Pre-Processing

In the Out-Bound Pre-Processing tactic surrogates collect data from mobile devices and pre-process the data – clean, filter, summarize, or merge – such that the data that is sent on to the enterprise cloud is ready for consumption and serves an immediate need, as shown in Fig. 7(a). This tactic can be identified in the GigaSight architecture as shown in Fig. 7(b). The out-bound pre-processing takes place as follows:

1–3. *GigaSight App* uploads stored video and metadata to the *Personal VM* identified by *Personal VM IP Address and Port* using the *File Uploader*.
4. The *GigaSight Server* receives the video, metadata and privacy settings for the user and sends these to the *Denaturing Process* for denaturing according to the user's privacy settings.
5–6. The *GigaSight Server* encrypts the original video and sends it to the *Data Manager* for storage.

[3] http://www.opencv.org.

[4] http://diamond.cs.cmu.edu.

[5] http://www.djangoproject.com.

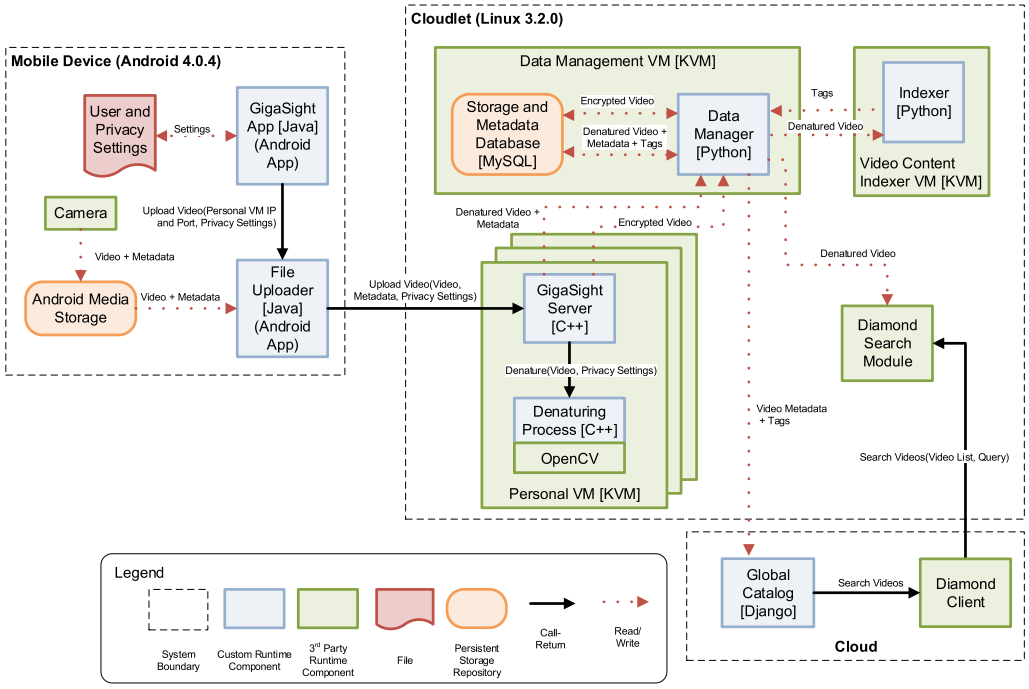


Fig. 6. High-level architecture of the GigaSight system.

7. The *GigaSight Server* sends the denatured video and metadata to the *Data Manager* for storage and indexing.
8–9. The *Data Manager* sends the denatured video to the *Indexer* for indexing, which returns the set of tags for elements identified in the video.
10. The *Data Manager* stores the denatured video, metadata and tags.
11. The *Data Manager* sends the video metadata and tags to the *Global Catalog* in the *Cloud*.

The Out-Bound Pre-Processing tactic supports all of the functional requirements because it maps well to sensing applications such as GigaSight. Because denaturing and indexing are extremely computation-intensive activities that are executed on the cloudlet and not on the mobile device (as demonstrated via experimentation in [7]), the tactic also supports energy efficiency (NFR1). Finally, the pre-processing that occurs on the cloudlet in the Denaturing Process, supports the privacy requirement (NFR4). Because the Personal VM is assigned to one and only one mobile device, there is a guarantee that the raw video is only processed by the Personal VM. Because the video is encrypted before it is stored in the Data Management VM, access to the raw video would only be possible via the Personal VM which is the only system component that knows the encryption key. The mapping between the tactic and the GigaSight implementation in Fig. 7 shows three main differences:

1. The GigaSight system has an additional *User and Privacy Settings* file that is read by the *GigaSight App* to obtain settings for uploading video to the cloudlet. This is reasonable and equivalent to the *App Metadata* component in the Computation Offload tactic, which is where the settings for the offload process are stored. An improvement for a future version of the Out-Bound Pre-Processing tactic is to include a more general *Settings* component that performs this role and mark it as optional.
2. The GigaSight system has an additional *Android Media Storage* component because video and metadata sent to the cloudlet are read from internal storage. This component makes sense for a system that stores data before sending it to the surrogate, as opposed to sending data as it is received. An improvement for a future version of the tactic is to include a more general *Data Storage* component that performs this role and mark it as optional.
3. The GigaSight system has an instance of the *GigaSight Server (Communications Manager)* for each user, as opposed to a single instance. This is done to support the privacy requirement and will be discussed shortly when the mapping to the Pre-Provisioned Surrogate tactic is analyzed.

4.4.2. Pre-Provisioned Surrogate

In the Pre-Provisioned Surrogate tactic surrogates are provisioned before their deployment with the capabilities that are offloaded by mobile clients, as previously shown in Fig. 3(a).

In the GigaSight system all data processing capabilities are provisioned on the cloudlet before deployment. However, this is a manual process. There is not the equivalent of a Surrogate Manager component to help with the provisioning process as shown in the tactic. In addition, because capabilities are not advertised, but rather each mobile device stores the IP Address and Port of its Personal VM as part of the User Settings, there is not the equivalent of a Capability Metadata component, nor the equivalent of a Capability Registry component. Prior to deployment the Terminal program that comes with the Linux distribution is executed locally or remotely to copy the Data Management VM Image File, the Video Content Indexer VM Image File, and the Personal VM Image File that contains the denaturing capabilities into the Linux Filesystem. The KVM Manager program that also comes with the Linux distribution is initially used to start one instance of the Data Management VM and one or more instances of the Video Content Indexer VM.[6] A Personal VM is then started for each mobile device that wants to use the GigaSight system for video offload. After starting the Personal VM the mobile device user is provided with its IP Address and Port, which needs to be added to the User Settings using the GigaSight app shown in Fig. 6.

[6] Deployment of more than one Video Context Indexer VM is an architectural design decision for scalability, as discussed in Section 4.5.

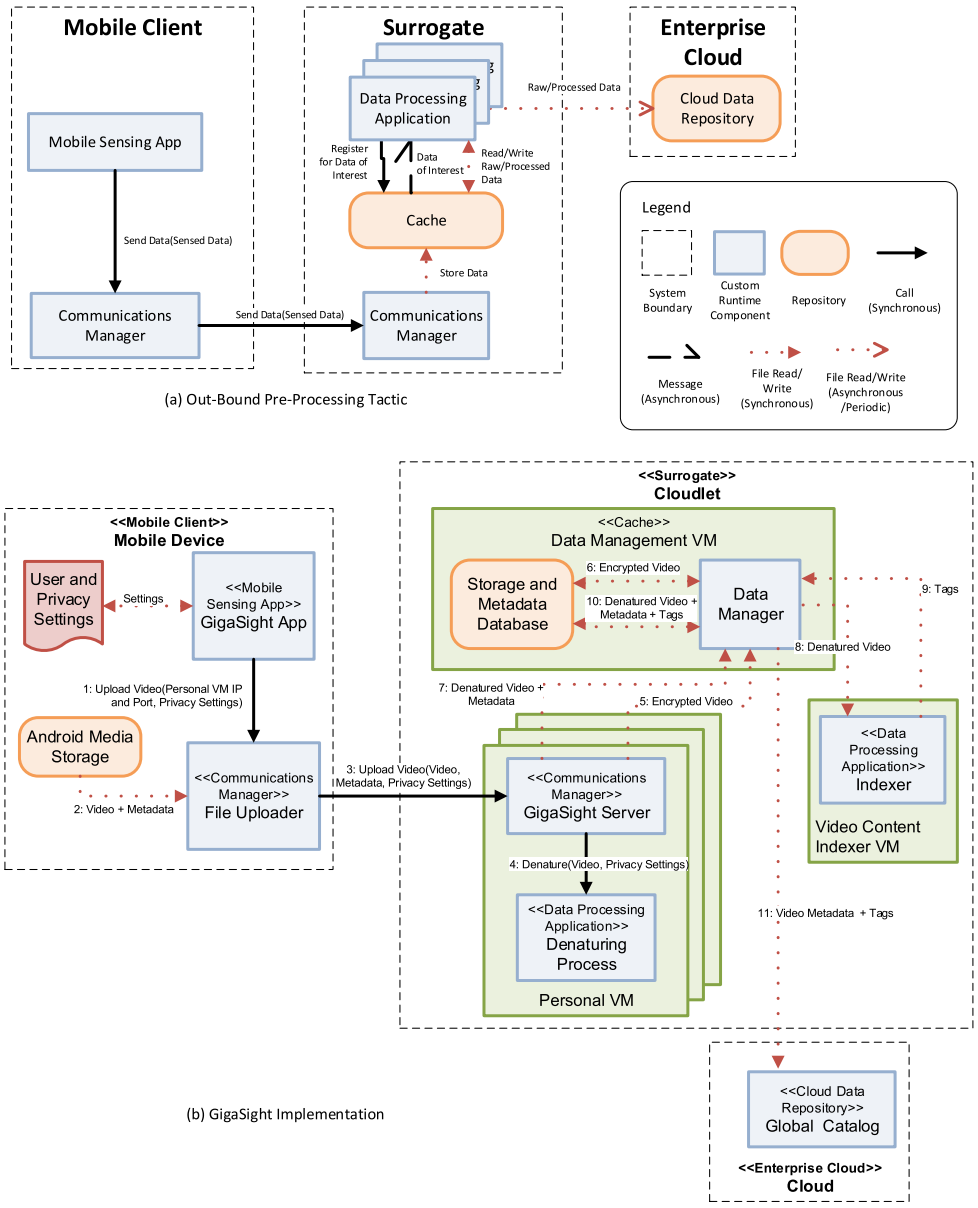


Fig. 7. GigaSight implementation of the Out-Bound Pre-Processing tactic.

The Pre-Provisioned Surrogate tactic supports computation offload to the cloudlet (FR4) and video index upload to the cloud (FR5). These are capabilities that are pre-provisioned on the cloudlet in the form of VMs. Because the capabilities already exist on the cloudlet, there is no need to transfer any extra computation from the mobile device or the cloud, leading to energy efficiency (NFR1). The mapping between the tactic and the GigaSight implementation shows two main differences:

1. The GigaSight system does not have the equivalent of the *Surrogate Manager*. Adding this component to the system would promote ease of deployment and manageability as GigaSight moves from a prototype to an operational system.
2. The GigaSight system does not have the equivalent of the *Capabilities Metadata* and *Capability Registry* components because (1) capabilities are not advertised and (2) capabilities on all surrogates are the same. Therefore, an improvement for a future version of the tactic would be to mark these two components as optional.

Even though having a pre-provisioned surrogate by itself does not support scalability (NFR2), in this particular system it does. Mobile devices are assigned a specific Personal VM on a cloudlet (by IP address and port) and therefore the number of mobile devices supported by a cloudlet can be controlled. Once a defined disk and memory threshold on a cloudlet has been reached, new mobile devices would need to be assigned to a different cloudlet. In essence, a pre-provisioned surrogate has more control over its load.

4.4.3. Local Surrogate Directory

In the Local Surrogate Directory tactic, mobile devices maintain a list of 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, as shown in Fig. 8(a).

The process is much simpler in the GigaSight system, as shown in Fig. 8(b). There is not a cloudlet selection process because every mobile device is assigned a Personal VM on a single Cloudlet. The location of a cloudlet for data upload takes place as follows:

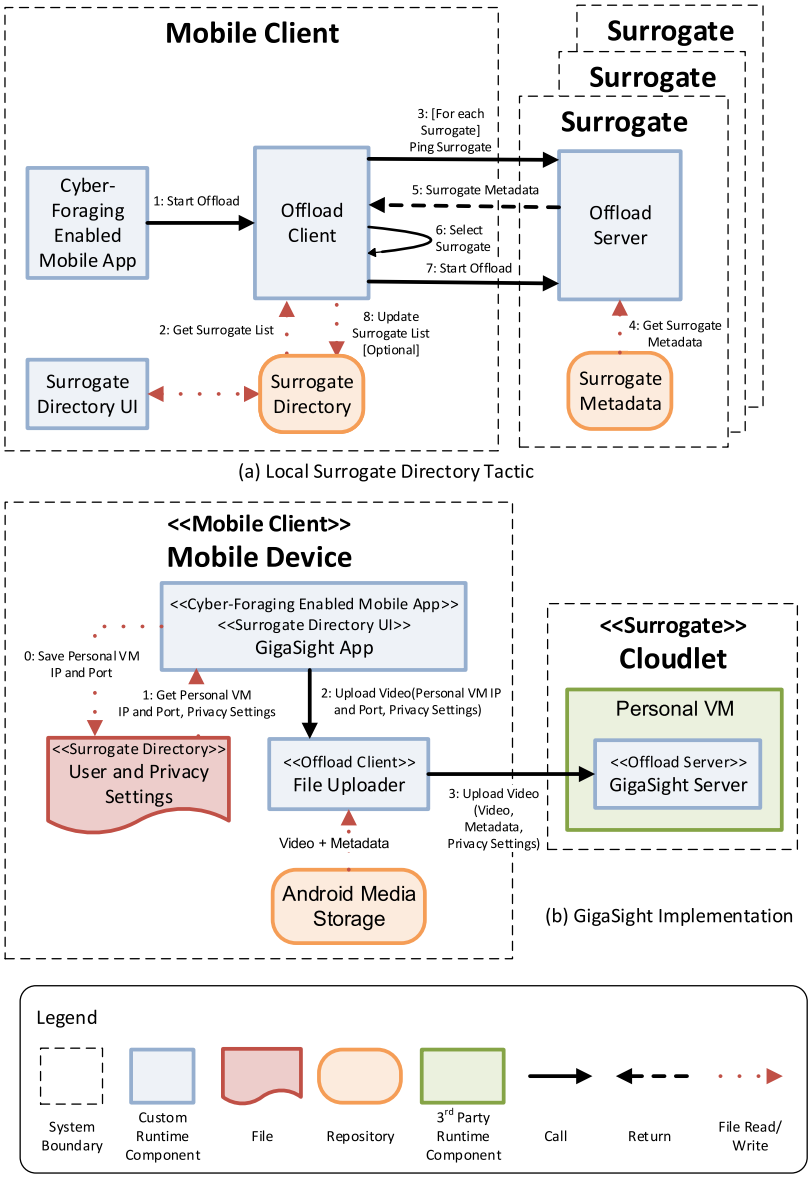


Fig. 8. GigaSight implementation of the Local Surrogate Directory tactic.

0. As indicated in the previous section, when a *Personal VM* is started for a *Mobile Device*, the *GigaSight App* in its role as *Surrogate Directory UI* is used to save the Personal VM IP Address and Port to the *User and Privacy Settings* file.
1. When the video upload process is started by the *GigaSight App* in its role of *Offload Client*, it reads the Personal VM IP Address and Port from the *User and Privacy Settings* file.

2–3. Video and Metadata are uploaded to the *Personal VM* at the provided IP Address and Port.

The **Local Surrogate Directory** tactic supports scalability as defined for the system (NFR2) because a cloudlet can support multiple users by instantiating multiple instances of a Personal VM. However, it is important to note that each cloudlet has an upper bound on the number of Personal VMs that it can run simultaneously. It also supports privacy (NFR4) because the Personal VM is the cloud-based counterpart of the mobile device: an entity that the user trusts to store personal content, but with much more computational and storage resources [7]. The mapping between the tactic and the GigaSight implementation in Fig. 8 shows two main differences:

1. The GigaSight system has an additional *Android Media Storage* component because video and metadata sent to the cloudlet are read from internal storage. This is not a gap in this particular tactic but an area for improvement for the Out-Bound Pre-Processing tactic, as discussed earlier.
2. The GigaSight system does not have a *Surrogate Metadata* component because there is not a cloudlet selection process. An improvement for a future version of the tactic would be to mark this component as optional, as well as the surrogate selection process (Steps 3–6 in Fig. 8(a)).

4.4.4. Client-Side Data Caching

The Client-Side Data Caching tactic is a variation of the Cached Results tactic. Data collected by a mobile client is cached on the mobile device and sent to the surrogate upon connection or reconnection, as shown in Fig. 9(a).

The Client-Side Data Caching tactic can be identified in the GigaSight architecture as shown in Fig. 9(b). The only difference between the implementation and the tactic is that the sensed data (video + metadata) is saved in the cache upon capture, instead of

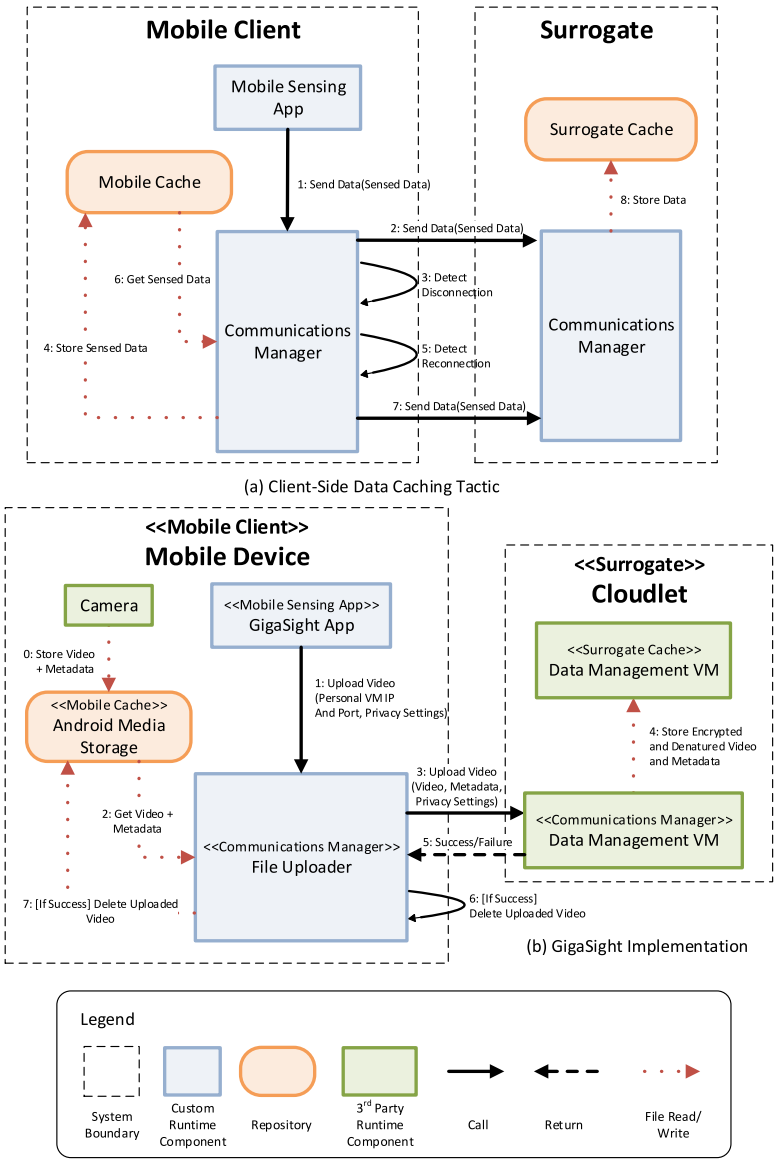


Fig. 9. GigaSight implementation of the Client-Side Data Caching tactic.

upon disconnection. The client-side data caching takes place as follows:

0. Video captured using the *Camera* on the *Mobile Device* is stored in the *Android Media Storage* along with metadata such as location.

1–4. The *GigaSight App* tries to upload video to its *Personal VM* to be encrypted, denatured, and stored in the *Data Management VM* on the *Cloudlet*.

5–7. Only if the operation is successful, the just uploaded video is deleted from the *Android Media Storage* to make room for new video. If it is not successful the user gets an error message and is asked to try the upload at a later time.

The **Client-Side Data Caching** tactic supports video capture and upload to a cloudlet (FR1 and FR2). It supports energy efficiency (NFR1) because uploading longer segments (instead of uploading as video is captured) requires the device to wake up less frequently from the sleep state, while the total number of bytes transmitted remains constant [7]. Finally, it supports fault tolerance (NFR3) because video is not uploaded until a cloudlet is available, and is not deleted from the device until the cloudlet confirms the upload.

The mapping between the tactic and the GigaSight implementation in Fig. 9 shows two differences:

1. The GigaSight system contains a *Camera* component as the data source. An improvement for a future version of the tactic would be to include a more general *Data Source* core component to indicate the source of the data that is stored in the *Mobile Cache*.
2. Sensed data is saved in the cache upon capture, instead of upon disconnection. This difference could be added as a variation of the Client-Side Data Caching tactic.

4.5. Analysis

4.5.1. Mapping between tactics and requirements

The mapping between the identified tactics and the GigaSight functional and non-functional requirements is shown in Table 2.

It is important to note that some non-functional requirements in GigaSight are supported by specific technology selection as opposed to the use of tactics. The use of VMs as containers for data and computation on the cloudlet promotes scalability and elasticity because of the ease for container creation, migration,

Table 2
Mapping of functional and non-functional requirements to the architecture of the GigaSight system.

Tactic	FR1: Video capture	FR2: Privacy settings	FR3: Video upload to cloudlet	FR4: Computation offload to cloudlet	FR5: Video index upload to cloud	FR6: User requests	NFR1: Energy Efficiency	NFR2: Scalability	NFR3: Fault Tolerance	NFR4: Privacy
Out-Bound Pre-Processing	X	X	X	X	X	X	X			X
Pre-Provisioned Surrogate				X	X		X	X		X
Local Surrogate Directory								X		X
Client Side Data Caching	X		X				X		X	

and destruction provided by VM management tools. For example, additional instances of the content indexer can be instantiated on one or more cloudlets to handle increasing loads. A Personal VM can also be easily moved to another cloudlet as long as the device is informed of its new address. The use of VMs as containers also promotes privacy in the system because Personal VMs are single-user and VM isolation is a well-known property of VMs. Although there are potential vulnerabilities and attacks, for the most part this property can be guaranteed [14].

To determine if the tactics meet their intended functional and non-functional requirements, the developers conducted extensive system testing and collected data to support their design and implementation decisions. In addition to successful test results, data collected included system throughput, cloudlet performance, algorithm accuracy, and energy consumption on the mobile device. All implementation details and supporting data are available in several publications [7,12].

4.5.2. Findings

The analysis of the GigaSight system identified four architectural tactics for cyber-foraging. Similar to the Tactical Cloudlets analysis, there were some gaps in the identified tactics (Section 4.5.1) that create opportunities for improvement of the tactics catalog:

1. Consistent with the Tactical Cloudlets system (Section 3.5.2), tactics should differentiate between core and optional components and interactions. Each optional component/interaction should contain rationale for when it is necessary to include it in the implementation of the tactic.
2. Consistent with the Tactical Cloudlets system, even if tactics are targeted at promoting a particular system quality, the tactics may have an effect on other system qualities.
3. Consistent with the Tactical Cloudlets system, as tactics are implemented in operational cyber-foraging systems it is likely that variations will arise. The GigaSight system introduced a potential tactic variation of the Client-Side Data Caching tactic that always caches data, as opposed to only caching data when a surrogate is not found.
4. Functional and non-functional requirements in cyber-foraging systems can also be met by technology selection, rather than by the use of a particular tactic. In the GigaSight system, the use of VMs as containers had a positive effect on scalability/elasticity as well as privacy. Insights that are gained from the implementation and evaluation of cyber-foraging systems could be added as notes to the tactics to provide even greater value to software architects.

The utility of the tactics was supported by the main developer for the GigaSight system in the following statement: "It is helpful for developers to have some 'best practices' in software architecture for cyber foraging. Today, we already have many patterns (e.g., Gang of Four [15]), but these are very focused on object-orientation, rather than on taking into account the actual deployment. Having a reference list of tactics, plus possibly coding elements in the future, would, in my view, be very helpful in designing production-grade cyber-foraging applications. So far, cyber-foraging has not truly left the lab prototype phase and typically good software design practices are second hand during this phase of the research. But with cloudlets, micro data centers, and edge clouds appearing everywhere, there will emerge a need from industry on this".

5. Case study 3: AgroTempus

5.1. System context

As many developing areas have to deal with the lack of proper access to resources such as Internet and electricity, cyber-foraging offers potential solutions to these resource challenges by leveraging proximate surrogates that can provide services that involve heavy computation such as image processing, store large sets of data collected in the field, or store information retrieved from data centers during scarce moments of Internet connectivity.

The goal of the AgroTempus system is to enable people involved in agriculture (e.g., farmers and non-governmental organization (NGO) employees helping farmers), who work in environments with little to no access to the Internet or electricity, to collect and retrieve data about the weather in their area. AgroTempus performs different types of computation on the collected data as examples of valuable services for its users.

End users interact with the system with smartphones, the proliferation of which is predicted to rise significantly in the coming years in developing regions [16,17]. The capabilities of the mobile applications running on the smartphone are extended by surrogates in the form of single-board computers running on solar power. To be able to eventually store all collected data in a cloud-based back-end, a mobile hub carrying a computer system with increased storage capabilities will connect to each surrogate periodically, and eventually connect to the Internet. This also makes it possible to propagate data from the Internet to the surrogates and mobile devices. This setup was inspired by the DakNet project in India [18].

5.2. System requirements

5.2.1. Functional requirements

- **FR1: Store weather data**: NGO employees and farmers can store weather data related to a certain area via a mobile app.
- **FR2: Retrieve weather data**: NGO employees and farmers can retrieve weather data related to a certain area using a mobile app. This data is derived from earlier reports (FR1), as well as from a third-party weather API accessible via the Internet.
- **FR3: Perform regressions on weather data**: NGO employees can select a weather information data set and perform a regression on it using the mobile app. A visualization of the results will be available when the operation completes.
- **FR4: Predict future weather data values**: NGO employees and farmers can obtain predictions of future values of variables related to the weather, based on data collected in the field, up to a week in the future.

- **FR5: Surrogate setup**: Surrogates are assigned to serve a certain region and as such need a setup procedure that enables NGO employees to enter the correct settings before it can be used.
- **FR6: Forecast delivery**: Weather forecasts for the region that the user is in can be retrieved using a mobile app.
- **FR7: Integration with cloud-based storage systems**: The system eventually stores all data collected from mobile devices in a cloud-based system such as ERS [19].
- **FR8: Voice interface**: The user interface for the farmers can support voice instructions to help users navigate the app.
- **FR9: Synchronize weather data**: Periodically, the latest weather forecasts and data for relevant regions are retrieved from a third-party weather API on the Internet. This data is eventually stored on the surrogates.
- **FR10: Surrogate registration on mobile hub**: When new surrogates are added to the system and are operational, their identification and location information (as provided in FR5) is stored on the mobile hub so that it can collect relevant data for this surrogate (FR9).

5.2.2. Non-functional requirements

- **NFR1: Fault tolerance and reliability**: The system should be able to recover from failures such as crashes and loss of connection between mobile devices and surrogates.
  - Because it is expected that there will be few people proficient in IT in the regions where the system will be used, surrogates should be able to detect failures in the services that they offer and restart them accordingly.
  - Losing connection during the interaction between surrogates and mobile hubs, as well as between surrogates and mobile devices, should not cause the services running on the surrogates to stop functioning.
  - Because it is expected that mobile app users will regularly be moving in and out of range of surrogates during use of the system, this should not cause users to lose results of completed computations or lose data that they have stored on the mobile app.
- **NFR2: Ease of deployment**: The system should be easy to deploy.
  - The mobile app can be installed through an app store and does not have to be configured. It should detect and connect to surrogates automatically.
  - Surrogates have to be configured locally (FR5), and this process should be able to be performed by NGO personnel with only basic IT knowledge. It should be a simple process, comparable to entering data in a form and confirming.
  - Active surrogates should register with the mobile hub automatically on first connection.
- **NFR3: Usability**: Literacy among users of mobile devices will vary. Most end users will have low technical knowledge as well. The interfaces to the functionality that they use should be understandable to them.
  - Text in English, including voice explanations.
  - Text in French, including voice explanations (one of the target languages, but will not be implemented in the AgroTempus system).
- **NFR4: Extensibility**: Developing new functionality and adding it to the system should be supported and made easy. A standard format for services that perform either computation offload or data staging should be available to future developers, including documentation and an example.
- **NFR5: Energy efficiency**: The mobile device and surrogate systems will run in an energy-challenged environment. Access to electrical power is limited and not always available.
  - Energy use on mobile devices should be minimized.
  - Energy use on surrogates should be minimized, but energy efficiency for mobile devices has higher priority.
- **NFR6: Capacity**: Low-end smartphones have low storage capacity and therefore storage should, for the most part, be the responsibility of the surrogates and mobile hubs.
  - The surrogate should be able to provide computation offload and data staging capabilities to multiple users at the same time.
  - Storage used on smartphones should be kept under 100 MB, not counting results for calculations that the user has saved.
  - Surrogates should be able to run 10 instances of services at the same time.
- **NFR7: Availability**: Capabilities provided by surrogates should, in principle, be available 24 h a day. However, because surrogates will run on solar energy, it is expected that they can run out of energy during heavy use, especially during periods with no or little sunshine.
  - Every 24-hour period, the surrogate should be able to deliver services amounting to 4 h of surrogate activity. This does not provide guarantees about unavailability due to crashes (which is discussed in NFR1 and NFR9).
  - When remaining battery life drops below 10% of the battery's capacity, computations that will take longer than 5 min should be queued until the battery is recharged to above 15%.
- **NFR8: Performance**: There are no hard performance requirements, except for the transfer of data between the mobile hub and the surrogate. This is because the window during which there is opportunity to interchange data is short and infrequent.
  - The transfer of data between the mobile hub and the surrogate should be prioritized over other offloaded computation or data staging operations that the surrogate is performing.
  - The only operation with higher priority is the registration of a new surrogate.
  - The mobile hub should check for a surrogate broadcast signal at least 10 times per second, as long as it is not interacting with one already.
  - The surrogate should broadcast its presence at least 10 times per second.
- **NFR9: Recovery**: When a surrogate has crashed, restarting the hardware should have it operational again within 10 min. Similarly, when a mobile hub has crashed, resetting the hardware should have it operational again within 10 min.
- **NFR10: Data integrity**: When weather data is entered on the mobile app, it should be checked for valid values, e.g., temperature values between certain valid limits. The same applies to setup data during the setup process.

5.2.3. Constraints and assumptions

The following constraints for the development of the AgroTempus system were identified:

- **C1: Low cost infrastructure and hardware**: End-users will mostly use low-end mobile devices, while the rest of the system will be deployed on hardware locally, for which the cost should be as low as possible.

- **C2: Use of FirefoxOS**: Agrotempus has to be developed for the FirefoxOS mobile operating system [20]. FirefoxOS is open source, based on standard Web APIs, and targeted at low-end smartphones and developing markets.
- **C3: Use of open standards**: There is a preference for open source components and the use of open standards where possible.
- **C4: Use of Java**: Because the implementation platform for surrogates is still evolving, the preference is to use Java due to its portability.

Only one assumption for the system was identified:

- **A1: Concurrent access to multiple surrogates**: Surrogate signals do not overlap because there is only one surrogate per village. This means the mobile devices and mobile hub can connect to different surrogates, but never at the same time.

5.3. Mapping of system requirements to architectural tactics

Based on the functional and non-functional requirements for the AgroTempus system, several tactics were identified by the developer as feasible for their fulfillment. The mapping of system requirements to architectural tactics is shown in Table 3. The rationale for the selection of each tactic, as indicated by the developer, is provided in the following sub-sections.

5.3.1. Computation Offload

The Computation Offload tactic enables mobile clients to offload expensive computation to surrogates. Regression and weather value prediction using extrapolation (FR3 and FR4) are computation-intensive operations that are initiated by the user on the mobile device, but the computation is offloaded to the surrogate. Data sets on which these operations are performed are located at the surrogates and can be reasonably large, while the input for the operations is a small set of variables of simple data types. Offloading small input/output, energy-intensive computations to the surrogate is the main method to minimize energy consumption on the mobile device (NFR5). Offloading from low-end mobile devices to surrogates with more computational power and data storage facilities increases the capacity of the system (NFR6).

5.3.2. Out-Bound Pre-Processing

In the Out-Bound Pre-Processing tactic surrogates collect data from mobile devices and pre-process the data – clean, filter, summarize, or merge – such that the data that is sent on to the enterprise cloud is ready for consumption and serves an immediate need. Weather data collected on mobile devices (FR1) is stored locally until it has been successfully transferred to a surrogate. The surrogates will store this data indefinitely, both to make it accessible to mobile users in the future, but also to make it available to the mobile hub, which will collect all data eventually. This data will not be saved on the mobile device after it has been successfully transferred to the surrogate because storage is limited on the low-end mobile devices (NFR6). The mobile hub will eventually be able to store new data that was entered on the mobile device in the cloud when it connects to the Internet (FR7). In the AgroTempus system there are therefore two levels of data staging: first at the surrogate and then at the mobile hub.

5.3.3. Pre-Fetching

The Pre-Fetching tactic anticipates data needs in order to minimize communication to the cloud and reduce latency. The mobile hub, according to a defined pre-fetch algorithm, retrieves weather data using a third-party weather API (FR9) based on the registered location of all the surrogates that it serves. Data retrieved from the mobile hub is stored on the surrogates and not the mobile devices to address storage limitations of low-end mobile devices (NFR6). Mobile devices that request weather data (FR2) will always obtain it from a surrogate where this data is staged, unless it has been explicitly saved on the mobile device by the user. The same is true for forecasts (FR6), which are calculated based on data downloaded from the mobile hub.

5.3.4. Pre-Provisioned Surrogate

In the Pre-Provisioned Surrogate tactic surrogates are provisioned before their deployment with the capabilities that are offloaded by mobile clients. All required functionality will be available on the surrogate from the start (FR1, FR2, FR3, FR4, FR6, FR7, FR9, FR10). Because all surrogates serving all regions have the same capabilities, it is easier to provision them using the same OS image (e.g., Raspberry Pi with cloned SD card) (NFR2). The only difference between surrogates is the location and identification settings provided during the setup procedure (FR5). Restarting a pre-provisioned surrogate is easier to do if started from a common OS image (NFR9).

5.3.5. Surrogate Broadcast

In the Surrogate Broadcast tactic surrogates advertise their availability and selected metadata to mobile devices for discovery. Mobile device users should be able to make use of system functionality as soon as they install the app and come in range of a surrogate. To increase the ease of deployment (NFR2), surrogates broadcast their presence and mobile devices in need of surrogate services can pick up on these broadcasts. Surrogate broadcast is also key for the automatic registration of newly deployed surrogates with the mobile hub as soon as they are in communication range (FR10). Lastly, because the opportunities for interaction between surrogates and the mobile hub are scarce, both the surrogate broadcasting its presence continuously and the mobile hub continuously trying to discover surrogates are key to the system's performance (NFR8).

5.3.6. Cached Results

The Cached Results tactic enables a system to cache results and state on a surrogate until the mobile device is able to reconnect. The interaction between mobile devices and surrogates is susceptible to loss of connection in the AgroTempus system. When computation offload (FR3, FR4) has been correctly initiated, but the mobile user moves out of range of the surrogate during the computation, results should be cached (NFR1) so they can be sent to the user as soon as the mobile device connects to the surrogate again to promote availability (NFR7).

5.3.7. Client-Side Data Caching

The Client-Side Data Caching tactic is a variation of the Cached Results tactic. Data collected by a mobile client is cached on the mobile device and sent to the surrogate upon connection or reconnection. Because mobile devices are not always in proximity of a surrogate, when entering weather data (FR1) without an available connection, data is cached on the mobile device (NFR1), which will periodically try to resend the data. In this case, caching is used to enable users to keep working with the app, saving new readings, and not having to worry about the data being saved immediately on the surrogate, therefore promoting availability (NFR7).

5.3.8. Just-in-Time Containers

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. Data regression (FR3) and weather value prediction (FR4) are heavy computations that will be used infrequently. Therefore, as opposed to the other services offered

Table 3
Mapping of system requirements to architectural tactics.

Tactic	FR1: Store weather data	FR2: Retrieve weather data	FR3: Data regression	FR4: Future weather values	FR5: Surrogate setup	FR6: Forecast delivery	FR7: Integration cloud storage	FR8: Voice interface	FR9: Synchronize weather data	FR10: Surrogate registration	NFR1: Fault tolerance	NFR2: Ease of deployment	NFR3: Usability	NFR4: Extensibility	NFR5: Energy efficiency	NFR6: Capacity	NFR7: Availability	NFR8: Performance	NFR9: Recovery	NFR10: Data integrity
Computation Offload			X	X											X	X				
Out-Bound Pre-Processing	X						X									X				
Pre-Fetching		X				X			X							X				
Pre-Provisioned Surrogate	X	X	X	X	X	X	X		X	X		X							X	
Surrogate Broadcast										X		X						X		
Cached Results			X	X							X						X			
Client-Side Data Caching	X										X						X			
Just-in-Time Containers			X	X											X	X				

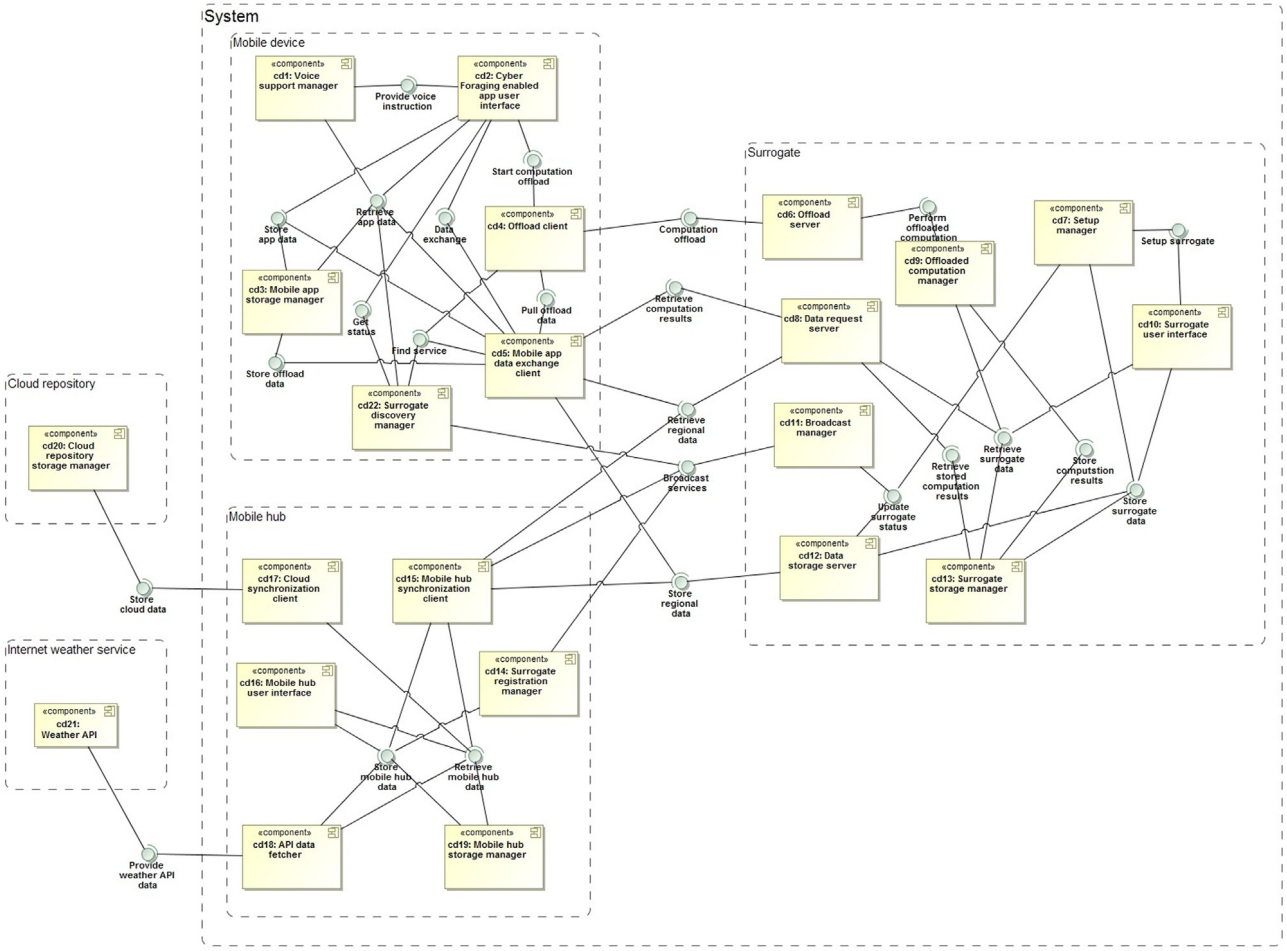


Fig. 10. High-level architecture of the AgroTempus system.

by surrogates, these services are better suited to run in their own containers, such that small operations will not get queued behind these large computations. To be able to handle multiple computation offload requests at the same time, as well as to not let these large computations cause small data transfers to have to wait for them (NFR6), each time a request for a computation offload is received at the surrogate, a container with the necessary functionality is created. Because requests for computation offload will be infrequent, often with long periods of time between requests, only creating containers for these capabilities when they are needed is a tactic that will save energy on the surrogate (NFR5).

5.4. System architecture and design

Based on the identified tactics, the developer created the high-level architecture for the AgroTempus system shown in Fig. 10 as a UML component diagram. Some components of the architecture were derived from the architectural tactics and others were

added to fulfill requirements not addressed by the tactics. The detailed components and mapping to the tactics are presented in Section 5.6. The main elements of the architecture are:

- Mobile Device Components
  - CD1: Voice Support Manager: Manages the voice snippets that map to the user interface elements.
  - CD2: Cyber-Foraging Enabled App User Interface: User interface component of the mobile app.
  - CD3: Mobile App Storage Manager: Manages storage of all permanent data and user settings on the mobile app, except for data that is being staged before moving to the surrogate. Storing and retrieving data is done through its interfaces: *Store app data* and *Retrieve app data*.
  - CD4: Offload Client: Handles computation offload from the mobile app to the surrogate, initiated through component CD2.
  - CD5: Mobile App Data Exchange Client: Handles staging data and transferring it from the mobile app to the surrogate after it has been entered via component CD2. It also handles requesting and receiving data from the surrogate.
  - CD22: Surrogate Discovery Manager: Finds available surrogate services.
- Surrogate Components
  - CD6: Offload Server: Handles requests for computation offload from mobile devices.
  - CD7: Setup Manager: Implements the setup process for newly deployed surrogates. Provides the interface *Setup surrogate*, which is used by component CD10 when the setup process is started.
  - CD8: Data Request Server: Handles requests for data stored on the surrogate from mobile devices, as well as from the mobile hub.
  - CD9: Offloaded Computation Manager: Creates containers that run offloaded computation and ensures that results are eventually stored in component CD13.
  - CD10: Surrogate User Interface: User interface component for the surrogate, available when a screen and mouse/keyboard are connected to the surrogate (e.g., during the setup process or to check console output).
  - CD11: Broadcast Manager: Broadcasts the presence of the surrogate and its capabilities through the interface *Broadcast services*. It is key for all requirements in which interaction between the surrogate and other system nodes is involved.
  - CD12: Data Storage Server: Handles requests from mobile devices and the mobile hub for storing data on the surrogate.
  - CD13: Surrogate Storage Manager: Manages storage of all permanent data, computation results, and settings on the surrogate. The interfaces for data retrieval include the possibility to delete data after a successful transmission.
- Mobile Hub Components
  - CD14: Surrogate Registration Manager: Handles the registration of surrogates that are new to the system by picking up broadcasts from component CD11 and storing new surrogate data in component CD19.
  - CD15: Mobile Hub Synchronization Client: Manages synchronization of data between the mobile hub and the surrogate.
  - CD16: Mobile Hub User Interface: User interface component for the mobile hub.
  - CD17: Cloud Synchronization Client: Ensures that data stored in the system is backed up to a cloud repository by interacting with component CD20.
  - CD18: API Data Fetcher: Retrieves weather data from a third party API and stores it on the mobile hub via component CD19. It also periodically checks whether the surrogate list stored by this component has new entries.
  - CD19: Mobile Hub Storage Manager: Handles storage and retrieval of data on the mobile hub, including settings, staged data, permanent weather data, and the list of known surrogates.
- Cloud Repository Component (External)
  - CD20: Cloud Repository Storage Manager: Third-party component that interacts with component CD17 to ensure that data stored in the system is backed up to a cloud repository.
- Internet Weather Service (External)
  - CD21: Weather API: Third-party component that provides weather data and forecasts through a REST interface.

5.5. Mapping of architectural components to system requirements

The mapping of functional and non-functional requirements to components of the architecture is shown in Table 4 (see end of manuscript). All requirements are implemented by one or more components, with the exception of *NFR4: Extensibility* because this requirement is related to the creation of artifacts to support developers, such as templates and documentation, and not to specific runtime components.

5.6. Mapping of architectural components to identified architectural tactics

The mapping between architecture components and the architectural tactics identified in Section 5.3 is provided in the following subsections to show component details, as well as the mapping to specific architectural tactic elements. All design decisions described at this point correspond to the as-initially-designed system. The final implementation decisions are described in Section 5.7. Implementation diagrams for tactics that have already been presented will not be included due to space limitations. The full set of implementation diagrams can be found in [13].

5.6.1. Computation Offload

The Computation Offload tactic is designed in the AgroTempus architecture for the offload operation to take place as follows. The stereotypes from Fig. 2(a) are used.

1. The Cyber-Foraging Enabled App User Interface (*Cyber-Foraging Enabled Mobile App*) requests to start an offloaded computation with input *Input*.
2. The Offload Server (component of *Offload Server*) receives the request and invokes the Offloaded Computation Manager (component of *Offload Server*).
3. The Offloaded Computation Manager starts the offloaded computation in a separate Java Thread (*Offloaded Code*) inside the JVM (*Container*).

The main difference between the Computation Offload tactic and the AgroTempus architecture is how the offloaded computation is executed. In the tactic shown in Fig. 2(a), after the offloaded computation is set up, the control returns to the *Cyber-Foraging*

Table 4
Mapping of system requirements to architecture components.

Component	FR1: Store weather data	FR2: Retrieve weather data	FR3: Data regression	FR4: Future weather values	FR5: Surrogate setup	FR6: Forecast delivery	FR7: Integration cloud storage	FR8: Voice interface	FR9: Synchronize weather data	FR10: Surrogate registration	NFR1: Fault tolerance	NFR2: Ease of deployment	NFR3: Usability	NFR4: Extensibility	NFR5: Energy efficiency	NFR6: Capacity	NFR7: Availability	NFR8: Performance	NFR9: Recovery	NFR10: Data integrity
CD1: Voice Support Manager								X					X		X					
CD2: App User Interface	X	X	X	X		X		X					X		X					X
CD3: App Storage Manager		X	X	X		X		X			X		X		X	X				
CD4: Offload Client			X	X							X		X		X					
CD5: App Data Exch Client	X										X		X		X					
CD6: Offload Server			X	X							X		X		X	X			X	
CD7: Setup Manager					X						X	X	X		X				X	X
CD8: Data Request Server	X	X	X	X		X					X		X		X	X			X	
CD9: Offloaded Comp Manager			X	X							X		X		X	X	X		X	
CD10: Surrogate UI					X						X	X	X		X				X	
CD11: Broadcast Manager	X	X	X	X		X	X		X	X	X	X	X		X			X	X	
CD12: Data Storage Server	X					X			X		X		X		X	X			X	
CD13: Surrogate Storage Mgr	X	X	X	X	X	X			X		X		X		X				X	
CD14: Surrogate Reg Mgr										X	X	X	X					X	X	
CD15: Mobile Hub Sync Client	X	X				X	X		X		X		X					X	X	
CD16: Mobile Hub UI													X						X	
CD17: Cloud Sync Client							X						X						X	
CD18: API Data Fetcher		X				X			X				X						X	
CD19: Mobile Hub Storage Mgr	X					X	X		X	X			X						X	
CD20: Cloud Repo Storage Mgr							X													
CD21: Weather API		X				X			X											
CD22: Surrogate Discovery Mgr	X	X	X	X		X					X	X	X		X					

Enabled Mobile App, which then executes the offloaded computation via the operation `4:Execute(Input)`. This is because the assumption is that the app interacts with the offloaded code in a request/response manner until the app closes. In the AgroTempus system, offloaded computation corresponds to a lengthy computation that is executed only once in an offload request. Therefore, the *Input* to the offloaded computation is sent in the initial request to offload.

5.6.2. Out-Bound Pre-Processing

The Out-Bound Pre-Processing tactic is designed in the AgroTempus architecture to support data staging from the mobile devices as follows. The stereotypes from Fig. 7(a) are used.

1. The Cyber-Foraging Enabled App User Interface (*Mobile Sensing App*) captures weather data and sends it to the Mobile App Data Exchange Client (*Communication Manager* on mobile device).
2. The Mobile App Data Exchange Client queues the weather data until a surrogate is in range and then sends it to the Data Storage Server (*Communications Manager* on the surrogate) for storage on the surrogate via the Surrogate Storage Manager (*Cache* on the surrogate).
3. The Data Request Server (*Data Processing Application*) on the surrogate waits for a weather data request from the Mobile Hub Synchronization Manager (*Communications Manager* on the surrogate). This happens when the mobile hub is in range of the surrogate.
4. The Data Request Server retrieves the weather data and sends it to the mobile hub for storage on the mobile hub via the Mobile Hub Storage Manager (*Cache* on the mobile hub).
5. Once the Cloud Synchronization Client (*Data Processing Application*) on the mobile hub has connectivity to the cloud repository, it retrieves the weather data from the Mobile Hub Storage Manager and sends it to the Cloud Repository Storage Manager for storage in the Cloud Repository (*Cloud Data Repository*).

The difference between the AgroTempus architecture and the Out-Bound Pre-Processing tactic is that the AgroTempus system performs data staging at two levels to get data from the mobile devices to the cloud: first at the surrogate and then at the mobile hub. Therefore, the Data Request Server on the surrogate and the Cloud Synchronization Client on the mobile hub perform two roles: data processing application for the cached data and communication manager for passing the information to the next level en route to the enterprise cloud.

5.6.3. Pre-Fetching

The Pre-Fetching tactic is designed in the AgroTempus architecture as shown in Fig. 11(b), with numbers to indicate the sequence of operations. The pre-fetching of data from the enterprise cloud to the surrogates serving mobile devices takes place as follows:

1. When the Mobile Hub has access to the Internet Weather Service, the Cloud Synchronization Client retrieves all weather data for the surrogates that it serves from the Weather API, based on the Surrogate Location List. It then caches the retrieved weather data.

2–3. When the Mobile Hub is in proximity of a Surrogate that it serves, the Mobile Hub Synchronization Manager reads the data for the surrogate location and pushes it to the Data Request Server on the Surrogate.

4. The Data Request Server caches the data on the Surrogate via the Surrogate Storage Manager.

5–7. When the mobile app has a request for weather data, the data is obtained from the Surrogate.

There are two differences between the AgroTempus architecture and the Pre-Fetching tactic:

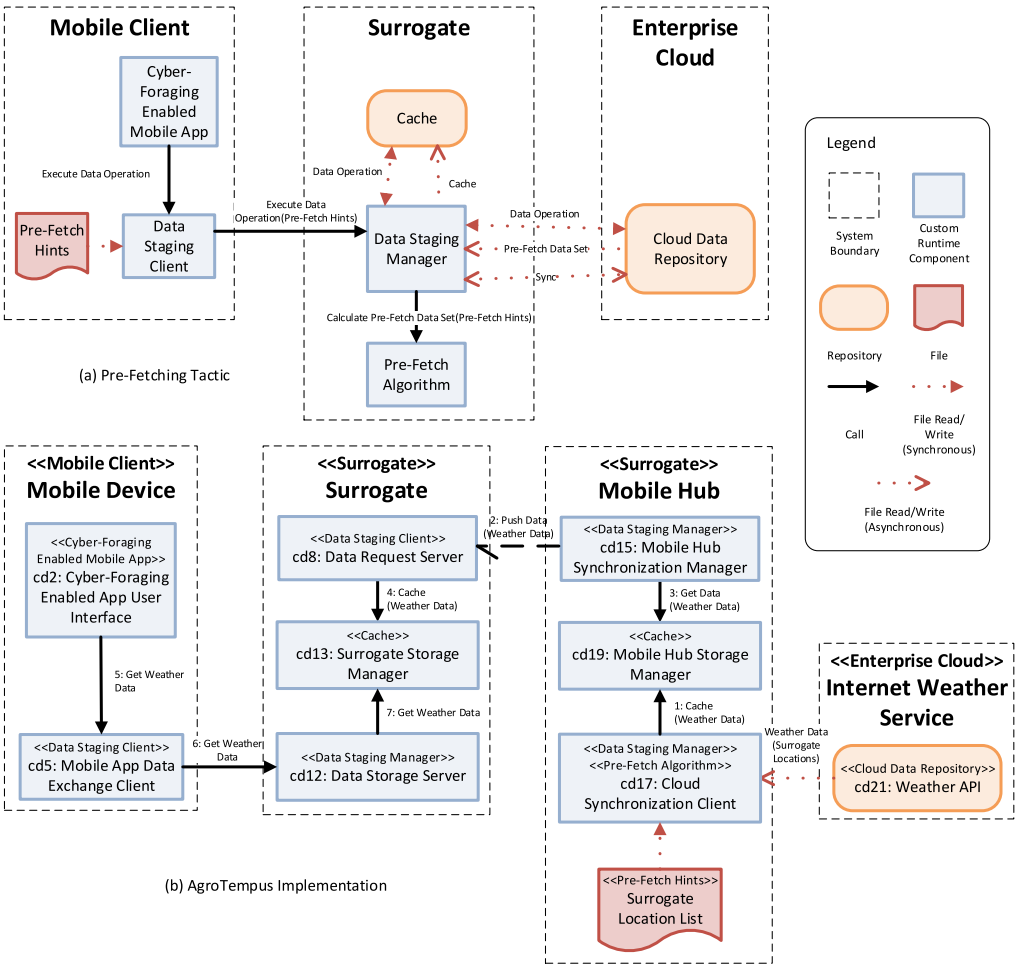


Fig. 11. Mapping of the AgroTempus architecture to the Pre-Fetching tactic.

1. The AgroTempus system performs data staging at two levels to pre-fetch data from the cloud and host it on the surrogates: first from the cloud to the mobile hub, and then from the mobile hub to the surrogate.
2. The Pre-Fetch Algorithm and Pre-Fetch hints reside on the mobile hub and not on the mobile client. This is because the mobile hub needs to fetch data from the cloud at a point in time when it is not likely that it will be near a surrogate or a mobile device. The Surrogate Location List is populated during Surrogate Registration (FR10).

5.6.4. Pre-Provisioned Surrogate

The Pre-Provisioned Surrogate tactic is designed in the AgroTempus architecture to provision capabilities on the surrogate as follows. The stereotypes from Fig. 3(a) are used.

1. A Terminal (component of *Local User Interface*) on the Surrogate is used to load the Surrogate Component Code Files (*Capability*) on the Surrogate.
2. The Terminal is used to start the Surrogate User Interface (component of *Local User Interface*) to obtain setup parameters for the surrogate, such as location.
3. The Surrogate User Interface invokes the Setup Manager (*Surrogate Manager*) to start the remaining surrogate components.

Step 1 of the provisioning process is only executed once prior to surrogate deployment. Step 2 is executed only once during surrogate deployment. Step 3 is executed manually during deployment, and then automatically on start/restart of the surrogate. There is not the equivalent of the Capability Metadata component nor a Capability Registry component because the capabilities provided to all mobile devices are the same and are not advertised. In addition, there is not the equivalent of a Remote User Interface because surrogates are envisioned to be low cost, low-end servers that are set up on site.

5.6.5. Surrogate Broadcast

The Surrogate Broadcast tactic is designed in the AgroTempus architecture for surrogate discovery as follows. The stereotypes in Fig. 4(a) are used.

0. The Broadcast Manager (*Broadcast Component*) running on the Surrogate broadcasts its address.
1. The Cyber-Foraging Enabled Mobile App User Interface (*Cyber-Foraging Enabled Mobile App*) requests an offload operation.
2. The Offload Client receives the request and obtains the surrogate address from the Surrogate Discovery Manager (components of *Offload Client*).
3. The Offload Client sends the offload operation to the Offload Server at the surrogate address.

The difference between the AgroTempus architecture and the Surrogate Broadcast tactic is that there is no need to find an optimal surrogate because only one surrogate is available for a mobile device. The assumption as stated in Section 5.2.3 is that there is only one surrogate per village, and surrogate signals do not overlap.

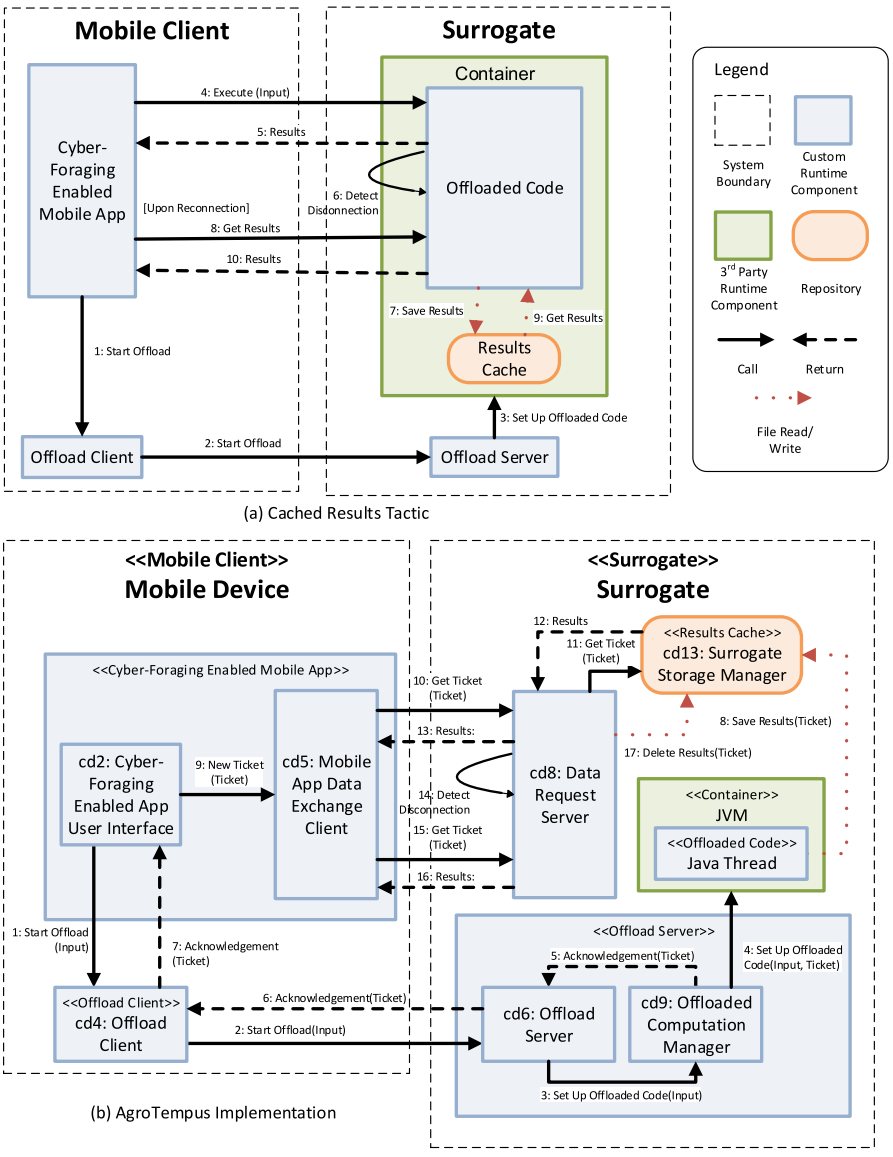


Fig. 12. Mapping of the AgroTempus architecture to the cached results Tactic.

The surrogate also broadcasts its presence to the mobile hub via the same mechanism.

5.6.6. *Cached Results*

The Cached Results tactic is designed in the AgroTempus architecture as shown in Fig. 12(b). The caching of results on a surrogate takes place as follows:

1–2. The Cyber-Foraging Enabled App User Interface requests to start an offloaded computation with input *Input*.
3. The Offload Server receives the request and invokes the Offloaded Computation Manager.
4–7. The Offloaded Computation Manager assigns the computation a unique identifier called a Ticket, starts the offloaded computation in a separate Java Thread inside the JVM, and returns an Acknowledgment to the Cyber-Foraging Enabled App User Interface with the assigned Ticket.
8. The Offloaded Computation executes and saves the results in the Surrogate Storage Manager with the assigned Ticket.
9–10. The Cyber-Foraging Enabled App User Interface, via the Mobile App Data Exchange Client, sends a request to the Data Request Server on the Surrogate for the results for the received Ticket.
11–12. The Data Request Server retrieves the results from the Surrogate Storage Manager.
13. The Data Request Server returns the results to Mobile App Data Exchange Client.
14–16. If the connection to the Mobile Device breaks during the transmission, the results remain on the Surrogate until they can be successfully sent to the Mobile Device.
17. After successful transmission the results associated with the Ticket are deleted from the surrogate.

There are two differences between the AgroTempus architecture and the Cached Results tactic:

1. Because the offloaded computation is expected to be a lengthy operation, the Surrogate always saves the results in the Results Cache instead of attempting the send the results to the Mobile Device immediately.
2. The Surrogate Storage Manager resides outside the Container because it is shared by all offloaded computation and other surrogate components.

5.6.7. Client-Side Data Caching

The Client-Side Data Caching tactic is designed in the AgroTempus architecture to store data collected on the mobile device until a surrogate is available as follows. The stereotypes in Fig. 9(a) are used.

1. The Cyber-Foraging Enabled App User Interface (*Mobile Sensing App*) requests the Mobile App Data Exchange Client (*Communications Manager* and *Mobile Cache*) to add collected weather data its outbound queue.
   [Repeat Until Outbound Queue is Empty]
2. The Mobile App Data Exchange client tries to find a surrogate (Section 5.6.5).
   [If a Surrogate is Found]

3–4. Queued data is sent to the Data Storage Server *Communications Manager*) for storage on the Surrogate Storage Manager (*Surrogate Cache*).

5–6. If the storage operation is successful the sent data is deleted from the queue.

There are two differences between the AgroTempus architecture and the Client-Side Data Caching tactic:

1. Because the collection of weather data is likely going to be in the field where there will not be a Surrogate in proximity, the Mobile Device always queues the results in the Mobile App Data Exchange Client instead of attempting to send the results to the Surrogate immediately.
2. The Mobile Cache, implemented as a queue, is part of the Mobile Data Exchange Client instead of a separate storage component.

5.6.8. Just-In-Time Containers

The Just-in-Time Containers tactic is designed in the AgroTempus architecture for creation and destruction of containers for offloaded computation as follows. The stereotypes in Fig. 5(a) are used as stereotypes.

1. The Cyber-Foraging Enabled App User Interface (*Cyber-Foraging Enabled Mobile App*) requests to start an offloaded computation with input *Input*.
2. The Offload Server receives the request and invokes the Offloaded Computation Manager (components of *Offload Server*).
3. The Offloaded Computation Manager starts the offloaded computation in a separate Java Thread (*Offloaded Code*) inside the JVM (*Container*).
4. Upon finishing the execution of the offloaded computation, the thread is terminated, therefore releasing allocated resources.

As with the Computation Offload tactic (Section 5.6.1), the main difference between the Just-in-Time Containers tactic and the AgroTempus architecture is that because the offloaded computation is only executed once, the *Input* to the offloaded computation is sent in the initial request to offload.

5.7. System implementation

A demo implementation of the AgroTempus system is available and documented at http://reuelbrion.github.io/AgroTempus/. Only the mobile app and surrogate components were developed as part of the demo because this is where the identified tactics are mainly implemented. The mobile hub and cloud components were simulated for the testing and evaluation of the system. The surrogate software was packaged for Raspberry Pi as a Raspbian OS image with an auto-start script. Raspbian is a Linux distribution optimized for Raspberry Pi [21]. The image was tested on a Raspberry Pi 2 Model B with a TP-Link TL-WN722N wireless adapter.

The mobile app (*Mobile Device* components in Fig. 10) is a Firefox OS app, which is essentially a Web app consisting of HTML pages, CSS style sheets, and Javascript code. Most of the app logic is written in plain Javascript with minimal use of the JQuery library [22].

The surrogate (*Surrogate* components in Fig. 10) was implemented in Java as a multi-threaded application. The component *CD9: Offloaded Computation Manager* that performs weather data regression and prediction makes use of the Java chart library JFreeChart [23] that offers tools to perform regression on data sets, as well as to generate plot images to visualize the results in common image formats. The same component also makes use of the Apache Commons Codec libraries [24] to convert images generated by JFreeChart into Base64[7] binary string format.

For communication between components residing on different nodes, JSON (JavaScript Object Notation) [25] was selected as the standard message and data storage structure. This format is used by free weather APIs such as OpenWeatherMap [26] and works well with Javascript. To be able to use JSON objects in the surrogate code, the system makes use of the JSON.simple toolkit [27]. JSON is also used by IndexedDB, the selected data storage API for FirefoxOS [28].

5.8. Analysis

5.8.1. System evaluation

The AgroTempus system implementation included seven of the eight tactics listed in Table 3. At implementation time, no working ad-hoc networking library was found for Firefox OS. Therefore, the Surrogate Broadcast tactic could not be used for surrogate discovery in the mobile app. The Local Surrogate Directory tactic was instead used for surrogate discovery. A list of surrogates, including connection details, is maintained on the mobile app. This way, whenever a surrogate service is needed, the mobile app tries to connect to each surrogate one by one until it can make a connection to a surrogate that provides the needed capabilities.

Extensive testing of the system, based on the scenarios defined for each requirement, was performed in order to verify that the implemented system satisfied its intended functional and non-functional requirements. Scenario details and test results are available at http://reuelbrion.github.io/AgroTempus/. The implementation details for each tactic are detailed below.

The **Computation Offload** tactic was implemented as designed and tested successfully. It is used to perform data regression (FR3) and prediction of future weather values (FR4), two computation-intensive operations. In addition, the generation of the regression chart images is another potentially computation-intensive operation that is also performed on the surrogate. Even though energy consumption was not measured on the mobile device to demonstrate energy efficiency (NFR5), these are two examples of operations that consume and produce small amounts of information compared to their computational requirements, which is known benefit from cyber-foraging [29]. The data regression operation takes as input a weather variable name (Temperature, Humidity, Pressure or Wind Speed), regression type (currently accepts only Linear, but can be easily extended to support other types such as Logistic and Polynomial), a start date, and the number of days to extrapolate, and produces a graph (PNG image) showing all the data points and the regression line. The weather value prediction operation has a weather variable name as input and produces a list of predictions for the variable for the next 7 days. Given that

[7] Base64 is a set of binary-to-text encoding schemes commonly used when sending binary data over a network.

the mobile devices that the AgroTempus app is intended to run on are low-end smartphones with limited computing and storage capabilities, the Raspberry Pi surrogate, although limited as well, still offers more computational power and data storage to increase the capacity of the system (NFR6). The smartphone used for test and evaluation was a ZTE Open C 4.0 with an MSM8210 Dual-Core 1.2 GHz CPU and 512 MB RAM [30]. The Raspberry Pi 2 Model B has a 900 MHz quad-core ARM Cortex-A7 CPU and 1 GB RAM [31], and supports SD cards up to 32 GB for storage. Given the successful implementation of the tactic as designed, an improvement for the tactics catalog would be to include a variation of the Computation Offload tactic for cases where there is a single request to offload instead of a continued request/response interaction between a mobile device and a surrogate.

The **Out-Bound Pre-Processing** tactic is used for intermediate storage of weather data on the surrogate (FR1) and eventual storage of weather data in the cloud (FR7). It was implemented as designed between the mobile device and the surrogate. Data captured on the mobile device was successfully transmitted and stored on the surrogate. Transmission of the weather data to the mobile hub and eventual storage on the cloud was simulated. As indicated in the evaluation of the Computation Offload tactic, data storage on the surrogate is larger than what is available on the mobile device, therefore increasing the storage capability of the system (NFR6). In addition, as will be described in the implementation of the Client-Side Data Caching tactic, weather data is deleted on the mobile device after successful transmission to the surrogate to also increase storage capacity. Although not tested end-to-end with real data, there is potential for the Out-Bound Pre-Processing tactic to implement more than one level of data staging as long as the client and surrogate roles are replicated across levels. An improvement for the catalog would be to include a variation of the Out-Bound Pre-Processing tactic for multi-level data staging.

The **Pre-Fetching** tactic was simulated in the demo implementation by loading a static set of weather data on the surrogate at startup time and tested successfully. The data was used and retrieved by the mobile app (FR2). Because of the lack of a mobile hub and cloud implementation, the complete fetching of data from the cloud to the surrogate (FR9) was not tested. However, the implementation of the fetch and store capabilities implemented in surrogate components *CD8: Data Request Server* and *CD13: Surrogate Storage Manager* would be equivalent to the discover and store capabilities on the mobile hub that would act as an intermediary between the cloud and the surrogate (*CD17: Cloud Synchronization Client* and *CD19: Mobile Hub Storage Manager*). As indicated in the evaluation of the previous two tactics, data storage on the surrogate is larger than what is available on the mobile device, therefore increasing the storage capability of the system (NFR6). Similar to the Out-Bound Pre-Processing tactic, there is potential for the Pre-Fetching tactic to implement more than one level of data staging as long as the client and surrogate roles are replicated across levels. An improvement for the tactics catalog would be to include a variation of the Pre-Fetching tactic for multi-level data staging.

The **Pre-Provisioned Surrogate** was implemented as designed and tested successfully. It enables all the functional requirements of the system, except for the voice interface (FR8) which was not implemented in the demo. All offloaded computation (short and long operations) is loaded on the surrogate upon setup and is packaged inside a Raspbian OS image with auto-start capabilities, as mentioned earlier, to support ease of deployment (NFR2). This same auto-start capability enables surrogate recovery after crashes (NFR9). Similar to the GigaSight system implementation of the Pre-Provisioned Surrogate tactic (Section 4.5.1), the AgroTempus implementation confirms that an improvement for a future version of the tactic would be to mark the *Capabilities Metadata* and *Capability Registry* components as optional because they are not necessary when capabilities are not advertised.

The **Surrogate Broadcast** tactic was not implemented in the AgroTempus system as indicated earlier. The **Local Surrogate Directory** tactic was used for surrogate discovery and implemented as indicated in the tactic. Ease of deployment (NFR2) is not as strongly supported by this tactic as would have been with the Surrogate Broadcast tactic. In the current implementation the list of surrogates is hard-coded in the mobile app. The original intent was to include surrogate metadata in a QR code on a sticker that would be placed on the surrogate. A mobile device that would want to make use of the surrogate would read the QR code, which would add the metadata to the list of available surrogates. However, as of the time of implementation, there were no QR libraries available for Firefox OS. Even though it was not tested with a mobile hub, there are multiple options for surrogate broadcast for Java which could be used by the surrogate to broadcast its presence to the mobile hub, such as the ZeroConf protocol used by the Tactical Cloudlets system (Section 3.4.3). To satisfy the performance requirement (NFR8), once a surrogate is contacted by a mobile hub, all running threads would be suspended until synchronization with the mobile hub is complete.

The **Cached Results** tactic was implemented in the surrogate as designed and tested successfully. Results of the data regression (FR2) and weather value prediction (FR4) operations are always stored on the surrogate and not sent to the mobile device until requested in order to support fault tolerance (NFR1). This is in case the mobile device moves out of the range of the surrogate before the computation completes. The results are saved until the mobile device connects to the surrogate, therefore promoting availability (NFR7). The change made in the design to always saves results on the surrogate when offloaded operations are expected to be lengthy, instead of attempting to send results to the mobile device immediately, could be added as a variation of the Cached Results tactic.

The **Client-Side Data Caching** tactic was implemented as designed and tested successfully. Data captured in the field (FR1) is stored on the mobile device until a surrogate is available, to promote fault tolerance (NFR1). The results are saved on the mobile device until it can connect to a surrogate, therefore promoting availability (NFR7). Similar to the Cached Results tactic, the change made in the design to always queue the results instead of attempting to send the results to the surrogate immediately could be added as a variation of the Client-Side Data Caching tactic.

The **Just-in-Time Containers** tactic was implemented as designed and tested successfully. When data regression (FR3) and prediction of future weather values (FR4) are offloaded, the system starts the computation in a separate thread, which is destroyed upon completion, therefore increasing the available capacity of the system (NFR6). In addition, because the computation only runs upon request, energy is saved on the surrogate (NFR5).

Based on this analysis, nine of the ten functional requirements were successfully supported through one or more of the available tactics, as shown in Table 3. The Voice Interface requirement (FR8) was not implemented due to project constraints but also because it was known that it would not be implemented through any of the tactics.

Similarly, seven of the ten non-functional requirements were successfully supported through one or more of the available tactics, as also shown in Table 3. The usability requirement to support multiple languages (NFR3), similar to the voice interface requirement, was not implemented due to project constraints, but also because it was known that it would not be implemented through any of the tactics. The extensibility requirement to support the development of new services (NFR4) was partially implemented outside of the tactics, through the initial implementation of the

project website that contains the mobile app and surrogate code, as well as documentation (http://reuelbrion.github.io/AgroTempus/). The current documentation needs to be augmented to fully support the requirement by providing more detailed guidance to developers (e.g., location of extension points, templates for new services). Finally, the data integrity requirement to provide data checks (NFR10) was not implemented due to project constraints, but could be easily be implemented outside of the tactics through input validation code in the user interface components.

5.8.2. Developer observation and feedback

Throughout the process we met with the developer once a week to check on project status and observe how the tactics were being used. The general development process that was followed is consistent with the structure of this section: (1) requirements elicitation, (2) mapping of requirements to tactics, (3) architecture, (4) mapping of components to architecture, (5) design, (6) implementation, and (7) testing and evaluation. Because of the nature of the case study, the developer was asked to document the project during the entire process.

The developer found the tactics easy to understand and use. The most difficult part for the developer was determining, based on the tactics, which of the components would be needed to implement the requirements. Feedback for a future version of the tactics is to provide differentiation between core and optional components of the tactic, consistent with the findings from the previous two case studies. Another recommendation from the developer was to include sample code and potentially a list of libraries/platforms that can be used to implement common requirements of cyber-foraging systems. The inclusion of sample code with the tactics is consistent with the feedback from the main developer of the GigaSight system (Section 4.5.2).

The developer also found the tactics to be useful in the development of the system. As stated by the developer: "The models that were used as a blueprint during development were in large constructed from the tactics; they were instrumental in providing a good foundation for the application".

5.8.3. Findings

Eight tactics were identified in AgroTempus to satisfy system requirements, of which seven were implemented in the system, and one had to be replaced by an alternative tactic due to a technology constraint. All the tactics were implemented as designed, but there were several changes that were made at design time to better fulfill requirements. Even though the essence of each tactic remained the same, these changes create opportunities for improvement of the tactics catalog. In particular, variations to the Computation Offload, Out-Bound Pre-Processing, Cached Results, and Client-Side Data Caching tactics were identified.

The case study shows that there are different ways to implement tactics, mainly determined by system constraints and assumptions, but also by mobile device and surrogate computing power and specifications, as well as usage contexts. For example, VMs are used as data and computation containers in the Tactical Cloudlets and GigaSight systems because of the flexibility that they provide, but also because the surrogates are expected to be high-end servers. For the AgroTempus system the selection of using JVMs as computation containers is a better choice because they have less overhead and consume less resources on the machine. They do not provide the flexibility of VMs, but this is not required in the more static usage context of AgroTempus.

The case study also showed that technology selection can sometimes be a barrier to the use of tactics and therefore effective satisfaction of requirements. The use of Firefox OS as the mobile device operating system did not allow the implementation of the Surrogate Broadcast tactic because of the lack of libraries for discovery in this platform. In addition, the lack of libraries for QR code reading also affected the ease of deployment requirement that was associated to the Local Surrogate Directory tactic that replaced that Surrogate Broadcast tactic for surrogate discovery. These technology insights that are gained from the implementation and evaluation of cyber-foraging systems could be added as notes to the tactics to provide even greater value to software architects.

Finally, as more real cyber-foraging systems are deployed, more tactics and non-functional requirements will emerge. For example, recovery was not a requirement that was identified as part of the SLR on architectural tactics for cyber-foraging [3]. However, it is highly likely that this will be a requirement for cyber-foraging systems in resource-challenged environments, such as the AgroTempus usage context. Recovery in the AgroTempus system was implemented via the use of Java threads combined with a monitoring capability. Because service instances run in separate threads after the initial connection, a failed service thread will not affect the main service thread. Passing data between threads happens through thread-safe queues (java.util.concurrent.ConcurrentLinkedQueue). The main surrogate process periodically checks whether all service threads are alive, and crashed threads are restarted. A generalization of this approach could easily be codified as a Surrogate Recovery tactic.

6. Threats to validity

There are two main threats to the validity of the results of the case studies. The first is related to *internal validity* because the data collection and analysis was conducted by a single researcher and therefore subjective interpretations might exist. To mitigate this threat for the Tactical Cloudlets and GigaSight case studies, collected data was reviewed by system developers that confirmed that the data collected was an accurate representation of the system. The developers also confirmed that the identified tactics were indeed present in the system. For the AgroTempus system, data collected from several sources (evolving system documentation, the code base, and ongoing developer interviews) was confirmed by the developer such that we could have immediate feedback.

The second threat is related to *external validity*, specifically whether the findings are generalizable given that the results reported for each case study are drawn from the analysis of a single system. To mitigate this threat we conducted three case studies. In addition, the system developers of the Tactical Cloudlets and GigaSight systems were provided the full set of tactics and asked to identify tactics that could be used to enhance the current system. The developers identified several tactics and recognized the potential for the tactics to build a better system. The developer of the AgroTempus system confirmed the usefulness of the tactics to build cyber-foraging systems.

7. Related work

Case studies are commonly used in software engineering to study aspects of real software systems, such as development processes, software artifacts, and development teams [32]. We used the case study methodology to validate the identified architectural tactics in real cyber-foraging systems.

The work that is most closely related to that presented in this article are case studies related to the identification of architectural tactics. Gesvindr and Buhnova [33] identify and evaluate a number of architectural tactics for PaaS cloud applications and demonstrate their findings with a case study of a private, cloud-based social network system. Although not targeted at the identification of tactics, Mirakhorli et al. [34] present a technique for automating the reconstruction of traceability links between classes and architectural tactics and validate their approach via a case study of the

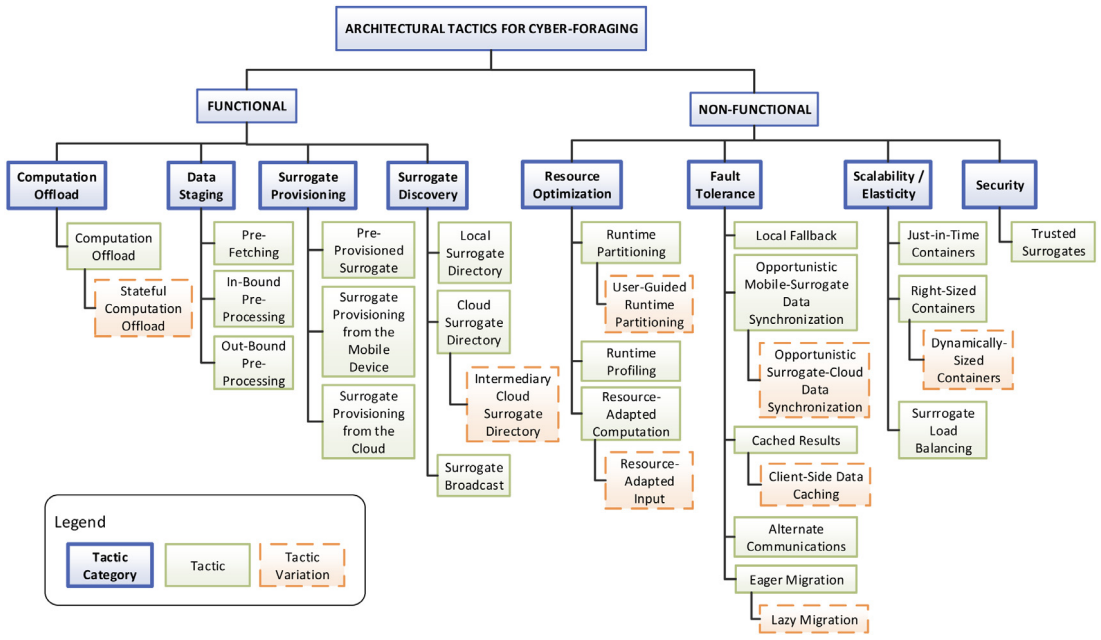


Fig. A.13. Architectural tactics for cyber-foraging.

Table A.5
List of functional architectural tactics.

Computation Offload	
Computation Offload	Mobile clients offload expensive computation to surrogates. Computation is self-contained in the form of a module, class, service, or complete application.
Stateful Computation Offload	Mobile clients offload expensive computation to surrogates. Because the granularity of the offload operation is at the process or at the method level, the state of the program or object that contains the process or method being offloaded has to be transferred to the equivalent program or object on the surrogate to guarantee that state is equivalent on both the mobile device and the surrogate.
Data Staging	
Pre-Fetching	Surrogate anticipates mobile device data needs in order to minimize direct communication to the cloud and reduce latency.
In-Bound Pre-Processing	Surrogate pre-processes the data that is retrieved or pushed from the enterprise cloud such that the mobile device receives data that is ready to be consumed, or filtered such that it only receives data of interest or relevance.
Out-Bound Pre-Processing	Surrogates collect data from mobile devices and pre-process the data – clean, filter, summarize, or merge – such that the data that is sent on to the enterprise cloud is ready for consumption and serves an immediate need.
Surrogate Provisioning	
Pre-Provisioned Surrogate	Surrogates are provisioned before their deployment with the capabilities that are offloaded by mobile clients.
Surrogate Provisioning from the Mobile Device	The mobile device sends the offloaded computation to the surrogate at runtime. The surrogate installs the computation inside an execution container.
Surrogate Provisioning from the Cloud	At runtime, the mobile device sends the location of the offloaded computation in the form of a URL for the surrogate to download and install.
Surrogate Discovery	
Local Surrogate Directory	Mobile devices maintain a list of 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.
Cloud Surrogate Directory	A centralized surrogate directory is maintained in the cloud and queried by mobile devices at runtime. All surrogate metadata is populated and updated in this central repository.
Intermediary Cloud Surrogate Directory	The cloud surrogate directory does not return the surrogate address to the mobile device, but rather forwards the offload request to a selected surrogate and then returns the results to the mobile device, therefore acting as an intermediary.
Surrogate Broadcast	Surrogates advertise their availability and selected metadata to mobile devices for discovery.

Apache Hadoop software framework. Our work uses a case study protocol similar to these studies, but our focus on cyber-foraging systems is novel. Finally, Koziolek et al. [35] propose an automated approach guided by architectural tactics to search the design space and help architects make decisions informed by quality tradeoffs. While they focus on providing guidance for tactics selection, we focus on how tactics can be integrated in architecture design and implementation to effectively realize the target functional and non-functional requirements.

Another area of related work is case studies and empirical experiments that analyze system qualities that are highly-relevant to cyber-foraging systems, such as energy efficiency. Jagroep et al. [36] developed a software energy profiling method and validated it via an empirical experiment on two consecutive releases of a commercial software product. Procaccianti et al. [37] developed a set of green architectural tactics for the cloud, and then empirically studied the energy impact of two best practices for energy-efficient software based on the identified tactics by applying them on MySQL Server and Apache Webserver [38]. These are just two examples of a large amount of work in this area. Our work illustrates that the benefits of cyber-foraging go beyond just energy efficiency.

Table A.6
List of non-functional architectural tactics.

Resource Optimization	
Runtime Partitioning	Mobile devices 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.
User-Guided Runtime Partitioning	Mobile devices make runtime offload decisions based on user preferences or input regarding what to optimize.
Runtime Profiling	Once the offload operation ends, or periodically, the system updates the profiling data and models that are used by the optimization functions to account for current operational conditions.
Resource-Adapted Computation	Mobile devices and surrogates have different versions of offloadable elements that match their resource characteristics, depending on whether code executes locally or remotely.
Resource-Adapted Input	Mobile devices and surrogates have identical versions of offloadable elements, but what varies is the input parameters depending on whether code executes locally or remotely. The assumption is that different input parameters will lead to different resource consumption.
Fault Tolerance	
Local Fallback	Mobile devices can revert to execution of the local copy of the offloadable computation in case the connectivity to the surrogate is lost.
Opportunistic Mobile-Surrogate Data Synchronization	Data is synchronized between mobile devices and surrogates during periods of connection such that data-reliant systems can continue operating in periods of disconnection.
Opportunistic Surrogate-Cloud Data Synchronization	Data is synchronized between mobile devices and the cloud during periods of connection such that data-reliant systems can continue operating in periods of disconnection.
Cached Results	Results are cached on the surrogate until the mobile device is able to reconnect.
Client-Side Data Caching	Data collected by a mobile client is cached on the mobile device and sent to the surrogate upon connection or re-connection.
Alternate Communications	Systems can switch to an alternate, potentially less energy-efficient communications mechanism, to continue serving mobile users in spite of disconnection (even if in a degraded mode due to less amount of information or less timely responses).
Eager Migration	Surrogates migrate offloaded computation to another connected surrogate when they detect that they might not be able to continue serving the mobile device that generated the offload request.
Lazy Migration	Surrogates retain execution of offloaded computation when they detect that they might not be able to continue serving the mobile device that generated the offload request, but the interaction with the mobile device is handed off to another connected surrogate. This means that all interaction between the mobile device and the original surrogate goes through the new surrogate that acts as an intermediary.
Scalability and Elasticity	
Just-in-Time Containers	A container and/or an instance of the offloaded code is created upon receipt of an offload request and then destroyed when the offload request is completed.
Right-Sized Containers	A container is created on the surrogate for the offloaded code that is of the smallest size possible in order to run the offloaded computation, based on computation requirements metadata related to the offloaded code, in order to optimize resource usage on the surrogate.
Dynamically-Sized Containers	If an error occurs at runtime that would indicate that the container does not have the necessary computing power for the offloaded computation, a new container is started and the offload request is moved to the new container.
Surrogate Load Balancing	Surrogates can send offloaded computation or data to other less-loaded, connected surrogates in order to provide a better user experience to mobile devices.
Security	
Trusted Surrogates	Surrogates provide credentials to mobile devices, and mobile devices provide credentials to surrogates, that aim to guarantee a trusted execution environment and interaction.

8. Conclusions and next steps

This article presented the results of three case studies to validate the architectural tactics for cyber-foraging documented in [10]. For the Tactical Cloudlets and GigaSight case studies, focusing on existing computation offload system and a data staging system, respectively, we addressed the following two research questions:

RQ1: Which of the architectural tactics for cyber-foraging can be identified in the system? The analysis of the **Tactical Cloudlets** system resulted in the identification of four architectural tactics for computation offload, cloudlet provisioning, cloudlet discovery and scalability/elasticity. In addition, elements of the Pre-Provisioned Surrogate tactic were also used to meet cloudlet management and ease of deployment and re-deployment requirements. The analysis of the **GigaSight** system resulted in the identification of four architectural tactics for data staging, cloudlet provisioning, cloudlet discovery and fault tolerance. In addition, elements of these tactics were also used to meet energy efficiency requirements as well as privacy requirements. Scalability requirements were met by a combination of tactics plus the selection of virtual machines as containers for data processing applications.

In addition, several gaps were identified that show that there is great potential to further extend the tactics catalog as more operational cyber-foraging systems are developed and evaluated.

RQ2: How do the implemented tactics support their intended functional and non-functional requirements? System testing and data collection show that the implemented tactics meet their intended functional and non-functional requirements. As indicated by the developers of the **Tactical Cloudlets** system, a catalog of architectural tactics would have been useful not only to discover ways to implement system requirements, but also to identify aspects of the system that had not been considered. Similarly, as indicated by the developers of the **GigaSight** system, a catalog of architectural tactics would definitely be an asset for the development of real cyber-foraging systems.

For the **AgroTempus** case study, focusing on the development of a cyber-foraging system for computation offload and data staging, we addressed the following research questions:

RQ3: Which of the architectural tactics for cyber-foraging can be used in the development of the system to fulfill its functional and non-functional requirements? The analysis of the AgroTempus system resulted in the identification of eight architectural tactics, seven of which were implemented in the system.

One tactic had to be replaced due to technology constraints. In addition, elements of these tactics were also used to meet energy efficiency, ease of deployment, and performance requirements. The recovery requirement was implemented via a mechanism that could easily be codified as a new tactic, especially applicable to cyber-foraging systems in resource-constrained environments. In addition, several tactic variations were identified.

RQ4: How do the selected tactics support their intended functional and non-functional requirements? System testing shows that the implemented tactics meet their intended functional and non-functional requirements. As indicated by the developer of the AgroTempus system, the architectural tactics constituted a strong foundation for the development of the system. In summary, the results of these three case studies show that there is potential for taking a tactics-driven approach to fulfill functional and non-functional requirements for cyber-foraging systems.

Continuation of this work has resulted in the characterization of usage contexts for cyber-foraging [39] as well as a decision model for cyber-foraging systems [40]. Next steps include a complementary quantitative component of the decision model to support for quantitative analysis of the impact of tactics selection, to more clearly understand the tradeoffs. As an example, we have started work to quantify the energy efficiency, bandwidth efficiency, and maintainability associated to the different tactics for surrogate provisioning (pre-provisioning, provisioning from the mobile device, and provisioning from the cloud).

Acknowledgment

We would like to thank Reuel Brion for his contribution to the AgroTempus case study.

Appendix. Architectural tactics for cyber-foraging

The complete catalog of architectural tactics is shown in Fig. A.13. 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. A short description of each functional tactic is provided in Table A.5, and of each non-functional tactic in Table A.6.

References

[1] M. Satyanarayanan, Pervasive computing: vision and challenges, IEEE Personal Commun. 8 (4) (2001) 10–17.
[2] G. Lewis, P. Lago, G. Procaccianti, Architecture strategies for cyber-foraging: Preliminary results from a systematic literature review, in: P. Avgeriou, U. Zdun (Eds.), Proceedings of the 8th European Conference on Software Architecture (ECSA 2014), in: Lecture Notes in Computer Science, vol. 8627, Springer International Publishing, 2014, pp. 154–169.
[3] G. Lewis, P. Lago, Architectural tactics for cyber-foraging: Results of a systematic literature review, J. Syst. Softw. 107 (2015) 158–186.
[4] G.A. Lewis, P. Lago, A Catalog of architectural tactics for cyber-foraging, in: Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, in: QoSA '15, ACM, New York, NY, USA, 2015, pp. 53–62.
[5] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, third ed., Addison-Wesley, 2012.
[6] S. Echeverria, G.A. Lewis, J. Root, B. Bradshaw, Cyber-foraging for improving survivability of mobile systems, in: 2015 IEEE Military Communications Conference (MILCOM 2015), 2015, pp. 1421–1426.
[7] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, M. Satyanarayanan, Scalable crowd-sourcing of video from mobile devices, in: Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services, in: MobiSys '13, ACM, New York, NY, USA, 2013, pp. 139–152.
[8] P. Brereton, B. Kitchenham, D. Budgen, Z. Li, Using a protocol template for case study planning, in: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering, 2008.
[9] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science & Business Media, 2012.
[10] G. Lewis, S. Echeverria, S. Simanta, J. Root, B. Bradshaw, Cloudlet-based cyber-foraging in resource-limited environments, Emerg. Res. Cloud Distrib. Comput. Syst. (2015) 92–121.
[11] G.A. Lewis, S. Echeverría, S. Simanta, B. Bradshaw, J. Root, Cloudlet-based cyber-foraging for mobile systems in resource-constrained edge environments, in: Companion Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 412–415.
[12] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, B. Amos, Edge analytics in the internet of things, IEEE Pervasive Comput. (2) (2015) 24–31.
[13] G.A. Lewis, Software Architecture Strategies for Cyber-Foraging Systems (Ph.D. thesis), Vrije Universiteit Amsterdam, 2016.
[14] R. Jithin, P. Chandran, Virtual machine isolation, in: Recent Trends in Computer Networks and Distributed Systems Security, in: Communications in Computer and Information Science, vol. 420, Springer Berlin Heidelberg, 2014, pp. 91–102.
[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Oftware, Pearson Education, 1994.
[16] Ericsson AB, Ericsson Mobility Report Appendix, Sub-Saharan Africa, Ericsson AB, 2013, URL http://www.ericsson.com/res/docs/2013/emr-nov13-rssa.pdf.
[17] G. Zhang, From feature phones to smartphones, the road ahead, GSMA Intelligence, 2015, URL https://goo.gl/2J1e67.
[18] A.S. Pentland, R. Fletcher, A. Hasson, Daknet: Rethinking connectivity in developing nations, Computer 37 (1) (2004) 78–83.
[19] M. Charlaganov, P. Cudré-Mauroux, C. Dinu, C. Guéret, M. Grund, T. Macicas, The Entity Registry System: Implementing 5-Star Linked Data Without the Web, arXiv preprint arXiv:1308.3357.
[20] Mozilla Developer Network, Firefox OS, 2015, URL https://developer.mozilla.org/en-US/docs/Mozilla/Firefox_OS.
[21] Raspbian, Welcome to Raspbian, 2015, URL https://www.raspbian.org/.
[22] The jQuery Foundation, jQuery, 2015, URL https://jquery.com/.
[23] Object Refinery Limited, JFreeChart, 2014, URL http://www.jfree.org/jfreechart/.
[24] The Apache Software Foundation, Apache Commons Codec, 2014, URL https://goo.gl/6HZLrZ.
[25] JSON.org, Introducing JSON, 2015, URL http://www.json.org/.
[26] OWM Inc., OpenWeatherMap, 2015 URL http://openweathermap.org/.
[27] Y. Fang, C. Nokleberg, D. Hughes, JSON.simple - A simple Java toolkit for JSON - Google Project Hosting, 2012, URL https://code.google.com/p/json-simple/.
[28] Mozilla Developer Network, IndexedDB, 2016, URL https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API.
[29] K. Kumar, Y.H. Lu, Cloud computing for mobile users: Can offloading computation save energy?, Computer 43 (4) (2010) 51–56.
[30] ZTE Corporation, ZTE Open C, 2015, URL http://www.ztedevice.com/product/b8fcc021-3e5b-4a96-8161-47ee8d4f42f1.html.
[31] Raspberry Pi Foundation, Raspberry Pi 2 Model B, 2015, URL https://www.raspberrypi.org/products/raspberry-pi-2-model-b/.
[32] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empir. Softw. Eng. 14 (2) (2009) 131–164.
[33] D. Gesvindr, B. Buhnova, Architectural tactics for the design of efficient paas cloud applications, in: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA 2016), 2016, pp. 158–167.
[34] M. Mirakhorli, Y. Shin, J. Cleland-Huang, M. Cinar, A tactic-centric approach for automating traceability of quality concerns, in: Proceedings of the 34th International Conference on Software Engineering, in: ICSE, IEEE Press, Piscataway, NJ, USA, 2012, pp. 639–649.
[35] A. Koziolek, H. Koziolek, R. Reussner, Peropteryx: Automated application of tactics in multi-objective software architecture optimization, in: Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS, in: QoSA-ISARCS, ACM, 2011, pp. 33–42.
[36] E.A. Jagroep, J.M.V.D. Werf, G. Procaccianti, P. Lago, S. Brinkkemper, C. Leen Blom, R. van Vliet, Software energy profiling: Comparing releases of a software product, in: Companion Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016.
[37] G. Procaccianti, P. Lago, G.A. Lewis, A catalogue of green architectural tactics for the cloud, in: Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2014 IEEE 8th International Symposium on the, IEEE, 2014, pp. 29–36.
[38] G. Procaccianti, H. Fernández, P. Lago, Empirical evaluation of two best practices for energy-efficient software development, J. Syst. Softw. 117 (2016) 185–198.
[39] G.A. Lewis, P. Lago, Characterization of cyber-foraging usage contexts, in: D. Weyns, R. Mirandola, I. Crnkovic (Eds.), Software Architecture, in: Lecture Notes in Computer Science, vol. 9278, Springer International Publishing, 2015, pp. 195–211.
[40] G.A. Lewis, P. Lago, P. Avgeriou, A decision model for cyber-foraging systems, in: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA 2016), IEEE, 2016, pp. 51–60.

Grace Lewis is a Principal Researcher at the Software Engineering Institute (SEI) at Carnegie Mellon University, where she is the principal investigator for the "Authentication and Authorization of IoT Devices in Edge Environments" and "High Assurance Software-Defined IoT Security" research projects. She also leads the Tactical Cloudlets work. Her current interests and projects are in edge computing, cloud computing, security, software architecture, and emerging technologies. She has a B.Sc. in Software Systems Engineering and a Specialization in Administration from Icesi University in Colombia; a Master in Software Engineering from Carnegie Mellon University; and a Ph.D. in Computer Science from Vrije Universiteit Amsterdam.

Patricia Lago is professor in software engineering at the Vrije Universiteit Amsterdam, where she leads the Software and Services (S2) research group. Her research is in software architecture and software quality with a special emphasis on sustainability. She has a Ph.D. in Control and Computer Engineering from Politecnico di Torino and a Master in Computer Science from the University of Pisa, both in Italy. She is a member of the Steering Committees of IEEE ICSA, ECSA and the ICT4S conference series, member of the IFIP 2.10 Working group on Software Architecture and the IFIP 2.14 Working group on Services-based Systems.

Sebastián Echeverría is a member of the technical staff at the Software Engineering Institute (SEI) at Carnegie Mellon University. He has worked for several years on projects related to Tactical Cloudlets, and is currently working on IoT security projects. Echeverria has over 8 years of professional software development experience, working in the commercial software industry. His main areas of expertise include mobile computing, wireless networks, distributed systems architecture, and cloud computing. He has a B.Sc. and a M.Sc. in Computer Sciences from Universidad Catolica de Chile, and a Master in Software Engineering from Carnegie Mellon University.

Pieter Simoens is a professor at Ghent University, Belgium and affiliated with imec. His research centers around the theme of providing intelligent services on distributed edge infrastructure. Current lines of research fog computing, the Internet-of-Robotic-Things, neuromorphic computing and resource-efficient deep learning. In these fields, he is author and co-author of approx. 100 papers published in international journals or in the proceedings of international conferences.