2009 11th IEEE International Symposium on Multimedia

eyeDentify: Multimedia Cyber Foraging from a Smartphone

Roelof Kemp, Nicholas Palmer, Thilo Kielmann, Frank Seinstra, Niels Drost, Jason Maassen and Henri Bal
Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
{*rkemp, palmer, kielmann, fjseins, niels, jason, bal*}*@cs.vu.nl*

***Abstract*—The recent introduction of smartphones has resulted in an explosion of innovative mobile applications. The computational requirements of many of these applications, however, can not be met by the smartphone itself.**

The compute power of the smartphone can be enhanced by distributing the application over other compute resources. Existing solutions comprise of a light weight client running on the smartphone and a heavy weight compute server running on, for example, a cloud. This places the user in a dependent position, however, because the user only controls the client application.

In this paper, we follow a different model, called cyber foraging, that gives users full control over all parts of the application. We have implemented the model using the Ibis middleware. We evaluate the model using an innovative application in the domain of multimedia computing, and show that cyber foraging increases the application's responsiveness and accuracy whilst decreasing its energy usage.

***Keywords*-content analysis; object recognition; mobile; middleware; cyber foraging**

I. Introduction

In the past 12 months both Google [1] and Microsoft [2] as well as Research In Motion [3], Sony Ericsson [4] and Nokia [5] have opened or announced application stores targeted at smartphone applications. These stores are much like and pushed by the success of the Apple App Store for iPhone applications [6], which already has exceeded 1 billion downloads. The various application stores have rapidly filled with applications that range from the very simple to highly sophisticated ones that make effective use of the available sensors on the smartphone [7], [8].

Although the smartphone is a suitable device to interact with users [9], it is significantly less suitable for heavy weight computation, due to the hardware limitations (processor, memory and, perhaps the most challenging, energy capacity). Smartphone applications that require extensive computation typically offload their computation to a service in a cloud and communicate with this service using a *client/server* model [10]. Users of such distributed applications can obtain the client application by purchasing it from the market. Although users pay for the client side of the application, they do not control the server side and therefore have no guarantee that this server will be available and will offer the expected quality of service, or any service at all. In other words, the user depends on the goodwill of the service provider, who can, for instance, change or upgrade the communication protocol anytime and charge for a new compatible client application. Furthermore, the service provider has to deal with keeping a server up and running and the developer of the server has to address non trivial scalability, if the application becomes popular very quickly as occurred with services such as Twitter.

In contrast, the *cyber foraging* model [11] offers users of mobile devices control over both the client and the server part of a distributed application. In this model users can exploit various compute resources called *surrogates*, which can be used to run the server part of the distributed applications. Using this model, the smartphone can offload tasks to a user's private compute resources such as laptops, desktops and home servers, or to public resources including clouds [12] and compute clusters.

As we argued in [13], middleware from the existing high performance distributed computing domain has much in common with mobile distributed computing. In this paper we explore the applicability of Ibis, a Java based high performance distributed middleware [14], for cyber foraging on smartphones. We use the Ibis Distributed Deployment System to deploy the heavy weight computations onto surrogates and the Ibis High-Performance Programming System to efficiently communicate with them.

To evaluate the applicability of Ibis as a cyber foraging middleware, we have made a smartphone application, named *eyeDentify*, on top of the Ibis middleware that can perform object recognition, using the camera sensor of the smartphone. Object recognition is a heavy weight computation task and representative for a much broader class of multimedia applications that one would wish to use on smartphones. We compare computation offloading with computing on the smartphone itself in terms of responsiveness, accuracy and energy usage.

The contributions of this paper are:

- We build a practical system that offers the means to deploy heavy weight computing from light weight devices to suitable compute resources.
- We show that multimedia applications with heavy weight computing components benefit from our system with respect to responsiveness and energy usage.

978-0-7695-3890-7/09 $26.00 © 2009 IEEE
DOI 10.1109/ISM.2009.21

- We build a multimedia application that can perform object recognition on a smartphone.

In section II we discuss the Ibis middleware and how it can be used for cyber foraging. Section III describes the object recognition application built on top of the Ibis Middleware. In section IV we present the measurements we performed with this application. Section V puts our work into the context of the related work and in section VI, we conclude.

II. Cyber Foraging with Ibis

In this section we describe the Ibis high-performance distributed computing middleware and how it is used for cyber foraging on smartphones. Since Ibis is a Java-based middleware and because the open source Android operating system offers the most complete virtual machine that can run Java code on smartphones, we focus on using Ibis on Android.

The Ibis middleware is originally designed for the domain of High-Performance Distributed Computing and Grid Computing, but also has potential for Mobile Computing [13]. It allows programmers to build powerful distributed applications using a very simple interface, one of the challenges for general smartphone middleware described by [15]. Other challenges are for smartphone middleware to be resource and energy aware. As such, it should use lightweight communication protocols, be aware of local processing pitfalls, and deal with periods of no connectivity and no sensor access. Furthermore it should not rely on adhoc networking and significantly simplify the software development process.

While the Android operating system itself offers continuous access to the sensors and comes with a good development environment, the Ibis middleware on smartphones fulfills a number of the middleware challenges through its two orthogonal components: the Ibis Distributed Deployment System and the Ibis High-Performance Programming System (see Figure 1).

A. Ibis Distributed Deployment System

The Ibis Distributed Deployment System offers the means to deploy a remote application, an essential feature needed to turn available resources into surrogates. Its main component is the Java Grid Application Toolkit (JavaGAT) [16], a toolkit that offers an API for remote File Management, remote Job Submission, Monitoring and Steering. Due to its flexible design it can bind to any middleware using an *adaptor* that maps the JavaGAT API calls to the calls for a particular middleware. JavaGAT contains adaptors for grid middleware among which Globus, gLite and SGE, and common middleware such as SSH and SFTP, while adaptors for cloud platforms are in progress (Amazon EC2). We have ported the JavaGAT to Android together with two adaptors, one to access the local resources and one to access resources using SSH. Using the JavaGAT version on Android we are

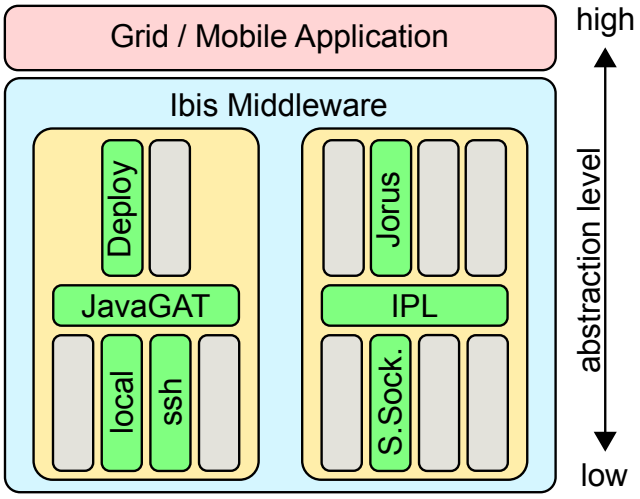

Figure 1. Overview of the Ibis middleware. The Ibis middleware consists of several subprojects, each implementing a part of grid middleware requirements. The left part of the Ibis middleware is the Ibis Distributed Deployment System, with JavaGAT as main component. The right part of the Ibis middleware is the Ibis High-Performance Programming System. Its main component is the Ibis Portability Layer (IPL). The boxes with texts are subprojects used for Cyber Foraging.

able to start any remote application on any machine that can be reached using SSH.

On top of the JavaGAT the Ibis Distributed Deployment System offers a deployment library, called IbisDeploy, tailored to start distributed applications built using the Ibis High-Performance Programming System. On top of the IbisDeploy library there is a graphical user interface (GUI) which can be used to deploy Ibis applications. Both the library and the GUI simplify the complex task of deploying a distributed application onto compute resources. The deployment of a distributed Ibis application consists of the following subtasks:

- Copy the application, the libraries and the input files to the compute resources
- Start an Ibis Server (registry) process
- Form an overlay network
- Construct middleware-specific job descriptions
- Submit the jobs to the compute resources
- Keep track of the job statuses
- When the jobs are done, retrieve the output files
- Clean up the remote filesystems

The IbisDeploy library and application use the concept of workspaces, in which applications, compute resources and job descriptions can be defined. Workspaces can be stored in files. A job description describes which application should be started on which compute resource. An application description describes the application itself, for instance its main class, the virtual machine options it needs and the arguments it gets. Finally, the compute resource description contains the details about how it should be accessed.

We have ported both the library and the graphical user interface tool to Android. IbisDeploy on Android offers a easy-to-use library together with an application to deploy remote Ibis applications. IbisDeploy is based on the philosophy that distributed application developers develop their application on a local machine and, when finished writing, deploy the application from the local machine. Therefore, IbisDeploy does not require any software to be available on the remote machines other than its default middleware where JavaGAT binds to and a Java Virtual Machine, used to run the application.

B. Ibis High-Performance Programming System

Having described how Ibis-based applications are deployed onto surrogates we now turn our attention to how Ibis applications are programmed. The Ibis High-Performance Programming System offers a programming environment to build distributed applications. The main component of this system is the Ibis Portability Layer (IPL), a communication library, that offers lightweight but powerful and efficient communication primitives. The IPL supports unidirectional communication streams, that can be connected between multiple endpoints (ports). The IPL ports support one-to-one, one-to-many and many-to-many connections. For each communication port, the programmer can specify the requirements for that port, thereby allowing the IPL to choose the most efficient communication implementation that satisfies the requirements. The IPL can do very efficient object serialization [17] and it also offers the means to implement fault-tolerance and malleability (the possibility to add and remove compute resources).

The IPL can communicate over normal TCP streams based on sockets, but there is also an implementation that communicates over SmartSockets streams. The SmartSockets library [18] is also part of the Ibis High-Performance Programming System and provides connections in difficult situations where normal TCP connections cannot be established. It can make connections through firewalls, it can effectively deal with Network Address Translation (NAT) issues and also solves the problem of connecting with machines with multiple network addresses. Since firewalls and NAT-boxes are common, we use the IPL implementation over SmartSockets for cyber foraging.

On top of the IPL several programming models are implemented like Remote Method Invocation, Group Method Invocation, Satin (a divide and conquer programming model) and Jorus (a Java implementation of Parallel-Horus [19]), a high performance multimedia programming model. We use the Jorus programming model to build the object recognition application. Through the IPL and the programming models, the Ibis High-Performance Programming System supports asynchronous as well as synchronous programming.

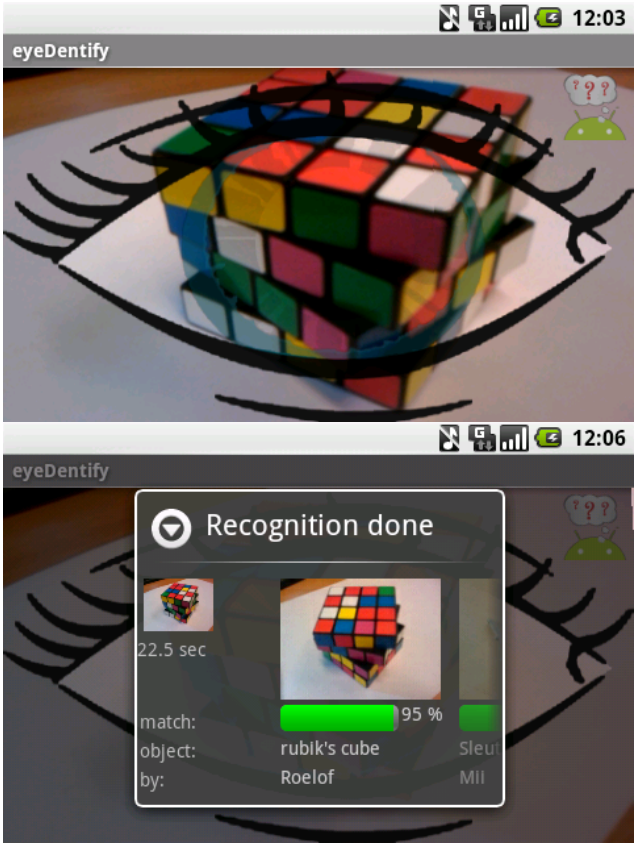

Figure 2. Two screenshots of the eyeDentify application. The upper one shows the eye looking at an object in recognition mode. The lower screenshot shows the result after the user has triggered the object recognition. The object is recognized correctly.

III. EYEDENTIFY

The system described above can be used for any distributed application that contains heavy weight computing. To illustrate this, we have implemented eyeDentify, a smartphone application that performs color based object recognition on images taken with the built-in camera sensor (see Figure 2). It is an interactive application in which the user can teach the application names of objects and subsequently let the application identify these objects. This application is a typical example from the general domain of Multimedia Content Analysis (MMCA) that aims to extract new knowledge from multimedia archives and data streams.

In *learning mode* the user takes a picture of an object and enters its name. The application stores the learning result into its internal database.

In *recognition mode*, the user takes a picture of an object, possibly under different viewpoint and lighting conditions, and eyeDentify will present the best match from the local database of learned objects.

Algorithms developed for object recognition try to mimic to some extent the way humans recognize objects. An

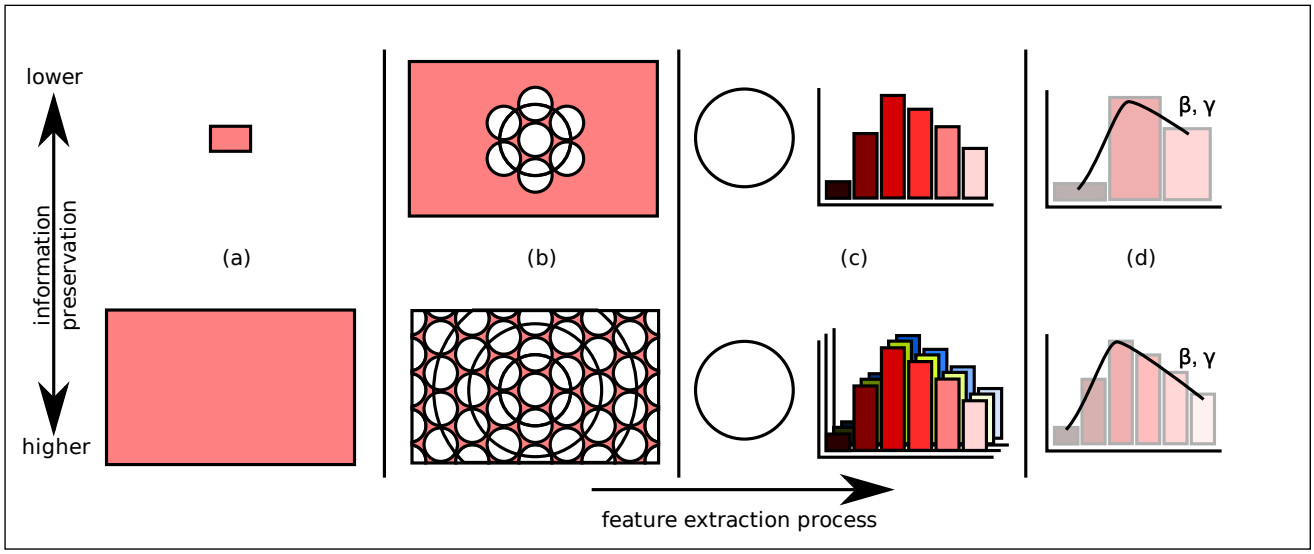

Figure 3. Schematic overview of feature extraction process of the image recognition algorithm used by eyeDentify. The process starts with a source image (a), with an overlay of rings with receptive fields (b). The bigger the source image, the more information of the original source is preserved. More receptive fields also increase the information preservation. Then, for each receptive field various color models (c) can be used to compute color histograms. Increase in the number of color models, increases the information about the image. The number of bins (d) in the histograms of the color models can be varied. The shape of the histograms can be approached by a weibull fit with two parameters, beta and gamma.

important difference between algorithms and humans is that algorithms operate on image input only, whereas humans use additional contextual information to identify an object. Therefore algorithms focus on getting as much useful information as possible out of the source image through a series of steps where the raw image information is converted into useful information, called *features*.

EyeDentify uses an object recognition algorithm taken from the Jorus high performance computing multimedia library. This compute intensive algorithm can run sequentially on a single compute resource as well as distributed on multiple compute devices, making it a good candidate to evaluate the use of cyber foraging, with multiple surrogates working together.

We implemented two versions of eyeDentify, one being a computation producer version that performs the computation on the smartphone itself, and another one being a computation consumer that deploys the entire computation server including the needed libraries to the surrogates using Ibis cyber foraging. Because Ibis offers a programming model for distributed multimedia computing and libraries for starting remote applications it was easy to implement the cyber foraging version of eyeDentify.

We evaluated the impact of remote execution on two important quality indicators of an application on the smartphone, the responsiveness and the energy usage, while we varied the input image size and the accuracy settings of the algorithm. The dimensions of the input image together with the accuracy settings determine the amount of compute power, memory and energy needed for the computation. The larger the dimensions of the input image are, the more resources the image recognition algorithm needs. The accuracy settings contain values for the parameters for each step in the feature extraction process. We will first describe the feature extraction process and then discuss the impact of its parameters on the resource usage.

A. The object recognition algorithm

For a given image the object recognition algorithm first designates a number of circular areas, called *receptive fields*, around the image center. Then, for each receptive field a number of color histograms is built, each for a different color model. Each color model has been selected for its invariance to specific imaging conditions, such as shadows, shading, and differences in the color of the light source. Subsequently, the shape of each color histogram is approached by a *weibull fit*. The resulting parameters for all histograms are combined in a single *feature vector*, thus forming a condensed description of the image scene. As each feature vector represents a point in a high-dimensional space, object recognition is achieved by finding the closest neighboring point in this space.

The accuracy parameters that can be changed without breaking the algorithm are the dimensions of the input image (Figure 3a), the number of receptive fields (Figure 3b), the number of color models (Figure 3c) and the number of the bins in the color histograms (Figure 3d). However, changes in these parameters will have an effect on the accuracy and performance of the algorithm. A higher number of receptive fields, color models or bins in the color histograms, will

Table I
ACCURACY PROFILES

	receptive fields	color models	bins
low ($\star$)	7	1	100
medium ($\star\star$)	19	2	500
high ($\star\star\star$)	37	3	1000

result in more accurate image recognition but also in higher resource usage. We made three accuracy profiles (Table I), one with high accuracy and strong performance requirements, another one with low accuracy and low performance requirements and one in between to evaluate the impact of the accuracy parameters on the resource usage.

IV. MEASUREMENTS

Ideally a smartphone application will be *responsive* and *energy efficient*. We show that by using cyber foraging, both the responsiveness and the energy usage of eyeDentify improve. We have performed measurements with two versions of the eyeDentify application. One that does all the computation on the phone itself (*standalone version*) and one that uses cyber foraging to offload the computation to surrogates (*foraging version*). In this section we will first briefly outline the environment in which we performed our measurements by describing the hardware resources we used and then proceed with a discription and a discussion of the measurements.

A. Hardware Environment

We have run our measurements on the T-Mobile G-1 smartphone. The G-1 has a 528 MHz Qualcomm ARM based processer and 192 MB RAM. It can communicate using GSM, 3G, Bluetooth and Wifi (802.11b/g), of which we use the Wifi for our experiments. Furthermore, it contains a number of sensors: a 3.2 MegaPixel camera, GPS, a compass and an accelerometer. The G-1 runs the Android Operating System.

We used 8 nodes of the VU cluster of the DAS-3[1] as surrogates for the cyber foraging version of eyeDentify. Each of the cluster nodes has a dual-CPU / dual-core 2.4 GHz AMD Opteron processor and 4 GB of RAM. The nodes run the Scientific Linux Operating System.

B. Responsiveness

We define *responsiveness* as the time used by an application to respond upon a user-triggered request. Whether a user is satisfied with the responsiveness offered by an application highly depends on personal preferences, but in general the lower the response time is, the higher the user's satisfaction.

In eyeDentify the user can trigger three actions which cause a considerable amount of computation and therefore are may not meet the responsiveness requirements of the user. These actions are: the initialization of the application and the learn and recognize actions. We will briefly describe the initialization measurements and then focus on the measurements of the learn and recognize actions.

[1]Distributed ASCI Supercomputer, http://www.cs.vu.nl /das3

Table II
INITIALIZATION (IN SEC.) VS. ACCURACY

version ▷	standalone			foraging
accuracy ▷	$\star$	$\star\star$	$\star\star\star$	$\star\star\star$
image size				
32 x 24	0.55	0.95	1.92	0.12
64 x 48	4.54	9.28	17.33	0.12
128 x 96	28.52	79.76	-	0.13
256 x 192	-	-	-	0.67
512 x 384	-	-	-	4.25
1024 x 768	-	-	-	42.1
2048 x 1536	-	-	-	451

During the initialization eyeDentify initializes the object recognition algorithm. For each receptive field a number of *Gaussian filters* is computed. The size of a receptive field depends on the dimensions of the images that are going to be used. The more and the bigger the receptive fields the longer the initialization process takes. Table II shows the algorithmic initialization times for different accuracy settings for both versions. Although the foraging version benefits from parallel algorithmic initialization on the surrogates, it needs additional foraging initialization time to start up the processes. The foraging initialization time in our experiments was about 34.8 seconds and in general will depend on factors like file transfer and queueing time on the surrogates.

Once the algorithm is initialized, the learn and recognize actions will be repeatedly invoked. They both trigger the same heavy weight computing task, the computation of a feature vector out of a captured image. The database lookup time for the recognize action is neglectable, and therefore is not included in the measurements. We measured the responsiveness of eyeDentify while varying the accuracy settings (i.e. the accuracy profile together with the image size) and the results are shown in Table III. The last column of the table shows the total response time of the foraging version and the portion of it caused by computation.

Due to the limited memory (16 MB) available to an application on the G-1 running Android, the standalone version can perform object recognition on images with sizes up to 128 x 96 pixels, but only with medium accuracy settings. For images larger than 128 x 96 pixels, the initialization process requires more memory. The memory size is also a limiting factor for the 128 x 96 image with the highest accuracy profile, although not during the initialization of the algorithm, but during the execution of the algorithm itself.

While the standalone version can only operate on fairly small images, the foraging version can use the maximum resolution of the camera (3.2 MegaPixels) with the highest accuracy profile. Even when operating on an image with 1024 times more pixels (64 x 48 vs. 2048 x 1536), the

Table III
RESPONSE TIME (IN SEC.) VS. ACCURACY

version ▷	standalone			foraging
accuracy ▷	⋆	⋆⋆	⋆⋆⋆	⋆⋆⋆
image size				total comp.
32 x 24	0.66	5.99	25.61	0.46 (0.12)
64 x 48	1.38	8.57	32.21	0.54 (0.12)
128 x 96	4.09	17.49	-	0.55 (0.13)
256 x 192	-	-	-	0.60 (0.19)
512 x 384	-	-	-	0.81 (0.41)
1024 x 768	-	-	-	2.06 (1.29)
2048 x 1536	-	-	-	6.51 (4.87)

Table IV
EXECUTIONS VS. ACCURACY

version ▷	standalone			foraging
accuracy ▷	⋆	⋆⋆	⋆⋆⋆	⋆⋆⋆
image size				
32 x 24	15,764	1,652	405	15,283
64 x 48	7,928	1,351	333	15,047
128 x 96	2,930	590	-	14,375
256 x 192	-	-	-	12,119
512 x 384	-	-	-	8,481
1024 x 768	-	-	-	3,746
2048 x 1536	-	-	-	1,712

cyber foraging version is still about 5 times faster than the standalone version. For small images, a major part of the response time of the cyber foraging version gets spent on communication, however, it is still about 56 times faster with the smallest image size and about 60 times faster with the 64 x 48 images. The computation itself is about 250 times faster on the surrogates than on the smartphone, which means that if the phone would have enough memory, the computation for a 2048 x 1536 image would take about 20 minutes.

We consider response times of up to 20 seconds still acceptable for the learn and recognize actions, which means that we consider running the standalone version with the high accuracy profile as not acceptable. The response times of the cyber foraging version, however, are all well below the 20 seconds. We conclude that cyber foraging proves to be a good technique that can drastically improve the responsiveness of smartphone applications that are compute intensive and that need only a limited amount of communication to offload the computation.

C. Energy Usage

Although offloading computation using cyber foraging increases the responsiveness of the application, it also introduces communication, which is known to be much more energy consuming than computation [20]. To evaluate the impact of cyber foraging on the smartphone's energy usage we performed experiments in which we measured the energy consumption on the smartphone of both the standalone and the foraging version of eyeDentify.

In each experiment we fully charged the phone and then let it repeatedly execute a feature vector computation until the battery was only 20 percent charged. We counted the number of executions for both the standalone and the foraging version, while varying the accuracy profile and the image size for each experiment.

The results of the experiments are shown in Figure IV and show that increasing the computation complexity for the standalone version by either increasing the image size or the accuracy settings results in a lower number of executions. For the foraging version, a larger image size and thus an increase in communication, results also in less executions.

Even for the smallest images (32 x 24 pixels) with low accuracy recognition where the computation for the standalone version is relatively small, the foraging version performs about the same number of executions, but then with high quality accuracy settings.

When both versions use the high quality accuracy settings the foraging version can do about 40 times more executions than the standalone version. The foraging version operating on the full 3.2 MegaPixel image can still do about 5 times more executions than the standalone version operating on 64 x 48 pixel images.

Battery lifetime is a very important aspect of todays smartphones and smartphone applications therefore should focus on consuming as little as possible energy. The additional costs for communicating in the foraging version are less than the costs that are saved by not doing heavy weight computation on the phone itself, making cyber foraging an attractive alternative to the local computing with respect to energy usage.

V. Related Work

In [11] cyber foraging is described as an effective way to deal with light weight mobile devices that need heavy weight computation. Important aspects of cyber foraging are the discovery of surrogates, the trust relation between the client and the surrogates, load balancing on the surrogates, scalability and seamlessness. The Ibis middleware provides several of these aspects through its various components.

The IbisDeploy library, built on top of the JavaGAT, specifies a format of resources that could be discovered. Currently, the discovery of resources is done manually by acquiring a file with the resource description or by a user configurated resource description. IbisDeploy deals with the trust relation, load balancing and scalability implicitly, because it makes use of existing middleware's trust models, load balancing and scalability through the JavaGAT adaptors.

The PeerHood environment is a middleware for cyber foraging. It has been used for a cyber foraging application that performs barcode analysis [21]. Using the PeerHood environment, mobile clients dynamically discover distributed resources offering a service to analyze barcodes. Other

than Ibis, PeerHood needs prestarted running servers on all compute resources, introducing a dependency on the maintainers of the servers for the user of the application. Furthermore, PeerHood is not available on Android.

The Split Smart Messages [22] smartphone middleware allows arbitrary code to be transfered from a smartphone to another resource, but requires the middleware to be already running on both the smartphone as well as the compute resource and therefore is not as powerful as Ibis, which can deploy itself to any accessable resource that has a Java version installed.

The design and implementation of a non Java cyber foraging system based on virtualization on the surrogates is describe by [23]. Using this cyber foraging system they made a smartphone application for speech recognition. Performance evaluation for this application shows increase in responsiveness of 200 times and a reduction of energy usage of 60 times compared to a local execution. The Ibis middleware offers a similar way to deploy distributed applications, but in addition it offers several easy to use programming models. Furthermore, the SmartSockets component solves connectivity issues, in case the surrogate is behind a firewall and can for example only be reached through SSH.

The CloneCloud architecture [24] replicates the phone state to the cloud and uses the compute power of the cloud to perform augmented execution on a phone emulator. CloneCloud supports five types of augmented execution. Primary functionality outsourcing is used to offload heavy weight computations to the clone. Background augmentation is used for heavy weight computations that do not necessarily need to respond quickly. Execution done on the phone itself concurrent with a slightly different execution in the cloud (for instance, a taint check version of the execution) is called mainline augmentation. Fourth, there is hardware augmentation, which compensates for the hardware shortcomings of the phone, such as the memory size. And finally the authors describe augmentation through multiplicity, where execution can be enhanced using parallel computing in the cloud. In contrast with CloneCloud, the Ibis middleware does not require synchronization with the cloud, but still can offer augmentend execution through primary functionality outsourcing, hardware augmentation and augmentation through multiplicity as shown in the eyeDentify application, presented in this paper.

VI. Conclusions

A well-known technique to make heavy weight applications run on smartphones is making the applications distributed. Current distributed applications follow the client/server model, in which the smartphone user only controls the client part of the distributed application and therefore is prone to future incompatibility with the server.

We show that an alternative model, cyber foraging in which the user gets control over both the client and the server, can be used effectively to overcome the dependency of the user on the service provider. In this model, the server will be deployed directly from the phone onto compute resources.

We implemented cyber foraging using the existing Ibis middleware. On top of this system we have built a smartphone application called eyeDentify that can perform object recognition using pictures taken with the phone's camera. We implemented two versions of eyeDentify, a standalone version that runs on the phone itself and a distributed version that uses Ibis cyber foraging to offload computation. The use of Ibis cyber foraging greatly improves the responsiveness with speedups around 60 times while being about 40 times more energy efficient. Furthermore, Ibis cyber foraging enables object recognition with high accuracy on full resolution images on a smartphone, which is not possible at all with a standalone version due to hardware limitations. Multimedia cyber foraging with Ibis provides easy and powerful means to create computation and memory intensive multimedia applications for smartphones

Acknowledgements

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ). This work has been supported by the Netherlands Organization for Scientific Research (NWO) grant 612.060.214 (Ibis: a Java-based grid programming environment).

References

[1] "Android Market," http://www.android.com/market/.

[2] "Windows Marketplace for Mobile," http://www.microsoft.com/presspass/press/2009/mar09/03-11WMMDevelopersPR.mspx.

[3] "BlackBerry App World," http://na.blackberry.com/eng/services/appworld/.

[4] "PlayNow Arena," http://www.playnow-arena.com/.

[5] "Ovi Store," http://store.ovi.com.

[6] "iPhone App Store," http://www.apple.com/iphone/ appstore.

[7] "Big in Japan," http://www.biggu.com/applications.

[8] "Mobilizy website," http://www.mobilizy.com/wikitude.php.

[9] R. Ballagas *et al.*, "The smart phone: A ubiquitous input device," *IEEE Pervasive Computing*, vol. 5, no. 1, p. 70, 2006.

[10] "Shazam website," http://www.shazam.com.

[11] M. Satyanarayanan, "Pervasive computing: Vision and challenges," *IEEE Personal Communications*, vol. 8, pp. 10–17, 2001.

[12] "Amazon Elastic Compute Cloud Website," http://aws.amazon.com/ec2.

[13] N. Palmer *et al.*, "Ibis for mobility: solving challenges of mobile computing using grid techniques," in *HotMobile '09: Proceedings of the 10th workshop on Mobile Computing Systems and Applications*. ACM, 2009, pp. 1–6.

[14] R. V. van Nieuwpoort *et al.*, "Ibis: a flexible and efficient java-based grid programming environment," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 7-8, pp. 1079–1107, 2005.

[15] O. Riva and J. Kangasharju, "Challenges and lessons in developing middleware on smart phones," *IEEE Computer*, vol. 41, no. 10, pp. 23–31, 2008.

[16] R. V. van Nieuwpoort *et al.*, "User-friendly and reliable grid computing based on imperfect middleware," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 2007, pp. 1–11.

[17] J. Maassen *et al.*, "Efficient java rmi for parallel programming," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 6, pp. 747–775, 2001.

[18] J. Maassen and H. E. Bal, "Smartsockets: solving the connectivity problems in grid computing," in *HPDC '07: Proc. of the 16th int. symposium on High performance distributed computing*. ACM, 2007, pp. 1–10.

[19] F. J. Seinstra *et al.*, "High-performance distributed video content analysis with parallel-horus," *IEEE MultiMedia*, vol. 14, no. 4, pp. 64–75, 2007.

[20] D. Estrin *et al.*, "Connecting the physical world with pervasive networks," *IEEE Pervasive Computing*, vol. 1, no. 1, pp. 59–69, 2002.

[21] T. Kallonen and J. Porras, "Use of distributed resources in mobile environment," *International Conference on Software in Telecommunications and Computer Networks*, vol. 0, pp. 281–285, 2006.

[22] N. Ravi *et al.*, "Split smart messages: Middleware for pervasive computing on smart phones," *Rutgers University Technical Report DCS-TR-565*, 2005.

[23] S. Goyal and J. Carter, "A lightweight secure cyber foraging infrastructure for resource-constrained devices," in *WMCSA '04: Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems and Applications*. IEEE Computer Society, 2004, pp. 186–195.

[24] B.-G. Chun and P. Maniatis, "Augmented smart phone applications through clone cloud execution," in *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS XII)*, 2009.