Remote Code Execution on Ubiquitous Mobile Applications

João Nuno Silva and Paulo Ferreira

INESC-ID / IST
Rua Alves Redol, 9, 1000 Lisboa, Portugal
{joao.n.silva,paulo.ferreira}@inesc-id.pt

Abstract. Today, most mobile devices (e.g. PDAs) are in some way associated to a fixed personal computer or server. In general this relation is only taken into account for synchronization purposes.
This is rather restrictive as, while away from these fixed computers, such mobile devices may require resources that are not available (e.g. network bandwidth, processing power or storage space). This lack of resources prevents the user from doing what he wants when he wants.
We propose a system in which, by enabling automatic remote code execution on a remote computer, these limitations are subdued. At run time it is decided whether some application code should run locally or on a remote computer. This is achieved using runtime meta-programming and reflection: we transform a centralized Python application so that some part of its code is run on another computer, where the needed resource is known to be available. This is accomplished without any manual code change. The performance results obtained so far, i.e. with no optimizations, are very encouraging.

1 Introduction

With the advent of mobile communication technologies, the resources available to mobile devices are no longer constant. In a certain moment the connectivity and bandwidth can be low and in the next instant these resources can be widely available. This variation on the environment and how the application must react poses a challenge to the development of environment aware applications. Adding to the mobility of these devices we now have a large number of personal computers with highly available resources (disk, bandwidth, screen, ...); when the user is roaming with its mobile device such resources are available but not used.

One kind of application that behaves independently of the environment resources is the download manager attached to browsers. In a portable device, while browsing a site, if the user wants to download a file, this action is always performed in the device. The browsing application, isn't aware of nearby computers that may have a larger display or better connectivity. If these computers could be used in a transparent and automatic way, the user would benefit from it, in terms of speed and ease of usage. If the network bandwidth is limited or the

P. Markopoulos et al. (Eds.): EUSAI 2004, LNCS 3295, pp. 172–183, 2004.
© Springer-Verlag Berlin Heidelberg 2004

storage space is insufficient, a better approach would be to execute the download to a remote computer.

For example, an application that displays or stores a file specified by its URL can benefit from the resources available in the remote computers. If the storage space in the PDA is not enough to save the file one should transparently save it in another computer. The same applies to the case where network bandwidth is reduced. If we are watching files on a PDA, the use of a large display is preferable. If a nearby computer has a larger display than the PDA we are running our application at, then the file should be presented in that remote computer.

One challenge we see in such environments is not only to adapt the mobile computer applications to its environment but also to use the wasted resources available in the surroundings. Adding to this challenge there are other problems: reconfiguration of the running objects when the environment changes, the automatic selection of the remote computer and issues concerning security and resource usage.

In order to accomplish this, the first problem to address is how to split an application in order to run part of it in a different computer. The first solution that comes to mind, and the hardest, is to explicitly program each application to make it mobile with the device. Such solution makes the application mobility impractical due to the different scenarios one must address.

1.1 Shortcomings of Current Solutions

The implementation of an environment aware distributed application can be accomplished using ordinary remote procedure calls or RMI libraries. The code to handle the environment observation and the results of these observations must be implemented by the programmer. The selection of the objects that are to be run remotely must be known at development time and this is hard coded to these objects. The programmer must also know in advance how the object should be executed in the remote computer.

When using some sort of agent API, handling the localization of the remotely executed object becomes easier. The agent libraries provide a way to let the programmer decide at a latter time where the code should run.

These solutions have the drawback that the original code must be changed and that the various possible scenarios (resources to take into account, available remote machines, different application configurations, ...) must be addressed while coding the transformed application.

By transforming the binary executable so that the resulting program can handle the environment changes, there is no longer the need to develop a new source version different from the original. This transformation tool must read the source code (or a binary version) and, in accordance to some configuration file, change the way the original code runs. These tools must insert the code that handle the environment awareness and transform the way the objects are created (remote versus local object creation)

By doing this transformation at compile time, we get two different binary versions, but the original source code remains constant. This approach is better

than the explicit programming mentioned earlier but still has some drawbacks. As the new binary versions have features and use resources that may not be available in all platforms there is still the need to match the correct binary version to the platform and to the execution needs.

1.2 Proposed Solution

When the code transformation is done at run-time, the problem that arises from having several binary versions disappears. Each platform has a different transformer that runs when executing the application. The same binary version has different behavior depending on the existence of a code transformer.

The system responsible for the code transformation, does not read the binary file; instead, it intercepts the loading of the code, and changes the class representation that is then stored in main memory.

With this approach the same binary can be ran using several transformation tools, generating different final programs, without the need to administer different versions.

By using metaclass programming we manage to intercept the class loading in an easy and straightforward way. We implemented a metaclass, defined it as the constructor of all loaded classes and made it responsible for the adaptation of the code being loaded.

In the next section we present some technologies systems that address similar issues as our work (remote code execution, mobility and reflection). In Sects. 3 and 4 we describe the architecture and implementation (respectively) of our system. Finally we show performance and functional evaluation as well as the conclusions and future work.

2 Related Work

A few years ago mobile computing was synonym of agent programming. Agent systems allowed the programmer to easily develop applications that would roam around several remote computers to perform a certain task.

Today, with the development of wireless technologies and portable devices, mobile computing has a new meaning. Now the applications are mobile because the devices they are running on are mobile. To take advantage of the full potential of the resources available in the surrounding environment these applications must adapt the way they execute.

The first approach to the adaptation of these mobile applications was to adapt the data transmitted to and from remote computers. The solutions proposed range from the development of specific proxies to the use of distributed middleware that handles the adaptation of the data transmitted. The proxy solution was first applied to the web contents [1][2] and allowed the transformation of the contents so that its download time is reduced. In the other edge we find systems that allow the development of mobile application that interact with a data source. In the work done by T. Kunz[3] the communication is done by a

series of objects: some local to the mobile device and others in the data source that adapt the information transmitted according to the resources available.

These solutions have the drawback that only the data is adapted to the device's environment. Another approach to the adaptation of the application execution is the development of middleware platforms that are environment aware. These systems range from the generic system discovery such as Satin[4] or ICrafter[5] to the more specific work done by Nakajima[6] with a middleware for graphical and multimedia applications. These solutions are valid but solve the problem of constrained resources by invoking services on remote computer but not executing the applications original code on a more suited host.

Some systems use reflection[7][4][8] to accomplish the transparent adaptation of the applications but its scope is the same as the systems described previously. Only logic mobility is performed, a task is executed in a remote host, but no application code moved to the remote computer, the service performed should be already present there.

Also related to the work we try to accomplish there are some migratory applications systems . The work done by Krishna Bharat[9] allowed the development of graphical applications that could roam between several computers, but in a monolithic way. Harter et al.[10] by using VNC server, the interface of an application could be moved from a display to another, but the code remained running in the same server.

The use of the Remote Evaluation Paradigm[11] overcomes the deficiencies of the previous systems by allowing the development of reconfigurable applications[12] with code mobility. There is no need to program where and when the objects should move to. The programmer must only develop the code taking into account that it will be mobile and state which requirements the remote host should satisfy. For instance, in Fargo[13] the programmer must develop a special kind of component: the complet. While developing these special components, the developer must program how the objects will roam among the computers: environment requirements, object aggregation, ... After the development of these components the application must be compiled so that the mobile code is inserted in the application.

Our work also relates to projects such as Javaparty[14] or Pangaea[15], in the sense that the distribution of the objects and the decision on where they should run is hidden from the programmer. In JavaParty the system hides the remote creation of objects and its mobility, but the programmer must tag these with a special keyword. A separate compiler is needed. In Pangaea, a special compiler analyzes the source code and transforms it so that some objects are created in remote hosts.

3 Architecture

The system must accomplish three different tasks in order to automatically transform a centralized application: load the application requirements, transform the

mobile classes according to the requirements and allow the remote execution and mobility of the code.

In our system, these tasks are executed by three different components (shaded in Fig. 1): the **Application Requirement Loader**, the **Code Transformer** and the **Remote Execution** engine.

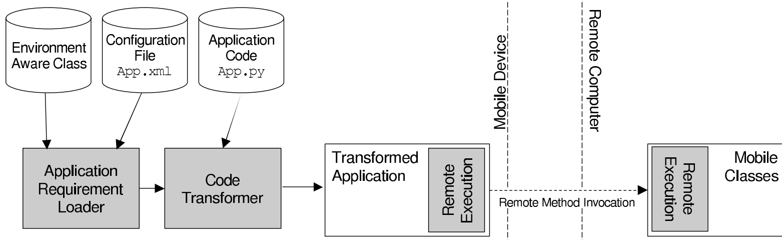

Fig. 1. System Architecture

In order to make a class mobile, during the execution of the application a configuration file must be present. This file states what classes are mobile and what their requirements are. The **Application Requirement Loader** reads the configuration file and stores the requirement rules associated to each class. From the information stored in the file, this module builds the rules that state whether a certain class instance should run locally or remotely. This module is also responsible for loading the code of the probes that observe the surrounding environment.

The **Code Transformer** module is responsible for transforming the classes that may run locally or remotely (those referred in the configuration file). If the **Application Requirement Loader** module knows about the class being loaded, the **Code Transformer** module changes the application code so that it is possible to create remote instances of that class. This module, besides changing the class code, also attaches to it the rule that should be evaluated when creating objects.

The **Remote Execution** module, runs both on the mobile device and on the remote computer. This module is responsible for the creation of the remote objects and the communication between the mobile device and the remote computer. This module must also upload the code into the remote computer if it was not previously installed in the remote computer. This module is contacted during the execution of the application whenever there is an object that should be created remotely and when these remote objects should execute any method.

The **Environment Aware Classes** must be supplied to our system at run time. Theseclasses must observe the surrounding environment (network bandwidth, display size, ...) and inform the application if the requirements are met. These classes must comply with a certain interface that will be described later (Sect. 4.2.)

3.1 Execution

The steps in the loading and transformation of an application are the ones shown on Fig. 2.

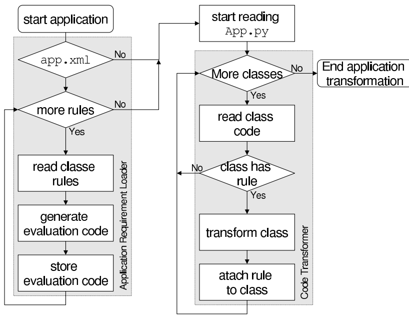

Fig. 2. Application transforming

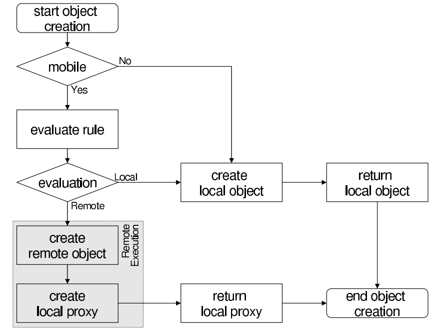

Fig. 3. Object creation

Even before the loading of the application code, the **Application Requirement Loader** is created. If a configuration file (`app.xml`) exists it is read. From the reading of the configuration file, the rules are created, stored and associated to the corresponding class. In order to later evaluate the environment, the **Environment Aware Classes** code is also loaded and it is instantiated. These first steps are performed by the **Application Requirement Loader** module as described in Fig. 2.

After the reading of the rules, the real transformation of the application is performed by the **Code Transformer**. After a class code is read, the **Code Transformer** module checks the existence of a rule associated to that class. If the class objects are to be mobile, the class code is transformed and the rules are also attached to it. This code and rules will be responsible for the evaluation of the environment and the creation of the object in a remote computer.

The code that is inserted into the transformed classes is executed when creating new objects as shown in Fig. 3.

If a certain doesn't have an evaluation rule associated (is not mobile), the object creation code was unchanged so its instances are always local.

If the class was transformed the associated rule is evaluated. This evaluation states if the local device has enough resources to execute the objects or if the object must be created on a remote computer.

If it was decided that the object can run locally, a normal local object is created. Otherwise, an instance of the class is created on a remote computer and on the local device a proxy to this new object is created. This proxy will replace a local object and forward the local method calls to the remote object.

The remote objects and its proxies are created in the context of the **Remote Execution** module.

4 Implementation

In the development of our system we used the Python[16] language. We took advantage of its portability and dynamic nature and the presence of introspection and reflection mechanisms on all tested platforms. There are fully functional Python interpreters to most desktop and server platforms and to most portable devices (Windows CE, QNX and Linux). In order to transform in run time the application we used the python reflection built-in mechanisms and to make communication between computers possible we used the Pyro[17] package.

4.1 Application Requirement Loader

The **Application Requirement Loader** module reads a configuration file, and from the information present in that file, assigns each class a rules that states whether its instances should run locally or remotely. The file is written in XML (Fig. 5), whose DTD is described in Fig. 4.

```
<!ELEMENT program (name, class*)>
<!ELEMENT class   (name,host,expr)>
<!ELEMENT expr (decis| and| or| not)>
<!ELEMENT decis   (name, config)>
<!ELEMENT and     (expr+)>
<!ELEMENT or      (expr+)>
<!ELEMENT not     (expr)>
<!ELEMENT host    (name)>
<!ELEMENT config  (#PCDATA)>
<!ELEMENT name    (#PCDATA)>
```

Fig. 4. Configuration file DTD

```
<program> <name> app1 </name>
 <class> <name> clss0 </name>
  <host> <name> host1 </name>
  </host>
  <expr> <decis>
    <name> BandWidth </name>
    <config> <more>1000</more>
    </config>
  </decis> </expr>
 </class></program>
```

Fig. 5. Configuration file

The classes that have some sort of environment requirement will have a rule in this file. These rules are written with the usual logical operators (OR, AND, NOT). After reading these rules a tree like structure will be generated and stored, so that they can be latter evaluated.

This module stores the rules in a hash-table, so that, during the loading of the classes (executed by the Code Transformation Module) these rules can be attached to the classes referred in the configuration file.

4.2 Environment Aware Classes

When evaluating the rules previously stored, there is always the need to evaluate the surrounding environment. This is stated by the `decision` element that appears in the XML configuration file. These nodes are the leafs of the rules.

These classes have only requirements: the constructor must receive a piece o XML that complies with its own definition and there must exist a method named

`decide` that returns `true` or `false` according to the sensors measurement and the information on the XML code.

The code that interact with the sensor that observes the surrounding environment must be supplied in the `decide` method and is executed when evaluating the rules. When the instances of these classes are created and attached to a rule, it is passed a XML snippet that informs the object about the requirements the application has. This XML code must comply with a certain DTD and defines the requirements the application has.

When evaluating the environment this object knows exactly what the application needs to be executed returning `true` or `false` whether a certain requirement (stated in the XML code) is met.

4.3 Code Transformation Module

This Code Transformation Module intercepts all class code loading and decides whether the class should run unmodified or not. This module uses the information generated by the **Decision Making Module** to know if a class must be transformed. The class loading interception is accomplished using a customized metaclass.

The `classCreator` metaclass (Fig. 6) is responsible for the interception of the program's classes loading.

```
1   class classCreator(type):
2     def buildClass(ClassName):
3       if (name in classCreator.remClss):
4         oldclass = type.buildClass(name+"old")
5         replaceClass = type.buildClass(name, remoteCodeCreator)
6         replaceClass.originalClass = oldclass
7         replaceClass.server = classCreator.remClss[name]
8         return replaceClass
9       else:
10        return type.builtClass(name)
```

Fig. 6. `classCreator` metaclass pseudo-code

When building a class, this code checks if the class being built was referred in the configuration file (Line 3) by looking at the `remClss` hash-table. This hash-table was previously built and populated while reading the configuration file (Sect. 4.1). If its name is not present in the hash-table, this class is built normally by the system default metaclass `type` (Line 10).

If the class being built was referred in the configuration file, this metaclass must replace it for a proxy class (Lines 4-8 in Fig. 6). The first action is to store the original class (Line 4), so that later it can be accessed to create local or remote objects. Next an instance of the `remoteCodeCreator` class is built (Line 5) and the original code and the evaluation rules are passed to it (Lines 6 and 7). This replacement class (`remoteCodeCreator`) is shown in Fig. 7.

```
1   class remoteCodeCreator (object):
2     def buildObject(this, *args):
3       decision = this.server['rule'].decide()
4       if decision == False:
5         obj = this.originalClass.createObject()
6       else:
7         hostname = cls.server['name']
8         URI=genURI(hostname)
9         proxy = getAttrProxyForURI(URI)
10        cdURI = proxy.createObject(this.originalName,args)
11        obj = getProxyForURI(cdURI)
12      return obj
```

Fig. 7. `remoteCodeCreator` replacement class

When creating an object that may run remotely, instead of running the original constructor, it is the code present in the `remoteCodeCreator` that runs. The rule associated to the class is evaluated (Line 3) and if the local device has enough resources to satisfy the rule, the result is negative and a local object is created (Line 5). In the opposite case a remote object must be created in the personal computer associated to this program. To create the new object in the remote computer, a connection to the **Remote Execution Module** present in the remote computer is made (Lines 8 and 9). Next, a remote instance of the original class is created (Line 10) along with its proxy (Line 11). Either a local object or a remote object proxy is returned in Line 12. This proxy is transparently created by the **Pyro System** and implements the same interface as the original class. From this point forward our original application interacts with a remote object by calling methods from the proxy without any change in the original code.

4.4 Remote Execution Module

The **Remote Execution** module runs part in the fixed computer where the remote code will execute and another part in the mobile device.

In the server side, this module is composed of a service that receives requests from the clients to create new objects, these new objects are then registered and made available to be remotely called.

The creation of remote objects is similar to other remote method execution systems, first a proxy to the server is obtained and the calls are forwarded to the located object. When requesting the creation of a remote object (Line 10 of Fig. 7), if its code is not present in the remote computer, the Pyro server automatically downloads the necessary code to create and execute the objects. After the code is loaded and the object is created, this new object is registered as a new server in order to receive the call made in the original application.

On the original application side, after the remote object creation, the URI returned is used to create a new Pyro proxy to the remote object. From this

point forward the call made supposedly to a local object is redirected by the Pyro proxy to the remote object stored and available in the server.

4.5 Execution Overview

In order to use this system and allow the automatic remote execution of the code, there isn't any need to change the original code, nor the Python code interpreter. Instead of calling the application the usual way (`python app.py`) one only needs to prepend our system file: `pyhton codetrans.py app.py`. One can also configure python so that our system is always loaded whenever a python program runs.

This `codetrans.py` file is responsible for bootstrapping our system by loading the configuration file associated with the python program and registering the `classCreator` metaclass (Fig. 6) as the default metaclass. After these initial steps the original python file is loaded, transformed as described in Sect. 4.3 and the application starts executing.

In order to create objects in remote computers, the daemon responsible for it must be running on those computers.

In this initial prototype the selected remote computer is hard coded in the XML configuration file. The **environment aware classes** must be coded so that we can evaluate the resources available: display size, disk capacity, network bandwith.

5 Evaluation

In order to evaluate the possible uses of our system, we developed two test applications. One application reads a URL from the keyboard and downloads that file to the hard disk. The second one receives also read a URL, downloads the correspondent text file and shows it in a graphical window.

By using our system we managed to execute part of the applications (user interface) in a Compaq IPAQ and execute the other part in a remote computer. In the first test the code responsible for the download of the file runs in the remote computer. In the second test we managed to run the download code and open a new graphical window on a remote computer. These distributed applications were executed without any change to the original code.

In order to evaluate the overhead incurred by our system, we developed a series of microbenchmarks that allows to measure the time spend in each stage of execution. Next is the description of the potential overheads and the applications used to measure them:

Bootstrap. In this test we measure the time to execute a simple application that only prints a message on the display. We managed to measure the bootstrap of our systems without loading any XML configuration file.

Rule loading. With this test we measure the time to load 100 rules present in configuration file. The rule provided to each class was shown in Fig. 5.

Class loading. The test program used to evaluate the time spent in the class loading, loads 100 classes that don't have any rule associated.

Class transforming. This test is similar to the previous one, but the loaded classes have a rule associated, allowing us to measure the time to transform a class.

Rule Evaluation. This final test loads 100 classes and creates one object of each class. To measure the rule evaluation time, the rules are evaluated but the objects are created locally.

These test applications were run on a Apple Ibook with an 800MHz processor and 640Mb of memory. We used a version 2.3 Python interpreter running on MacOsX. The results are presented in Table 1.

Table 1. Mac OS X execution times (sec)

	unaltered Python	adaptation system
bootstrap overhead	0.12	0.50
rule loading	0.12	0.96
class loading	0.22	0.60
class transforming	0.22	1.14
rule evaluation	0.22	1.18

From the values shown in Table 1 we conclude that our system overhead comes from the bootstrap (about 0.38s) and from the XML rule loading (about 0.46s for 100 rules). All the other tasks have a minimal impact on the execution time. The loading of 100 classes instead of taking 0.10s takes 0.18s, when the classes are transformed. The evaluation of the rules takes a minimal time to perform (about 0.04s).

6 Conclusions

We developed a system that allows us to experiment the possibility to create a reflective platform to dynamically adapt applications depending on the resources available. This platform, without any application source code change, modifies the way an application behaves. With the inclusion of configuration files the programmer can make the objects of a class remote. When creating these objects, depending on the resources required by the objects and available at the device, these objects are created locally or on a remote computer. These adaptations are made with minimal loss in performance.

We now have a system that allows ordinary applications to use the best resources available around a mobile computer. To use these resources the programmer neither has to change the source code of the application nor handle multiple binary versions.

This experiment with application adaptation is promising. Using this platform we can now work on the development of a transparent and automatic application reconfiguration system using mobile objects.

References

1. Han, R., Bhagwat, P., LaMaire, R., Mummert, T., Perret, V., Rubas, J.: Dynamic adaptation in an image transcoding proxy for mobile web browsing. IEEE Personal Communications Magazine **vol. 5, no. 6** (1998)
2. Fox, A., Gribble, S.D., Brewer, E.A., , Amir, E.: Adapting to network and client variability via on-demand dynamic distillation. In: Proceedings of the Seventh Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII). (1996)
3. Kunz, T., Black, J.: An architecture for adaptive mobile applications. In: Proceedings of Wireless 99, the 11th International Conference on Wireless Communications. (1999)
4. Zachariadis, S., Mascolo, C., Emmerich, W.: Adaptable mobile applications: Exploiting logical mobility in mobile computing. In: Proceedings of 5th Int. Workshop on Mobile Agents for Telecommunication Applications (MATA03), Marrakech, Morocco (2003)
5. Ponnekanti, S., Lee, B., Fox, A., Hanrahan, P., Winograd, T.: Icrafter: A service framework for ubiquitous computing environments. In: Proceedings of the 3rd international conference on Ubiquitous Computing, Springer-Verlag (2001)
6. Nakajima, T., Hokimoto, A.: Adaptive continuous media applications in mobile computing environment. In: Proceedings of 1997 International Conference on Multimedia Computing and Systems (ICMCS '97). (1997)
7. Capra, L., Emmerich, W., Mascolo, C.: Carisma: Context-aware reflective middleware system for mobile applications. IEEE Transactions on Software Engineering **29(10)** (2003)
8. Grace, P., Blair, G.S., , Samuel, S.: Remmoc: A reflective middleware to support mobile client interoperability. In: On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE, Springer-Verlag (2003)
9. Bharat, K.A., Cardelli, L.: Migratory applications. In: Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag (1995)
10. Harter, A., Hopper, A., Steggles, P., Ward, A., Webster, P.: The anatomy of a context-aware application. Wireless Networks **8** (2002)
11. Fuggetta, A., Picco, G.P., Vigna, G.: Understanding code mobility. IEEE Transactions on Software Engineering **24** (1998)
12. Raatikainen, K., Christensen, H.B., Nakajima, T.: Application requirements for middleware for mobile and pervasive systems. SIGMOBILE Mob. Comput. Commun. Rev. **6** (2002)
13. Holder, O., Ben-Shaul, I., Gazit, H.: Dynamic layout of distributed applications in fargo. In: Proceedings of the 21st international conference on Software engineering, IEEE Computer Society Press (1999)
14. Philippsen, M., Zenger, M.: JavaParty — transparent remote objects in Java. Concurrency: Practice and Experience **9** (1997)
15. Spiegel, A.: Automatic distribution in pangaea. In: Proc. Workshop on Proc. Workshop on Communications-Based Systems, CBS 2000. (2000)
16. van Rossum, G.: Python Programming language, (http://www.python.org)
17. de Jong, I.: PYRO - Python remote Objects, (http://pyro.sorceforge.net)