Towards a new model for cyber foraging

Diogo Lima, Hugo Miranda, François Taïani

Towards a new model for cyber foraging

Diogo Lima
Universidade de Lisboa,
LaSIGE
Lisbon, Portugal
dlima@lasige.di.fc.ul.pt

Hugo Miranda
Universidade de Lisboa,
LaSIGE
Lisbon, Portugal
hmiranda@di.fc.ul.pt

François Taïani
Université de Rennes 1,
IRISA, ESIR
Rennes, France
francois.taiani@irisa.fr

ABSTRACT

Cyber foraging seeks to expand the capabilities and battery life of mobile devices by offloading intensive computations to nearby computing nodes (the *surrogates*). Although promising, current approaches to cyber foraging tend to impose a strict separation between the application state maintained on the mobile device, and data processed on the surrogates. In this paper, we argue that this separation limits the applicability of cyber foraging, and explore how state sharing could be implemented in practice.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Client/server; D.4.7 [**Software**]: Operating System—*Organization and Design*

General Terms

DESIGN, MEASUREMENT

Keywords

Cyber Foraging, Surrogate, Remote Execution, Wireless

1. INTRODUCTION

Cyber foraging seeks to overcome the limitations of wireless mobile devices [1] by opportunistically offloading resource intensive tasks on nearby resourceful surrogate computers. Cyber foraging is particularly useful for CPU-intensive computations running on devices with either limited capabilities or constrained energy resources. Such computations include mobile interactive multimedia applications, that combine rich media with on-line analysis tasks. An example is the cognitive assistance scenario presented in [13], where a face and object recognition application embedded in smart glasses helps assisting Alzheimer's patients in remembering people and everyday tasks.

Cyber foraging system must balance the benefits of remote executions with the additional communication costs (energy, latency) introduced by delegating parts of an application's computation to remote surrogates. One source of overhead stems from the need to exchange enough information about the state of a computation to allow its remote execution. Existing cyber foraging systems are generally oblivious or make little use of the applications' state at the client's side. In this paper, we argue that this knowledge can be used to enhance performance by reducing the number of remote invocations and the volume of data transferred, thus reducing the latency of the computation and the mobile device's energy consumption.

In this paper, we first introduce cyber foraging in more detail (Sec. 2). In Sec. 3 we present a case study of an existing face recognition application and discuss how cyber foraging with state-sharing could help improve its execution. Section 4 sketches the high-level blueprint of a cyber foraging architecture designed to enable state sharing. Finally, in Sec. 5 we discuss how this blueprint offers a path forward to materialize the benefits identified by our case study. Section 6 concludes the paper.

2. BACKGROUND

Existing cyber foraging systems have been exploiting three main mechanims to realize their benefits: Virtual Machine images, mobile code and Remote Procedure Calls (RPC). In the following, we discuss each of them in more detail.

2.1 Virtual Machines

In this approach, the mobile device prepares a virtual machine image of its own execution and sends it to the surrogates [4, 5, 8, 14, 9, 13]. This solution is flexible in the sense that the set of services used by the device can be uploaded to the surrogates upon demand or in anticipation.

In CloneCloud [4, 5] the VM is created in the cloud, and is synchronized either periodically or on-demand with the mobile device. Applications can either be entirely offloaded or executed cooperatively on both entities.

In [8], a centralised registrar helps mobile devices locate adequate surrogates, for example to ensure portability. An interesting aspect of this model is that surrogates are expected to retrieve from the Internet the application software required by the clients for remote execution.

Slingshot [14] assumes that both clients and surrogates have ubiquitous Internet connectivity and defines two classes of surrogates. The first-class replica is an Internet connected server controlled by the mobile user. Second-class replicas are temporary servers available on the mobile device's local network and which retrieve the virtual machine image from the first-class replica. To circumvent the relatively low speed and high latencies of Internet links when compared to local Wi-Fi, Slingshot prefers second-class replicas.

Satyanarayanan's vision [9, 13] equally relies on full Virtual Machine migration. Rather than relying on a distant "cloud", a device can obtain a fast response from a nearby resource-rich cluster, here called *cloudlet*. The mobile device operates as a thin client, with all significant computation occurring in the *cloudlet*. The use of low latency, high-bandwidth network link allows for near realtime interactions, a powerful feature of this approach.

These various approaches demonstrate the flexibility of Virtual Machines when applied to cyber foraging. However, designing a VM-based approach is not a trivial task. Their main drawback is the initialization and migration overhead they typically incur due to the time and data size required to transfer and boot up a virtual machine image elsewhere.

2.2 Mobile Code

Mobile code offers an alternative to Virtual Machines in which only part of the code is transferred to and executed on surrogate machines. This avoids the need to replicate the full state of the mobile device, as a Virtual machine would.

In Scavenger [10], for instance, the source code to be executed as part of the task offloading is uploaded to the surrogates by the mobile devices. Surrogates run a daemon which is responsible for providing an execution environment for the mobile code. The offloading process is automatic. The application programmer annotates each method that may be remotely executed and lets the system decide which functions are transferred to the surrogates. Additionally, the developer can also create mobile code tasks manually. This is done by asking for available surrogates, check whether the task is already installed on that surrogate, install it if necessary and finally, invoke the task at the surrogate.

Compared to the Virtual Machine approach, mobile code is more lightweight. Given that the source code should not exceed a few kilobytes in size, mobile code has a negligible initialization overhead, in contrast with virtual machine images, one order of magnitude larger. However, mobile code is bounded to a specific language, thus limiting the portability needed to ensure a seamless and transparent execution environment for all mobile devices.

2.3 Remote Procedure Calls (RPC)

As with mobile code, RPC-based cyber foraging avoids transferring a device's full state. In the RPC approach, client applications are partitioned into locally executable code and remotely executable services. In contrast to Mobile Code approaches, these services are pre-installed on surrogate computers who offer a RPC like API.

In Spectra [7], for example, the mobile clients run a specific Spectra client, which follows one of a set of possible execution plans for an application. An execution plan lists one or more services, which provide the actual application code that is executed on the surrogates. The execution plan concept was refined in Chroma [1, 2] with the introduction of *tactics*. In a tactics file, the developer specifies the RPC functions that may be called during that operation execution, and the different ways that these functions may be combined to solve the operation. Nevertheless, on both of these systems it is up to the programmer to code the remotely executable services as stand-alone applications to be installed on the surrogates.

Other RPC-based systems include MAUI [6] and Odessa [12] which aim at automatically partitioning application's methods. Applications rely on the MAUI framework to decide which methods should be offloaded. This decision is based upon a call graph created off-line to assess the computing and energy costs of each method and the size and energy consumed to transfer the state remotely. Odessa, on the other hand, proposes a runtime solution that automatically and adaptively makes offloading and parallelism decisions. Instead of estimating costs based on off-line graphs, Odessa uses a greedy algorithm that periodically gathers information from a profiler to estimate the bottleneck of applications in the current configuration. This information is used to estimate whether offloading or increasing the parallelism level of the bottleneck stage would improve performance.

Like the mobile code approach, this model induces a negligible initialization overhead since services are already installed and made available on surrogates. RPCs also offer the portability needed to ensure a seamless and transparent execution environment for all mobile devices by being platform and programming language agnostic.

2.4 Limitations

We consider that a cyber foraging system should exhibit the following properties:

Flexibility so that it can address the requirements of a broad range of applications. Each mobile application has its own set of requirements and the system must provide enough services, functions or libraries to fulfill those requirements.

Lightweight in terms of initialization and transfer overhead. If the boot time or the volume of data transferred is significant, the system becomes unattractive to users since they might have left the surrogate's area by the time initialization is finished or because setting up the remote execution would consume as much energy as executing applications locally.

Portability so that mobile applications do not have to be bounded to a specific execution environment. The system must be able to operate with mobile devices with distinct characteristics and applications using several programming languages.

Table 1 summarizes the characteristics of the cyber foraging paradigms surveyed. Both mobile code and the RPC approaches are more lightweight than Virtual Machine images, which require a significant initialization overhead from the data transfer. RPC is unique in that it is not bounded to a specific platform or programming language. This is of particular importance given that a cyber foraging system must be able to deal with a broad range of mobile device manufacturers and programming languages. In this sense, we consider the RPC-based approach to be the most well suited to implement a cyber foraging system.

Interestingly, most of the systems we have presented are oblivious or have very little knowledge of the application's state at the client side. However, this knowledge can contribute to improve the offloading process performance by reducing the volume of data transferred. Exceptions

Model	Flexible	Lightweight	Portable
VM image	✓		
Mobile code	✓	✓	
RPC	✓	✓	✓

Table 1: Proprieties of each paradigm

are MAUI, CloneCloud and the architecture proposed by Paluska et al. [11]. In MAUI, each method invocation has an additional input parameter used to transfer the application state from the mobile device to the surrogate. The system only transfers incremental deltas of the application state, thus contributing to reduce the latency and traffic. In CloneCloud, when an execution of a process on the mobile device reaches a migration point, the executing thread is suspended and its state sent to the synchronized clone. This state is composed by the virtual state, program counter, registers, and stack. In the work presented in [11], the application state is used within a more general abstraction called *tasklet* which represents a thread of computation. Tasklets are composed of *chunks*, fixed-sized blocks that include data, code and the machine runtime state needed to execute the tasklet on the surrogates.

We envision developing a more advanced version of state sharing by allowing surrogates to keep a local copy of an application's state between successive remote calls. Furthermore, we envisage that this approach can be extended to share the state of multiple application instances running on the same surrogate on behalf of different users, and exploit synergies between these users' different needs.

3. CASE STUDY

Face and speech recognition as well as language translation are some of the applications that would benefit from low-latency, high-bandwidth wireless access to computing resources. In [13], the authors present the *Cognitive Assistance* scenario, where software for scene interpretation, face and object recognition are combined with a camera embedded into the eyeglass frame of a user and with earphones for audio feedback. The goal of the application is to whisper objects and person names to assist Alzheimer's patients in recognizing people and everyday objects.

Face Recognition with OpenCV is an open source application available on Google Play that relies on the OpenCV[1] computer vision software library. When running the application, the mobile device's camera is always on. The application has two operation modes: *training* and *searching*. The training mode allows the user to register a new person. When the application detects a face, the user saves it together with a name to be associated. The searching mode corresponds to the face recognition stage where each time a face is detected in a frame, the application compares it with the faces recorded and returns the name of the matched one (if any).

3.1 Application Experimentation

To understand the impact of face recognition on the resource consumption of mobile devices, the *Face Recognition with OpenCV* application was experimented in a Nexus 7 Android Virtual Device (AVD) running Android version 4.4 with API level 19. Performance was evaluated using the Dalvik Debug Monitor Server (DDMS) debugging tool.

[1]http://opencv.org

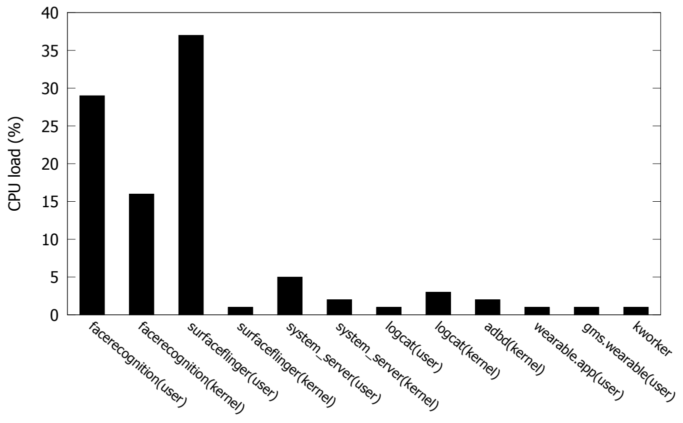

Figure 1: CPU load by process when running the Face Recognition application

Evaluation showed that when neither the face recognition nor any other application is running on the device, the only processes running belong to the Operating System and consume 10% of the CPU time. Once the face recognition application is running, the CPU usage rises to 98%. This confirms that face recognition makes an important use of the CPU, making it a good candidate to benefit from cyber foraging. Figure 1, which depicts the CPU load distribution by the processes when the *Face Recognition with OpenCV* application is running, shows that almost 90% of the CPU load that was previously idle is taken mainly by three different processes: *facerecognition* at the client and kernel level and *surfaceflinger*.

Surfaceflinger is an Android system service responsible for displaying all application and system surfaces on the mobile device's screen. The 37% of the CPU time it consumes can be attributed to its constant display of the frames captured by the device's camera. Therefore, the availability of a cyber foraging system could not help reducing this overhead. However, a cyber foraging system could contribute to reduce the 45% of CPU load spent by both *facerecognition* processes. Figure 2 depicts the top branches of the graph obtained from method profiling in run-time. Graph nodes are of the form: *[ref] callname (<inc-ms>, <exc-ms>,<numcalls>)*, where *ref* is the call reference number; *inc-ms* (resp. *exc-ms*) are the inclusive (resp. exclusive) elapsed time in milliseconds, which counts the time spent in the current method and child methods (resp. not including any child methods); and *numcalls* is the number of calls.

The figure shows two main branches starting from the program's entry point (the root of the tree), each consuming around 50% of the application's CPU time. The right-hand side branch executes methods from the *android.view* package, resposible for the screen layout and user interaction. The importance of this package is attributed to the camera's continuous usage, whose frames are displayed in real-time to the user. The left-hand size branch is concerned with OpenCV library methods. The leaf of this branch is a method called *detectMultiScale*, which belongs to the *CascadeClassifier* class from the OpenCV

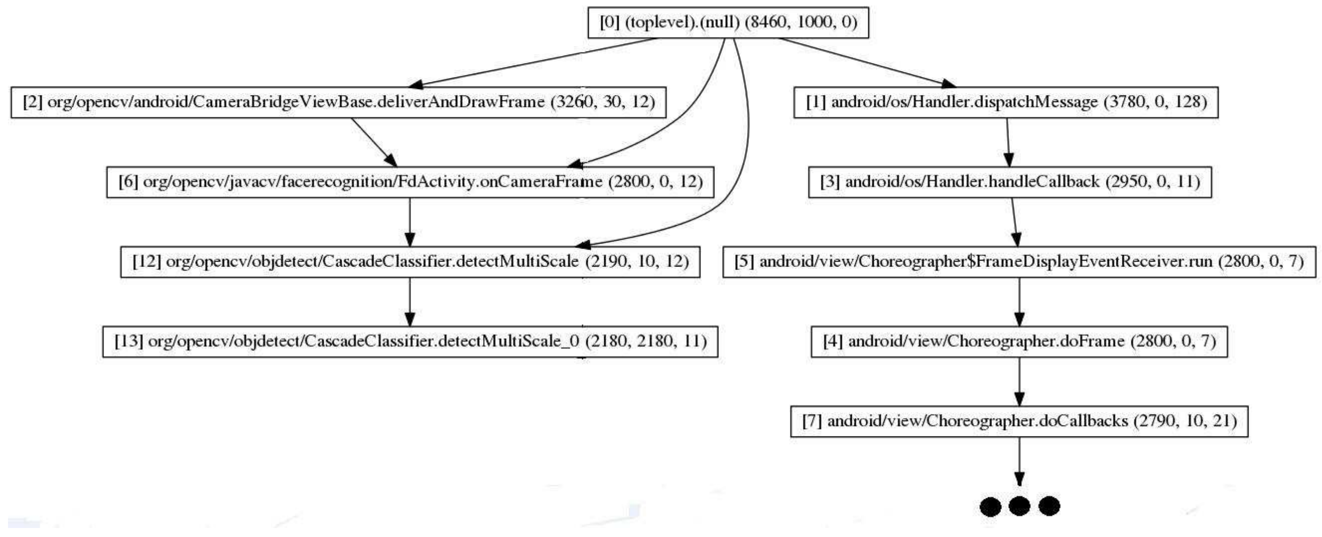
[0] (toplevel).(null) (8460, 1000, 0)
[2] org/opencv/android/CameraBridgeViewBase.deliverAndDrawFrame (3260, 30, 12)
[1] android/os/Handler.dispatchMessage (3780, 0, 128)
[6] org/opencv/javacv/facerecognition/FdActivity.onCameraFrame (2800, 0, 12)
[3] android/os/Handler.handleCallback (2950, 0, 11)
[12] org/opencv/objdetect/CascadeClassifier.detectMultiScale (2190, 10, 12)
[5] android/view/Choreographer$FrameDisplayEventReceiver.run (2800, 0, 7)
[13] org/opencv/objdetect/CascadeClassifier.detectMultiScale_0 (2180, 2180, 11)
[4] android/view/Choreographer.doFrame (2800, 0, 7)
[7] android/view/Choreographer.doCallbacks (2790, 10, 21)

Figure 2: Method invocation graph of the *facerecognition* process

library. This is the most time consuming method accounting for more than 2000 milliseconds, roughly corresponding to 27% of the total execution time.

3.2 Discussion

The method *detectMultiScale* detects objects in a video stream. Its CPU usage suggests that it would be a good candidate to be installed and invoked on a surrogate computer to extend the battery life of the mobile device. In such case, providing the surrogate with the application's active operation mode (training or searching), the face recognition algorithm being used and the list of the user's trained faces would allow the surrogate to continue executing the application once the complex task of face detection is finished. State shared could equally include the personal notes associated to each face as well as a list of the latest faces observed to speed up face recognition.

To decide whether offloading this method would save energy and improve responsiveness, one must consider whether the cost of sending frames (compressed using an appropriate video encoding) to the surrogate would be balanced out by the CPU time saved from not executing *detectMultiScale* on the mobile device. This comparison remains to be performed. However, it should be noted that the application use of streaming does not change this equation considerably, when compared to a design that would only use on-demand static frames for face recognition. Streaming introduces the possibility to use standard video compression techniques to communicate with the surrogate, and renders the potential benefits of cyber foraging more important, but the above trade-off remains.

4. SYSTEM ARCHITECTURE

In our system model we assume that surrogates are resourceful devices, without power constraints, making available both computing power and storage to the clients. Our system is composed of surrogates connected to each other and located at one-hop WiFi distance of the clients. Clients invoke services available on surrogates to alleviate their computing power using an RPC like model. This model is supported by experiences observed in [3] which claim that smartphone users are almost 50% of the time connected to a WiFi access point.

4.1 Application Partitioning

We claim that it should not be entirely up to the mobile application developer to assume the responsibility of application partitioning. With the proliferation of various development tools for mobile applications, the number of application programmers has largely increased with very uneven knowledge basis. As a consequence, the awareness to structure an application to take advantage of cyber foraging may significantly vary, which could result in under-performing mobile applications. Moreover, even the most experienced developer cannot fully anticipate the environment conditions in which the application will run. On the other side, fully automatic solutions may risk to be inaccurate or add a significant overhead on mobile devices from building complex cost assessment models. Therefore, we opted for an adaptive programmer driven partitioning model. We propose to use annotations written by the developer to partition an application from its bytecode into remotely executable methods. A novelty of our model comparing to previous solutions is the presence of an offloading interest for each method. In contrast with previous systems which only used a binary ("yes" or "no") offloading decision, we assign each annotated method with an offloading interest between 0 and 1. Decisions will weight the offloading interest with criteria such as the current network conditions (periodically measured), the current execution load and the list of available services provided by a surrogate, which are announced at discovery time.

4.2 State Sharing

In our approach, we envision clients invoking different services on different surrogates for a same application instance and surrogates having access and keeping application state to provide several services. MAUI, CloneCloud and the model proposed by Paluska et al. [11] are examples of frameworks that consider state sharing between clients and surrogates. As firstly proposed by Paluska et al., we envision that the applications' state will be expressed as *chunks*, a fixed-sized data block with an unique identifier. However, Paluska's et al. solution requires that the machine runtime state is always sent along with the code and the data, and CloneCloud requires sending the virtual state, program counter, registers and stack. MAUI improves this process by sending incremental deltas to its stored state.

Chunks are built by the clients and then sent to the surrogates who keep them. However, in contrast with [11] our approach decouples the control flow from the data flow. Moreover, with the objective of reducing even further the communication burden, our model defines different synchronization levels for chunks: *Eager* chunks will always have to be up to date at the surrogates. In contrast, *lazy* chunks can be used for execution by a surrogate even with previous versions and updated at a more favorable moment (e.g. when bandwidth or RSSI is high).

Considering the execution of the application discussed in the previous section, these chunks could be instances of the objects used by the OpenCV methods. In particular, OpenCV uses mainly two methods for face detection and face recognition. The *detectMultiScale* that allows the face detection and the *predict* method that executes face recognition. This last method relies on an object from the *FaceRecognizer* class. In a cyber foraging scenario, chunks could be attributes or fragments of such object. For example, *FaceRecognizer* can operate with different recognition algorithms such as: LBPH, Fisher or Eigen and maintain the list of *trained* faces. Keeping on surrogates the active face recognition algorithm along with the set of *trained* faces as chunks would avoid the user to send them along with every frame, as well as every time the user starts the application. In such case, since the active recognizing algorithm may be transparent for most users, this could be implemented as a *lazy* chunk, while images and labels would be *eager* chunks since missing their changes would have a direct impact on the user's experience.

4.3 Architecture overview

Figure 3 depicts a broader view of the invocation and state sharing model followed on our approach using the *Face Recognition with OpenCV* application as an example. After the application partitioning is done and methods such as *detectMultiScale* and *predict* are identified as offloading candidates, the decision of delegating those methods is taken by a scheduler present in the client's mobile device, called *Task Distributor*. For example, the *detectMultiScale* method can be executed either locally or remotely depending of the current bandwidth, the surrogate's current execution load and the number of cached faces. Whether offloaded or not, its execution is performed in a component called *Tasklet Executor*, available on both parties.

If offloaded to a surrogate, the method invocation must contain the name of the function requested and the IDs of the chunks needed to complete that execution. To illustrate our example, let us assume that we have a Chunk C1 representing the application's current operation mode (set to *searching*), a Chunk C2 containing the current face recognition algorithm used and a Chunk C3 containing the image recorded and the associated label of a user's relative. Upon receiving a method invocation, the surrogate starts by requesting the needed functions (*detectMultiScale* for example) and C1, C2 and C3 to their respective manager components *Service Locator* and *Chunk Manager*. To overcome the possibility of a missing chunk, the surrogate can ask the mobile device to build and send it. In the figure, we are assuming that C1 is missing in the surrogate. In response to the request sent by the surrogate, the *Chunk Builder* component at the mobile device creates and send it back to the surrogate. Once the execution is finished, results are sent back to the mobile device along with state changed *eager* chunks. An order for searching known faces is an interesting example of a *lazy* chunk, given that having it up to date at the surrogate could speed up the recognition process but it would not be critical for performance.

Clients may communicate with surrogates using distinct protocols (GSM, Bluetooth or WiFi). The *Network Dispatcher* layer is responsible for abstracting these network conditions from the rest of the components.

5. FUTURE WORK

A cyber foraging model with the characteristics above raises a number of interesting research challenges. This section addresses some of them, arranged in different lines.

One class is related with mobile application experimentation, which is currently at an early stage. A larger scale study has to be conducted on mobile applications in order to identify the common characteristics of applications that are suitable for cyber foraging and those that are not. Metrics have to be defined for computing complexity, data transfer sizes among others that might help classify applications as good remote execution candidates. Problems rest on the identification and development of the library that the surrogates should make available to their users. Research must weight efficiency (knowing that the most complete functions require few interactions between mobile devices and surrogates) and applicability (given that the more simple the function, the bigger is the probability that the function can be used in more than one application).

Even if a mobile application is well suited for cyber foraging, the performance of the system will be strongly dependent of the latency of the communication between clients and surrogates. A second line of research consists in investigating an efficient algorithm allowing surrogates to anticipate a client's arrival, thus contributing to improve latency. This will imply the combination of route prediction algorithms with movement pattern studies and profile history to build predictions about users arriving close to a determined surrogate or the sequence of surrogates to be followed (consider for example a shopping mall, where the majority of users follow one of a few routes).

Ensuring user privacy is one of the more challenging questions. End-to-end encryption of messages is a basic security measure addressing this problem. However, security needs to be considered on a broader scale to include chunk security. Consider for example the reuse of a face recognition application by different users in proximity, trying to identify some bystander. In such a scenario, one must consider distinct privacy levels. The biometrics that are fed to the task as well as its results can be made publicly available. However, personal comments about the bystander, retrieved from a user's personal database must be kept confidential and disclosed exclusively for the user that created them. The mechanisms to associate the different privacy levels to tasks and chunks at the programming level and its enforcement in run-time will be equally subject of analysis.

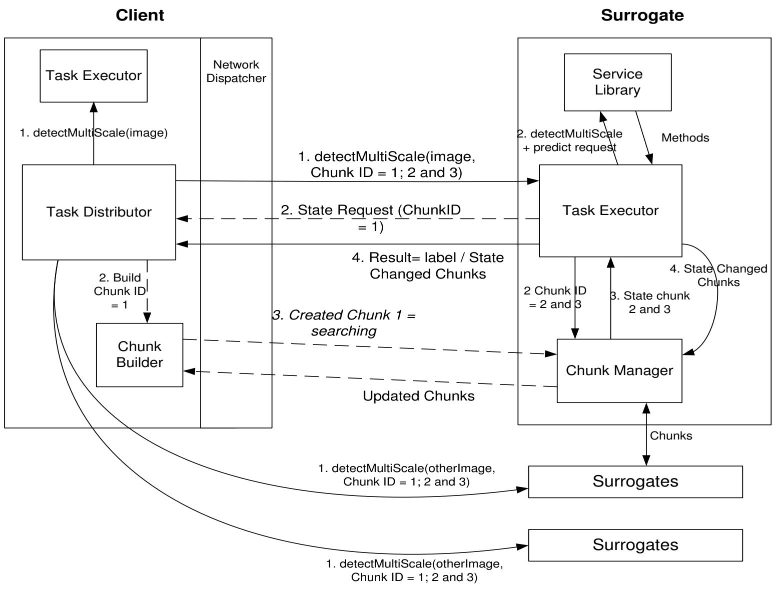
Client
Surrogate
Task Executor
Network Dispatcher
1. detectMultiScale(image)
Task Distributor
2. Build Chunk ID = 1
Chunk Builder
1. detectMultiScale(image, Chunk ID = 1; 2 and 3)
2. State Request (ChunkID = 1)
4. Result= label / State Changed Chunks
3. Created Chunk 1 = searching
Updated Chunks
Service Library
2. detectMultiScale + predict request
Methods
Task Executor
4. State Changed Chunks
2 Chunk ID = 2 and 3
3. State chunk 2 and 3
Chunk Manager
Chunks
1. detectMultiScale(otherImage, Chunk ID = 1; 2 and 3)
Surrogates
Surrogates
1. detectMultiScale(otherImage, Chunk ID = 1; 2 and 3)

Figure 3: Invocation and state sharing model (dashed arrows indicate optional steps).

6. CONCLUSIONS

In this paper we have argued that state sharing can be used in cyber foraging to enhance performance by reducing the volume of data transferred. To make this point we presented a case study of an existing face recognition application and discussed how cyber foraging with state sharing would help improving its execution. We then sketched a high-level blueprint of a cyber foraging architecture designed to enable that state sharing. Finally, we have described the issues that will have to be addressed in order to materialize our proposed high-level architecture into a concrete framework.

7. ACKNOWLEDGMENTS

This work has been partially funded by the French and Portuguese Conferences of University Presidents resp. Rectors (CPU and CRUP) within the project *Cloudlets* of the PAUILF program, by the French ANR project *SocioPlug* (ANR-13-INFR-0003), and by the *DeSceNt* project of the Labex CominLabs (ANR- 10-LABX-07-01).

8. REFERENCES

[1] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The case for cyber foraging. In *10th ACM SIGOPS European Workshop*, 2002.

[2] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying cyber foraging for mobile devices. In *MobiSys '07*. ACM, 2007.

[3] M. Barbera, S. Kosta, A. Mei, and J. STEFA. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *INFOCOM'13*. IEEE, 2013.

[4] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *EuroSys '11*. ACM, 2011.

[5] B.-G. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *HotOS'09*. USENIX Association, 2009.

[6] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code offload. In *MobiSys '10*. ACM, 2010.

[7] J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *ICDCS'02*. IEEE, 2002.

[8] S. Goyal and J. Carter. A lightweight secure cyber foraging infrastructure for resource-constrained devices. In *WMCSA 2004*. IEEE, 2004.

[9] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time provisioning for cyber foraging. In *MobiSys '13*. ACM, 2013.

[10] M. Kristensen. Scavenger: Transparent development of efficient cyber foraging applications. In *PerCom'2010*. IEEE, 2010.

[11] J. Mazzola Paluska, H. Pham, G. Schiele, C. Becker, and S. Ward. Vision: A lightweight computing model for fine-grained cloud computing. In *3rd ACM Workshop on Mobile Cloud Computing and Services*. ACM, 2012.

[12] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *MobiSys '11*. ACM, 2011.

[13] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4), Oct. 2009.

[14] Y.-Y. Su and J. Flinn. Slingshot: Deploying stateful services in wireless hotspots. In *MobiSys '05*. ACM, 2005.