A new system model for cloud offloading

Diogo Lima[1], Hugo Miranda[1], François Taïani[2]

[1] Lasige, Universidade de Lisboa
dlima@lasige.di.fc.ul.pt
hmiranda@di.fc.ul.pt

[2] Université de Rennes 1, IRISA
francois.taiani@irisa.fr

Abstract. As mobile applications become increasingly complex, the ability to offload computation from mobiles to the cloud gains relevance as a technique to alleviate the computing power of resource constrained mobile devices. The existing approaches can either execute independent tasks within a process or full processes through the creation of virtual machine images. However these systems are oblivious or have very little knowledge of the application's state at the client side. This knowledge can be used to enhance the performance by reducing the number of invocations and data transfered. In this paper, we present an architecture towards remote execution of mobile applications that is able to share applications' state between clients and surrogates. This model leverages offloading responsiveness through a lighter communication model, with the possibility to cache and pre-fetch results in anticipation of a client's arrival. We then describe how this system and all of its components will operate on a real case scenario.

1 Introduction

As communication and sensing capabilities of mobile devices increase, mobile applications become increasingly complex and a new class of latency-sensitive computations such as mobile multimedia applications are arising. Cloud offloading aims to alleviate the computing power of resource constrained mobile devices in such situations. Cloud offloading alleviates the cost of executing applications locally, what drains battery and computation resources of the mobile device. However, offloading tasks also consumes energy resources of the mobile device due to sending and receiving data and results, and still adds the latency inherent to this process. Being aware of this trade-off and estimating when each option compensates is key to designing a remote execution system. In contrast with Mobile Cloud Computing, which provides cloud based services to mobile users through the Internet, remote execution systems offer an opportunistic use of available computing resources that are reachable on the local network. However the existing remote execution systems are oblivious or have very little knowledge of the application's state at the client side. This knowledge can be used to enhance the performance by reducing the number of invocations and data transfered.

In this paper, we present a novel cloud offloading system model. Our model assumes that surrogates, i.e. the providers of computing power and storage, are connected to the Internet, can communicate with each other and communicate with clients using a short range, low power protocol like WiFi. The main innovation for our architecture is the sharing of processes' state among users and surrogates. This allows surrogates to be proactive and cache or pre-fetch past results in anticipation of user movement. In addition, the system evaluates the cost of local and remote execution and selects the best option among them.

The remainder of this paper starts with a discussion of the related work on remote execution for mobile applications (Sec. 2). Next, we present a high-level description of our system architecture (Sec. 3) and its application in a real life scenario (Sec. 4). The final two sections are dedicated to the discussion of future work (Sec. 5) and conclusion (Sec. 6).

2 Related Work

Cloud offloading or cyber foraging seeks to overcome the resource limitations of wireless mobile devices [1]. The goal is to dynamically augment the computing resources of a mobile device by offloading their resource intensive tasks on resourceful surrogate computers. This concept may also be referred as remote execution, and it is particularly useful for latency-sensitive computations. To ensure the remote execution of tasks, existing systems can follow three main paradigms: Remote Procedure Calls (RPC), the creation of Virtual Machine images that replicate the mobile devices' execution platforms or Mobile Code. These paradigms will be discussed in separate in the following subsections.

2.1 Remote Procedure Calls (RPC)

In the RPC-based approach, client applications are partitioned into locally executable code and remotely executable services. These services are pre-installed on surrogate computers who then offer a RPC like API. In Spectra [2], the mobile clients run a specific Spectra client which specifies a set of possible execution plans for an application. An execution plan uses one or more services, which are the actual application code that is executed on the surrogates. Chroma [1, 3, 4] was proposed as a refinement of Spectra and so both approaches share similar architectures. The main innovation brought by Chroma was the use of *tactics*, a new way of defining Spectra's execution plans. 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 [5] and Odessa [6] which aim at automatically partitioning application's methods. Applications rely on MAUI framework to decide which methods should be offloaded. This decision is based upon a call graph that is created off-line to assess the computational 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 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.

2.2 Execution environment replication through Virtual Machines

Another class of cloud offloading systems creates virtual machine images that replicate the execution environment present on mobile devices. Users prepare an image of their devices and then send it to the surrogates for execution. CloneCloud [7,8] distributes the computing effort of the resource-intensive parts of an execution with a more powerful clone. This clone is created on the cloud, and the user's mobile device and the clone are synchronized periodically or on-demand. Applications can either be offloaded as a whole to the clone or may be partitioned into pieces and executed on both entities.

In the system proposed by Goyal & Carter [9], the surrogates begin by registering themselves on a central service registrar, detailing which virtual machines they offer. Once a client queries the registrar to find a suitable surrogate for his execution, he receives an IP address of a possible surrogate. The client is then responsible for contacting the surrogate and establish a contract to provide the service for a specified amount of time. Goyal & Carter's system may require an Internet connection in the surrogates, in case they have to fetch the application specific software from the Internet.

Slingshot [10] follows a different vision of a VM-based cyber foraging system. An Internet connection is mandatory for both client devices and surrogates at all time. Slingshot also assumes that there is always one surrogate available: the first-class replica, which is an Internet connected machine controlled by the mobile user used when no surrogates are available on the mobile client local network. First-class replicas have the setback of the relatively low speed and high latencies of Internet links when compared to local Wi-Fi, especially in terms of upload speed from the mobile device to the surrogate. To alleviate this problem, Slingshot assumes that surrogates are capable of receiving virtual machine images from the first-class replica, thus becoming second-class replicas.

Satyanarayanan's vision of *cloudlets* [11, 12] equally relies on full Virtual Machine migration. Rather than relying on a distant "cloud", a device can obtain a real-time interactive response by low latency, one-hop, high-bandwidth wireless access to the nearby *cloudlet*. The mobile device functions as a thin client, with all significant computation occurring in the *cloudlet* .

2.3 Mobile Code

The last approach that can be followed to implement a remote execution system is to have the source code traveling from the user to the surrogates to be executed. In this approach, proposed in Scavenger [13], surrogates run a daemon which is responsible for offering remote access to the mobile code execution environment as well as device discovery. The offloading process can be done in two ways. The first is the *manual* option, where the application can itself ask for a list of available surrogates and install code on those surrogates. The second possibility is an automated mode, where the application programmer annotates each method that may be remotely executed, and lets the system schedule and decide which functions are transferred to the surrogates.

2.4 Limitations

When comparing the three remote execution models, we observe that the use of virtual machines images allows more flexibility than the RPC-based approach in the sense that users can send their own functionality to surrogates on-demand. The installation of "services" in the VM-based approach is done by giving the user complete control over an entire virtual machine image running on a surrogate computer. However, this solution has an obvious setback in terms of initialization overhead due to the time and data size required to transfer and boot up an entire virtual machine image on another computer.

In terms of overhead, the RPC and Mobile Code models are more lightweight, hence more suitable for mobile devices, since they have negligible initialization overhead. Either because the services are already installed and made available on surrogates or because the code transfer required by the Mobile Code model is, in most cases, only a few kilobytes in size. However, the latter approach is bounded to a specific language in which the mobile code must be expressed, thus limiting the portability needed to ensure a seamless and transparent execution environment for all mobile devices. For these reasons, we consider that the RPC-based approach is the most suitable model to implement a remote execution system.

It is interesting to notice that the existing solutions are oblivious or have very little knowledge of the application's state at the client side. This knowledge is however useful to enhance the offloading process performance by reducing the number of invocations and data transfered, thus making it more effective and transparent. Exceptions are MAUI and the architecture proposed by the computing model vision for application cloud computing presented by Paluska et al. in [14]. 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 the work presented in [14] the application state is used within a more general abstraction called *tasklet* which represents a thread of computation. Tasklets are composed of *chunks*, which are fixed-sized arrays that include data, code and the machine runtime state needed to execute the tasklet on the surrogates. We envision extending this knowledge of application's state sharing between clients and surrogates to leverage offloading responsiveness through a lighter communication model with the clients and the possibility to cache and pre-fetch results in anticipation of a client's arrival.

3 System Architecture

Our system is composed of surrogates connected to each other and located at one-hop WiFi distance of the clients. Surrogates are resourceful devices, without power constraints, making available both computing power and storage to the clients. Clients can use different transmission modes to reach surrogates (GSM, Bluetooth or WiFi), although the short range and bandwidth make WiFi the preferred model. The remote execution mechanism is implemented through RPC, with the clients invoking services available on surrogates to alleviate their computing power.

The mobile applications that clients run are first partitioned into methods that can later be executed locally or remotely. It is up to the client to decide which methods will be remotely executed, but it is up to the surrogates to announce their availability to the users. This is done with periodical announcements. Clients can invoke different services on different surrogates for a same application. The main innovation of our system is that the surrogates have access to and keep the needed applications' state locally to execute the invoked services. As firstly proposed in [14], the applications' state will be expressed as *chunks*, a fixed-sized array with an unique identifier. Chunks are built by the clients (by a component called *Chunk Builder*) and then sent to the surrogates who keep them (on a component named *Chunk Manager*). But, in opposition to MAUI, states may be on different synchronization levels. *Eager* chunks always have to be up to date at the surrogates. If a chunk is tagged as *lazy*, surrogates can execute tasks with previous versions. Our design is comparable to MAUI in the worst case scenario where every invocation needs to receive the update of an *eager* chunk. Otherwise, since we maintain *lazy* synchronizations, surrogates can execute tasks with chunks that might not have been synchronized yet, thus reducing the volume of transfers and the burden of keeping data up to date. By keeping state at the surrogates, this model is also expected to outperform [14] where the machine runtime state is always included in the tasklets along the code and the data needed to execute them.

Before the client is be able to offload his tasks, a few operations must be performed. Surrogates periodically announce their services. Advertisement messages are produced by a surrogate component named *Service Announcer* and include information about the current execution load of the surrogate (provided by the *Load Monitor* component) and the list of available services from the *Service Library* component. Since surrogates are connected to each other, this list of available services can be updated by requesting services present on other surrogates. The *Service Locator* component keeps the index of the own services of a surrogate and also forms a service graph of the other surrogates services.

On the client side, the application targeted for remote execution must firstly be partitioned from its bytecode into remotable execution methods. This partition is made by identifying the annotations written by the developer. For each annotated method, the decision of offloading is taken by the the scheduler, named *Tasklet Distributor*. The methods decided to be offloaded are transformed into service invocations to the surrogate. Otherwise, these methods are executed locally by the *Tasklet Executor* component (present both on the client and on the surrogate). An addition of our model comparing to previous solutions is the presence of offloading interest for each method. In contrast with previous systems which only used a binary ("yes" or "no") offload. We envision to assign each annotated method with an offloading interest between 0 and 1, that will help the client scheduler to decide which methods will be offloaded. Decision will weight the criteria with the current network conditions (periodically measured by an *Environment Monitor*) and the current execution load announced by the surrogates at discovery time.

After discovering which surrogates are available and the running application properly partitioned, a user can invoke services of the surrogates. Since clients can communicate with surrogates using distinct protocols (GSM, Bluetooth or WiFi), communication is handled on the client side by a *Network Dispatcher* component, responsible for abstracting the network conditions from the rest of the components. Invocations contain the IDs of the chunks needed to complete those executions. The surrogates ask their *Chunk Manager* component to return the referred chunks and pass them to the *Tasklet Executor* component, where executions are preformed. If the chunks needed are not available on a surrogate, it asks the mobile device to build and send them. Once the execution is finished and if the states of the chunks changed, the chunks are updated back to Chunk Manager. If appropriate (considering future use and network conditions) modified chunks are also returned to the client along with the results from the invocations. Figure 1 depicts the overall architecture of the system. Surrogate discovery, tasklet invocation, chunk creation and result return are identified by the arrows numbered from 1 to 5. Dashed arrows indicate optional steps.

Finally, the surrogates have additional features that allow to cache and prefetch previous tasklets results for a user that usually requests a determined tasklet or a set of taskslets at a certain location and time. This is done with the help of a *Mobility Profiler* and a *Route Prediction* components that rely on the historical data of remote executions of each user and movement prediction algorithms to anticipate the client's arrival to the surrogate region.

4 Application to a real life scenario

This section describes an application of our framework in a face and object recognition application, embedded for example in smart glasses assisting Alzheimer's patients [11]. In this example, a camera for scene capture is built into the eyeglass frame of a user along with earphones for audio feedback. These hardware components are combined with software for scene interpretation, facial and object recognition and context awareness in such a way that, when a user looks at a person or an object for a few seconds, that person or object's name is whispered in the user's ear along with possible additional cues. In order to implement this scenario on our model we assume that the surrogate possesses a set of services required to identify faces and objects on an image, classify them, obtain relevant

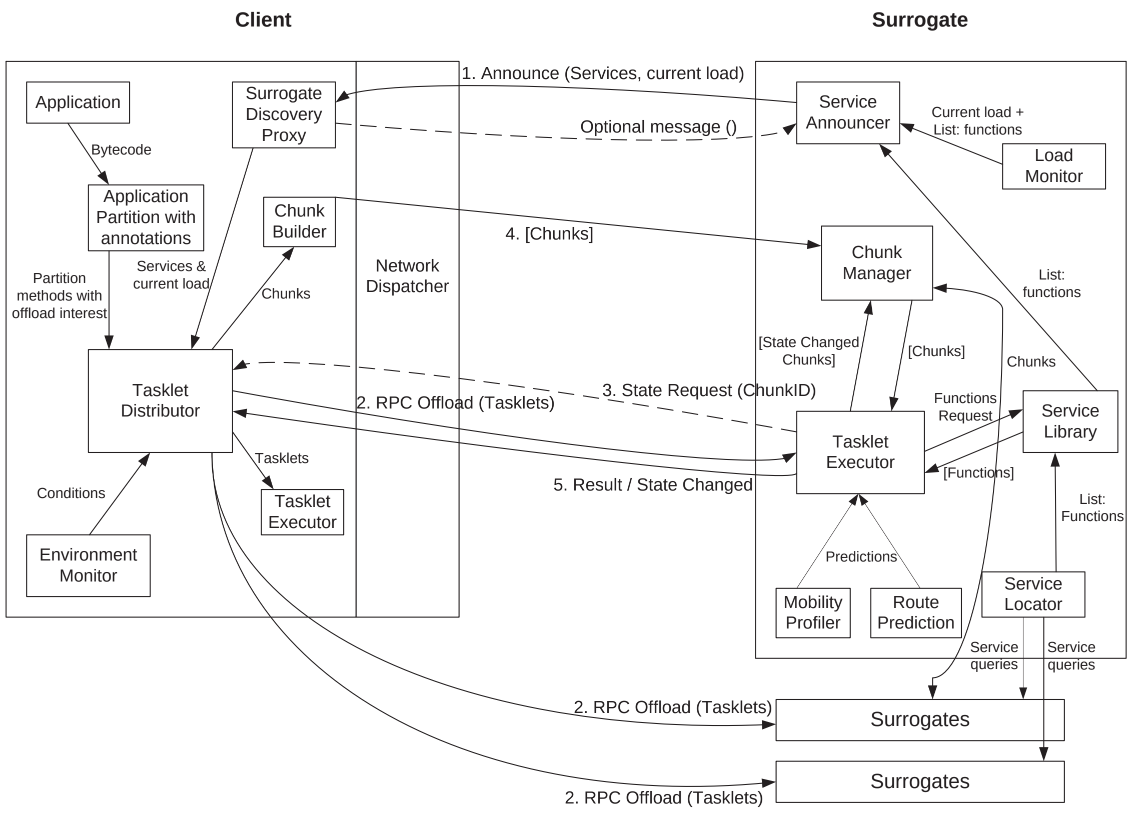

Fig. 1. Architecture design

characteristics of the items, and compare such features with existing ones on familiar faces and objects.

Following this example, the system at the client side starts by statically analyzing the byte code to identify partition methods and offloading interests for those methods. In such scenario, these may include extracting faces and objects from scenes; obtain relevant features from the data (such as face regions, variations, angles, and measurements like eye spacing); comparison methods; and classification algorithms to form the identification task. These partition methods are sent to the Tasklet Distributor component.

Meanwhile, the surrogate's Service Announcer periodically announces its availability by sending its available service list and its current execution load.

The client receives the announcement broadcast through his Surrogate Discovery component. The later sends the list of available services provided by the newly discovered surrogate and its current execution load to the Tasklet Distributor component.

The Tasklet Distributor component gathers the partitions identified earlier and the surrogate information received from the discovery module, and decides which will be executed locally and which will be offloaded, thus creating tasklets. The local tasklets are sent to the Tasklet Executor present on the client's device. The offloaded tasklets are invoked on the Tasklet Executor of the surrogate through RPC. In this case, such invocation may contain the first service ID (extracting faces and objects from a scene) and the images that are being gathered by the client's eyeglasses.

As a result of an invocation, the surrogate Tasklet Executor receives the chunks for that execution from its Chunk Manager (responsible for keeping and managing the chunks shared with the client). If necessary, in case where the surrogate does not have a specific chunk or needs its latest version to execute a tasklet, the surrogate reaches the client for the needed chunks. For example, in the case of the comparison methods used for face and object recognition, the surrogates may query the client to determine whether the client has familiar faces or objects to be compared with the scenes received.

On the client side, the request of a missing or an outdated chunk from the surrogate is transferred to the Chunk Builder component that is responsible to build and send it back to the surrogate. A new chunk can be built with the characteristics of a new familiar face or object that the scenes being analyzed by the surrogate need to be compared with.

As the tasklets are being executed on the surrogate's Tasklet Executor, the results are sent back to the client along with possible new chunk states, changed during execution. For example, such state change can be the identification of a person or an object at a determined location or the notification that a set of scenes have already been analyzed.

5 Future Work

A remote execution model with the characteristics above raises a number of interesting research challenges. This section addresses some of them, arranged in different classes.

One class is related with the performance of the system, which will be strongly dependent of the latency of the communication between clients and surrogates. The framework must find, for example, an efficient algorithm allowing surrogates to anticipate a client's arrival. 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).

The second class is related with the capability of the framework to address the requirements of a broad range of applications. Problems rest on the identification and development of the library functions that the surrogates should provide to their users. Responses 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).

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 larger scale, given that chunks or tasklet results could contribute to improve the overall performance of the system. Consider for example the reuse of a face recognition tasklet by different users in proximity, trying to identify some bystander. In such a scenario, one must consider distinct privacy levels. The metrics that are feed to the tasklet 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 tasklets and chunks at the programming level and its enforcement in run-time will be equally subject of analysis in the future work.

6 Conclusion

In this paper we have proposed an architecture towards remote execution of mobile applications that is able to share applications' state between clients and surrogates. This model leverages offloading responsiveness through a lighter communication model, with the possibility to cache and pre-fetch results in anticipation of a client's arrival. We have described each component of our system and how they intercommunicate with each other. We have also described how our system would operate on a real life scenario and how the users would benefit from using a remote execution system with application state sharing for a communication and performance improvement. Finally, we have described our initial issues raised from designing such system and that will be addressed in the future.

References

1. Balan, R., Flinn, J., Satyanarayanan, M., Sinnamohideen, S., Yang, H.I.: The case for cyber foraging. In: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop. EW 10, New York, NY, USA, ACM (2002) 87–92
2. Flinn, J., Park, S., Satyanarayanan, M.: Balancing performance, energy, and quality in pervasive computing. In: ICDCS. (2002) 217–226
3. Balan, R.K., Satyanarayanan, M., Park, S.Y., Okoshi, T.: Tactics-based remote execution for mobile computing. In: Proceedings of the 1st International Conference on Mobile Systems, Applications and Services. MobiSys '03, New York, NY, USA, ACM (2003) 273–286
4. Balan, R.K., Gergle, D., Satyanarayanan, M., Herbsleb, J.: Simplifying cyber foraging for mobile devices. In: Proceedings of the 5th International Conference on Mobile Systems, Applications and Services. MobiSys '07, New York, NY, USA, ACM (2007) 272–285

5. Cuervo, E., Balasubramanian, A., Cho, D.k., Wolman, A., Saroiu, S., Chandra, R., Bahl, P.: Maui: Making smartphones last longer with code offload. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services. MobiSys '10, New York, NY, USA, ACM (2010) 49–62
6. Ra, M.R., Sheth, A., Mummert, L., Pillai, P., Wetherall, D., Govindan, R.: Odessa: Enabling interactive perception applications on mobile devices. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services. MobiSys '11, New York, NY, USA, ACM (2011) 43–56
7. Chun, B.G., Maniatis, P.: Augmented smartphone applications through clone cloud execution. In: Proceedings of the 12th Conference on Hot Topics in Operating Systems. HotOS'09, Berkeley, CA, USA, USENIX Association (2009) 8–8
8. Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: Elastic execution between mobile device and cloud. In: Proceedings of the Sixth Conference on Computer Systems. EuroSys '11, New York, NY, USA, ACM (2011) 301–314
9. Goyal, S., Carter, J.: A lightweight secure cyber foraging infrastructure for resource-constrained devices. In: Mobile Computing Systems and Applications, 2004. WMCSA 2004. Sixth IEEE Workshop on. (Dec 2004) 186–195
10. Su, Y.Y., Flinn, J.: Slingshot: Deploying stateful services in wireless hotspots. In: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services. MobiSys '05, New York, NY, USA, ACM (2005) 79–92
11. Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N.: The case for vm-based cloudlets in mobile computing. IEEE Pervasive Computing **8**(4) (October 2009) 14–23
12. Ha, K., Pillai, P., Richter, W., Abe, Y., Satyanarayanan, M.: Just-in-time provisioning for cyber foraging. In: Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services. MobiSys '13, New York, NY, USA, ACM (2013) 153–166
13. Kristensen, M.: Scavenger: Transparent development of efficient cyber foraging applications. In: Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on. (March 2010) 217–226
14. Mazzola Paluska, J., Pham, H., Schiele, G., Becker, C., Ward, S.: Vision: A lightweight computing model for fine-grained cloud computing. In: Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services. MCS '12, New York, NY, USA, ACM (2012) 3–8