Tactical Cloudlets: Moving Cloud Computing to the Edge

Grace Lewis, Sebastián Echeverría, Soumya Simanta, Ben Bradshaw, James Root
Carnegie Mellon Software Engineering Institute
Pittsburgh, PA USA
{glewis, secheverria, ssimanta, bwbradshaw, jdroot}@sei.cmu.edu

***Abstract*—Soldiers and front-line personnel operating in tactical environments increasingly make use of handheld devices to help with tasks such as face recognition, language translation, decision-making, and mission planning. These resource-constrained edge environments are characterized by dynamic context, limited computing resources, high levels of stress, and intermittent network connectivity. Cyber-foraging is the leverage of external resource-rich surrogates to augment the capabilities of resource-limited devices. In cloudlet-based cyber-foraging, resource-intensive computation and data is offloaded to cloudlets. Forward-deployed, discoverable, virtual-machine-based tactical cloudlets can be hosted on vehicles or other platforms to provide infrastructure to offload computation, provide forward data staging for a mission, perform data filtering to remove unnecessary data from streams intended for dismounted users, and serve as collection points for data heading for enterprise repositories. This paper describes tactical cloudlets and presents experimentation results for five different cloudlet provisioning mechanisms. The goal is to demonstrate that cyber-foraging in tactical environments is possible by moving cloud computing concepts and technologies closer to the edge so that tactical cloudlets, even if disconnected from the enterprise, can provide capabilities that can lead to enhanced situational awareness and decision making at the edge.**

Keywords—mobile cloud computing; cloudlets; cloud computing; tactical cloudlets; mobile computing; edge computing

I. INTRODUCTION

Mobile applications are increasingly used by military personnel and others operating in crisis and hostile environments in support of their missions. These environments are not only at the edge of the network infrastructure, but are also resource-constrained due to dynamic context, limited computing resources, intermittent network connectivity, and high levels of stress. Applications that are useful to 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.

Cyber-foraging is the leverage of external resource-rich surrogates to augment the capabilities of resource-limited mobile devices [1]. Most existing cyber-foraging solutions rely on conventional Internet for connectivity to the cloud or strategies that tightly couple mobile clients with servers at deployment time. These solutions are not appropriate for resource-constrained environments because of their dependence on multi-hop networks to the cloud and static deployments.

Cloudlet-based cyber-foraging relies on discoverable, generic, forward-deployed servers located in single-hop proximity of mobile devices. The goal of this paper is to propose *tactical cloudlets* as a strategy for providing infrastructure to support computation offload and data staging at the tactical edge. Section II presents a short summary of related work in this area. Section III describes cloudlet-based cyber-foraging. Section IV describes cloudlet discovery. Section V presents five mechanisms for cloudlet provisioning. Section VI describes the generic process for application execution. Section VII presents experimental data that shows the pros and cons of each cloudlet provisioning mechanism. Finally, Section VIII summarizes the potential for tactical cloudlets to support operations in resource-constrained edge environments, next steps and future work.

II. RELATED WORK

Multiple cyber-foraging systems have been developed that differ in terms of the strategy that they use to leverage remote resources — where to offload, when to offload, and what to offload. Where to offload varies between remote clouds and local servers located in proximity of mobile devices. When to offload varies between a runtime decision or an "always offload" strategy. To support runtime offload decisions, one strategy is to manually or automatically partition code into portions that either run on the mobile device or on a remote machine. At runtime an optimization engine — typically targeted at optimizing energy efficiency, performance, or network usage — decides whether the code should execute locally or be offloaded to a remote machine (surrogate). An example of such cyber-foraging system is MAUI [2]. CloneCloud [3] follows the same code partitioning principle but automatically partitions code at the thread level without the need for manual code annotation. Other cyber-foraging solutions assume that the computation-intensive code exists in a remote machine and the cyber-foraging task therefore becomes one of service discovery and composition. HPC-as-a-service [4] is an example of this "always offload" strategy. What to offload is what has the most variation, ranging from threads [3] to methods [2] to services [4] to full programs [1], with many other options in between. Our work is based on cloudlets, as described in [1]. Despite all the work in cyber-foraging, our research has showed that (1) there is emphasis on the algorithms to support code offload and state synchronization with minimal focus on software architecture and quality attributes beyond energy efficiency and performance, (2) there is little guidance on how to support quality attributes such as survivability, resilience, trust and ease of deployment, critical in tactical environments.

III. Cloudlet-Based Cyber-Foraging

Cloudlets are discoverable, generic, stateless servers located in single-hop proximity of mobile devices, that can operate in disconnected mode and are virtual-machine (VM) based to promote flexibility, mobility, scalability, and elasticity [1]. In our implementation of cloudlets, applications are statically partitioned into a very thin client that runs on the mobile device and a computation-intensive S*erver* that runs inside a *Service VM*. A reference architecture for cyber-foraging is presented in Fig. 1. The main elements of the architecture are the *Mobile Client* and the *Cloudlet Host*. A *Discovery Service* running inside the cloudlet host publishes *Cloudlet Metadata* that is used by the *Cloudlet Client* to determine the appropriate cloudlet for offload and to connect to the cloudlet (Section IV). Cloudlet metadata can range from a simple IP address and port to connect to the cloudlet server to more complex data structures describing cloudlet capabilities. Every application is composed of a *Cloudlet-Ready Client App* that corresponds to the client portion and the *Client App Metadata* that contains information that is used by the *Cloudlet Client* and the *Cloudlet Server* to negotiate and carry out the offload process. Once a cloudlet is identified for offload, the *Cloudlet Client* sends the *Client App Metadata* and *Provisioning Data* to the *Cloudlet Server*. The provisioning data varies depending on the cloudlet provisioning process (Section V), and can range from parameters to start a *Service VM* that already resides on the *Cloudlet Host*, to provisioning instructions, to actual server code. The *Cloudlet Server* then configures and starts the corresponding *Service VM* inside the *VM Manager* according to the defined cloudlet provisioning process and data. Once the *Service VM* is started, the client app is notified that it is ready for execution (Section VI).

IV. Cloudlet Discovery

The scenarios in which tactical cloudlets are deployed are very dynamic because both the mobile devices and the cloudlets can be mobile. Therefore a key feature of a cyber-foraging solution is for mobile devices to be able to locate cloudlets around them. Our implementation of cloudlet discovery is based on Zeroconf (Zero Configuration Networking) [5]. It uses DNS Service Discovery (DNS-SD) along with Multicast DNS so that a client can request a service without knowing the IP addresses of the servers that provide the service, as shown in Fig. 2. When a cloudlet starts, its *Discovery Service* joins a particular *Multicast IP Address* as a listener. When the *Cloudlet Client* wants to discover cloudlets, it sends a DNS-SD query about cloudlet services (defined as a "_cloudlet._tcp" service in our implementation) through Multicast DNS to the multicast IP address. The query reaches the *Discovery Service* of any cloudlets in the network through Multicast DNS, which reply with a DNS-SD response indicating the IP address and port of the *Cloudlet Server*.

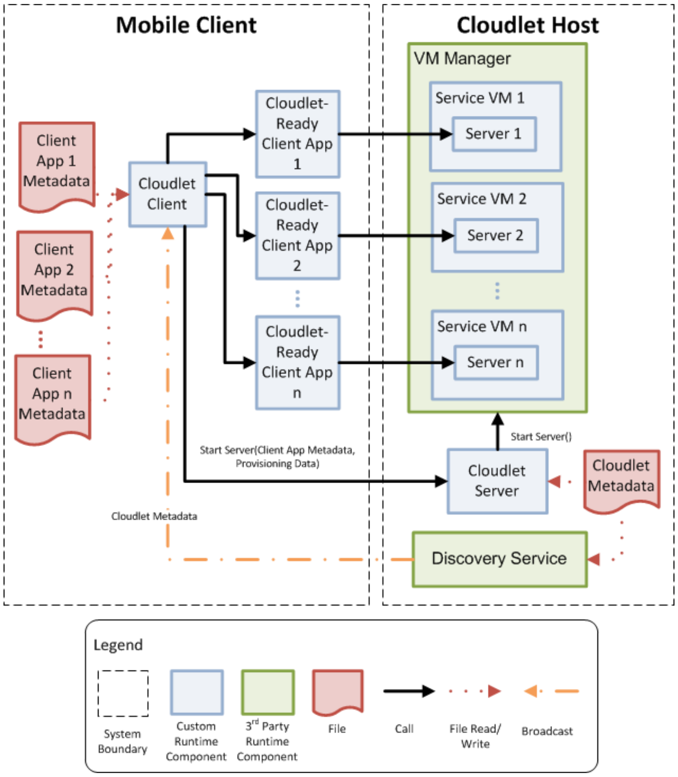

Fig. 1. Reference Architecture for Cloudlet-Based Cyber-Foraging

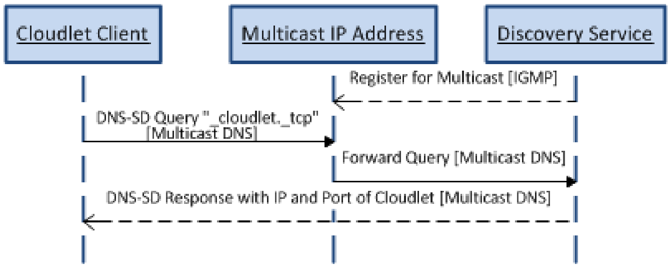
Cloudlet Client
Multicast IP Address
Discovery Service
Register for Multicast [IGMP]
DNS-SD Query "_cloudlet._tcp" [Multicast DNS]
Forward Query [Multicast DNS]
DNS-SD Response with IP and Port of Cloudlet [Multicast DNS]

Fig. 2. Cloudlet Discovery Details

V. Cloudlet Provisioning

In addition to cloudlet discovery, a key aspect of cloudlet-based cyber-foraging is cloudlet provisioning—configuring and deploying the Service VM that contains the server code on the cloudlet so that it is ready to use by the client running on the mobile device.

A. Optimized VM Synthesis

In VM synthesis the cloudlet is provisioned from the mobile device at runtime. Our original implementation of VM synthesis is fully described in [6].

In our Optimized VM Synthesis implementation the goal was to reduce application-ready time – time between the cloudlet provisioning request and the notification that the server is ready for execution. An application overlay that corresponds to the server portion of a client-server application is created once offline by starting a VM instance from a base VM disk image file (that uses QEMU copy on write 2 (qcow2) [7] as the VM disk image file format) and a base memory image, installing the server on the base VM image, and suspending the VM. When suspended, there are two files that are created as part of the application overlay: one corresponds to the disk image differences between the suspended VM and the base VM (the qcow2 file) and another that corresponds to the binary difference between the suspended memory image and the base memory image (calculated using VCDIFF [8]). These calculated overlays (disk and memory) are compressed using LZMA2 with the XZ stream compression format [9] and loaded on the mobile device.

At runtime, as shown in Fig. 3, the *Cloudlet Client* checks if the cloudlet has the *Base VM* from which the overlay was created. If it does, then the compressed application overlay is sent in chunks to the *Cloudlet Server*. Each chunk is placed in a queue, decompressed, and appended to a file. This optimization, called pipelining, enables overlay decompression to be done incrementally as opposed to having to wait until the complete overlay is received. In the *Synthesize VM* step, because of another optimization to use qcow2 as the VM file image format, the disk overlay that is received already corresponds to the changes with respect to the Base VM, which means that there is no need for extra processing after decompression. For the memory image, the received memory overlay is applied to the base memory overlay in order to create the complete memory image (the opposite of what was done in the memory overlay creation process). The Base VM disk image file, the qcow2 file, and the complete memory image file are saved as the *Service VM*, which now corresponds to the suspended VM from which the application overlay was created. A copy of the *Service VM* is created and started. All the cloudlet provisioning mechanisms create a copy of the Service VM (called the *Transient Service VM*) so that all Service VM instances are started from same baseline. Finally, the IP address and port to connect to the *Service VM* are sent back to the *Cloudlet Client*.

B. *Application Virtualization*

In Application Virtualization the cloudlet is also provisioned from the mobile device at runtime. Application Virtualization uses an approach similar to operating system (OS) virtualization, by “tricking" the software into interacting with a virtual rather than the actual environment. A runtime component intercepts all system calls from an application and redirects these to resources inside the virtualized application. Virtualized applications are created in advance for server portions of applications using tools that package the application with all its dependencies. We used CDE (short for Code, Data and Environment) as the application virtualizer for Linux [10] and Cameyo for Windows [11]. CDE virtualizes applications by monitoring their execution and Cameyo by monitoring their installation process. Both tools produce virtualized applications that are loaded on the mobile device and at runtime are sent to the cloudlet to be deployed in a VM that matches the OS of the virtualized application. The full implementation is described, analyzed, and compared to VM synthesis in [12].

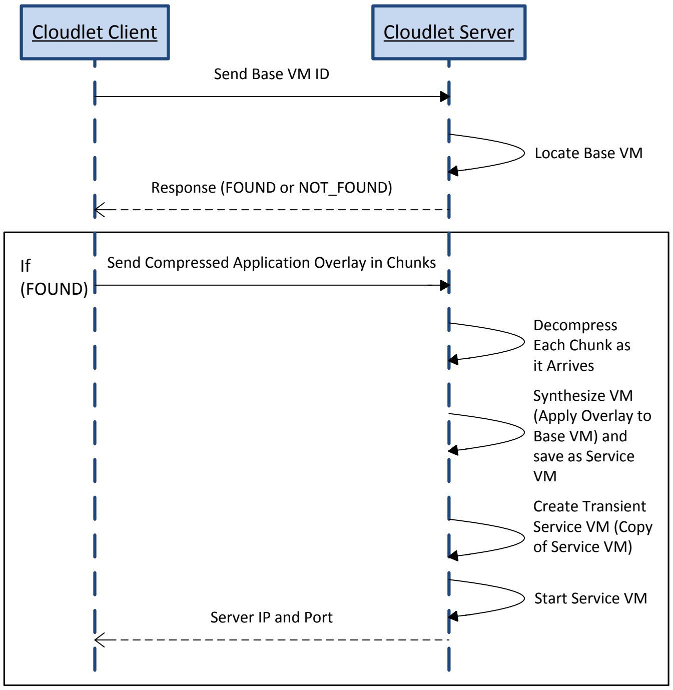
Cloudlet Client
Cloudlet Server
Send Base VM ID
Locate Base VM
Response (FOUND or NOT_FOUND)
If (FOUND)
Send Compressed Application Overlay in Chunks
Decompress Each Chunk as it Arrives
Synthesize VM (Apply Overlay to Base VM) and save as Service VM
Create Transient Service VM (Copy of Service VM)
Start Service VM
Server IP and Port

Fig. 3. Cloudlet Provisioning Using VM Synthesis

At runtime, as shown in Fig. 4, the *Cloudlet Client* checks if the cloudlet has a *Guest VM* that matches the OS required by the application. If it does, it sends the virtualized application to the *Cloudlet Server*, which then deploys the application into a copy of the matching guest VM. The application is started and the IP and port to connect to it are sent back to the *Cloudlet Client*.

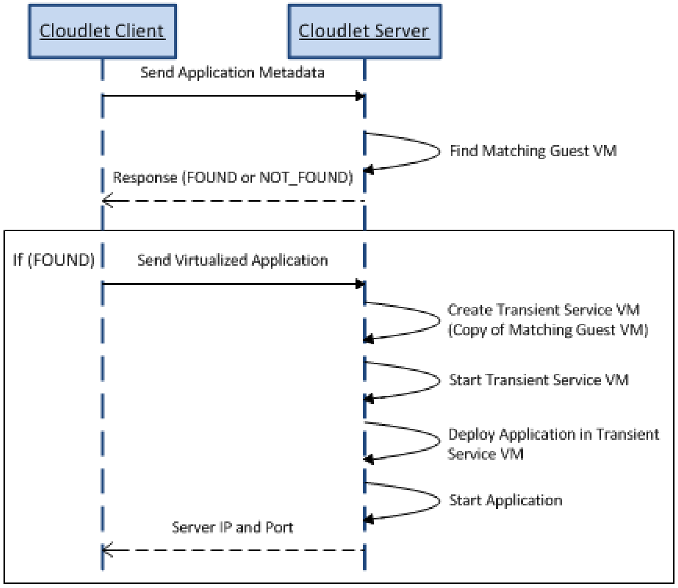
Cloudlet Client
Cloudlet Server
Send Application Metadata
Find Matching Guest VM
Response (FOUND or NOT_FOUND)
If (FOUND)
Send Virtualized Application
Create Transient Service VM (Copy of Matching Guest VM)
Start Transient Service VM
Deploy Application in Transient Service VM
Start Application
Server IP and Port

Fig. 4. Cloudlet Provisioning Using Application Virtualization

C. *Cached VM*

In Cached VM the cloudlet is pre-provisioned with Service VMs that correspond to mission-specific capabilities that match the client apps on the mobile device. Each Service VM has a unique service identifier.

At runtime, as shown in Fig. 5, the *Cloudlet Client* checks if the cloudlet has a *Service VM* that matches the client app. If it does, the *Cloudlet Server* creates a copy of the matching *Service VM* and starts it. When ready, the IP address and port to connect to the *Service VM* are sent back to the *Cloudlet Client*.

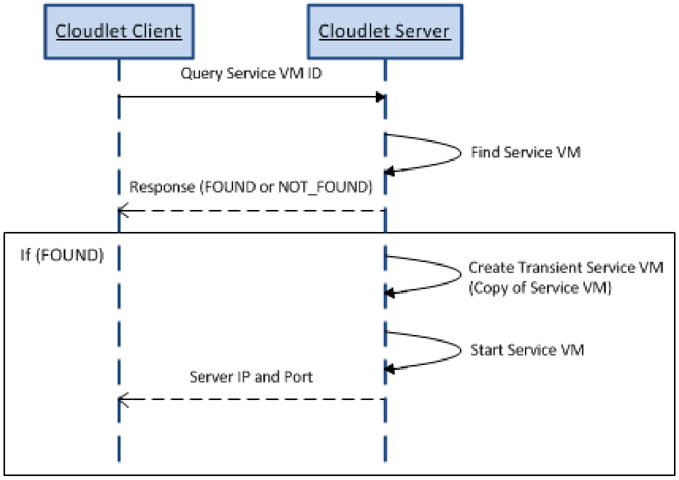

Fig. 5. Cloudlet Provisioning Using Cached VM

D. *Cloudlet Push*

In Cloudlet Push, the cloudlet is not only pre-provisioned with Service VMs for mission-specific capabilities, but also the corresponding mobile client apps.

At runtime, as shown in Fig. 6, the *Cloudlet Client* obtains a list of available applications on the cloudlet, similar to accessing an app store. It then checks if the selected application exists for the mobile device's OS. If so, the cloudlet client receives the app and installs it on the mobile device while the *Cloudlet Server* starts the corresponding *Service VM*. When ready, the IP address and port to connect to the *Service VM* are sent back to the *Cloudlet Client*.

E. *On-Demand VM Provisioning*

In On-Demand VM Provisioning a commercial cloud provisioning tool is used to "assemble" a Service VM. In this case the cloudlet has access to all the elements to put together a Service VM based on a provisioning script. Our implementation uses Puppet [13] and the provisioning script is a "manifest" that is written in Puppet's declarative language.

At runtime, as shown in Fig. 7, the *Cloudlet Client* sends a provisioning script to the *Cloudlet Server*, which verifies that it can execute the provisioning script. The *Cloudlet Server* creates a transient *Service VM* as a copy of a *Baseline Service VM* which corresponds to a VM defined by an organization as the basic configuration from which all VMs are created. The *Cloudlet Server* then runs the provisioning script that will find and install inside the *Service VM* all the components that make up the server capabilities required by the client app. After executing the script, the *Cloudlet Server* starts the newly assembled *Service VM* and sends its IP address and port to the *Cloudlet Client*.

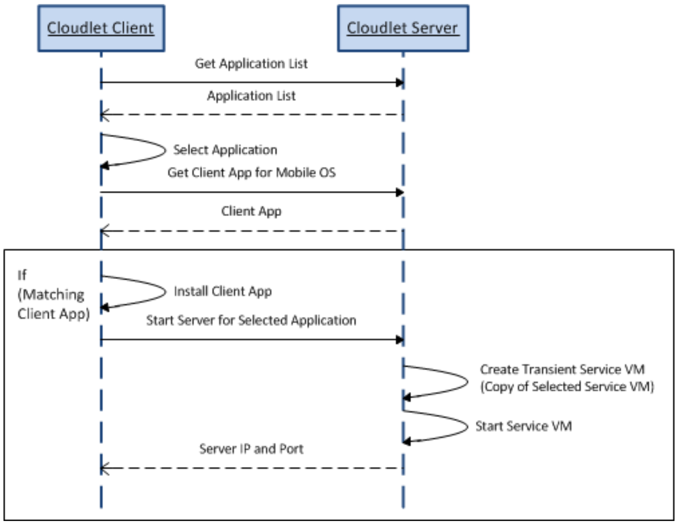

Fig. 6. Cloudlet Provisioning Using Cloudlet Push

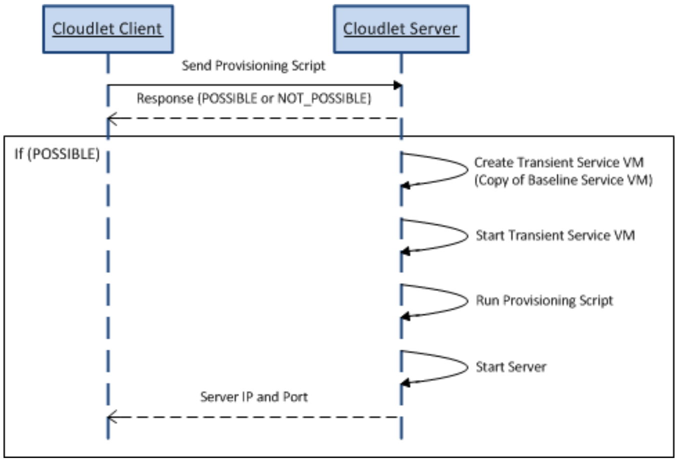

Fig. 7. Cloudlet Provisioning Using On-Demand Provisioning

VI. Application Execution

As shown in Fig. 8, after the *Cloudlet Client* receives the IP address and port for the *Service VM* it passes this information on to the *Cloudlet-Ready App*. The *Cloudlet-Ready App* then opens a socket to the IP address and port and starts the interaction with the *Service VM*. The interaction depicted in Fig. 8 is simple request-response. Even though other types of interaction could be supported by cloudlets, this is the most energy-efficient type of interaction because it limits communication between the mobile device and the cloudlet. In general, offloading is beneficial when large amounts of computation are needed with relatively small amounts of communication [14]. Even though not shown in Fig. 8, an optional step after the *Cloudlet-Ready App* is closed is to stop the *Service VM* on the cloudlet to promote elasticity. This would release resources on the cloudlet that could be used by other mobile devices.

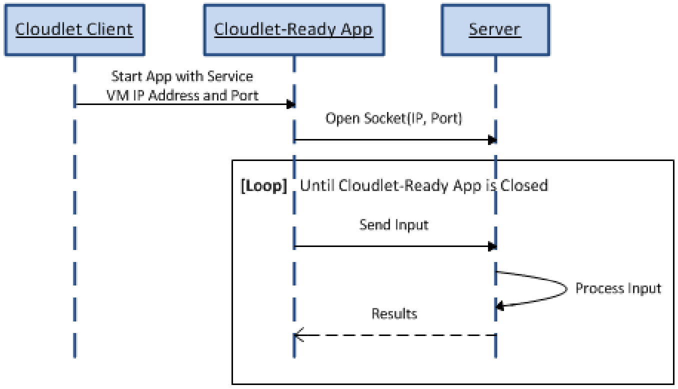

Fig. 8. Application Execution Details

VII. Quantitative and Qualitative Comparison of Cloudlet Provisioning Mechanisms

To perform a quantitative and qualitative comparison of the five different cloudlet provisioning mechanisms, we conducted a set of experiments using three computation-intensive applications: face recognition (FACE), speech recognition (SPEECH), and object recognition (OBJECT). We used a Galaxy Nexus with Android 4.3 as a mobile device and a Core i7-3960x based server with 32 GB of RAM running Ubuntu 12.04 as the cloudlet. We created a self-contained wireless network (using Wi-Fi 802.11n at 2.4 GHz, 65 Mbps) to be able to isolate network traffic effects. Energy was measured using a Power Monitor from Monsoon Solutions [15]. The results of these experiments are shown in TABLE I. The first column under each mechanism is the size of the payload in MB that is sent from the mobile device to the cloudlet for provisioning. The second column is application-ready time, measured as the time in seconds from the start of the provisioning process until the cloudlet responds that it is ready. The third column in the energy consumed on the mobile device during application-ready time.

TABLE I. shows that the largest amount of energy is consumed by *VM Synthesis* and *On-Demand VM Provisioning*. In VM Synthesis this is due to the large payload. Our experiments confirm that payload size is directly proportional to energy consumption, as has been stated by many others. In On-Demand VM Provisioning, even though the payload is very small, the high energy consumption is due to the longer application-ready time. The power monitor measures total energy consumption and does not distinguish between energy consumed during communication and during idle time. Application-ready time is also variable, as can also be seen in TABLE I. For example, for Windows applications the application-ready times are much longer because the component installation processes are more complicated.

For *Application Virtualization*, although payload size is between 8% and 46% of the payload for VM synthesis, it is still large for transmission in resource-constrained environments. In addition, the size of the payload is very variable because it depends on the OS and the number of external dependencies of the application that is being virtualized.

Cached VM and *Cloudlet Push* consume less energy because payload size is smaller, which in turn leads to shorter and more consistent application-ready times across applications. In Cached VM the payload size is very small (Service ID) and application-ready time is the time that it takes to start the corresponding Service VM. In Cloudlet Push the payload is small (client app from cloudlet to mobile device) and the application-ready time is the time that it takes to install the app on the mobile device.

TABLE II. shows a qualitative comparison of the cloudlet provisioning mechanisms. For *VM Synthesis* the advantage is that the cloudlet can run any server code that can be installed in a VM. In addition, because cloudlet is provisioned by the mobile device the cloudlet does not have to be pre-provisioned with any mission-specific capabilities. However, as noted earlier, the large payload size is a disadvantage for tactical environments. In addition, because the VM synthesis process requires the exact same base VM from which the overlay was created, any changes to the base VM, due to for example security patches, would require changes to every application overlay that was created with that base VM.

For *Application Virtualization* the advantage is also that the cloudlet can be provisioned from the mobile device, with an advantage over VM synthesis with respect to payload size. In addition, the dependency is on the operating system running inside the VM which enables portability across OS distribution boundaries. However, all server code dependencies have to be captured at packaging time which is a challenge for any application virtualization tool [12].

For *Cached VM*, in addition to the small payload size, the advantage is that it supports server code updates as long as service interface remains the same. However, the assumption is that the cloudlet is provisioned with Service VMs required by client apps or has access to them either at deployment time or at runtime (i.e., an enterprise-level Service VM repository).

For *Cloudlet Push*, in addition to the small payload size, the advantage is that it supports most client mobile devices with distribution at runtime. However, similar to Cached VM, the assumption is that the cloudlet is provisioned with Service VMs in addition to Client Apps. In addition, the cloudlet would need to have a client app version that matches mobile client OS version.

Finally, for *On-Demand VM Provisioning*, in addition to small payload size, the advantage is that the Service VM can be assembled at runtime which provides the greatest flexibility. However, as noted earlier, in addition to longer application-ready time the constraint is that the cloudlet has all required server code components, or access to the components from enterprise repositories or code distribution sites.

VIII. Conclusions, Next Steps and Future Work

Forward-deployed, discoverable, virtual-machine-based tactical cloudlets can be hosted on vehicles or other platforms to provide infrastructure to offload computation, provide forward data-staging for a mission, perform data filtering to remove unnecessary data from streams intended for dismounted users, and 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). If tactical cloudlets are pre-provisioned, there are many applications that can function disconnected from the enterprise or can synchronize with the enterprise if and when there is connectivity. The fact that cloudlets are discoverable enables mobile devices to locate mission-specific capabilities as personnel and cloudlets move and missions change. Finally, virtual machine technology not only simplifies the distribution and rapid deployment of capabilities, but also enables the leverage of any legacy application that can be packaged inside a VM.

The results of the experiments led us to combine *Cached VM* with *Cloudlet Push* as the cloudlet provisioning mechanism for our current working prototype to enable lower energy consumption on the mobile device, place less requirements on mobile devices, and simplify provisioning in tactical environments. An additional advantage of combining both mechanisms is that if the mobile device already has the client app it can simply invoke the matching Service VM; if not it can obtain the client app from the cloudlet, similar to accessing an app store, and then invoke the matching Service VM. The tradeoff is that it relies on cloudlets that are pre-provisioned with server capabilities that might be needed for a particular mission, or that the cloudlet is connected to the enterprise, even if just at deployment time, to obtain the capabilities. We argue that this requirement is not unreasonable in tactical edge environments and that it makes cloudlet deployment in the field easier and faster while leveraging the state of art and best practices from the cloud computing industry. A pre-provisioned-VM-based solution also promotes resilience and survivability by supporting rapid live VM migration in case of cloudlet mobility, discovery of more powerful or less-loaded cloudlets, or unavailability due to disconnection or disruption. It supports scalability and elasticity by starting and stopping VMs as needed based on number of active users (which is typically bounded in edge environments because group size is known). In addition, the request-response nature of many of the operations needed in the field also lends itself to an asynchronous form of interaction in which the cloudlet can continue processing and send results back to a mobile device (directly or by re-routing) as network conditions change. Although not part of the presented prototype implementation, an added feature would be to have “dual-mode” cloudlet-ready apps that exploit cloudlets when and if available but rely on a local implementation as a fallback mechanism. The local implementation could be identical or could be a version that is adapted for resource-constrained devices that may not provide the same precision or quality of results but would provide some result even if a cloudlet is not available.

We are currently working on a standard packaging of Service VMs so that they can be easily installed from the cloudlet manager (web-based interface to the Cloudlet Server and Service VM repository), an enterprise Service VM repository, a thumb drive, or the mobile device connected via USB to the cloudlet. We are also adding the following capabilities to adapt to cloudlets to the characteristics of tactical environments:

- Optimal cloudlet selection: We are extending the cloudlet discovery protocol to use metadata from the client app, Service VM, and the cloudlet so that in the case that there is more than one cloudlet in range, the mobile device can automatically select the cloudlet that maximizes a pluggable utility function. This function can be based on cloudlet load, signal strength, or any other parameter.

- Manual and automated cloudlet handoff: We are adding VM migration capabilities to enable manual and automated handoff of data and computation between cloudlets that are within range of each other. Manual handoff would enable scenarios in which a user is migrating capabilities from a fixed cloudlet to a mobile cloudlet to support field operations, as well as reintegration back to the fixed cloudlet. Automated migration would enable load balancing, similar to what is done in cloud data centers for resource optimization.

- Data synchronization between cloudlets and the enterprise: Even though cloudlets can operate fully-disconnected from the enterprise if they are pre-provisioned at deployment time, there are situations when cloudlet capabilities (Service VMs) need access to a master data source located in the enterprise. We plan to add support for integration with distributed, networked filesystems such as Coda [16] to support disconnected operations with opportunistic synchronization when connectivity becomes available.

Our future work is related to security, in particular establishing the initial trust between mobile devices and cloudlets; that is (1) as a mobile device, is what I discovered really a "friendly" cloudlet? and (2) as a cloudlet, did that offloading request really come from a "friendly" mobile device? The solution presented in this paper relies on the underlying network to provide the secure communication between the mobile device and the cloudlet. While this may be enough in some scenarios, it is not enough for many military scenarios. A common solution for establishing trust between two nodes is to use a third-party online trusted authority that validates the credentials of the requester or a certificate repository. However, the characteristics of tactical edge environments do not consistently provide access to that third-party authority or certificate repository because tactical cloudlets operate in what is known as DIL environments (disconnected, interrupted, low bandwidth). The goal is to explore solutions for establishing trusted identities in disconnected environments with the advantage/constraint that tactical cloudlets are not meant to be long-lived, meaning that they are pre-provisioned and eventually deployed to support a mission. This constraint may enable us to explore more dynamic identity solutions.

Acknowledgment

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-

0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution (DM-0001206).

REFERENCES

[1] Satyanarayanan, M., Bahl, P., Cáceres, R., Davies, N. 2009. The Case for VM-Based Cloudlets in Mobile Computing. IEEE Pervasive Computing vol.8, no.4, 14–23.

[2] Cuervo, E., Balasubramanian, A., Cho, D.-K., Wolman, A., Saroiu, S., Chandra, R., Bahl, P. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10), pp. 49–62. ACM, New York.

[3] Chun, B., Ihm, S., Maniatis, P., Naik, M., Patti, A. 2011. CloneCloud: Elastic Execution between Mobile Device and Cloud. Proceedings of the 6th Conference on Computer Systems (EuroSys '11), pp. 301–314. ACM, New York.

[4] Duga, N. 2011. Optimality Analysis and Middleware Design for Heterogeneous Cloud HPC in Mobile Devices. Doctoral Thesis. Addis Ababa University.

[5] Zeroconf. Zero Configuration Networking (Zeroconf). http://www.zeroconf.org/ (2014)

[6] Simanta, S, Lewis, G., Morris, E., Ha, K., and Satyanarayanan, M. 2012. A Reference Architecture for Mobile Code Offload in Hostile Environments Proceedings of the Joint Working IEEE/IFIP Conference Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), pp.282 - 286.

[7] Gnome. The QCOW2 Image Format https://people.gnome.org/~markmc/qcow-image-format.html (2014)

[8] IETF. 2002. RFC 3284: The VCDIFF Generic Differencing and Compression Data Format. http://tools.ietf.org/html/rfc3284.

[9] Python Software Foundation. lzma – Compression Using the LZMA Algorithm. https://docs.python.org/dev/library/lzma.html (2014)

[10] Guo, P.J., Engler, D. 2011. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In: Proceedings of the 2011 USENIX Annual Technical Conference, p. 21. USENIX Association, Berkeley.

[11] Cameyo User Guide, Version 2.0. 2012. http://cameyo.com/doc/CameyoManual.pdf

[12] Messinger, D., Lewis, G. 2013. Application Virtualization as a Strategy for Cyber Foraging in Resource-Constrained Environments (Technical Report CMU/SEI-2013-TN-007). Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

[13] Puppet Labs. Puppet Entrprise. http://puppetlabs.com/puppet/puppet-enterprise (2014)

[14] Kumar, K.., Lu Y.H. 2010. "Cloud computing for mobile users: Can offloading computation save energy?" Computer, vol. 43(4), pp. 51-56.

[15] Monsoon Solutions Inc. Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/ (2014)

[16] Carnegie Mellon University. Coda File System. http://www.coda.cs.cmu.edu/ (2014)

TABLE I. EXPERIMENT DATA FOR CLOUDLET PROVISIONING MECHANISMS

Applications	Optimized VM Synthesis[a]			Application Virtualization			Cached VM			Cloudlet Push			On-Demand VM Provisioning		
	(1)	*(2)*	*(3)*	*(1)*	*(2)*	*(3)*	*(1)*	*(2)*	*(3)*	*(1)[b]*	*(2)*	*(3)*	*(1)[b]*	*(2)*	*(3)*
FACE (Windows)	55	53.4	57.8	14	14.3	10.5	0.00	8.2	10.3	0	7.9	13.8	0	112.7	129.1
OBJECT (Linux)	332	175.7	333.3	29	21.9	24.5	0.00	11.6	13.5	0	11.7	16.9	0	211.0	244.0
SPEECH (Windows)	194	85.9	175.5	66	62.5	66.6	0.00	12.2	14.7	0	12.8	18.2	0	237.6	269.2
SPEECH (Linux)	147	99.0	172.5	68	38.3	54.2	0.00	12.2	14.9	0	12.8	18.2	0	94.1	109.3

[a] Columns under each mechanism are (1) Payload Size (MB), (2) Application-Ready Time (s), and (3) Client Energy (J) [b] Size of payload is less than 1KB

TABLE II. QUALITATIVE COMPARISON OF CLOUDLET PROVISIONING MECHANISMS

	Optimized VM Synthesis	Application Virtualization	Cached VM	Cloudlet Push	On-Demand VM Provisioning
Cloudlet Content[c]	Exact base VM	VM compatible with server code	Service (VM) repository	Repository of paired VMs (server code) and client apps	• VM provisioning software • Server code components
Mobile Device Content	• Application overlay • Client app and metadata	• Virtualized server code • Client app and metadata	Client app and metadata	None	• VM provisioning script • Client app and metadata
Payload	Application overlay	Virtualized server code	Service ID	Client app and metadata	VM provisioning Script
Advantages	Cloudlet can run any server code that can be installed on a Base VM	Portability across OS distribution boundaries	Supports server code updates as long as service interface remains the same	Supports most client mobile devices with distribution at runtime	Service VM with server code can be assembled at runtime
Constraints	Requires exact base VM which limits distributions and patches	All server code dependencies have to be captured at packaging time	Cloudlet is provisioned with service VMs required by client apps (or has access to them)	Cloudlet has a client app version that matches mobile client OS version	Cloudlet has all required server code components (or access to them)

[c] In addition to Cloudlet Server and Metadata