Adaptive Online Deployment for Resource Constrained Mobile Smart Clients

Tim Verbelen, Raf Hens, Tim Stevens, Filip De Turck, and Bart Dhoedt

Ghent University - IBBT, Department of Information Technology,
Gaston Crommenlaan 8 bus 201, 9050 Gent, Belgium

Abstract. Nowadays mobile devices are more and more used as a platform for applications. Contrary to prior generation handheld devices configured with a predefined set of applications, today leading edge devices provide a platform for flexible and customized application deployment. However, these applications have to deal with the limitations (e.g. CPU speed, memory) of these mobile devices and thus cannot handle complex tasks. In order to cope with the handheld limitations and the ever changing device context (e.g. network connections, remaining battery time, etc.) we present a middleware solution that dynamically offloads parts of the software to the most appropriate server. Without a priori knowledge of the application, the optimal deployment is calculated, that lowers the cpu usage at the mobile client, whilst keeping the used bandwidth minimal. The information needed to calculate this optimum is gathered on the fly from runtime information. Experimental results show that the proposed solution enables effective execution of complex applications in a constrained environment. Moreover, we demonstrate that the overhead from the middleware components is below 2%.

Keywords: Middleware, Pervasive computing, Offloading, Software partitioning, Smart clients.

1 Introduction

Although mobile devices gain more and more capabilities, mobile applications still have to cope with much less resources than their desktop or server counterparts. Limited memory capacity, CPU speed, network bandwidth and battery power constrain the complexity of the applications. For advanced applications such as augmented reality the programmer has to trade accuracy or robustness for an acceptable framerate [14].

One solution to cope with device limitations is to use the thin client computing model. The mobile device then only handles input from and output to the user, while the application logic is executed on a remote server. This concept dates back to the era of mainframe computers, but recently revived for business desktop applications because it facilitates centralized management of software and reduces hardware cost of client devices. Examples of such systems are Citrix ICA (Independent Computing Architecture) [25] and Sun Ray [24]. The biggest

Y. Cai et al. (Eds.): Mobilware 2010, LNICST 48, pp. 115–128, 2010.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2010

problem to use the thin client setup in a mobile environment is to cope with the varying properties of a wireless network. It is shown that latency is an important limiting factor for thin clients over a WAN [15] and they are also not resilient to data bursts as discussed in [23]. Hence, executing all application logic on the server is not the optimal solution.

Another solution consists of adapting the application to the capabilities of the mobile device by replacing some parts of the software by other, more lightweight components. In [10] a framework is presented which switches between components depending on the context. However, this solution will mostly result in a degraded application and heavily depends on the application developers' willingness to provide different versions of the components. It's also impossible to run an application that needs more resources than the maximum available on the device, which is still fairly limited.

In this paper we propose a middleware solution for smart clients where the application is dynamically divided between the mobile client and a remote server. By choosing the optimal deployment we lower the CPU usage and minimize the consumed bandwidth to be able to run demanding applications on the device. The optimal deployment can change over time as the context in which the application is executed will also change. Contrary to the local adaptation approach discussed above, the proposed solution will not result in a degraded version.

The remainder of this paper is structured as follows. In the next section we discuss related work. Section 3 presents a typical use case used throughout this paper and Section 4 will outline the architecture of the system. In Section 5 the implementation details and design issues of the different components are discussed. Our experimental results are presented in Section 6 and finally Section 7 concludes this paper.

2 Related Work

Since the rise of state-of-the-art middleware such as CORBA [8] and Java RMI [20], research has been done to transform legacy software into distributed applications. JavaParty [17], Doorastha [3] and AdJava [7] make a Java application distributed by preprocessing its source code and generating remote invocation code. The programmer decides which part of the software will run remotely by using special keywords. Big drawbacks of this approach are of course that the source code has to be available and the deployment is fixed at compile time.

Addistant [26] and J-Orchestra [27] try to solve this by manipulating Java bytecodes. The first requires a policy file to decide where to partition, while the latter also does offline profiling of the application to aid finding a good partitioning. A similar approach is used in Coign [12] where an application built from Microsoft COM objects is distributed using offline profiling and binary rewriting. Still these systems end up with static partitions, which are not optimized for the mobile use case.

Gu et al. present an adaptive offloading framework that can offload parts of the software at runtime [9]. The goal is to cope with the limited memory capacity of mobile devices. A fuzzy control model is used to trigger offloading. To get runtime

information about the software an application execution graph is maintained by extensive monitoring of objects and method calls, which introduces a significant overhead.

A widely used approach to calculate the optimal partitioning, is to represent the software as a weighted graph, and transform the deployment problem into a graph partitioning problem. This way diverse algorithms from graph theory can be used to solve the problem. Stoer and Wagner [21] describe an algorithm to find the minimum cut to divide the graph in two partitions. The Kernighan-Lin heuristic is an iterative procedure that converges to a local optimum [13].

Ou et al [16] and Han et al [11] present graph partitioning algorithms aimed specificly at the problem of partitioning software for mobile devices. The first describes the (k+1) partitioning algorithm that results in one unoffloadable partition and k disjoint offloadable partitions. The latter transforms the graph to a flow network and computes the maximum flow to find the optimal deployment.

To offload parts of the application there has to be a server infrastructure available. Storz found a synergy between pervasive computing and grid computing, introducing the Grid as a platform for ubiquitous applications [22]. Buyya et al. envision Cloud computing as the technology to offer computing services anywhere in the world on demand [2]. Emerging Cloud platforms like Amazon Elastic Compute Cloud (EC2)[4] or OpenNebula [18] will offer us the necessary computing power to enhance the abilities of our mobile devices.

The solution presented in this paper does not modify the original source code nor Java bytecodes. Instead, it uses the extensible and service oriented architecture of OSGi [1] to offload parts of the software. While others offload to reduce memory usage [9] or battery consumption [11], we investigate how to improve performance for CPU intensive applications, while minimizing the needed bandwidth. We collect data from runtime profiling in order to offload without a priori knowledge of the software and to be able to adapt at runtime when the device context changes.

3 Use Case - Augmented Reality Game

As an example use case we present an augmented reality (AR) game. On a head mounted display the player sees the environment captured by a webcam, augmented with virtual items. The user must be able to move freely in the environment and thus all processing is done by a mobile device that is connected wirelessly to a back end server. Besides the images also other sensor information, such as GPS or accelerometers can be used to determine the location of the player. Objects can be recognized from the image stream and trigger virtual objects to be displayed.

In order to do all this processing and still achieve an acceptable framerate it will be necessary to offload some of the processing components to a remote server. However, a pure thin client model will fail because of the high bandwidth requirements to send all the image and sensor data to the server, and the latency that will be introduced between the capturing the environment and displaying the video stream.

4 Smart Client Architecture

Figure 1 presents the architecture of our management framework. On both client and server a Resource Monitor tracks the resource usage of the system (step 1) and a Bundle Monitor gathers information about individual software bundles (step 2 & 3). The Client Agent will call these bundle monitors to get an overview of the current resource usage and to construct a weighted graph of the components. In this weighted graph, different components are represented by nodes and communication between components results in edges between their corresponding nodes. Nodes are weighted with the CPU usage of the components and edge weights reflect the amount of data exchanged.

This graph is then offered to the Graph Cutter (step 4) that will calculate the best graph cut. This is the cut that minimizes the bandwidth, while making sure the CPU usage on the mobile devices does not exceed a predefined threshold. By putting this threshold on 80% we can make sure there is no resource contention on the client. One could also lower this threshold when the objective is for example to extend battery life.

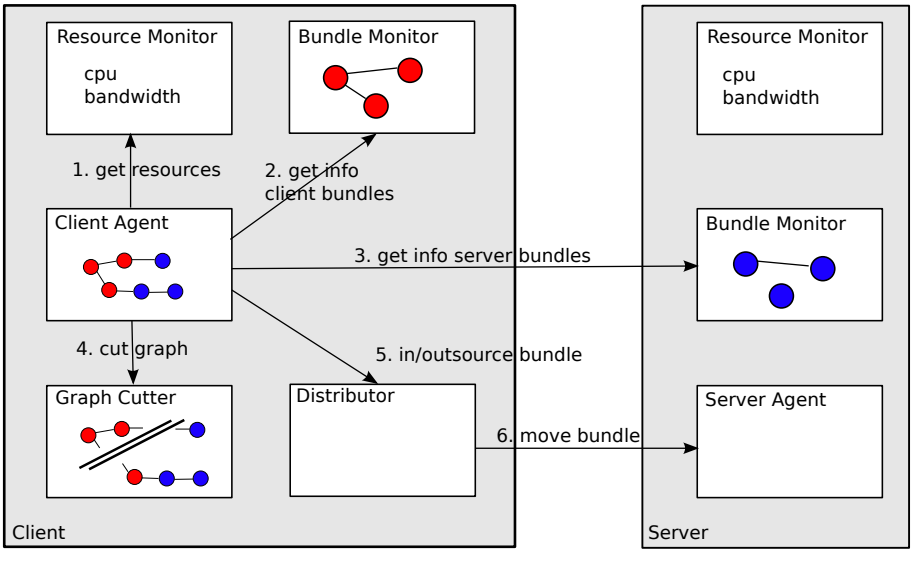

Fig. 1. Smart Client Architecture: The Client Agent gets the resource usage of the system and individual software bundles from the Resource Monitor and Bundle Monitors. The Graph Cutter then calculates the best partitioning after which the Distributor is instructed to in- or outsource some components. The Server Agent takes care of the initialization of bundles at the server side.

Although the Graph Cutter is deployed on the client side in Figure 1, this is not a requirement and the calculation of the best cut can also be offloaded to the server to conserve client resources when the application structure complexity

increases. However for small graphs the required CPU time to calculate the best cut is negligible and it isn't worth the communication cost of outsourcing this calculation. Subsequently the Client Agent will instruct the Distributor to migrate some components to or from the server if necessary (step 5). The Server Agent will make sure that the migrated components are started or stopped correctly at the server side (step 6).

The Client Agent performs these actions in a control loop. Periodically it fetches the monitor information and builds up a global graph of the distributed application. It then decides whether to recalculate a better deployment or not.

The Graph Cutter calculates the minimal cut where the sum of the node weights on the client partition cannot exceed a certain threshold. The algorithm used is an extension of the Stoer and Wagner [21] minimum cut algorithm. When the minimum cut found by Stoer and Wagner does not meet the maximum client weight constraint we add the graph together with the found cut to the queue. For each of the edge found in the cut, we investigate the graph with that edge's weight set to infinity. That way this edge will not be in the new solution. If the new solution still does not meet the constraint we add it to the queue. This algorithm will search in a breadth-first manner to find a cut that satisfies the maximum client weight contraint. It is shown in pseudocode below. The $MINCUT$ subroutine calls the minimum cut algorithm of Stoer and Wagner and *threshold* represents the maximum client weight contraint.

```
INF_CUT(G : Graph, threshold : Number)
Graph G', G''
GraphCut result, previous
result ← MINCUT(G)
if result.GET_CLIENT_WEIGHT ≤ threshold then
  return result
else
  Queue.ADD(result, cut)
  while Queue ≠ ∅ do
    G', previous ← Queue.POP_FIRST()
    for all edges e ∈ previous do
      G'' ← G'
      G''.SET_EDGE_WEIGHT(e, ∞)
      result ← MINCUT(G'')
      if result.GET_CLIENT_WEIGHT ≤ threshold then
        return result
      end if
      Queue.ADD(result, G'')
    end for
  end while
end if
```

5 Implementation Details

In this section we discuss the implementation issues for the architecture components depicted in Figure 1.

5.1 Core

A high level view of our implementation is shown in Figure 2. The middleware builds upon OSGi [1], a popular module management API. We adapted the OSGi framework to get runtime information about method calls between software bundles. This allows for fine grained monitoring necessary for making the right outsourcing decisions.

OSGi adopts a service oriented model in which an application is built from loosely coupled components called bundles. OSGi bundles communicate through services, which are Java classes published under a service interface in a central service registry. Through this service registry bundles look up services they want to use. OSGi provides a light-weight component model that is well suited for use on mobile devices. We use the OSGi bundle as a unit of deployment that can be deployed either at the client or at the server.

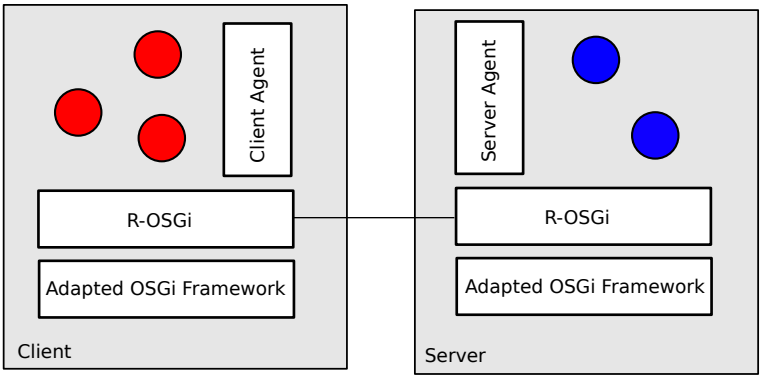

Fig. 2. The client and server run R-OSGi upon an adapted OSGi framework. The agents on both machines monitor the resources and deploys the application bundles (represented by the circles) according to the optimal partition.

On top of OSGi we use R-OSGi [19], which extends the OSGi paradigm for distributed systems. R-OSGi manages interaction between bundles located on different devices by maintaining its own service registry for services that are remotely available. When a bundle requests such a remote service, R-OSGi will generate a local proxy bundle that exposes that service interface locally. When this proxy bundle is called, R-OSGi initiates a remote invocation to forward the method call to the original bundle. This is the core for our management bundles at the client and the server. These management bundles will gather the information about the different running application components, build up the

weighted graph, calculate the best graph cut and migrate bundles from the client to the server or vice versa.

5.2 Resource Monitor

The Resource Monitor tracks the resource usage of the system over time. At predefined time intervals it fetches the used memory, the percentage of CPU usage and the number of bytes sent and received over the network.

To get this information this component has to interact with the operating system which necessitates a platform-dependent solution. We implemented it by reading the */proc/* filesystem on Linux based machines. Alternatively, one could also use native bindings through JNI to interface with the underlying operating system.

5.3 Bundle Monitor

The Bundle Monitor monitors bundle-specific information to estimate the node and edge weights for the application graph. Each time interval we calculate the percentage of CPU time used by each bundle and the amount of data exchanged between all bundles in order to be able to assign graph node weights and edge weights respectively.

To gather this information we adapted the Felix OSGi implementation [5] to intercept all calls between bundles. Instead of registering a service object bound to a certain service interface, we create a dynamic proxy for this interface and register this proxy as service object. The proxy will then send events to $MonitoredCallListeners$ when a method is called and then forward the call to the original service object. The event will notify a listener of the method called, the thread in which the method was called and the arguments used or the object returned. Our Bundle Monitor listens to these events and calculates the size of the data exchanged as if it would have been serialized. This represents the bandwidth cost of an edge if a bundle would be outsourced.

The $java.lang.management.ThreadMXBean$ is used to calculate the CPU usage, which exposes an interface to the JVM and gives us the CPU usage of each thread. For each thread we keep a bundle call stack. When we receive an event of a bundle method call, we push this bundle on the thread's stack, and when we receive an event of a bundle method return, we pop it off the stack. Thus we have to account the execution time of the thread to the bundle that was on top of the stack on that moment. However, we still have to find the bundle that started a thread, since that does not necessarily involve a bundle method call. We do this using the fact that every bundle in OSGi has its own classloader. By matching the classloader that loaded the Thread object to the bundle classloaders we can identify the root bundle of each thread call stack. That way we can map the execution time in a thread to execution time in a bundle. The accuracy of the measurements is dependend on JVM implementation and of course the underlying operating system. Experiments show that we reach an accuracy of tens of milliseconds.

5.4 Client Agent

The Client Agent fetches the information of all monitor bundles periodically and uses it to build up a weighted graph of the application. When the CPU usage exceeds the defined threshold it will request the Graph Cutter to calculate a better graph cut and migrate the necessary bundles.

5.5 Graph Cutter

The Graph Cutter implements the algorithm discussed in Section 4. We implemented two optimizations to the algorithm. As queue we implemented a priority queue that is sorted on the client weight of the cut. This ensures that cuts are processed in order of increasing cut weight.

A second optimization is pruning of the search tree by keeping track of the graphs that already have been partitioned by the minimum cut algorithm. This prevents that equivalent graphs (i.e. with the same edge configuration) are visited more than once by $MINCUT$.

5.6 Distributor and Server Agent

The Distributor and Server Agent will handle the migration of the bundles. At this moment only the migration of stateless software bundles is supported. Stateful migration would mean that a bundle has to be able to serialize its state on the client and restore this state on the server side. Also state changes at the client during the migration have to be propagated to the server. This introduces many difficulties and is considered as future work.

When a bundle is outsourced from the client to the server:

- The Distributor will send the .jar file to the server.
- The Server Agent generates proxy bundles of the services used by the migrated bundle that are only available on the client.
- The Server Agent installs and starts the migrated bundle and makes its services remotely available.
- The Distributor generates proxy bundles of the migrated bundle.
- The Distributor uninstalls the local version of the migrated bundle.

The R-OSGi bundle takes care of the proxy generation, remote invocation and remote service lookup.

Some optimizations can be done to this system by also considering duplication instead of migration. The server could for example keep a bundle cached when it is moved back to the client. When later the client wants to outsource it again to the server this would cut back the migration cost.

6 Prototype Evaluation Results

6.1 Evaluation Setup

We evaluate our framework on a Nokia N900 mobile device with a 600 MHz ARM Cortex A8 processor and 256 MB RAM. This device runs Maemo 5 Linux

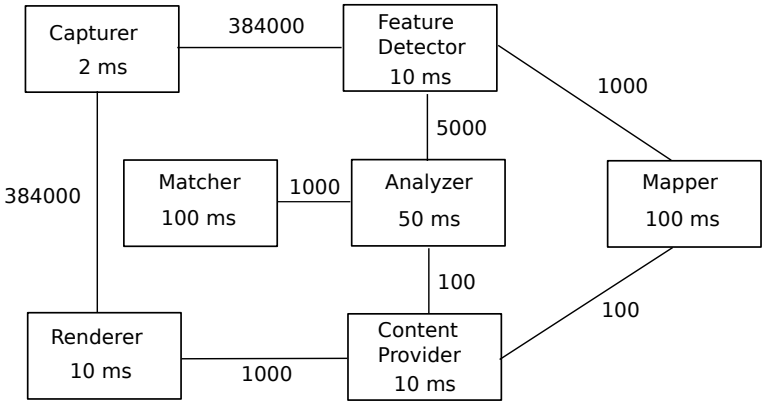

Fig. 3. The architecture and cost graph of our AR application. Different components are annotated with the time it takes to execute a method call and the edges are annotated with the size of the method's arguments in bytes.

as operating system and we used Sun Java SE for Embedded 6 JVM [6]. It also has a camera capable of video recording at a resolution of 800x480. The server machine is equipped with an Intel Core 2 DUO P8400 CPU clocked at 2.26GHz and runs Ubuntu Linux.

To illustrate the operation of the proposed solution, we created a dummy AR application based upon the use case discussed in Section 3. Figure 3 presents the architecture of this application.

The application consists of three concurrent threads. A first thread starts with the Capturer that simulates the fetching of 800x480 images of the camera. It pushes these frames to the FeatureDector and the Renderer. The Renderer will then request the ContentProvider for virtual content and renders it together with the image from the camera on the display to create the augmented reality effect. A second thread starts with the FeatureDetector. This thread grabs the latest image pushed by the Capturer and does a rough first detection step. It then sends image parts to the Analyzer which will analyze it further and generate a pattern characterizing the properties of the image part. This will be matched against a list of known patterns to recognize certain objects by the Matcher. When an object is found it will be notified to the ContentProvider to activate some virtual content. A third and final thread is started by the Mapper component. This thread gets feature points as input of the FeatureDetector and uses an iterative optimization algorithm to estimate the position of the camera in the 3D space. This data is used by the ContentProvider to estimate a correct pose for the virtual content. The more steps the Mapper can perform, the better the estimation of the pose will be.

We implemented this application as stub components that generate a predefined CPU load. We estimated the time to render or to detect some feature rather small (10 ms), and identified some more CPU intensive actions such as the analyzing of an image patch (50 ms), the matching of a pattern (100 ms)

or an iteration in the Mapper thread (100 ms). Moreover the Mapper thread will be greedy and try to fill up all remaining CPU time to get an accurate solution of many iterations. We also estimated the size of the data exchanged between the components. In the example the Capturer will push 800x480 uncompressed grayscale images, while the FeatureDetector will only send cropped parts of 5000 bytes to the Analyzer and some feature points totalling 1000 bytes to the Mapper. The other edges are estimated in a similar way.

We started this application on the mobile device and measured CPU and bandwidth usage of the system. As monitor interval we used one second. We also recorded the number of calls per second to the Renderer, which would reflect the frames per second shown in a real application. After one minute, we activated the Client Agent with a threshold of 80% for the CPU load.

6.2 Experimental Results

Beforehand one can easily see that it will be difficult to get good performance of this application by running it on the mobile device. Depending on the thread scheduling strategy of the JVM, we expect a low framerate, a low rate of analyzing features or a low accuracy of the Mapper as not all threads can be active all the time. It's also clear that a thin client approach would introduce an image stream to the server as input and an augmented image stream back to the client as output and thus would have too high bandwidth requirements.

The resulting graph of the CPU usage is presented in Figure 4. Four phases are marked in the figure. The first minute the Client Agent is inactive and the usage on the client is 100%. In the second phase the Client Agent calculates a

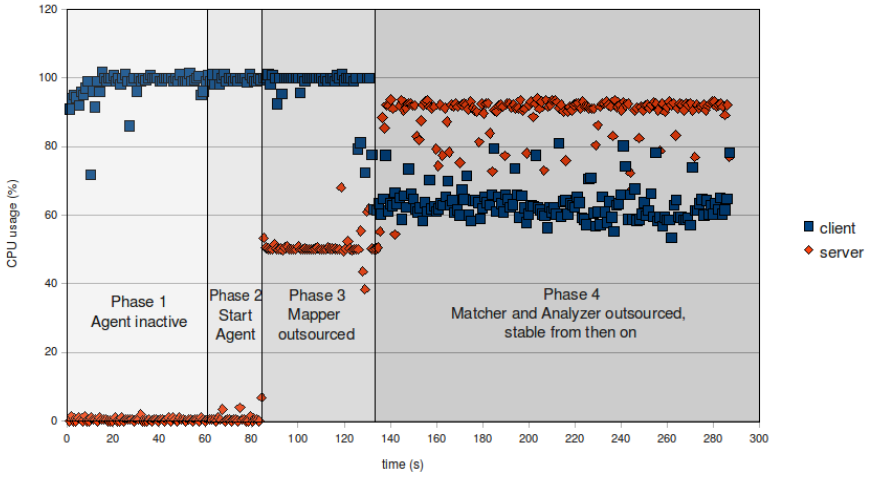

Fig. 4. CPU usage over time. After one minute the Client Agent is activated and starts offloading components to the server until the CPU usage on the client is below the threshold of 80%.

new deployment and decides to outsource the Mapper component. When this outsourcing is complete we see in in the third phase that the CPU usage on the server rises to 50%. This is because the Mapper tries to do as many iterations as possible and thus it uses a complete core of the dual core processor of the server. However, the CPU usage on the client remains 100% so there is still resource contention. The Client Agent will recalculate again and decides to outsource two more components: the Matcher and the Analyzer. When these are outsourced we see in the fourth and final phase that the resource usage on the server rises even more to 90% , but more important, the CPU usage of the client lowers to 60%. Now the CPU usage of the client is below the defined threshold of 80% and the Client Agent will not attempt to outsource any more components.

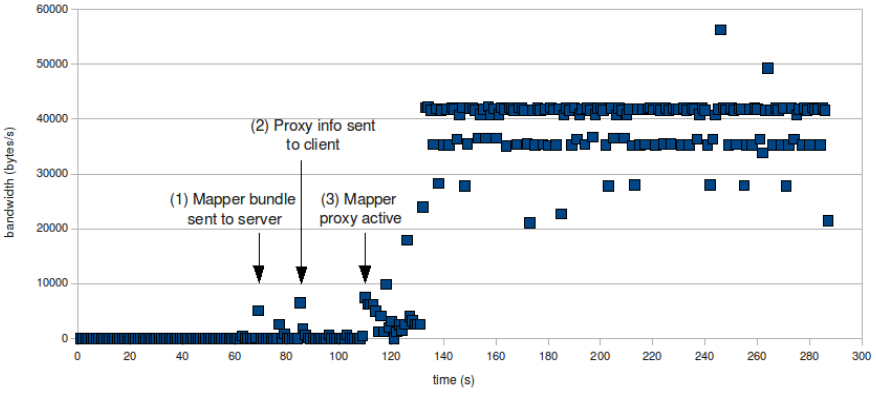

Fig. 5. Bandwidth usage over time. Peaks around 69, 85 and 120 seconds show the outsourcing process of the bundles. The more bundles are outsourced, the more bandwidth is used for the remote method calls.

Figure 5 shows us the bandwidth usage over time. The first 60 seconds there is no bandwidth usage as the Client Agent is inactive. When the Client Agent is activated, there is a peak of around 5 kilobytes after 69 seconds, which is the size of the Mapper jar file that is sent to the server (1).

After 80 seconds the Mapper bundle is started at the server and the the client fetches proxy information from the server, which explains the second peak at 85 seconds (2). However, after the migration the bandwidth drops to zero until 110 seconds (3). This shows that although the Mapper bundle is allready started at the server, the proxy bundle has to be generated and started at the client. Because of the resource contention at the client side it takes a while before the proxy is ready to use.

One also notices the 2 peaks around 120 seconds that represent the sending and receiving of the Matcher and Analyzer bundles and their proxies respectively. After that the communication between the client and server bundles uses about 40 kilobytes per second. Note that this much smaller then the bandwidth required to send the whole video stream in a thin client configuration.

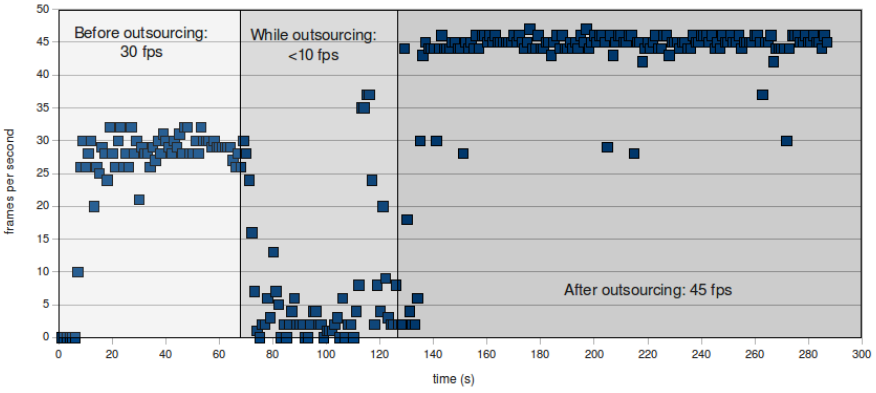

Fig. 6. Frames per second over time. During the outsourcing of the bundles there is a drop in performance. However, when the outsourcing is done there is a gain of 15 frames per second.

Figure 6 presents the frames that would be rendered per second, measured by counting the calls to the Renderer each second. The outsourcing of bundles causes a temporary drop in the performance, because the CPU is used for generating and starting the proxy bundles. However, after the outsourcing of the three bundles the system stabilizes and we see a gain of 15 frames per second while the mobile device only uses 60 % CPU.

Note that a component configured to use 50 ms CPU time will use 50 ms whether it is on the client or the server. Thus, in case of a real application there would also be a performance gain due to the higher clock frequence of the processor at the server side.

Of course the monitoring of the bundles also introduces a certain overhead. However, in our experiments the BundleMonitor never uses more than 2% CPU, which is better than the 7% stated in [9], where a more fine grained monitoring solution is used. To get more detailed data about the monitoring overhead we conducted the following experiment. We used a dummy application of two components that execute 1000 method calls. We ran this application on the mobile device using both the unmodified Felix OSGi framework and our modified framework performing monitoring. By comparing the execution time of the application in the two configurations we can estimate the overhead per method call. We find values between 20 and 40 ns overhead per method call. This shows a very small overhead, knowing that only method calls between different bundles are monitored.

7 Conclusion and Future Work

In this paper we presented an offloading framework for mobile devices that partitions an application and outsources components to a remote server. Built upon the OSGi framework it uses runtime monitoring information to decide which

bundles should be outsourced. By calculating the best partition that restricts the client's CPU usage and minimizes the bandwidth the framework converges to the best deployment of the application, without a priori knowledge. Experimental results with a relevant use case of augmented reality show the effectiveness of our approach, while the overhead introduced by monitoring is small.

Future work consists of optimizing the framework in order to make the impact of migrating bundles as small as possible, for example by caching or pro-active generation the of proxy bundles. In the future we also want to migrate the state of components and deal with fault-tolerance in case of network failures.

References

1. The OSGi Alliance. OSGi Service Platform, Core Specification, Release 4, Version 4.2. aQute (September 2009)
2. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems 25(6), 599–616 (2009)
3. Dahm, M.: Doorastha – a step towards distribution transparency. In: JIT (2000)
4. Amazon elastic compute cloud (EC2), `http://www.amazon.com/ec2/`
5. Apache Felix, `http://felix.apache.org/site/index.html`
6. Sun Java SE for Embedded 6, `http://java.sun.com/javase/embedded/index.jsp`
7. Fuad, M.M., Oudshoorn, M.J.: Adjava: automatic distribution of java applications. In: Proceedings of the Twenty-Fifth Australasian Conference on Computer Science, ACSC 2002, pp. 65–75. Australian Computer Society, Inc., Australia (2002)
8. Object Management Group. Common object request broker architecture: Core specification, `http://www.corba.org`
9. Gu, X., Messer, A., Greenberg, I., Milojicic, D., Nahrstedt, K.: Adaptive offloading for pervasive computing. IEEE Pervasive Computing 3(3), 66–73 (2004)
10. Gui, N., Sun, H., De Florio, V., Blondia, C.: Accada: A framework for continuous context-aware deployment and adaptation. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 325–340. Springer, Heidelberg (2009)
11. Han, S., Zhang, S., Cao, J., Wen, Y., Zhang, Y.: A resource aware software partitioning algorithm based on mobility constraints in pervasive grid environments. Future Gener. Comput. Syst. 24(6), 512–529 (2008)
12. Hunt, G.C., Scott, M.L.: The coign automatic distributed partitioning system. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation, OSDI 1999, Berkeley, CA, USA, pp. 187–200. USENIX Association (1999)
13. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell System Technical Journal 49(2), 291–307 (1970)
14. Klein, G., Murray, D.: Parallel tracking and mapping on a camera phone. In: Proc. Eigth IEEE and ACM International Symposium on Mixed and Augmented Reality ISMAR 2009, Orlando (October 2009)
15. Lai, A.M., Nieh, J.: On the performance of wide-area thin-client computing. ACM Trans. Comput. Syst. 24(2), 175–209 (2006)
16. Ou, S., Yang, K., Zhang, J.: An effective offloading middleware for pervasive services on mobile devices. Pervasive Mob. Comput. 3(4), 362–385 (2007)
17. Philippsen, M., Zenger, M.: Javaparty – transparent remote objects in java. Concurrency: Practice and Experience 9(11), 1225–1242 (1997)

18. OpenNebula Project, http://www.opennebula.org/
19. Rellermeyer, J.S., Alonso, G., Roscoe, T.: R-osgi: distributed applications through software modularization. In: Cerqueira, R., Campbell, R.H. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 1–20. Springer, Heidelberg (2007)
20. Java RMI, http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp
21. Stoer, M., Wagner, F.: A simple min-cut algorithm. J. ACM 44(4), 585–591 (1997)
22. Storz, O., Friday, A., Davies, N.: Towards 'Ubiquitous' ubiquitous computing: an alliance with the grid. In: First Workshop on System Support for Ubiquitous Computing Workshop (Ubisys 2003) in association with Fifth International Conference on Ubiquitous Computing. Citeseer, Washington (2003)
23. Sun, Y., Tay, T.T.: Analysis and reduction of data spikes in thin client computing. J. Parallel Distrib. Comput. 68(11), 1463–1472 (2008)
24. Sun Ray Sun Microsystems, http://www.sun.com/sunray
25. Citrix Systems, www.citrix.com
26. Tatsubori, M., Sasaki, T., Chiba, S., Itano, K.: A bytecode translator for distributed execution of "legacy" java software. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 236–255. Springer, Heidelberg (2001)
27. Tilevich, E., Smaragdakis, Y.: J-orchestra: Enhancing java programs with distribution capabilities. ACM Trans. Softw. Eng. Methodol. 19(1), 1–40 (2009)