Cyber-Foraging for Improving Survivability of Mobile Systems

Sebastián Echeverría*†, Grace A. Lewis†, James Root†, Ben Bradshaw†
†Carnegie Mellon Software Engineering Institute
Pittsburgh, PA USA
{secheverria, glewis, jdroot, bwbradshaw}@sei.cmu.edu
*Universidad de los Andes
Santiago, Chile

***Abstract*—Cyber-foraging is a technique for dynamically augmenting the computing power of resource-limited mobile devices by opportunistically exploiting nearby fixed computing infrastructure. Cloudlet-based cyber-foraging relies on discoverable, generic, forward-deployed servers located in single-hop proximity of mobile devices. In particular, we define tactical cloudlets as the infrastructure to support computation offload and data staging at the tactical edge. However, the characteristics of tactical environments, such as dynamic context, limited computing resources, disconnected-intermittent-limited (DIL) network connectivity, and high levels of stress pose a challenge for the continued operations of mobile systems that leverage cloudlets in tactical environments. This paper presents an architecture and experimental results that demonstrate that cyber-foraging using tactical cloudlets increases the survivability of mobile systems, defined as the capability of a system to be able to continue functioning in spite of adversity.**

I. Introduction

Cyber-foraging is an area of work within mobile cloud computing that leverages external resources (i.e., cloud servers, or local servers called surrogates) to augment the computation and storage capabilities of resource-limited mobile devices while extending their battery life [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 capability. The second is data staging to improve data transfers between mobile devices and the cloud by temporarily staging data in transit.

Cloudlet-based cyber-foraging relies on discoverable, generic, forward-deployed servers located in single-hop proximity of mobile devices [2]. We further define *tactical cloudlets* as cloudlets that support computation offload and data staging at the tactical edge [3]. However, the characteristics of tactical environments, such as dynamic context, limited computing resources, disconnected-intermittent-limited (DIL) network connectivity, and high levels of stress pose a challenge for the continued operations of mobile systems that leverage cloudlets in tactical environments.

Battery savings and better response times (lower latency) are demonstrated benefits of cyber-foraging and contributors to mission success [3]. However, mission success also requires cloudlet-deployed capabilities to be available when they are required, despite the challenges of tactical environments.

This paper presents our implementation of tactical cloudlets with a specific focus on survivability, defined as the capability of a system to continue functioning in spite of adversity. Section II presents a summary of related work in cyber-foraging and software systems survivability. Section III presents the architecture of our tactical cloudlet implementation. Section IV focuses on the tactical cloudlet features that promote survivability of mobile systems in the field. Finally, Section V concludes the paper and outlines next steps.

II. Related Work

A. Cyber-Foraging

We recently conducted a systematic literature review (SLR) on architectures for cyber-foraging systems that identified 57 primary studies and a total of 60 cyber-foraging systems in these studies (52 computation offload and 8 data staging systems) [4]. The SLR showed that there is indeed active work in this area but also that there is (1) a lack of understanding of system qualities attributes beyond energy, performance, network usage, and fidelity of results, and (2) a lack of focus on system-level concerns that are required when moving from experimental prototypes to operational systems such as ease of distribution and installation, survivability and security. The goal of the work presented in this paper is to fill in some of these gaps.

As far as the use of cyber-foraging and cloudlets in tactical environments, there is indeed a motivation for their use as outlined in [5]. Satyanarayanan et al propose proximate, VM-based cloudlets as a way to overcome the lack of a reliable, high-bandwidth, end-to-end network, which is difficult to guarantee in hostile environments. More specifically, there is recent work in deploying cloudlet-like servers at the tactical edge to support mobile devices. Wang et al [6] propose the use of mobile micro-clouds and in particular mobility-induced service migration so that computation can follow its mobile users. Sookoor et al [7] propose the use of smartphones as data collection platforms that leverage mobile high-performance computers deployed on humvees for data processing. This work focuses on optimal selection of the processing node based on minimizing job completion time. Even though computation migration and optimal cloudlet selection are two features that are implemented in our tactical cloudlets to promote survivability, the uniqueness of our work is the combination of lightweight versions of these features with additional cloudlet management capabilities that in addition support very simple cloudlet deployment in the field.

B. *Software Systems Survivability*

Survivability is traditionally defined in military systems as the capability to avoid or withstand the interaction between a system and a given hostile environment [Richards2007]. More specifically, the Air Force Cyber Vision 2025 document [8] lists survivability as an enduring principle and defines it in terms of fitness/readiness, awareness, anticipation, speed (responsiveness), agility, and evolvability. While this is only one definition of survivability, it is an example of the many system attributes that fall under the category of survivability.

Survivability of software systems is viewed differently among research and practice communities, but in general it refers to the capability of a system to be able to continue functioning in spite of adversity. For example:

- The Self-Healing Systems (Autonomic Computing) community defines survivability as the capability of a system to continue functioning in spite of system disruptions. More specifically, self-healing is the capability to discover, diagnose, and react to system disruptions with the goal of maximizing availability, survivability, maintainability, and reliability of a system [9].
- The Security community defines survivability as the capability of a system to continue functioning in spite of faults caused by cyber-attacks and to continue to provide essential services, even if in degraded mode [10].
- The Software Architecture community views survivability as a system quality that is related to other system qualities such as dependability, availability, reliability, fault-tolerance, and trustworthiness [11]. The goal of work in this area is to define architectural elements that promote survivability of software systems.

Our work takes the view of the software architecture community, which is to define architectural elements that promote survivability. In particular, it is aligned with Thuraisingham et al who simply define survivability as being able to adapt to changing environments [12].

III. Tactical 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. The forward-deployed, single-hop proximity to mobile devices promotes energy efficiency as well as lower latency (faster response times) [3].

The architecture for our tactical cloudlet implementation is presented in Figure 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 or data to a cloudlet.
  - Cloudlet Client GUI: Mobile app that is used to access the app store capability described in Section IV-A.
  - Cloudlet Client Lib: Library that is used by the two components above to discover cloudlets, retrieve data from cloudlets, and offload computation or data. It interacts with the Cloudlet Host using HTTP.
- Cloudlet Host: Linux server that runs a tactical cloudlet. The main components are:
  - PyCloud Lib: Python component that implements the core cloudlet functionality.
  - Cloudlet API: Python component that is used by the Cloudlet Client Lib to start Services as Service VMs.
  - 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 (Section IV-A). 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.
- Admin (PC): Browser that is used to access the Cloudlet Manager web application.

The implementation and additional documentation for tactical cloudlets is available on GitHub at https://github.com/SEI-AMS/pycloud.

IV. Tactical Cloudlet Features That Promote Survivability

Sheard states that implementing survivability in a software system requires knowledge of known or predictable threats [13]. Banjeree et al state that survivability deals with three basic kinds of threats: attacks, failures, and accidents [14]. However, systems in tactical environments have to deal with an additional threat which is the environment itself. To account for the characteristics of tactical environments we mapped these characteristics to system requirements and then to cloudlet features described in the following subsections.

A. *Pre-Provisioned Cloudlets with App Store*

In previous work [3] we presented several options for cloudlet provisioning and selected to combine pre-provisionng (*Cached VM*) with app store capabilties (*Cloudlet Push*). This combination leads to lower energy consumption on the mobile device during provisioning, places less requirements on mobile devices, and simplifies provisioning in tactical environments. The following characteristics provide the rationale for meeting system requirements:

- Capabilities as Services: Based on the definition of a service in the context of service orientation, each Service

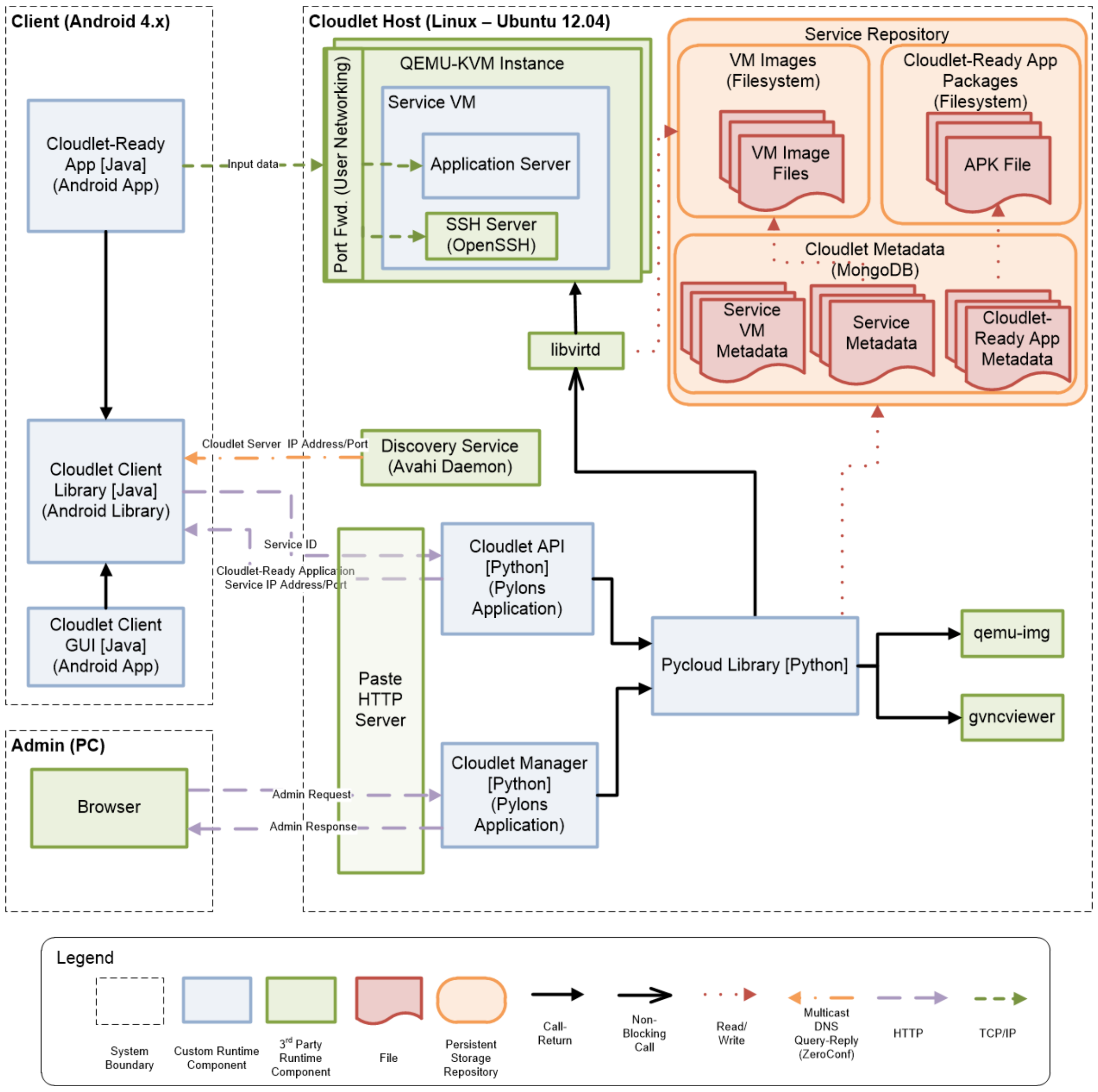
Client (Android 4.x)
Cloudlet-Ready App [Java] (Android App)
Input data
Cloudlet Client Library [Java] (Android Library)
Cloudlet Client GUI [Java] (Android App)
Cloudlet Host (Linux – Ubuntu 12.04)
Port Fwd. (User Networking)
QEMU-KVM Instance
Service VM
Application Server
SSH Server (OpenSSH)
libvirtd
Service Repository
VM Images (Filesystem)
VM Image Files
Cloudlet-Ready App Packages (Filesystem)
APK File
Cloudlet Metadata (MongoDB)
Service VM Metadata
Service Metadata
Cloudlet-Ready App Metadata
Cloudlet Server IP Address/Port
Discovery Service (Avahi Daemon)
Service ID
Cloudlet-Ready Application Service IP Address/Port
Cloudlet API [Python] (Pylons Application)
Paste HTTP Server
Pycloud Library [Python]
qemu-img
gvncviewer
Admin (PC)
Browser
Admin Request
Admin Response
Cloudlet Manager [Python] (Pylons Application)
Legend
System Boundary
Custom Runtime Component
3rd Party Runtime Component
File
Persistent Storage Repository
Call-Return
Non-Blocking Call
Read/Write
Multicast DNS Query-Reply (ZeroConf)
HTTP
TCP/IP

Fig. 1. Tactical Cloudlet Architecture

VM provides a self-contained capability and exposes a simple interface. Service Metadata is used by the cloudlet discovery protocol to inform mobile devices of available capabilities (Section IV-C).

- Virtual Machines as Service Containers: VMs can be started and stopped as needed based on number of active users (which is typically bounded in tactical environments because group size is known) therefore providing scalability and elasticity to efficiently support as many active users as possible.
- Request-Response Nature of Interactions Between Clients and Cloudlets: In the case of computation offload, tactical cloudlets are best fit for applications based on stateless, request-response, client/server interactions. This type of interaction enables easy detection of failed communication between mobile devices and cloudlets, as well as minimal effect on mobile devices if computation needs to be restarted or migrated.

B. *Standard Packaging of Service VMs*

To support ease of deployment and re-deployment of cloudlets, there is a standard format for Service VMs so these can be easily loaded from the cloudlet disk drive, an enterprise Service VM repository, a thumb drive (if allowed), or a mobile device connected via USB to the cloudlet. The file format is .csvm, which is a .tar.gz file that contains a JSON file with Service metadata, and two image files (disk and state) that compose the VM Image associated to the Service.

- Service Metadata
  - Service ID (string): unique identifier
  - Service Port (integer): port on which the server inside the VM is listening
  - Version (string): version number
  - Description (string): text description

  - Tags (string): tags used by service queries
  - Number of Clients Supported (integer): number of clients that the server is designed to support
  - Minimum Memory (MB) (decimal): minimum amount of RAM required to run properly
  - Ideal Memory (MB) (decimal): ideal amount of RAM
- VM Image: two image files — one for the disk image and one for the state/memory image — that contain a suspended Service VM and are used to create Service VM Instances
  - Disk Image file (.qcow2): qcow2 file containing an image of the hard drive of a Service VM
  - VM State Image file (.lqs): VM state image in the format that libvirt.save() generates when saving a memory image. It includes the description of the VM that was suspended and the memory state of the VM.

C. Optimal Cloudlet Selection

In a cyber-foraging scenario, there may be more than one cloudlet available for use. In this case, it would be useful to know the characteristics of the available cloudlets so that the "best" cloudlet for a Cloudlet-Ready App can be used. To achieve this, we extended our cloudlet discovery protocol so that cloudlet metadata can be used by the Cloudlet Client Lib to execute an algorithm that selects the "best" available cloudlet. One of the goals of the architecture and implementation is to be able to easily change cloudlet metadata as well as the algorithm for cloudlet selection.

The discovery protocol used by our tactical cloudlets uses Zeroconf (www.zeroconf.org), which in turn uses DNS Service Discovery (DNS-SD) along with Multicast DNS. DNS-SD uses DNS queries and replies in a specific format defined for service discovery. A multicast address is used in order to allow a client to request a specific service without knowing the IP addresses of the available cloudlets (equivalent to a more focused broadcast request). The Zeroconf discovery protocol is limited in the type and size of the information that can be returned. In particular, broadcast information is static. The data that is needed by Cloudlet-Ready Apps to evaluate cloudlets, such as CPU, memory, disk space, is data that must be dynamically collected/calculated by the server and therefore difficult to broadcast with Zeroconf. To address this, we added an HTTP GET request for metadata that is invoked after a cloudlet has been discovered. The cloudlet responds with JSON-formatted metadata. The process steps are:

1) When the cloudlet starts, its Discovery Service (the Avahi Daemon in our implementation) joins a particular Multicast IP Address as a listener (Cloudlet Multicast IP). Note that this is only done when the cloudlet is started regardless of how many discovery queries are received.
2) When the application using the cloudlet client library wants to discover cloudlets, it sends a DNS-SD query for cloudlet services through Multicast DNS to the Cloudlet Multicast IP address.
3) The query reaches the Discovery Service of all cloudlets in the network through Multicast DNS.
4) Each Discovery Service replies with a DNS-SD response indicating the IP and port of its cloudlet.
5) The client requests cloudlet metadata through an HTTP GET request to each cloudlet that it has discovered.
6) Each cloudlet collects the required metadata and replies to the HTTP request with the information.

The cloudlet metadata model consists of a number of objects that are transmitted in JSON format and can be used by a mobile device to select an optimal cloudlet for offloading:

- CPU metadata (assumes all cores are equal)
  - **cpu.usage** (float, in %): % of CPU in use on the cloudlet, as a whole
  - **cpu.maxCores** (int): number of cores in the system
  - **cpu.speed** (double, MHz): frequency that the cores are working at
  - **cpu.cache** (int, KBs): the amount of cache per core
- Memory metadata
  - **memory.maxMemory** (int, bytes): total amount of physical memory on the cloudlet
  - **memory.freeMemory** (int, bytes): amount of free memory on the cloudlet

The first ranking algorithm we implemented is called CPUBasedRanker. It evaluates the cloudlet metadata to calculate the percentage of CPU power available to execute the offloaded code. This is done using the formula below which represents the percentage of CPU available on the cloudlet at that particular moment.

$$(100.0 * cpu.maxCores) - cpu.usage$$

We performed experiments to evaluate the behavior of the CPUBasedRanker. These were performed on 3 machines acting as cloudlets, with the following characteristics:

- **Cloudlet 1**: i7-4700MQ @ 2.40GHz, 32 GB, 0.1% base CPU load, 1.5 GB base mem usage. Ubuntu 12.04.4 LTS.
- **Cloudlet 2**: Core 2 Q9550 @ 2.83Ghz, 4 GB, 0.2% base CPU load, 1.1 GB base mem usage. Ubuntu 14.04 LTS.
- **Cloudlet 3**: Core 2 Q9550 @ 2.83Ghz, 32 GB, 0.1% base CPU load, 0.7 GB base mem usage. Ubuntu 14.04 LTS.

The purpose of the experiments was to evaluate the response time of a request to a cloudlet and test if the algorithm correctly selected the less loaded cloudlet for each scenario. We used three test apps: FACE-OPENCV (face recognition), SPEECH (speech recognition) and OBJECT (object recognition). Results are shown in Table I. App-Ready Time measures the roundtrip request-response time in seconds of the first request, including the setup of the Service VM on the cloudlet. Average Response Time measures the average roundtrip request-response time in miliseconds of each subsequent request (all the Service VMs are capable of handling multiple users). Each row corresponds to a different scenario in which the three cloudlets in the network are working under different loads. Table I shows that the CPUBasedRanker indeed always selects the cloudlet with the lowest load. However, it not always selects the cloudlet that would deliver the smallest response time because not all the cloudlets in our experiments are equal. Cloudlet 1 is the most powerful cloudlet and has the smallest app-ready and response times. Data from additional experiments showed that even at higher loads Cloudlet 1 would still have smaller app-ready and response times than Cloudlets 2 and 3. Therefore, the CPUBasedRanker is ideal only if all cloudlets have the same characteristics.

This led us to another approach for a ranking algorithm which is to calculate a value that measures the overall performance of the cloudlet. which is not to calculate a single overall performance value, but to calculate a value for CPU power or memory only, in order to measure the computational performance or the memory performance of the system, respectively. Because a Cloudlet-Ready App can choose the selection algorithm to use, it could select based on the feature that is more useful for its particular needs. As such, we could have the following ranking algorithms:

- **CPUPerformanceRanker**: This could be done with an over-simplification: measuring performance using the CPU frequency as a proxy. Though this is not very accurate, it could be used as a rough comparison. A simple formula that could be used to calculate the *available CPU performance* of a cloudlet is
$$((100.0 * cpu.maxCores) - cpu.usage) * cpu.speed$$
This is the same formula used to calculate the percentage of available CPU performance in the CPUBasedRanker, but using that percentage to calculate the ”available speed” for the offloaded code.
- **MemoryPerformanceRanker**: Information on free memory and CPU cache could be combined to measure the *avalable memory performance* of a cloudlet. The following formula could be used, in which A and B are constants that could be calculated to give different weights to cache and RAM memory:
$$(cpu.cache * cpu.maxCores) * A + \\ memory.freeMemory * B$$

D. Cloudlet Management Component

The Cloudlet Manager shown in Figure 1 is a lightweight, web-based interface that enables easy deployment and redeployment of capabilities. It provides the following functionality:

- Service VM creation, edit and deletion
- Service VM import and export
- Service VM Instance start, stop and migration
- Cloudlet-Ready App repository (i.e., app store)

E. Cloudlet Handoff/Migration

A characteristic of cloudlets in resource-constrained environments is that (1) they can be mobile because they could reside on vehicles and (2) clients can easily become disconnected due to mobility and to intermittent network connectivity. A strategy to deal with these problems is to enable manual 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. Our tactical cloudlet implementation uses QEMU+KVM through libvirtd, as shown in Figure 1. The migration process leverages libvirtd to control the migration of a Service VM. The steps of the migration process are:

1) The Cloudlet Server (pycloud) sends Service VM Instance Metadata to the Remote Cloudlet Server.
2) The Remote Cloudlet Server (pycloud) adds the Service VM Instance Metadata to its list of Service VM Instances in the Service Repository.
3) The Cloudlet Server pauses (suspends) the Service VM Instance so that the disk image is not changed while it is being sent.
4) The Cloudlet Server sends the Service VM Instance disk image file to the Remote Cloudlet Server.
5) The Remote Cloudlet Server points (rebases) the Service VM Instance Disk Image File to the path of the Service base VM image file on that Remote Cloudlet.
6) The Cloudlet Server calls the libvirtd function to migrate the VM (XML description and state of the VM, and memory).
7) The Remote Libvirt Daemon receives and stores the VM.
8) The Cloudlet Server notifies the Remote Cloudlet Server that the migration has finished.
9) The Remote Cloudlet Server resumes the migrated Service VM Instance.

To test migration times we used two of the test apps from Section IV-C — SPEECH and OBJECT — plus a third app called FLUID which is a fluid simulation app. The reason for including FLUID is because it is a continuous interaction app rather than a request-response app so we could see the effects of migration on this type of application. Migration was performed between Cloudlet 1 and Cloudlet 3 as defined in Section IV-C. Both cloudlets were connected to the same switch with a 1Gbps link. The roundtrip time for a ping between the two machines is on average 0.2 ms. Service VM Instance image file sizes and average migration times for the test applications are shown in Table II. Note that the VM disk image is not a full image, but a qcow2 Service VM instance image that only contains the differences between the disk image of the Service and the current running instance. Migration time is measured from the moment that the migration button is pushed in the Cloudlet Manager until the Service VM is running again on the Remote Cloudlet Server.

The VM image file sizes are approximately the same, as well as the VM migration times. For the SPEECH and OBJECT apps that have a request-response interaction with the cloudlet the user simply experiences a longer response time if executing the application at the moment of migration. For the

TABLE I
CLOUDLET SELECTION AND EXECUTION DATA

CPU Load (%)			FACE-OPENCV			SPEECH			OBJECT		
Cloudlet 1	Cloudlet 2	Cloudlet 3	Selected Cloudlet	App-Ready Time (s)	Avg Resp. Time (ms)	Selected Cloudlet	App-Ready Time (s)	Avg Resp. Time (ms)	Selected Cloudlet	App-Ready Time (s)	Avg Resp. Time (ms)
0	**50**	**95**	1	3.41	11	1	3.37	3,083	1	3.07	1,815
50	**0**	**95**	2	15.65	9	2	20.09	5,374	2	21.21	3,649
50	**95**	**0**	3	18.13	18	3	19.73	4,575	3	16.00	3,970
0	**95**	**50**	1	2.67	10	1	2.65	3,054	1	3.13	2,066
95	**0**	**50**	2	14.99	10	2	18.96	4,049	2	20.81	3,844
95	**50**	**0**	3	24.84	9	3	19.17	5,017	3	15.70	4,170

TABLE II
VM MIGRATION DATA

App	Service VM Instance Disk Image Size (MB)	Service VM Instance Memory Image Size (MB)	Average Migration Time (s)
SPEECH	8.6	492	6.33
OBJECT	8.7	442	6.23
FLUID	13	919	5.47

FLUID application which constantly receives data from the cloudlet (continuous interaction) the app freezes momentarily during the migration. The migration times for FLUID are slightly better than for the other cases, even though FLUID has the largest VM image sizes. This may be caused by the way in which the memory state is transferred, which is done directly by the libvirt daemons without creating a saved state file. The information in the FLUID memory state may be easier to compress or to efficiently transfer than for the other apps thus decreasing the migration times slightly.

V. CONCLUSIONS AND NEXT STEPS

The paper presents an architecture and experimental results that demonstrate that cyber-foraging using tactical cloudlets increases the survivability of mobile systems, defined as the capability of a system to be able to continue functioning in spite of adversity. The characteristics of tactical environments were mapped to system requirements for survivability and then to tactical cloudlet features. The next steps in this area of our research are to (1) develop and evaluate a set of rankers for different service characteristics, and (2) add capabilities for automated migration that would enable load balancing, similar to what is done in cloud data centers for resource optimization, or to enable migration to a more powerful or nearby cloudlet to improve response time and provide continued operations. Other areas of our research are exploring trusted identities for secure communications and the use of tactical cloudlets for efficient data staging.

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-0002371).

REFERENCES

[1] M. Satyanarayanan, “Pervasive computing: vision and challenges,” *Personal Communications, IEEE*, vol. 8, no. 4, pp. 10–17, Aug 2001.
[2] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, “The case for VM-based cloudlets in mobile computing,” *Pervasive Computing, IEEE*, vol. 8, no. 4, pp. 14–23, 2009.
[3] G. Lewis, S. Echeverria, S. Simanta, B. Bradshaw, and J. Root, “Tactical cloudlets: Moving cloud computing to the edge,” in *Military Communications Conference (MILCOM), 2014 IEEE*, Oct 2014, pp. 1440–1446.
[4] G. A. Lewis, P. Lago, and G. Procaccianti, “Architecture strategies for cyber-foraging: Preliminary results from a systematic literature review,” in *Software Architecture*. Springer International Publishing, 2014, pp. 154–169.
[5] M. Satyanarayanan, G. Lewis, E. Morris, S. Simanta, J. Boleng, and K. Ha, “The role of cloudlets in hostile environments,” *Pervasive Computing, IEEE*, vol. 12, no. 4, pp. 40–49, Oct 2013.
[6] S. Wang, R. Urgaonkar, T. He, M. Zafer, K. Chan, and K. Leung, “Mobility-induced service migration in mobile micro-clouds,” in *Military Communications Conference (MILCOM), 2014 IEEE*, Oct 2014, pp. 835–840.
[7] T. Sookoor, D. Doria, D. Bruno, D. Shires, B. Swenson, and L. Pollock, “Offload destination selection to enable distributed computation on battlefields,” in *Military Communications Conference (MILCOM), 2014 IEEE*. IEEE, 2014, pp. 841–848.
[8] M. T. Maybury, “Cyber vision 2025,” United States Air Force Cyberspace Science and Technology Vision, Tech. Rep. AF/ST TR 12-01, 2012.
[9] M. Salehie and L. Tahvildari, “Self-adaptive software: Landscape and research challenges,” *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.
[10] A. Serageldin, A. Krings, and A. Abdel-Rahim, “A survivable critical infrastructure control application,” in *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, ser. CSIIRW ’13. New York, NY, USA: ACM, 2013, pp. 34:1–34:4.
[11] M. Pokharel, S. Lee, and J. S. Park, “Disaster recovery for system architecture using cloud computing,” in *Applications and the Internet (SAINT), 2010 10th IEEE/IPSJ International Symposium on*, July 2010, pp. 304–307.
[12] B. Thuraisingham and J. Maurer, “Information survivability for evolvable and adaptable real-time command and control systems,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 11, no. 1, pp. 228–238, Jan 1999.
[13] S. Sheard, “11.2. 2 a framework for system resilience discussions,” in *INCOSE International Symposium*, vol. 18, no. 1. Wiley Online Library, 2008, pp. 1243–1257.
[14] S. Banerjee, C. A. Mattmann, N. Medvidovic, and L. Golubchik, “Leveraging architectural models to inject trust into software systems,” in *Proceedings of the 2005 Workshop on Software Engineering for Secure Systems&Mdash;Building Trustworthy Applications*, ser. SESS ’05. New York, NY, USA: ACM, 2005, pp. 1–7.