A Reference Architecture for Mobile Code Offload in Hostile Environments

Soumya Simanta[1], Kiryong Ha[2], Grace Lewis[1],
Ed Morris[1], and Mahadev Satyanarayanan[2]

[1] Software Engineering Institute
Pittsburgh, PA USA
[2] School of Computer Science
Carnegie Mellon University
Pittsburgh, PA USA
{ssimanta,glewis,ejm}@sei.cmu.edu,
{krha,satya}@cs.cmu.edu

Abstract. Handheld mobile technology is reaching first responders and soldiers in the field to aid in various tasks such as speech and image recognition, natural language processing, decision making, and mission planning. However, these applications are computation-intensive and we must consider that 1) mobile devices offer less computational power than a conventional desktop or server computer, 2) computation-intensive tasks take a heavy toll on battery power, and 3) networks in hostile environments such as those experienced by first responders and soldiers in the field are often unreliable and bandwidth is limited and inconsistent. While there has been considerable research in code offload to the cloud to enhance computation and battery life, most of this work assumes reliable connectivity between the mobile device and the cloud—an invalid assumption in hostile environments. This paper presents a reference architecture for mobile devices that exploits cloudlets—VM-based code offload elements that are in single-hop proximity to the mobile devices that they serve. Two implementations of this reference architecture are presented, along with an analysis of architecture tradeoffs.

Keywords: reference architecture, mobile architecture, mobile systems, code offload, virtual machines, cloud computing.

1 Introduction

The number of apps and mobile devices created specifically for use in hostile environments, such as those experienced by first responders operating in crisis situations and military personnel, continues to grow [1][2][3]. First responders and soldiers can use handheld devices to help with tasks such as speech and image recognition, natural language processing, decision making and mission planning. However, several obstacles impede achieving the needed capabilities. First, mobile devices offer less computational power than conventional desktop or server computers, and are therefore not an ideal computation platform for tasks such as natural language

D. Uhler, K. Mehta, and J.L. Wong (Eds.): MobiCase 2012, LNICST 110, pp. 274–293, 2013.

processing and complex decision making. Second, computation-intensive tasks, such as image recognition or even global positioning systems (GPS), take a heavy toll on battery power. Third, networks in hostile environments are often unreliable and bandwidth is limited and inconsistent [4].

To overcome some of these challenges, cyber-foraging is employed to leverage external resources to augment the capabilities of resource-limited mobile devices [5][6][7][8][9][10][11][12][13]. However, many cyber-foraging strategies rely on the conventional internet or on environments that tightly couple handheld applications and servers on which code is offloaded (see Section 6). Cyber-foraging therefore addresses the challenges of conserving battery power and limited computing power, but does not address the challenges of unreliable networks and dynamic environments.

This report presents a strategy to overcome the challenges of cyber-foraging for mobile platforms in hostile environments by using *cloudlets* — discoverable, localized, stateless servers running one or more virtual machines (VMs) on which mobile devices can offload expensive computation. Cloudlets enhance processing capacity and conserve battery power, and simultaneously provide ease of deployment in the field.

2 Cloudlets as Intermediate Offload Elements

Code offload from mobile devices to cloud environments is the topic of much recent research [14][15][16][17][18][19][20]. This is also an approach used commercially by applications such as Siri for voice recognition [21]. However, an underlying assumption in these approaches is connectivity to the cloud, which is not always available or reliable in hostile environments.

A high-level architecture for code offload in hostile environments is proposed in [22] and presented in Figure 1. This architecture inserts an intermediate layer between the central core (i.e., enterprise cloud) and the mobile devices. At the heart of this architecture is a large centralized core that could be implemented as one of Amazon's data centers or a private enterprise cloud. At the edges of this architecture are offload elements for mobile devices. These elements, or cloudlets, are dispersed and located close to the mobile devices they serve [23]. This architecture decreases latency by using a single-hop network and potentially lowers battery consumption by using WiFi or short-range radio instead of broadband wireless which typically consumes more energy [24][25].

A key attribute of this architecture is that the offload elements are stateless. A mobile device does not need to communicate with the core during an offload operation; it only needs to communicate with its closest offload element. Communication between the offload element and the core is only needed during setup and provisioning. Once an offload element is provisioned, it can work in disconnected mode. Adding or replacing an offload element involves little setup or configuration effort.

One approach to offload is VM synthesis [23][26]. In this approach, an application overlay is offloaded from the mobile device to a cloudlet. An application overlay represents the difference between a base VM with only an operating system installed and the same VM with the application installed. In effect, the mobile device becomes the vector by which the needed application is deployed to the field. This approach

takes advantage of properties of VMs that reduce hardware dependencies—the same VMs can operate on several hardware platforms, and a single hardware platform can support multiple VMs—to provide flexibility in highly volatile environments where it is difficult to assure the right hardware or OS platform for cyber-foraging.

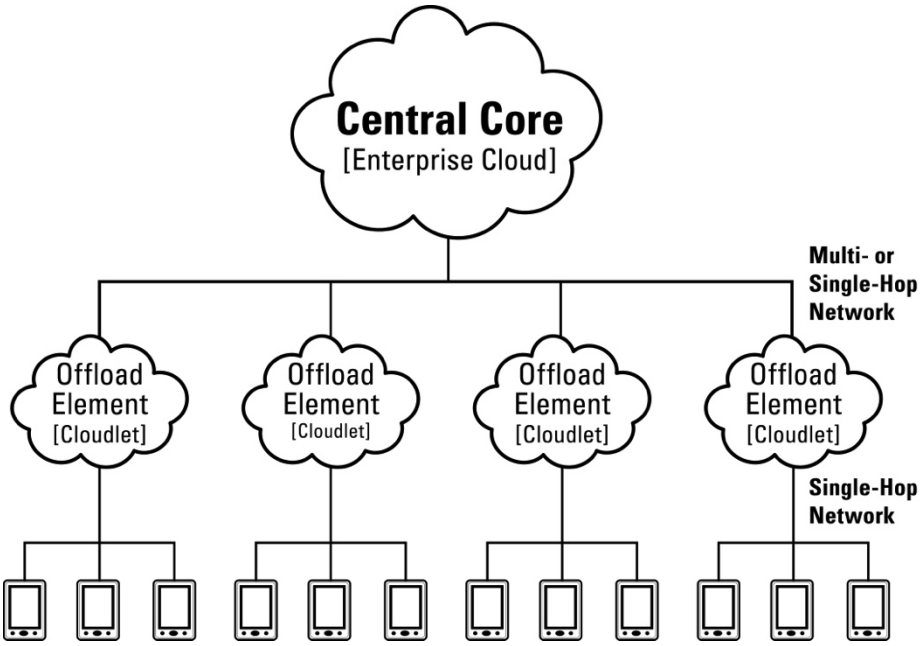

Fig. 1. Three-tier architecture for code offload

An application overlay is created once per application, as described in Figure 2. The Base VM is a VM disk image file that is obtained from the Central Core and saved to a cloudlet that runs a VM manager compatible with the Base VM. The VM manager starts the Base VM and the application is installed. After installation, the VM is shut down. A copy of the modified Base VM Disk Image is saved as the Complete VM Disk Image. The Application Overlay is calculated as the binary diff (VCDIFF rfc3284) between the Complete VM Disk Image and the Base VM Disk Image. The Base VM is then deployed to any platform that will serve as a cloudlet. The cloudlet may support multiple VMs, thereby reducing hardware dependencies on the offload element, and between the offload element and mobile device. A mobile device carrying application overlays will be able to execute these applications on any cloudlet that has the corresponding Base VM.

VM synthesis is particularly useful in hostile environments characterized by unreliable networks, loss of cyber foraging platforms, and a need for rapid deployment. For example, imagine a scenario where a first responder must execute a computation-intensive app configured to work with cloudlets. At runtime, the app discovers a

nearby cloudlet located in a rescue camp and offloads the computation-intensive portion of code to it. However, due to a natural disaster, loss of network connectivity, or exhaustion of energy sources on the cloudlet, the mobile app is disconnected from the cloudlet. The mobile app can locate a different cloudlet and have the app running in a short time, with no need for any configuration on the app or the cloudlet. This runtime flexibility enables the use of opportunistically available resources, replacement of lost cyber-foraging resources, and dynamic customization of newly-acquired cyber-foraging resources.

The following sections present the reference architecture for code offload in hostile environments and details of two prototype implementations.

3 Reference Architecture

Hostile environments are characterized by uncertainty in available resources such as computational capability and bandwidth. In addition, many applications that are useful in these environments are computation-intensive; these include face recognition, natural-language processing, route calculation, and text recognition, all of which require some form of input from sensors on the device. Soldiers and first responders executing missions are often away from their bases for many hours and cannot afford to carry many extra batteries. Therefore, a solution for code offload in hostile environments must consider 1) native apps that exploit device sensors, 2) code offload elements that can be quickly configured and deployed, and 3) battery consumption on the mobile device.

Figure 3 presents a reference architecture for mobile devices that exploit cloudlets for code offload. The major components of this architecture are the *Cloudlet Host* and the *Mobile Client.*

The *Cloudlet Host* is a physical server that hosts 1) a discovery service that broadcasts the cloudlet IP address and port to allow mobile devices to find it 2) the Base VM Image that is used for VM synthesis 3) a Cloudlet Server that handles code offload in the form of application overlays, performs VM synthesis and starts guest VM instances with the resulting VM images, and 4) a VM Manager that serves as a host for all guest VM instances that contain the computation-intensive server component of the corresponding mobile app.

The Mobile Client is a handheld or wearable device that hosts 1) the Cloudlet Client app that discovers cloudlets and uploads application overlays to the cloudlet and 2) a set of Cloudlet-Ready Apps that operate as clients of the server code running in the cloudlet. The Mobile Client stores an application overlay for each cloudlet-ready app that a user would conceivably want to execute and for which computation offloading is appropriate. Each application overlay is generated from the same Base VM Image that resides in the cloudlet. In an operational setting, these Base VM Images could be retrieved from the central core shown in Figure 1.

To validate the feasibility of the proposed reference architecture for hostile environments, we constructed a prototype as described in the next section.

4 Initial Prototype

The initial prototype is an implementation of a face recognition application in which the client is an Android app and a cloudlet-based server that contains computation-intensive code that performs face recognition. The client locates a cloudlet via a discovery protocol, sends the application overlay (Face Recognition Server code) to the cloudlet for VM synthesis and captures images and sends them to the Face Recognition Server on the cloudlet. This initial prototype was created based on the reference architecture presented in Figure 3. The prototype architecture is shown in Figure 4.

4.1 Base VM Image Creation

The Base VM Image for this prototype is a 3GB image with Microsoft XP plus the necessary system updates. To keep the image size small, "system restore" was turned off and unnecessary system components were removed using the DiskCleanup utility. Additional components were included in the Base VM Image to enable communication between the Guest VM and the *Cloudlet Server*. This additional complexity was one of the reasons for revising the initial prototype (see Section 5).

A major difference between the prototype and the cloudlet work presented in [23] is the type of client involved. Satyanaranyan's work uses a virtual network computing (VNC) client that acts as a remote desktop for the VM [27]. In our prototype, the Face Recognition Client is a rich client (native app) that interacts with a Face Recognition Server inside the VM. The Face Recognition Client requires the IP address and port of the Face Recognition server to establish a network connection. However, the Cloudlet Server does not directly know the IP address and port of the synthesized VM because the IP address is assigned by the Dynamic Host Configuration Protocol (DHCP) server executing in bridged mode and the VM host has no visibility into that assignment. The Cloudlet Client uses an HTTP request to start the cloudlet setup, and expects an HTTP response from the Cloudlet Server that contains the IP address and port of the Face Recognition Server.

To solve this problem, a CloudletStartup Windows service (implemented in Python) is included in the Guest VM that performs three functions:

1. starts the Face Recognition Server in a separate thread
2. reads the IP address and port from the Face Recognition Server configuration and communicates it to the Cloudlet Server in an HTTP POST
3. sends a periodic heartbeat to the Cloudlet Server in an HTTP POST

After completion of these steps, the VM was shut down and the resulting image file was saved as the Base VM Image.

4.2 Tools for Overlay Creation

The overlay for the face recognition application was created by following the steps presented in Figure 2. The VM Manager was started using the Base VM Image, the Face Recognition Server was installed, and the VM shut down. The specific tools used to "Calculate Diff between Complete and Base VM Image" are

- xdelta3: open-source binary diff tool that generates a file as the difference between the Base VM Disk Image and the Complete VM Disk Image [29]

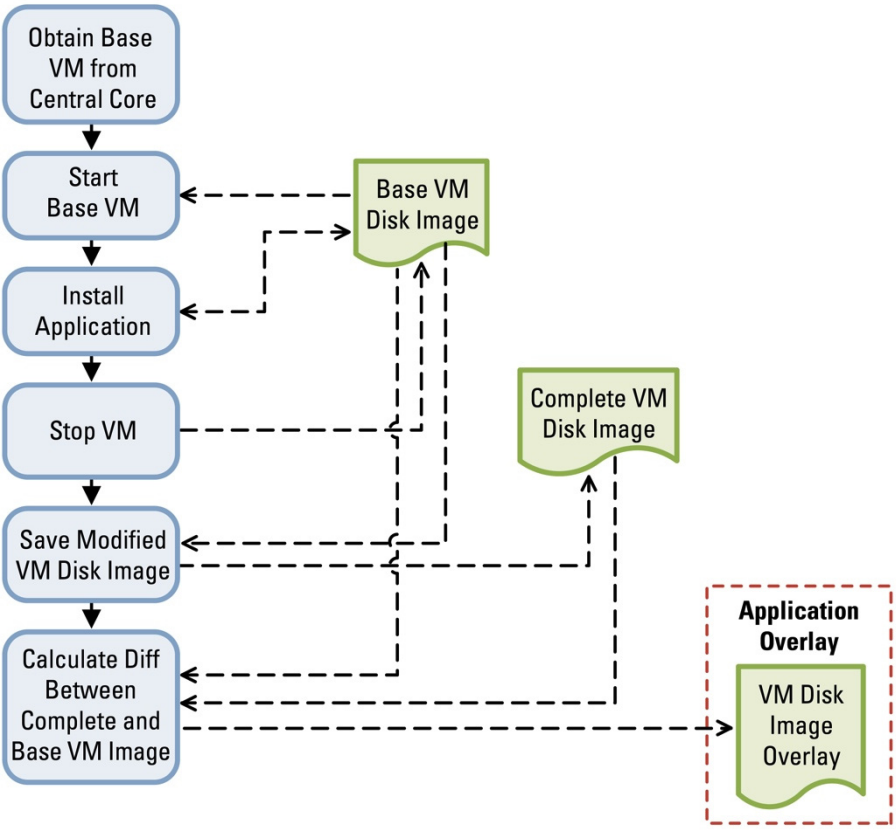

Fig. 2. Application overlay creation process

- lzma: set of public-domain libraries and tools that compress the diff file using the Lempel-Ziv-Markov chain data-compression algorithm [30]
- OpenSSL: open-source SSL (Secure Sockets Layer) implementation that encrypts the compressed file [31]

The resulting file is the application overlay. The Cloudlet Server uses these same tools in reverse order when it performs VM synthesis.

4.3 Cloudlet Host

The Cloudlet Host is an Ubuntu 10.10 Linux server that hosts the following sub-components.

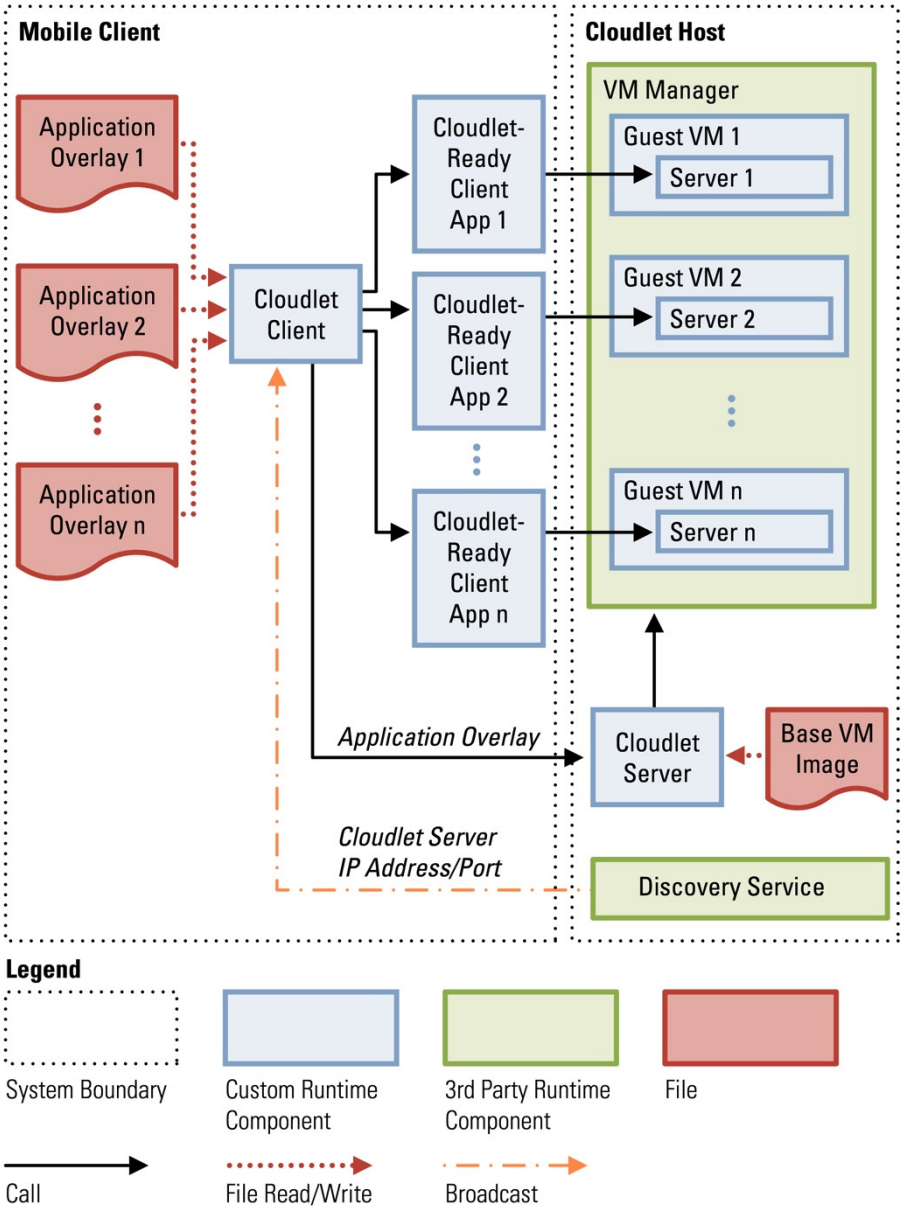

Fig. 3. Reference architecture for cloudlet-based code offload

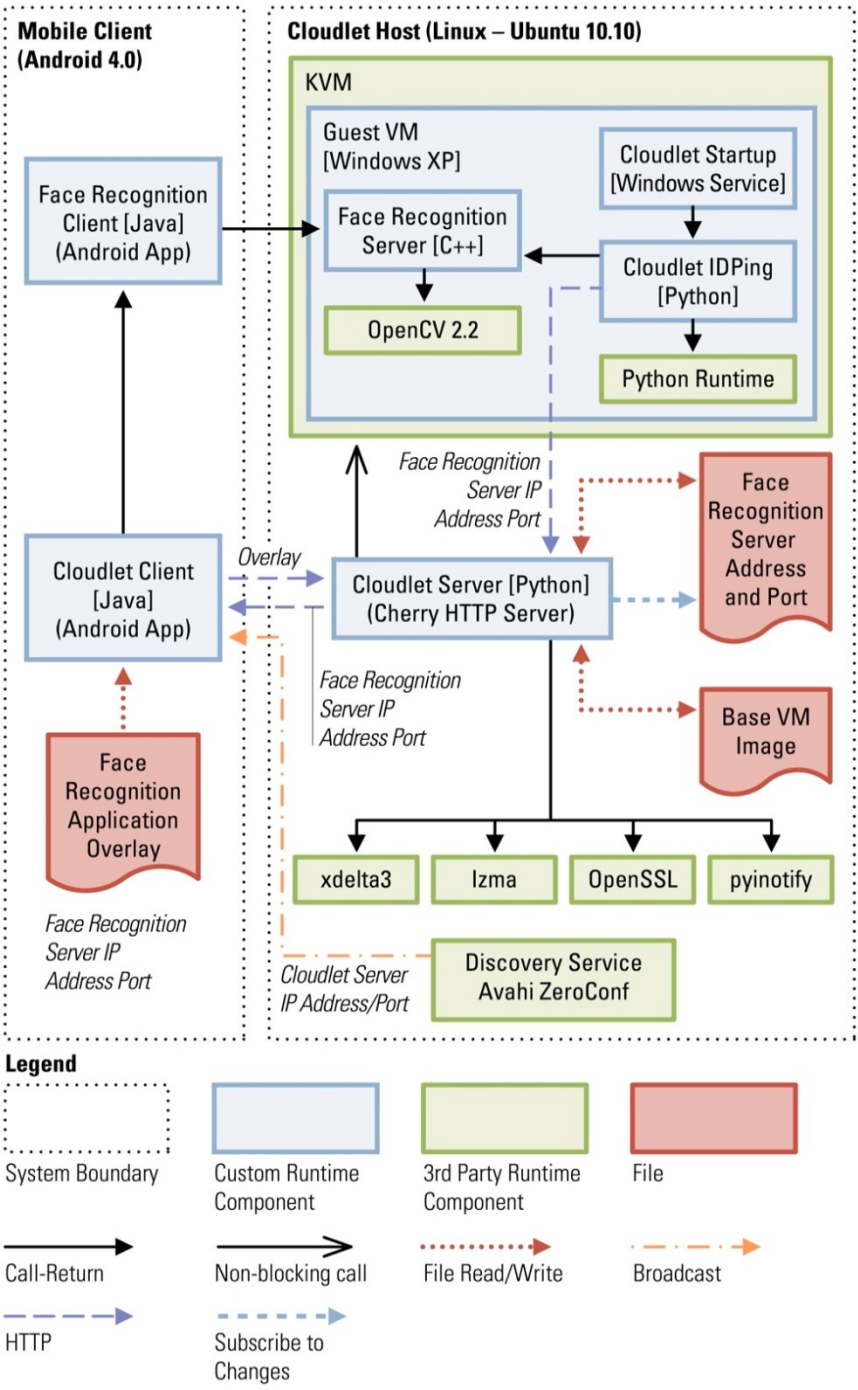

Fig. 4. Architecture for initial prototype

Kernel-Based Virtual Machine (KVM)

The virtualization infrastructure used in the prototype is KVM [32]. KVM is open-source and has good community support. KVM runs a Guest VM for each offloaded application. The Guest VM for the face recognition application is Windows XP, but KVM supports most popular operating systems [33].

Cloudlet Server

The core of the Cloudlet Server is an HTTP Server implemented using CherryPy, an extensible HTTP server [34]. The Cloudlet Client sends the encrypted and compressed overlay as an HTTP POST request. Upon receipt of the overlay, the overlay is decrypted and decompressed using the tools listed in Section B. VM synthesis is performed by using xdelta3 to apply the overlay to the Base VM Image and create the VM Image that contains the Face Recognition Server. The synthesized VM is started in bridged network mode, so that the Guest VM has a unique network-accessible IP address [35] and waits for notification that the Guest VM has successfully started. The Cloudlet Server uses pyinotify—a Python file system change monitoring utility—to subscribe to changes in the face recognition server IP address and port file [36]. On startup, the Guest VM executes the CloudletStartup Windows service (see Section A) to communicate the IP address and port back to the Cloudlet Server where another thread writes the IP and port information to a file. As soon as the file changes, the waiting thread in Step 3 receives notification. It reads the file and sends a response to the Cloudlet Client that contains the IP address and port on which the Face Recognition Server will be listening.

Discovery Service

The Discovery Service is based on the Avahi implementation of Zero Configuration Networking (ZeroConf) [37]. Zeroconf is a local network-discovery protocol for creating an IP network [38]. The Discovery Service broadcasts the Cloudlet Server IP address and port.

4.4 Cloudlet Client

The Cloudlet Client is an Android app that

1. Discovers cloudlets through information broadcast by the Discovery Service residing in the Cloudlet Host
2. Creates a HTTP connection to the Cloudlet Server for overlay transmission and uploads the overlay
3. Obtains the IP address and port that the Face Recognition Server is listening on from the Cloudlet server
4. Communicates the IP address and port of the Face Recognition Server to the Face Recognition Client app using a shared local file
5. Launches the Face Recognition Client.

4.5 Face Recognition Client

The Face Recognition Client is an Android app that executes in client/server mode with the Face Recognition Server running in the Guest VM. On launch, the Face Recognition Client reads the local file that contains the IP address and port of the Face Recognition Server and opens a TCP/IP connection to it. Images that the camera captures are sent to the Face Recognition Server.

4.6 Face Recognition Server

The Face Recognition Server is implemented in C++ using the OpenCV image recognition library that supports training or recognition modes [39]. When in recognition mode, it returns coordinates for the recognized faces plus a measure of confidence.

4.7 Prototype Evaluation

We evaluated the prototype using the previously described face recognition application and two additional computation-intensive applications that are representative of capabilities needed in hostile environments. Each application has an Android app (client) and a server corresponding to the computation-intensive offloaded code. The server-side applications are described below.

- OBJECT: Linux C++ application based on the CMU MOPED object recognition libraries [40]
- FACE: Windows XP C++ face recognition application described in Section 0
- SPEECH: Windows XP Java application based on the CMU Sphinx-4 speech recognition toolkit [41]

In addition, an overlay corresponding to a NULL application (VM simply started and stopped) serves as a baseline for the analysis of transmission overhead and battery consumption. Table 1 displays the application data.

Table 1. Application Data for Initial Prototype

Appl.	Platform	Lang.	Appl. Size (MB)	Base VM Disk Image (MB)	Compressed VM Disk Image Overlay (MB)
OBJECT	Linux	C++	27.50	3546	165.32
FACE	Windows XP	C++	17.65	3073	43.55
SPEECH	Linux	Java	51.04	3546	176.23
NULL	Linux	N/A	N/A	3546	0.12

We conducted all experiments using the configuration shown in Figure 5. We measured energy usage with a Monsoon Solutions Power Monitor and the corresponding Power Tool software [42]. To ensure good experimental control, interactive inputs were scripted.

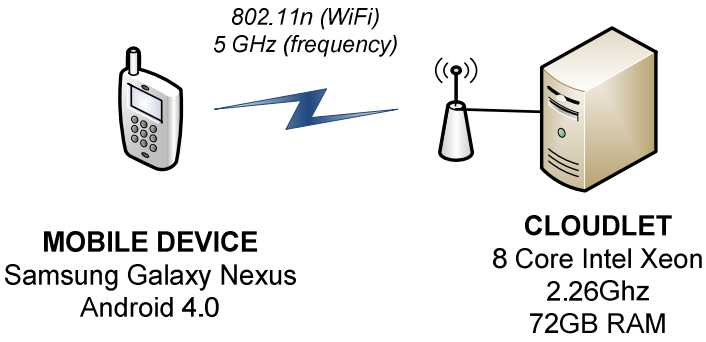

Fig. 5. Evaluation infrastructure setup

Average times for each step of the process and average energy consumption are shown in Figure 6. All times are measured from the client perspective and include HTTP Request/Response time.

The largest amount of time is consumed by the Upload Overlay operation and depends on overlay size (Figure 6). VM Synthesis and Start VM almost equally consume the second-largest time. VM Synthesis time is smaller for FACE because the sum of the base VM size and overlay size is smaller. FACE is also the outlier for Start VM because it is the only application that runs on Windows and therefore has a longer boot time. Energy consumption depends largely on overlay size. Average application ready time (time between upload overlay and Start VM) is between 101 and 166 seconds for the non-null applications. Given the dependence of these numbers on base VM image size and application overlay size, reducing the size of these files would reduce both application ready time and energy consumption. File size and implementation complexity are addressed in a revised prototype described in the next section.

5 Revised Prototype

In revising the initial prototype, our goals were first, to reduce application complexity and dependencies, and second, to decrease overall application ready time. The revised architecture shown in Figure 7 uses the same implementation of face recognition as for the initial prototype.

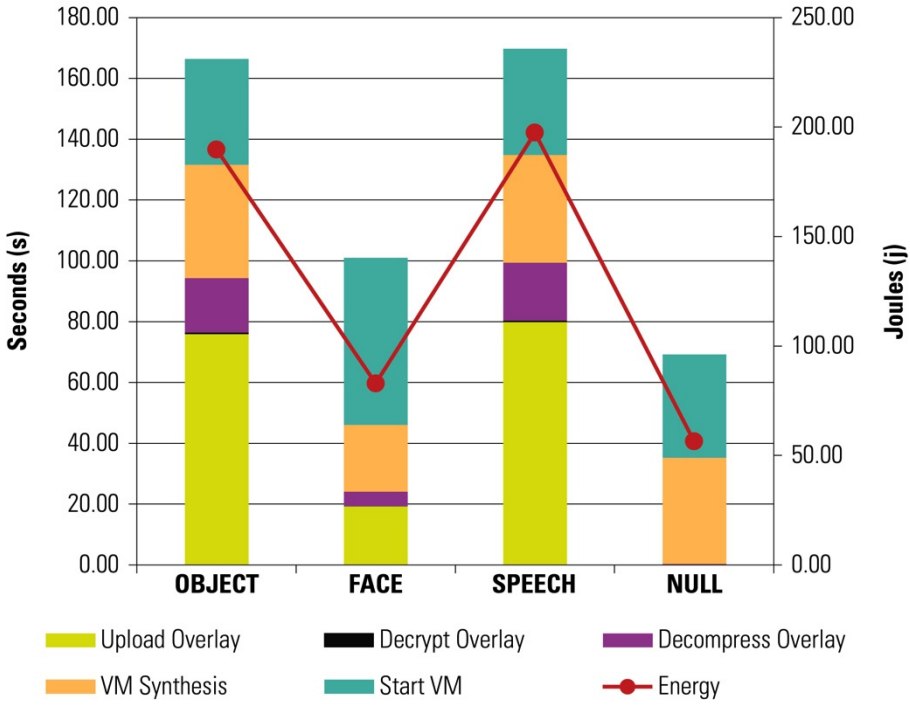

Fig. 6. Evaluation infrastructure setup

5.1 Main Changes

To address limitations of the initial prototype, we made the following changes.

Disk Image Format

The default format for KVM disk images is raw images. Storage for raw images is allocated during creation of the image. If a 3GB image is created, the VM manager writes data inside the image file that is fixed in size. This results in large pre-allocated images.

Another disk-image format that KVM supports is QEMU copy on write 2 (qcow2). The advantage of qcow2 is that storage allocation is delayed until actually needed. This means both size of the disk and overlay will be optimized. A disadvantage of qcow2 is the overhead caused by storage allocation at runtime. Reduction in the amount of time for overlay transmission made this tradeoff acceptable for the revised prototype.

Memory Snapshot Overlay Plus Disk Image Overlay

The initial prototype only transfers the disk image overlay. This means that the VM is always cold started and requires application-specific scripts to start the application and

send connection information back to the client, as explained in Section 4. In the revised prototype, a memory snapshot is also created. The three files (Base VM Disk Image, Base Memory Snapshot, and Base Disk Snapshot) now constitute the base VM image. When the VM is started, as shown in Figure 8, Base Memory Snapshot and Base Disk Snapshot are immediately applied. The application is installed, the VM is suspended, and a second set of snapshots is saved. Overlays are created as the difference between the sets of snapshots, which accounts for a very small set of overlays. While the client has to send two overlays, there are fewer base VM dependencies. In addition, because the VM starts from a suspended state instead of from a stopped state, the VM Start time is faster.

KVM in NAT Mode and Port Redirection

One of the main complexities of the initial prototype was the KVM-to-Cloudlet-Host communication explained in Section 3 that allowed the mobile device to connect directly with the Face Recognition Server inside the VM.

The revised prototype starts the synthesized VM in NAT (Network Address Translation) mode instead of bridged mode so that the Guest VM can share the same IP address as the Cloudlet Host. However, in NAT mode the Guest VM is not directly accessible from the client. When the Cloudlet Server starts the synthesized VM in NAT mode, it includes the port that the Guest VM should listen on as a parameter. The Cloudlet Server maps an externally accessible port on the Cloudlet Host to the port assigned to the Guest VM. The externally accessible port number is sent to the client and the cloudlet host redirects all communication to the Guest VM. The tradeoff is that NAT is restricted to certain protocols (e.g., HTTP, SMTP, FTP) and cannot be bound to ports numbered lower than 1024 without root privileges.

Scalability could become an issue if the port request exceeds the number of available ports. However, our revision greatly simplifies deployment because the Windows service, Python runtime, CloudletIDPing Python script, and pyinotify are no longer necessary. In addition, NAT is more secure than bridged mode because the VM is firewalled from the outside.

5.2 Evaluation of Revised Prototype

We evaluated the revised prototype using the same applications as listed in Section 4.7. Application data is shown in Table 2. Overlay size is considerably smaller, except for FACE, due to application characteristics such as language, use of DLLs, configuration files, etc.

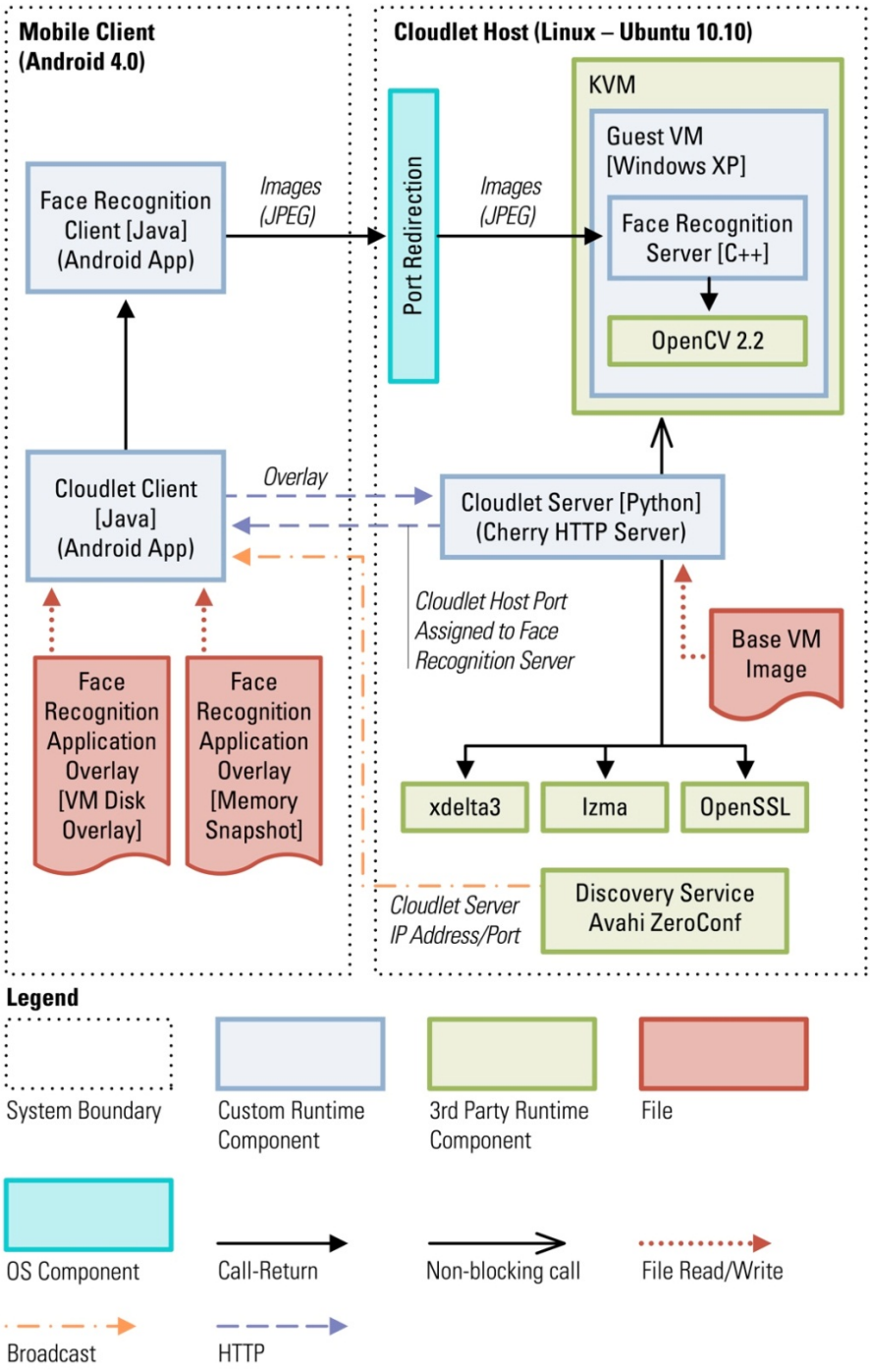

Fig. 7. Architecture for revised prototype

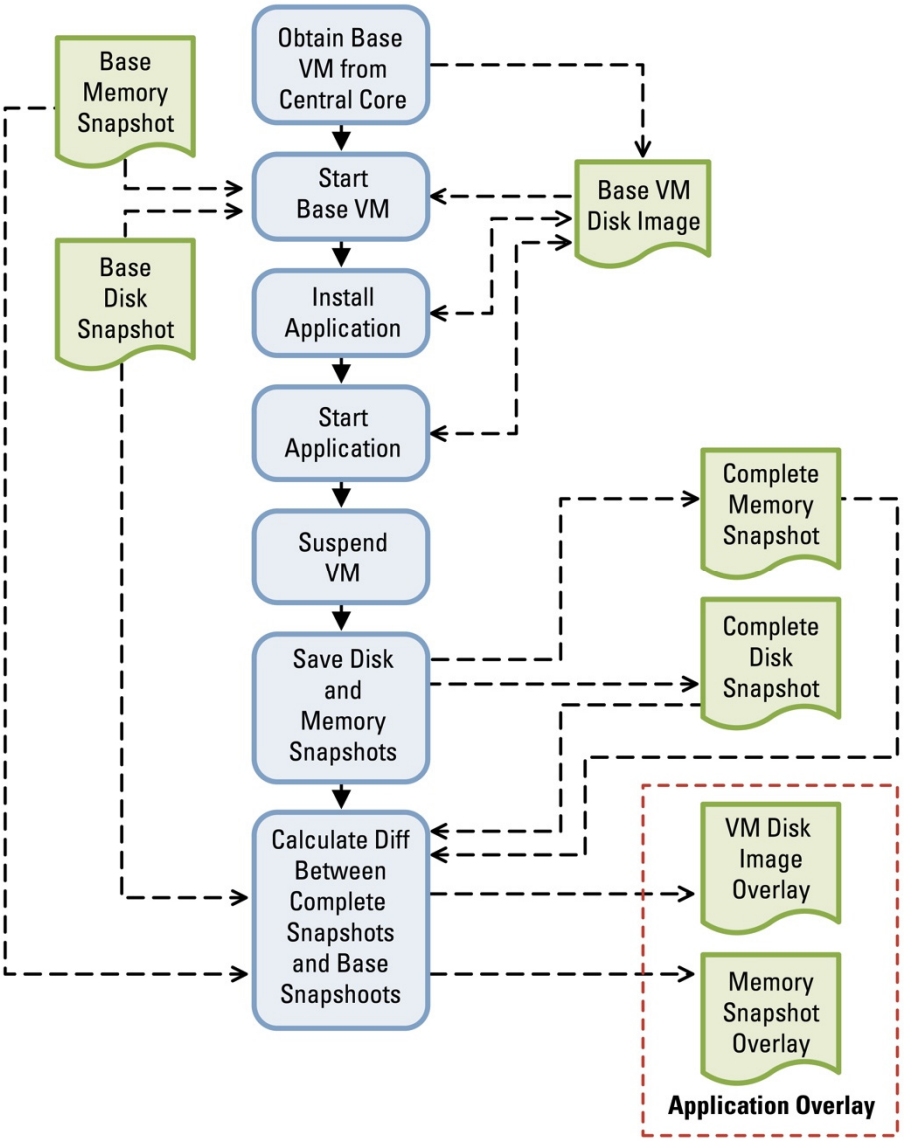

Fig. 8. Revised application overlay creation process

Table 2. Application Data for Revised Prototype

Application	Base VM Disk Image qcow2 (MB)	Base Disk Snapshot qcow2 (MB)	Base Memory Snapshot (MB)	Compressed VM Disk Image Overlay (MB)	Compressed Memory Snapshot Overlay (MB)
OBJECT	3558	17	554	94	293
FACE	2421	15	278	71	101
SPEECH	3558	17	554	86	257
NULL	3558	17	554	1	3

Average times for each step of the process and average energy consumption are shown in Figure 9.

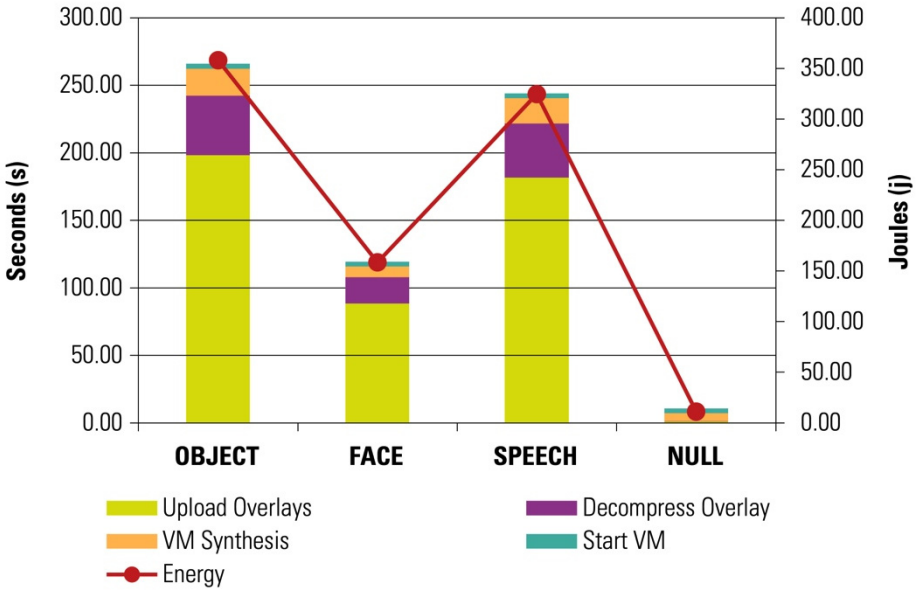

Fig. 9. Measures per application for revised prototype

Although the reduction in VM Synthesis and Start VM times are considerable with respect to the initial prototype, Upload Overlay and Decompress Overlay times are higher because of the total size of the combined overlays (disk plus memory). Consequently, energy consumption increased because of increased overlay size. However, implementation, application configuration, and VM-host-guest communication are simplified, and Start VM time is more consistent regardless of the operating system running inside the VM.

The bandwidth between the mobile device and the cloudlet during the experiments was approximately 13 Mbps, even when using 802.11n wireless. This lower-than-expected data rate may be caused by radio interference in the environment where the experiments were conducted. For the revised prototype to pay off, the efficiencies gained in VM Synthesis and Start VM would require supplementation with greater bandwidth.

6 Related Work

A considerable amount of work, conducted as early as 2001, relates to code offload from mobile devices to cloud environments [5][6][7][8][9][10][11][12][13][14][15][16][17][18][19][20]. However, this work in cyber-foraging from mobile devices assumes that acceptable networking conditions prevail between a mobile device and its offload site. Although the bandwidth and latency of this connectivity varied, a universal assumption in previous work was that connectivity was "good enough." To the best of our knowledge, our work is the first to investigate the challenges of cloud offload in hostile environments and to propose an architectural solution to the problem.

One example of closely related work is MAUI [25]. MAUI is a system that enables the fine-grained energy-aware offload of mobile code to offload elements, with minimum programmer effort—code annotations indicate methods that could be executed remotely. However, this approach is platform-specific (Microsoft .NET) and therefore would limit the applications that could be offloaded.

Another closely-related effort is CloneCloud [43]. Unlike with MAUI, applications do not require modification. This work also assumes connection to the cloud, but it could be implemented such that threads are migrated to a cloudlet instead of a cloud. However, supported platforms are limited (e.g., Java VM, Android Dalvik VM, Microsoft .NET) and deployment and hardware requirements would be difficult to achieve in some hostile environments.

Other work that establishes a foundation for cyber-foraging includes

- Goyal and Carter propose a VM-based approach in which a discoverable virtual machine server acts as a surrogate to run client application code [10]. This approach addresses security issues but requires the surrogate to be connected to the internet to locate and download client application code and also requires code to be partitioned at development time.
- Similar to this approach is the Locusts framework that enables the discovery of cyber-foraging resources (surrogate peers). Peers can offload coarse-grained tasks to other peers as well as process offloaded tasks from other peers [44]. Applications must be partitioned in advance into locally executed code and remotely executable tasks.
- Work proposed by Chen et al is not VM-based, but includes a mechanism for deciding whether to execute locally or remotely based on size of method input data and wireless channel conditions, and whether to interpret bytecode or compile native code [45]. The solution only works for Java code and applications must be modified to enable code offload.
- Kemp et al. propose an approach that leverages the Ibis high-performance distributed computing middleware [11]. The server portion is sent from the mobile device to the surrogate at runtime but applications must be written as distributed applications using the Ibis programming environment and the server.

Our work implements an instance of the cloudlet strategy presented by Satyanarayanan that uses a thick-client approach (native app) instead of a thin, VNC-based client [23]. The advantage of the VNC approach is that applications would not require any modification because they would run completely on the server. However, many applications use sensors to capture information about the environment. A better fit for our scenario involves splitting the application into a very simple native app that runs on the mobile device to capture manual or sensed input and a server portion that runs the expensive computation. The advantage of the VM-based approach is simplicity of setup and deployment that relies on generic servers running pre-configured Base VMs (cloudlets) and cloudlet-ready mobile devices loaded with overlays for computation-intensive applications. There is no need for special hardware or middleware.

7 Conclusions and Future Work

There is substantial consensus that cloud offload of resource-intensive application execution is a core technique in mobile computing. In this paper, we have described a reference architecture for code offload in hostile environments and presented two viable implementations along with architectural tradeoffs. A difficult problem exposed by this architecture involves rapid delivery of large application overlays to offload sites as well as rapid application ready time.

Current and future work includes 1) other forms of code offload, such as demand paging from the cloud, that consume less energy and provide better launch times, but require reliable communication between cloudlets and the cloud [22] 2) rapid VM synthesis, and 3) extension of the discovery protocol to enable VM caching so that overlays do not always have to be transmitted.

Acknowledgements. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

References

1. Frenzel, L.: Street-Ready Smart Phone Enhances First Responder Communications. Electronic Communications (2012), http://electronicdesign.com/article/communications/streetready-smart-phone-enhances-responder-communications-73646
2. Kozlowski, J.: Smartphone and Tablet Apps for First Responders. EMS1.com (2012), http://www.ems1.com/ems-products/communications/articles/1129715-Smartphone-and-tablet-Apps-for-first-responders/
3. Morris, E: A New Approach for Handheld Devices in the Military. Software Engineering Institute Blog (2011), http://blog.sei.cmu.edu/post.cfm/a-new-approach-for-handheld-devices-in-the-military
4. Satyanarayanan, M.: Fundamental Challenges in Mobile Computing. In: Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing, pp. 1–7. ACM, New York (1996)

5. Balan, R., Flinn, J., Satyanarayanan, M., Sinnamohideen, S., Yang, H.: The Case for Cyber Foraging. In: Proceedings of the 10th ACM SIGOPS European Workshop, pp. 87–92. ACM, New York (2002)
6. Balan, R., Gergle, D., Satyanarayanan, M., Herbsleb, J.: Simplifying Cyber Foraging for Mobile Devices. In: Proceedings of the 5th International Conference on Mobile Systems Applications and Services, pp. 272–285. ACM, New York (2007)
7. De Lara, E., Wallach, D.S., Zwaenepoel, W.: Puppeteer: Component-based Adaptation for Mobile Computing. In: Proceedings of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems, p. 14. USENIX Association, Berkely (2001)
8. Flinn, J., Narayanan, D., Satyanarayanan, M.: Self-Tuned Remote Execution for Pervasive Computing. In: Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems, pp. 61–66. IEEE, New York (2001)
9. Flinn, J., Park, S., Satyanarayanan, M.: Balancing Performance, Energy Conservation and Application Quality in Pervasive Computing. In: Proceedings of the 22nd International Conference on Distributed Computing Systems, pp. 217–226. IEEE, New York (2002)
10. Goyal, S., Carter, J.: A Lightweight Secure Cyber Foraging Infrastructure for Resource-constrained Devices. In: Proceedings of the 6th IEEE Workshop on Mobile Computing Systems and Applications, pp. 186–195. IEEE, Washington D.C (2004)
11. Kemp, R., Palmer, N., Kielmann, T., Seinstra, F., Drost, N., Maassen, J., Bal, H.: eyeDentify: Multimedia Cyber Foraging from a Smartphone. In: Proceedings of the 11th IEEE International Symposium on Multimedia, pp. 392–399. IEEE, New York (2009)
12. Ok, M., Seo, J.-W., Park, M.-S.: A Distributed Resource Furnishing to Offload Resource-Constrained Devices in Cyber Foraging Toward Pervasive Computing. In: Enokido, T., Barolli, L., Takizawa, M. (eds.) NBiS 2007. LNCS, vol. 4658, pp. 416–425. Springer, Heidelberg (2007)
13. Satyanarayanan, M.: Pervasive Computing: Vision and Challenges. IEEE Personal Communications, 10–17 (2001)
14. Christensen, J.: Using RESTful Web Services and Cloud Computing to Create Next-Generation Mobile Applications. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object-Oriented Programming Systems Languages and Applications, OOPSLA 2009, pp. 765–766. ACM, New York (2009)
15. de Leusse, P., Periorellis, P., Watson, P., Maierhofer, A.: Secure and Rapid Composition of Infrastructure Services in the Cloud. In: Proceedings of the 2nd International Conference on Sensor Technologies and Applications, pp. 770–775. IEEE, New York (2008)
16. Kumar, K., Lu, Y.: Cloud Computing for Mobile Users: Can Offloading Computation Save Energy? IEEE Computer 43, 51–56 (2010)
17. Li, X., Zhang, H., Zhang, Y.: Deploying Mobile Computation in Cloud Service. In: Jaatun, M.G., Zhao, G., Rong, C. (eds.) Cloud Computing. LNCS, vol. 5931, pp. 301–311. Springer, Heidelberg (2009)
18. Marinelli, E.: Hyrax: Cloud Computing on Mobile Devices using MapReduce. Technical Report, Carnegie Mellon University (2009)
19. Palmer, N., Kemp, R., Kielmann, T., Bal, H.: Ibis for Mobility: Solving Challenges of Mobile Computing Using Grid Techniques. In: Proceedings of the 10th Workshop on Mobile Computing Systems and Applications, article No. 17. ACM, New York (2009)
20. Zhang, X., Schiffman, J., Gibbs, S., Kunjithapatham, A., Jeong, S.: Securing Elastic Applications on Mobile Devices for Cloud Computing. In: Proceedings of the 2009 ACM Workshop on Cloud Computing Security, pp. 127–134. ACM, New York (2009)
21. Dilger, D.: First Look: Using iPhone 4S with Siri Voice Assistant (with Videos). Apple Insider (2011), http://www.appleinsider.com/articles/11/10/14/first_look_using_iphone_4s_with_siri_voice_assistant.html

22. Ha, K., Lewis, G., Simanta, S., Satyanarayanan, M.: Code Offload in Hostile Environments. Technical Report, Carnegie Mellon University (2011)
23. Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N.: The Case for VM-Based Cloudlets in Mobile Computing. IEEE CS Pervasive Computing, 14–23 (2009)
24. Lehr, W., McKnight, L.: Wireless Internet Access: 3G vs. WiFi? Center for eBusiness @ MIT (2002)
25. Cuervo, E., Balasubramanian, A., Cho, D., Wolman, A., Saroiu, S., Chandra, R., Bahl, P.: MAUI: Making Smartphones Last Longer with Code Offload. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, pp. 49–62. ACM, New York (2010)
26. Wolbach, A.: Improving the Deployability of Diamond. Technical Report, Carnegie Mellon University (2008)
27. Richardson, T., Stafford-Fraser, Q., Wood, K.R., Hopper, A.: Virtual Network Computing. IEEE Internet Computing 2(1), 33–38 (1998)
28. Microsoft Corporation: TCP/IP and NBT configuration parameters for Windows XP, http://support.microsoft.com/kb/314053
29. xdelta.org, http://www.xdelta.org
30. 7-Zip.org: LZMA SDK (Software Development Kit), http://www.7-zip.org/sdk.html
31. OpenSSL.org: OpenSSL: The Open Source Toolkit for SSL/TLS, http://www.openssl.org/
32. KVM: Kernel-Based Virtual Machine, http://www.linux-kvm.org/
33. KVM: Guest Support Status, http://www.linux-kvm.org/page/Guest_Support_Status
34. CherryPy: Cherry Py – A Minimalist Python Web Framework, http://www.cherrypy.org/
35. Ubuntu: KVM/Networking – Community Ubuntu Documentation, https://help.ubuntu.com/community/KVM/Networking
36. Github: seb-m/pyinotify—Github, https://github.com/seb-m/pyinotify
37. Avahi.org: Avahi, http://avahi.org/
38. Zeroconf: Zero Configuration Networking (Zeroconf), http://www.zeroconf.org
39. OpenCV: Welcome – Open CV Wiki, http://opencv.willowgarage.com/wiki/
40. MOPED: MOPED: Object Recognition and Pose Estimation for Manipulation, http://personalrobotics.ri.cmu.edu/projects/moped.php
41. SPHINX-4. Sphinx-4 – A Speech Recognizer Written Entirely in the Java Programming Language, http://cmusphinx.sourceforge.net/sphinx4/
42. Monsoon Solutions: Power Monitor, http://www.msoon.com/LabEquipment/PowerMonitor/
43. Chun, B., Ihm, S., Maniatis, P., Naik, M., Patti, A.: CloneCloud: Elastic Execution between Mobile Device and Cloud. In: Proceedings of the 6th European Conference on Computer Systems, pp. 301–314. ACM, New York (2011)
44. Kristensen, M.D.: Execution Plans for Cyber Foraging. In: Proceedings of the 1st Workshop on Mobile Middleware, article no. 2. ACM, New York (2008)
45. Chen, G., Kang, B., Kandemir, M., Vijaykrishnan, N., Irwin, M., Chandramouli, R.: Studying Energy Trade Offs in Offloading Computation/Compilation in Java-Enabled Mobile Devices. IEEE Transactions on Parallel and Distributed Systems 15(9), 795–809 (2004)