Fast dynamic deployment adaptation for mobile devices

Tim Verbelen
Ghent University - IBBT,
Department of Information Technology
Gaston Crommenlaan 8/201
9050 Gent, Belgium
tim.verbelen@ugent.be

Tim Stevens
Ghent University - IBBT,
Department of Information Technology
Gaston Crommenlaan 8/201
9050 Gent, Belgium
tim.stevens@ugent.be

Filip De Turck
Ghent University - IBBT,
Department of Information Technology
Gaston Crommenlaan 8/201
9050 Gent, Belgium
filip.deturck@ugent.be

ABSTRACT

Mobile devices that are limited in terms of CPU power, memory or battery power are only capable of executing simple applications. To be able to run advanced applications we introduce a framework to split up the application and execute parts on a remote server. In order to dynamically adapt the deployment at runtime, techniques are presented to keep the migration time as low as possible and to prevent performance loss while migrating. Also methods are presented and evaluated to cope with applications generating a variable load, which can lead to an unstable system.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Management, Measurement, Performance, Experimentation

Keywords

Smart client, Adaptive software, Software partitioning

1. INTRODUCTION

As mobile devices get more CPU power, memory and network connectivity, more and more applications are developed for the mobile platform. However, these devices still fall short to execute the same applications as their desktop counterparts due to resource constraints and ever increasing resource demands. Moreover the limited battery power also constrains the possible applications on these devices. This puts limits to the complexity and the quality of mobile applications.

In order to cope with the mobile device limitations the thin client concept has regained intrest [3]. The mobile device then only acts as input/output device and all processing is done on a remote server. However, this approach usually uses much bandwidth and always introduces extra latency. In [14] we propose a smart client approach to solve this problem. Here the resources on the mobile device are used to process a part of the application, while other parts are outsourced at runtime to remote servers while the bandwidth needed is kept as low as possible. A middleware layer continuously monitors the running applications and decides which components are outsourced. A proof-of-concept implementation showed the effectiveness of our approach in a scenario of an application generating a high and constant load. However, the redeployment of the application takes some time (more than 50 seconds) during which the application performance drops significantly.

In this paper we extend the framework presented in [14] to achieve truly dynamic deployment adaptation. Therefore two main issues are tackled. First we reduce the migration time and the performance loss involved. Second we experiment with an application model generating variable load. This model is more appropriate for most applications that rely on user input to trigger computations. Based on these experimental data we propose optimizations to improve the offloading decision of the framework.

The paper is structured as follows. In the next section important related work is discussed. Section 3 presents the architecture of our offloading middleware and Section 4 introduces the augmented reality use case used to evaluate the framework. Section 5 discusses the steps taken to speed up migration while Section 6 presents results concerning variable load. Section 7 ends this paper with conclusions and future work.

2. RELATED WORK

The rise of mobile computing sets the need to cope with changing contexts (e.g. network connectivity) and limited resources, which introduced the paradigm of adaptive software [1]. In this paper we adapt the software by partitioning the application and by outsourcing parts of it to remote servers at runtime. In this section we discuss important related work on software adaptation by partitioning.

Early research on how to transform legacy software into distributed applications uses an extra preprocessing step before compiling to insert remote invocation code. JavaParty [9], Doorastha [2] and AdJava [4] expect the programmer to insert special keywords indicating parts of the software to be run remotely. This approach has two major drawbacks: the

source code of the application has to be available and the deployment is fixed at compile time.

The first problem is addressed by Addistant [11] and J-Orchestra [13], by impacting the Java bytecodes to create a distributed application. To determine a good partition the first uses a policy file, while the latter uses an offline profiling phase to find the best partition. Similarly, Coign [7] distributes Microsoft COM objects using offline profiling and binary rewriting. However, these systems still end up with static partitions, while in the mobile context the partitions should be able to adapt to context changes.

An adaptive offloading framework is presented by Gu et al. [5]. This research focuses on memory constraints, and in order to deal with limited memory capacity on the mobile device a fuzzy control model is used to offload classes at runtime to a remote server. Runtime information of the application is fetched by extensive monitoring of objects and method calls, introducing a significant overhead. Ou et al. focusses on minimizing response time to the user by offloading classes, proposing a (k+1) partitioning algorithm that results in one part to execute on the device and k parts to execute on k remote servers [8]. Han et al. present a flow based algorithm to partition software which is evaluated by simulation [6].

Our middleware presents a solution that, as opposed to [9], [2], [4], [11] or [13] does not modify the original source code nor the bytecodes. The service oriented architecture of OSGi [12] is used to offload parts of the application on a software component level. Instead of optimizing memory usage [5] or response time [8], our main goal is to improve performance for CPU intesive applications, while minimizing the needed bandwidth. Lightweight profiling is used to instruct the offloading decision at runtime and to be able to react on changing device context.

3. OFFLOADING MIDDLEWARE

In this section we will describe the base framework for adaptive offloading. The architecture is shown in Figure 1. The client agent runs the main management loop and periodically checks the monitor information. The Resource Monitor provides information about the complete system, monitoring the total CPU and bandwidth used. The Bundle Monitor is a more fine grained monitor that observes the managed software bundles' CPU and communication cost. Using this information the Client Agent builds up a weighted graph in which the software bundles are represented as nodes and the communication between components is represented as edges. When a decision is made to redeploy the system the Graph Cutter calculates the optimal deployment that outsources enough CPU cost to the server while minimizing the needed bandwith. The Distributor makes sure the software bundles are in- and/or outsourced as needed. When a bundle is outsourced a proxy bundle is generated on the client side that provides the same interface as the original but forwards calls to the server bundle. The Server Agent initializes and manages software components that are moved to the server.

Currently the framework is configured to cope with CPU intensive applications that need to be partly outsourced in order to be able to run the application. An offloading decision is made when the percentage CPU time used is above a predefined threshold of 80%. The optimal cut is the one that keeps the CPU cost of the bundles at the client side below the threshold and minimizes the used bandwidth. The framework is implemented in Java because a cross platform solution was needed in order to execute components on mobile devices as well as on server machines. The OSGi framework [12] offers us a component model and forces applications to be built up from seperate units of deployment. It is used in combination with R-OSGi [10] which handles proxy generation and remote method invocations. Until now the framework also only supports migration of stateless components.

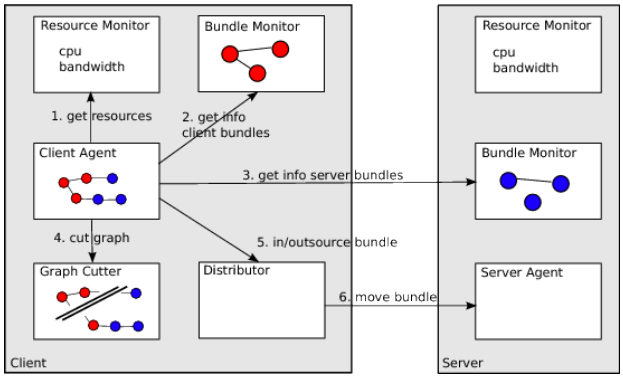

Figure 1: The architecture of our offloading framework.

The framework as described in [14] has two main drawbacks in order to allow true dynamic deployment adaptation. The first one is the migration time that takes up to 10 seconds per bundle and the performance loss while migrating. The second one is the fact that it is evaluated using application components generating a constant load, while this is almost never the case in a real application. Load variations in bundles could lead to a constant in- and outsourcing of bundles which is not desired. The first drawback will be tackled in Section 5, while the second problem will be handled in Section 6.

4. USE CASE: AUGMENTED REALITY

As an example use case we consider an augmented reality (AR) application. By showing the images captured by a mobile phone's camera on the display the user can look to his device as if it were a window on reality. On this display the reality can be augmented with virtual objects or overlayed with useful information about the user's surroundings. The architecture of an example AR application is shown in Figure 2.

The Capturer will continuously fetch 800x480 images of the camera and push these to the Renderer, that combines this image with virtual content given by the ContentProvider. The FeatureDetector will pull the latest available image for processing and detect some feature to analyze. The Analyzer will take time to analyze the found features and instructs the Matcher to match them against known templates. When information is found the ContentProvider will be activated and the content will be rendered.

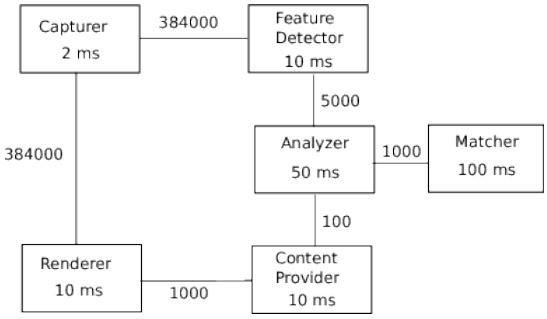

Figure 2: The architecture of an example AR application. The components are annotated with the time it takes to execute a method call and the edges with the communication cost (bytes) of such a call.

To evaluate our framework we created a synthetic application reflecting this scenario. The framework will outsource CPU intensive components (i.e. Matcher, Analyzer) while keeping some processing local to minimize the bandwidth (FeatureDetector). The performance of the application is measured by counting the amount of method calls to the Render bundle which reflects the achieved frames per second in a real application.

5. IMPROVING MIGRATION TIME

To evaluate the effectiveness of the framework, experiments were done using the synthetic application presented in the previous section. Tests were run on a Nokia N900 mobile device with a 600 MHz ARM Cortex A8 processor and 256 MB RAM running Maemo 5 Linux. The server machine is equipped with an Intel Core 2 DUO P8400 CPU clocked at 2.26GHz running Ubuntu Linux. The monitor info is fetched every second and every 5 seconds the framework evaluates if a new deployment is needed. The synthetic application is executed on the mobile device and after a minute the outsourcing framework is activated. This calculates the best cut and outsources components until the CPU usage drops below the threshold of 80%. We measured performance of the application as the number of frames that would be rendered per second. The results are shown in Figure 3. The framework chooses – as expected – to outsource both the Matcher and the Analyzer bundle, which takes almost half a minute during which the performance drops significantly. This is unacceptable for real-time use, so optimizations are needed in order to lower the outsource time and performance drop.

In order to optimize the outsourcing of a bundle we take a detailed look at all necessary steps that need to be done.

1. **Connect to the ServerAgent service.**
   The ServerAgent will take care the initialization and management of bundles at the server side. We use R-OSGi to get a reference to this service. R-OSGi will look for the service and return the reference of a proxy service at the client that will forward calls to the remote ServerAgent. If a proxy service is not yet instantiated R-OSGi will generate one on the fly using bytecode manipulation.
2. **Resolve dependencies of the bundle to migrate.**
   Next the dependencies of the bundle that will be migrated are inspected. These are needed to link the right services between the client and the server after the migration.
3. **Send the bundle to the server.**
   The bytecodes of the bundle to migrate are serialized and sent to the server.
4. **Install the bundle at the server.**
   The ServerAgent installs and initializes the bundle at the server side. R-OSGi is used to expose the service interface to the client.
5. **Generate proxy of the outsourced bundle.**
   R-OSGi will search for the exposed service interface on the server and generate a local proxy that forwards calls to the remote instance of the bundle.
6. **The local instance of the bundle is stopped.**
   On the client device the bundle is stopped. All bundles using this service will now use the proxy implementation and method calls will be forwareded to the remote instance.

Measuring time spent in each of these steps showed that more than 60% of the migration time was spent in step 5. Also the first step took significantly longer for outsourcing the first bundle. This indicates that most time is used to generate proxy bundles. The reason is that the procedure to generate and initialize proxy bundles at runtime uses bytecode manipulation, reflection and custom class loading, which are rather slow. Also knowing that the migration procedure is called when the CPU usage is above 80%, CPU resources on the device are scarce to perform these operations.

To avoid the expensive proxy generation at migration time we introduce proactive generation of the proxy bundles. Instead of letting R-OSGi take care of the proxy generation when we first need the remote service, we generate proxy bundles for each application bundle upfront when the application is started. At migration time step 5 is limited to injecting the properties of the server (e.g. ip address and port number) in the proxy bundle corresponding with the application bundle to migrate.

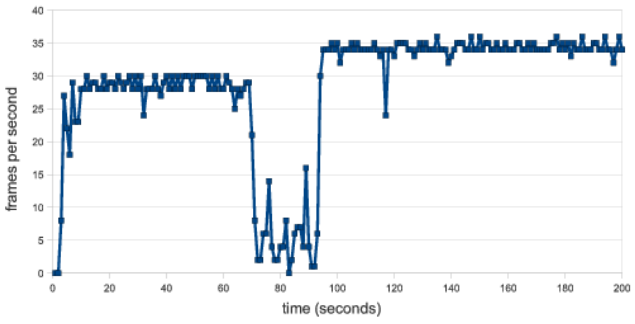

Figure 3: The performance graph of the synthetic application before optimizations. The outsourcing of 2 components takes almost 30 seconds.

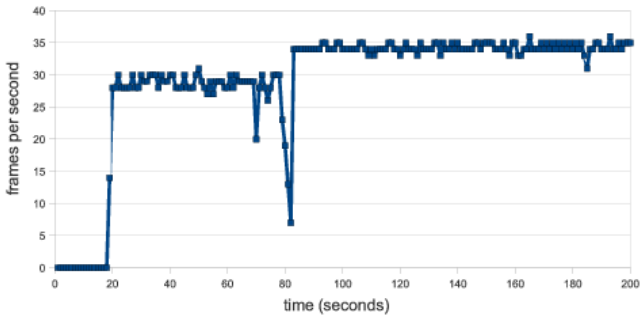

Figure 4: The performance graph of the synthetic application with proactive proxy generation. The outsourcing of components now only takes 3 seconds.

The effect is shown in Figure 4. The migration of the 2 components now takes less then 3 seconds, and the framerate drop is less intrusive. This comes at the cost of an extra startup time of around 15 seconds, but this cost is only introduced once, while time is saved for each migration.

6. EFFECTIVENESS UNDER VARIABLE LOAD

Until now we always used components generating a constant load. In real applications however the load generated will vary and depend on the user input, for example in our augmented reality application the time spent analyzing and matching will depend on the number of features found in the input images. This could lead to an unstable system where components are constantly in- and outsourced. To illustrate this we adapted the Analyzer component from Figure 2 to generate a changing load. Every 10 seconds the time spent in a call to the Analyzer will change between 100ms and 2ms. Every 5 seconds the client agent calculates the optimal deployment and in- or outsources bundles as needed. The result is shown in Figure 5 where the percentage CPU time used is plotted for the client device and the remote server.

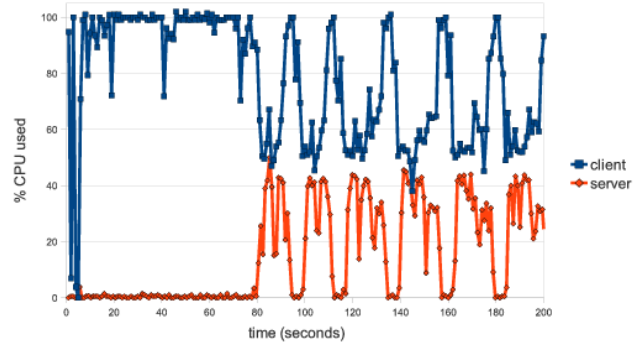

Figure 5: The percentage CPU time used on the client and server. The Analyzer bundle will constantly be in- and outsourced leading to an unstable system.

Each time the Analyzer is in the 100ms phase the framework will decide to outsource the component to the server. On the other hand, when the bundle is in the 2ms phase, it will be insourced again to the client in order to save bandwidth. When a component quickly varies in generated load this can lead to an unstable system that is constantly migrating bundles form client to server and vice versa. This indicates that a snapshot of the generated load is not enough to instruct the offloading decision, but that history should be taken in account. In the following we introduce two ways to cope with the variability of the load, both working on the history of the past monitor information. The first one uses the average of the load generated in a past time window, a second one uses the maximum value.

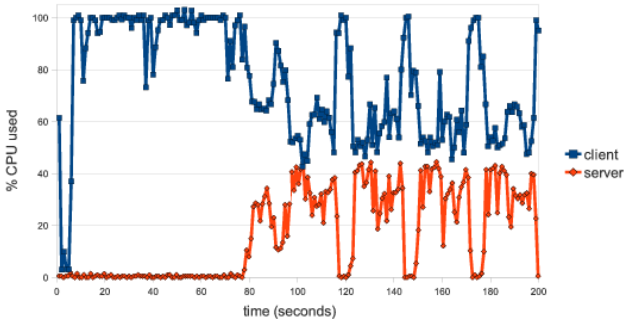

Figure 6: The percentage CPU time used on the client and server. Because the monitor values are averaged, the framework will be more robust to changes in the load.

6.1 Average load generated in the past

One way to stabilize a varying signal and eliminate peaks is by averaging over a time window. This way a short drop in load of a component will not immediatly lead to insourcing that component. The result of averaging the monitored values is shown in Figure 6. A time window of 15 seconds is used.

When the load of the Analyzer bundle drops, the average will slowly decrease and the bundle will not be insourced immediatly. As more and more small values come in the time window the bundle load will keep on decreasing until the decision is made to insource the bundle. This is shown in Figure 6 where less frequently a wrong insource decision is made. This shows that averaging the monitor values makes the framework more robust to varying load. The amount of history used will influence how fast the framework reacts to changes. However, using the average load will lead to putting components at the client side who's peak load can not be handled. Thus, a better approach would be to watch for the peak load of a bundle inside the past time window.

6.2 Maximum load generated in the past

Another approach for preventing the system to react on short variations in the load is to take the maximum value from the past time window. Using a time window of 15 seconds the system now becomes stable as depicted in Figure 7. Once outsourced the framework will not try to insource the bundle because the time window is longer than the period of low load. Altough the generated load keeps varying, it stays below the 80% threshold at the client.

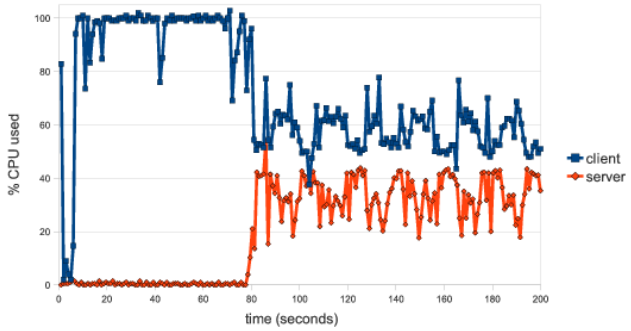

Figure 7: The percentage CPU time used on the client and server. Using the peak monitor value of from the past time window results in a stable system.

A drawback of this method is that one rare short peak in the load will cause the system to outsource the bundle. The question to ask then is which situation is preferable: execute the short peak load on the device and save bandwidth or outsource the bundle and keep the CPU used at the client low. This question can be answered by taking in account the battery power used in both situations and use minimal energy as a goal.

7. CONCLUSIONS AND FUTURE WORK

In this paper we extended our offloading framework to make fast dynamic redeployment possible. By proactively generating proxy bundles the migration time is decreased tremendously, while also diminishing the performance loss during migration. In order to cope with bundles generating varying load the history of the monitor information is used to prevent the system from becoming unstable.

Intresting future work is to trade of the use of CPU power against the use of bandwidth. In this context energy would be a better goal to minimize. Also the support of stateful migration is considered as future work.

8. ADDITIONAL AUTHORS

Additional authors: Bart Dhoedt (Ghent University - IBBT, email: `bart.dhoedt@intec.ugent.be`).

9. REFERENCES

[1] A. Al-bar and I. Wakeman. A Survey of Adaptive Applications in Mobile Computing. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 246–251. IEEE Computer Society, 2001.

[2] M. Dahm. Doorastha – a step towards distribution transparency. In *JIT*, 2000.

[3] L. Deboosere, B. Vankeirsbilck, P. Simoens, F. De Turck, B. Dhoedt, Demeester Piet, M. Kind, F.-J. Westphal, T. Abdeslam, and T. Plantier. MobiThin management framework: design and evaluation. In *Proceedings of the 3rd international workshop on Adaptive and dependable mobile ubiquitous systems*, pages 25–30. ACM, 2009.

[4] M. M. Fuad and M. J. Oudshoorn. Adjava: automatic distribution of java applications. In *ACSC '02: Proceedings of the twenty-fifth Australasian conference on Computer science*, pages 65–75, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[5] X. Gu, A. Messer, I. Greenberg, D. Milojicic, and K. Nahrstedt. Adaptive offloading for pervasive computing. *IEEE Pervasive Computing*, 3(3):66–73, 2004.

[6] S. Han, S. Zhang, J. Cao, Y. Wen, and Y. Zhang. A resource aware software partitioning algorithm based on mobility constraints in pervasive grid environments. *Future Gener. Comput. Syst.*, 24(6):512–529, 2008.

[7] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 187–200, Berkeley, CA, USA, 1999. USENIX Association.

[8] S. Ou, K. Yang, and J. Zhang. An effective offloading middleware for pervasive services on mobile devices. *Pervasive Mob. Comput.*, 3(4):362–385, 2007.

[9] M. Philippsen and M. Zenger. Javaparty – transparent remote objects in java. *Concurrency: Practice and Experience*, 9(11):1225–1242, December 1997.

[10] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-osgi: distributed applications through software modularization. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 1–20. Springer-Verlag New York, Inc., 2007.

[11] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of "legacy" java software. In *Object-Oriented Programming*, pages 236–255. Springer-Verlag, 2001.

[12] The OSGi Alliance. *OSGi Service Platform, Core Specification, Release 4, Version 4.2*. aQute, September 2009.

[13] E. Tilevich and Y. Smaragdakis. J-orchestra: Enhancing java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19(1):1–40, 2009.

[14] T. Verbelen, R. Hens, T. Stevens, F. De Turck, and B. Dhoedt. Adaptive online deployment for resource constrained mobile smart clients. In *MOBILWARE '10: Proceedings of the 3rd international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*. ICST, 2010.