A Reflective Middleware Architecture for Adaptive Mobile Computing Applications

Celso Maciel da Costa [1], Marcelo da Silva Strzykalski [1] and Guy Bernard [2]

[1] Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul,
Av. Ipiranga, 6681, prédio 30,
bloco 4, Porto Alegre, Brazil

[2] Département Informatique, Institut National des Télécommunications,
Rue Charles Fourier, 91000,
Evry, France

Abstract. Mobile computing applications are required to operate in environments in which the availability for resources and services may change significantly during system operation. As a result, mobile computing applications need to be capable of adapting to these changes to offer the best possible level of service to their users. However, traditional middleware is limited in its capability of adapting to the environment changes and different users requirements. Computational Reflection paradigm has been used in the design and implementation of adaptive middleware architectures. In this paper, we propose an adaptive middleware architecture based on reflection, which can be used to develop adaptive mobile applications. The reflection-based architecture is compared to a component-based architecture from a quantitative perspective. The results suggest that middleware based on Computational Reflection can be used to build mobile adaptive applications that require only a very small overhead in terms of running time as well as memory space.

1 Introduction

Wireless devices such as laptop computers, mobile phones and personal digital assistants are becoming very popular. The combined use of wireless networks technologies on these personal devices enables its owners to access their personal information as well public resources anytime and anywhere. Such combination results in a new computational paradigm: mobile computing.

However, developing applications targeted to these types of devices presents challenging problems to designers and developers [4]. Such devices face temporary loss of network connectivity when in movement. They have scarce resources, such as low battery power, slow CPU speed and little memory, which should be exploited in an efficiently manner. Besides, they are required to react to frequent changes in the environment, such as new location and high variability of network bandwidth. As a result, from these limitations, building mobile distributed applications on the top of network layer would be tedious and error-prone. In this case, applications designers and developers would have to deal explicitly with these non-functional requirements. Such constraints are not just artifacts of current mobile devices technology, but they are intrinsic to mobility. Therefore, mobile elements must be adaptive to function in a successful way [21].

Conventional middleware technologies, which reside between the network operating system and the distributed application and implement the session and presentation layer of the ISO/OSI reference model, provide application developers with a higher level of abstraction, hiding the complexity introduced by distribution using the network programming primitives of the network operating system. These technologies have designed for stationary distributed systems built with fixed networks, but they do not appear to be suitable for mobile computing environments, which have an intermittent network connection and a dynamic execution context. Current generation of mainstream middleware is heavyweight, inflexible, monolithic and does not provide the support needed for dealing with the new dynamic aspects in which mobile computing applications need to operate nowadays [3][14].

The development of mobile distributed applications require a middleware that can be adapted to changes in their execution context and customized to fit in many kinds of mobile devices. However, conventional middleware is limited in its capability of supporting adaptation. Adaptive middleware has evolved from conventional middleware to solve this problem. Such next middleware generation should be run time configurable and allow inspection and adaptation of the underlying software. In order to support adaptation, adaptive middleware can employ the Computational Reflection engineering paradigm in addition to object-oriented programming paradigm. Such combination enables middleware to inspect and adapt itself at runtime.

This paper presents an adaptive middleware architecture based on reflection to support adaptation. Section 2 introduces Computational Reflection related concepts. Section 3 analyzes a set of requirements that future middleware platforms should incorporate in its architecture to supporting adaptive mobile applications. Section 4 discusses some adaptation techniques that can be employed in mobile computing applications to reduce energy consumption and to allow user interaction when disconnected from the remote server. Section 5 presents a reflective middleware architecture to support adaptation. Section 6 describes a prototype developed to validate the architecture proposed. Section 7 evaluates the performance of the prototype. Section 8 briefly presents and compares our approach with related work and we outline conclusions and directions for future works in section 9.

2 Computational Reflection

Smith [23] and Maes [15] introduced reflective computing systems in the context of programming language community to support the design of more open and extensible languages. Such computing systems can be made to manipulate representations of itself in the same way as it manipulates representations of its application domain. This self-representation is constituted of both its state and behavior, and can be used for inspection and adaptation of the system’s internals.

More specifically, reflection refers to the capability of a system to reason about, and possibly, alter its own behavior [22]. It is the ability of a system to watch its computation and possibly change the way it is performed. A reflective system provides a representation of its own behavior, which can be used to inspection (i.e., the internal behavior of the system is exposed) and adaptation (i.e., the internal behavior of a system can be dynamically changed) and is causally connected to the underlying behavior it describes. Causally connected means that changes made to the self-representation are immediately mirrored in the underlying system's actual state and behavior and vice-versa.

A reflective system is logically structured in two or more levels, constituting a reflective tower. The first level is the base-level and describes the computations that the system is supposed to do. The second one is the meta-level and describes how to perform the previous computations. The entities (objects) working in the base-level are called base-entities, while the entities working in the other levels (meta-levels) are called meta-entities.

Each level is causally connected to adjacent levels, i.e., entities working into a level have data structures reifying the activities and the structures of the entities working into the underlying level and their actions are reflected into such data structures.

A reflective computation can be separated into two logical aspects: computational flow context switching and meta-behavior. A computation starts with the computational flow in the base-level; when the base-entity begins an action, such action is trapped by the meta-entity and the computational flow rises at the meta-level (shift-up action). Then the meta-entity completes its meta-computation, and when it allows the base-entity to perform the action, the computational flow goes back to the base level (shift-down action).

3 Adaptive Architecture Requirements

Efstratiou *et al.* [10] suggests that there are limitations of current approaches for supporting adaptive applications. Specifically, these approaches lack of support for enabling applications to adapt to numerous different attributes in an efficient and coordinated way. Thus, a new approach is required, which must provides a common space for the coordinated, system-wide interaction between adaptive applications and the complete set of attributes that could be used to trigger adaptation.

This new approach is based on a set of requirements that could be used to develop an appropriate architecture for supporting adaptive applications. The first key requirement of the architecture is to provide a common space for handling the adaptation attributes used by the system in which new attributes can be introduced as and when they become important. The second requirement is to be able to control adaptation behavior across all components involved in the interaction on a system-wide level. A further requirement is to support the notion of system-wide adaptation policies that should enable a system to operate differently given the current context and the requirements of the user. A final requirement arises from the fact that most mobile applications operate in a distributed environment, reason for what the adaptation mechanism need to coordinate all elements involved in the system distributed operation.

4 Adaptation Strategies

Future mobile environments will require software to dynamically adapt to rapid and significant fluctuations in the communication link quality, frequent network disconnections, device resource restrictions and power limitations. Such scenario implies in the fact that software will have to include adaptation techniques in its design and implementation.

Several adaptation techniques can be triggered in all levels of an adaptive application, from system level to user level. In the middleware level, three approaches can be identified [11]: middleware services can attempt to reduce application bandwidth requirements by using data compression techniques before transmission, data can be prefetched and cached during periods of bandwidth high availability in preparation to future bandwidth reduction or service disconnection and clients can be redirected to services available in the local context until network connectivity can be established. In addition, we can include in the middleware level some adaptation techniques to reduce energy consumption and to allow users to continue working when in disconnected state.

4.1 Power Management

Since one of the main limitations on mobile computing is battery life, minimizing energy consumption is essential for maximizing the utility of wireless computing systems. Adaptive energy conservation algorithms can extend the battery life of portable computers by powering down devices when they are not needed. The disk drive, the wireless network interface, the display, and other elements of mobile computing devices can be turned off or placed in low power modes to conserve energy.

Many physical components are responsible for ongoing power consumption in a mobile device. The top three items are, in this order [26]: CPU, screen and disk. Due the fact that hardware technology in this area is still rapidly evolving, power management techniques to reduce display consumption are not explored in this section.

4.1.1 Processor

Power consumed by the CPU is related to the clock rate, the supply voltage and the capacitance of the devices being switched. The reduction in CPU power consumption as the clock rate decreases is a result of the switching characteristics of the logic gates in CMOS VLSI circuits. When a complementary transistor pair in a VLSI circuit switches state, energy is wasted. Unfortunately, most of logic gates in a CMOS VLSI circuit will switch states on every clock circle. Therefore, higher the CPU clock rate, more frequently logic gates are switching, and more energy is wasted.

The power wasted by logic gates during switching is equal to the supply voltage squared divided by circuit's resistance. Because the switching resistance is commonly fixed, the wasted power is proportional to the square of the operating voltage. Besides, the total power required by CPU is proportional to C V^2 F, where C is the total capacitance of the wires and transistor gates, V is the supply voltage and F is the clock frequency. While C can only be changed during the VLSI circuit design, newer devices are beginning to make possible to vary F and V during runtime, which allows achieving linear and quadratic savings in power.

Dynamic Voltage Scaling (DVS) has been a key technique in exploiting the characteristics above mentioned to reduce processors energy dissipation by lowering the supply voltage and operating frequency if it is expected a large amount of CPU idle time [17]. DVS tries to address the tradeoff between performance and battery life taking into account two important features of most current computing systems. First, the peak computing rate needed is much higher than the average throughput, so, high performance is only needed for a small fraction of the time, which allows lowering the operating frequency of the processor when full speed is not a requirement. Second, processors are based on CMOS logic. Such technology allows scaling the operating voltage of the processor along with its frequency. In this manner, by dynamically scaling both voltage and frequency of the processor based on computation load, DVS can provide the performance to meet peak computational demands, while on average, providing the reduced power consumption.

Weiser *et al.* [25] propose an approach that balances CPU usage between periodic bursts of high utilization and the remaining idle periods under the control of the operating system scheduling algorithms by predicting the upcoming workload requirements and adjusting the processor voltage and frequency accordingly. Three algorithms derivate from this approach: OPT, FUTURE and PAST. Each of these algorithms adjusts the CPU clock speed at the same time scheduling decisions are made by the operating system scheduler with the goal of decreasing time wasted in idle loops while retaining interactive response for the user. OPT is completely optimistic (and impractical) because it requires perfect future knowledge of the work to be done in an interval. FUTURE is similar to OPT, except by the fact it looks to the future only in a small window. Unlike OPT, it is practical because it only optimizes over short windows (it is assumed that all idle time in the next interval can be eliminated). PAST is a practical version of FUTURE that uses the recent past as a predictor of the future. Instead of looking a fixed window into the future, it looks a fixed window into the past, and assumes the next window will be like the previous one. Obviously, such approach depends on an effective way of predicting workload to save power by the adjustment of processor speed fast enough to accommodate the workload.

4.1.2 Disk

Spinning down the disk when it is not being used can save power. Such technique is possible by the fact that most mobile computers disk drives have a new mode of operation called SLEEP mode (in such mode, a drive can reduce its energy consumption to near zero by allowing the disk platter to spin down to a resting state). Most, if not all current mobile computers use a fixed threshold specified by the manufacturer to determine when to spin down the disk: if the disk has been idle for some predetermined amount of time, the disk is spun down. The disk is spun up again upon the next access. The fixed threshold is typically about many seconds or minutes to minimize the delay from on demand disk spin-ups.

Spinning a disk for just a few seconds without accessing it can consume more power than spinning it up again upon the next access since spinning the disk back up consumes a significant amount of energy. Therefore, spinning down the disk more aggressively reduce the power consumption of the disk in exchange for higher latency upon the first access after the disk has been spun down.

Douglis *et al.* [9] investigated two types of algorithms for spinning a disk up and down minimizing power consumption and response time: off-line, which can use future knowledge and on-line, which can use only past behavior. Off-line algorithms are useful only as a baseline for comparing different on-line algorithms. On the other hand, on-line algorithms are implementable. A perfect off-line algorithm can reduce disk power consumption by 35-50% when compared to the fixed threshold suggest by manufacturers. On the other hand, an on-line algorithm with a threshold of 10 seconds reduces energy consumption by about 40% compared to the 5-minute threshold recommended by manufacturers.

4.2 Disconnected Operation

Wireless networks are very susceptible to suffering disconnections, so this is a very important aspect to keep in the mind when designing architectures to support mobile computing. We can classify the disconnections in two mainly types: forced disconnections (usually accidental and unavoidable, that takes place when the user enters in an out-of-coverage area) and voluntary disconnections (when the user decides to disconnect from the network to saving energy).

Forced disconnections as well as voluntary disconnections are frequent in a mobile computing environment. But, as can been seen in Coda File System project [20], the use of caching and server replication techniques can mitigate the undesirable effects of disconnections, which allows users to be able to continue working even in disconnection states. In this mode of operation, a client continues to have read and write access to data in its cache during temporary network outages, being the system responsible to propagating modifications and detecting conflicts when connectivity is restored. In addition, disconnected operation can extend battery life for avoiding wireless transmission and reception.

5 A Reflective Middleware Architecture

Welling [27] asserts that adaptive techniques must be decoupled from basic application functionality due to the complexity of building adaptive applications for mobile computing. Such principle allows both applications and adaptive techniques be designed and implemented independent of each other.

In this direction, Zhang and Jacobsen [28] observe that middleware platforms architectures have been evolving exactly by the necessity of a software layer that decouples applications from the concern of handling the complexity related to distributed computing environments.

The architecture proposed in this section employs this principle of decoupling adaptation techniques (meta-level) from application basic functionality (base-level), as can be seen in figures 1 and 2.

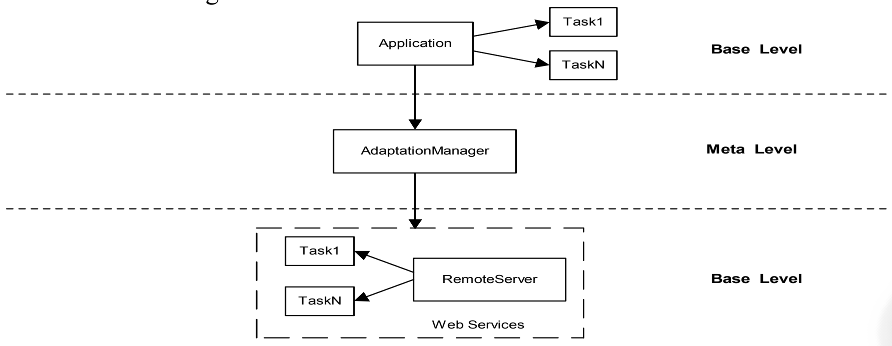

Fig. 1. Separation of concerns in the proposed reflective middleware architecture.

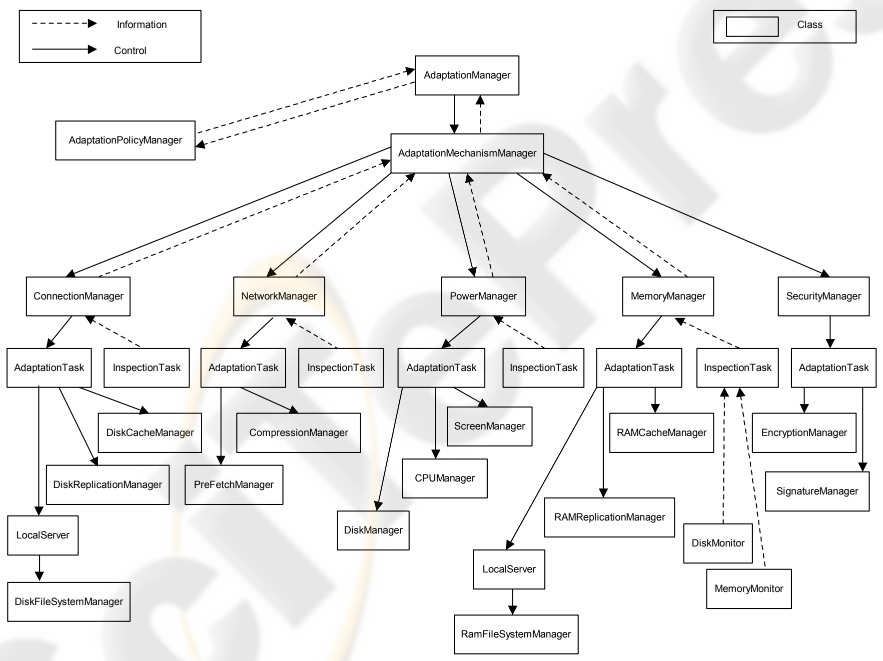

Fig. 2. The proposed reflective middleware architecture.

The Adaptation Manager (AM) decides which adaptation strategies should be executed from data supplied by the inspection task of each resource managed by the same. Attention might be given to this module in the architecture. Because conflicts may arise during the execution of a specific adaptation strategy, this module must guarantee that such conflicts can be solved in a coordinate manner. The set of attributes (computational resources) to be managed by the AM must be extensible; by the way, new attributes can be added in the proposed architecture. The AM depends on two modules which responsibilities are complementary, described below.

The Adaptation Policy Manager loads the adaptation policies described in the application profile, which is defined by the application's user. The application profile (which describes the application nonfunctional behavior) is encoded using the Extensible Markup Language (XML) due the fact such language supports a representation of information that is both easily handled by machines and readily understandable by humans. The application profile presented in figure 3, for example, inspect the battery resource each 10 seconds and spindown the hard disk and scale CPU frequency when the amount of energy available in the battery is below 10%. In addition, the Adaptation Policy Manager updates dynamically the adaptation policies using statistical learning methods.

```
<?xml version="1.0"?>
<ApplicationProfile>
<InspectionTasks>
<Task name="BatteryInspector">
<Frequency value="10"></Frequency>
</Task>
</InspectionTasks>
<AdaptationTasks>
<Task name="DiskManager">
<Resource name="Battery">10</Resource>
<Threshold value="10"></Frequency>
</Task>
<Task name="CpuManager">
<Resource name="Battery">10</Resource>
<Threshold value="10"></Frequency>
</Task>
</AdaptationTasks>
</ApplicationProfile>
```

Fig. 3. An example of the application profile.

The Adaptation Mechanism Manager inspects and adapts the following attributes: connectivity, network bandwidth, energy and memory. The following modules manage such attributes: Connection Manager, Network Manager, Power Manager and Memory Manager. Besides, this module should guarantee the security of data exchanged between the application and the Remote Server. Such behavior is encapsulated in the Security Manager module. It should be noted that the attributes managed by this module can be extended dynamically by changes in the application profile. At the run time, the Adaptation Mechanism Manager checks the conditions of each adaptation rule described in the application profile to determine if an adaptation task should be performed. To execute this operation, the class `AdaptationMechanismManager` provides two methods that systematically iterates through each rule coded in the application profile to check conditions and load an unload adaptation tasks in accordance to these rules. The Connection Manager monitors the connectivity between the application and the Remote Server. In case of disconnection, the data handled by the application are gotten from the Local Server module. The methods invoked toward the Remote Server are stored in the local cache and will be deferred to execute by the Replication Manager adaptation task when the Remote Server was in connected mode again. The Network Manager monitors the network bandwidth. This module compresses the body message that will be sent to the Remote Server. Besides, this module pre-fetches messages that were posted in the Remote Server and were not replicated to the Local Server yet. The Power Manager monitors energy. If the amount of energy in a moment were below a boundary expressed in the application profile, the interface between the application and the user is text based. Besides, the hard disk can be put in spindown mode and the CPU frequency can be scaled to save battery power. The Memory Manager monitors memory and disk space. If the amount of disk space in a moment were below a boundary expressed in the application profile, the messages posted in the Local Server as well as data from local cache will be stored in the RAM memory, not more in the hard disk. The deferred methods will be stored in RAM memory either. The Security Manager module should guarantee the confidentiality (XML Encryption) and the integrity (XML Signature) of the XML messages exchanged by the application and the Remote Server.

6 Implementation

A simplified mail server prototype was implemented based on Web Services and Java technologies to evaluate the proposed architecture. The prototype only implements the Connection Manager and the Power Manager modules. Therefore, the platform's evaluation performance reflects only these attribute's measures. We have employed a collaborative relationship between the operating system and the application by the middleware level in such prototype, which modifies its behavior to conserve energy to meet user-specified goals for battery duration. In addition, our approach predicts future energy demand from measurements of past usage.

The Power Manager measures energy consumption by using the ACPI subsystem in Linux to get an accurate evaluation of the remaining capacity of the battery. The Screen Manager changes the user interface from text to graphic mode and vice-versa in accordance to the current energy level available. The CPU Manager implements the PAST algorithm using the CPUFreq loadable kernel module framework [7], which is a project for adding support for CPU frequency and voltage scaling to the Linux kernel. The PAST algorithm calculates that the upcoming interval will be as equally busy as the previous interval. The speed policy is as follow: if the prediction is for a mostly-idle interval, PAST decreases speed; and if the prediction is for a busy interval, PAST increases speed. To avoid excessive fluctuations in processor speed (variable performance for the user), PAST will limit the amount in change of speed in a decision to a maximum of 20% of the maximum speed. The Disk Manager implements the on-line disk spindown algorithm by the use of noflushd [16], which is a Linux daemon that monitors disk activity and spins down idle disks. It then blocks further writes to the disk to prevent it from spinning up again. Writes are cached and flushed to disk when the next read request triggers a spin-up.

The application environment is composed by a Local Server and by a Remote Server. The Remote Server was implemented with the use of the Axis server API [1]. It is a Java class that exposes public methods for invocation and is responsible by the following operations: create mailboxes, delete mailboxes, delete messages, list messages, read a message and send a message. The Remote Server encapsulates in its public interface the semantics of the main SMTP commands as exposed in the Simple Mail Transfer Protocol (RFC2821) as well as the semantic of the main IMAP client commands as exposed in the Internet Message Access Protocol (RFC3501). The Local Server is responsible by delete messages, list messages, read a message and send a message. It implements partly both specifications in the local system context and employs a queued remote procedure call based technique [12] that permits applications to continue to make non-blocking remote procedure calls even when the Remote Server is off-line enabling the system to operating in disconnected mode. In this case, requests and responses are exchanged upon network reconnection. The consistency between data replicated from Remote Server to Local Server is based in some clustering principles employed in mobile databases context [18]. Clustering maintains two copies of every object: a strict version, which is globally consistent, and a weak version, which can be globally inconsistent, but must be locally consistent. Weak versions are transformed in strict by the Replication Manager, which compares the version number of both weak and strict objects to decide what is the more recent.

The proposed system operates as follows. At the mail's client first request the mailbox data is copied from the Remote Server to the Local Server. The read operations are always local and in this case, the disconnections from the Remote Server are not important. The write operations are executed simultaneously in the Remote Server and Local Server. The mail client is a Java class that implements the user interface and uses Axis client API to call the services exposed by the Remote Server. The class that implements the AM, which allows the adaptation techniques to take place, intercepts all the calls sent by the client.

7 Evaluation

In this section, we try to prove that computational reflection can be used to develop adaptive mobile applications that require only a very small overhead in terms of running time as well as memory footprint. The following measures are the mean value gotten after 5000 executions for each operation below in an AMD K6-2 500 MHz machine with 184 Mb of memory running a Fedora Core Linux 2.0 kernel 2.6.19.

Before the data analysis, it should be noted that in the component-based implementation the remote methods are invoked directly in the Remote Server class by the intermediation of a Proxy class. In this implementation, there no exist meta-levels between the client and the remote server. In this manner, the component-based implementation is not designed to adapting to environment changes.

Table 1. Running time to respond a service request.

Implementation	Create a mailbox	Delete a mailbox	Delete a message
Component-based	2181.50 ms	2096.50 ms	2163.30 ms
Reflection-based	3027.33 ms	2268.50 ms	2658.50 ms

Table 2. Running time to respond a service request (continuation).

Implementation	List messages	Read a message	Send a message
Component-based	511.60 ms	197.50 ms	2678.50 ms
Reflection-based	370.33 ms	76.66 ms	3279.50 ms

The collected results presented in tables 1 and 2 suggest that reflection can be used to develop mobile adaptive applications which require only a small overhead in terms of running time. The write operations (create a mailbox, delete a mailbox, delete a message and send a message) require more time to running (the code size is bigger than component-based). However, the read operations require less time due the fact that these operations are served by the local system.

Table 3. Code size.

Implementation	Code size (disk)	Code size (memory)	Number of classes
Component-based	85231 bytes	2429544 bytes	46
Reflection-based	149507 bytes	2066232 bytes	64

From table 3, it can be verified that reflective-based code size in disk is 42,3 % bigger than component-based code. However, reflective-based code size in memory is 14,5 % smaller than component-based code. It should be emphasized that reflective-based code has a code overhead of 18 classes (64276 bytes), but in this case the usage of memory is lower (the classes are loaded dynamically, on demand, by the Adaptation Manager).

8 Related Work

The middleware community has already investigated the principle of reflection during the past years, mainly to achieve flexibility and dynamic reconfigurability of the Object Request Broker (ORB) of CORBA. Examples include OpenCorba [24], DynamicTAO [13] and OpenORB [2]. OpenCorba is a CORBA compliant ORB that uses reflection to expose and modify some internal characteristics of CORBA. OpenCorba is implemented in NeoClasstalk, a reflective language based on Smalltalk. OpenCorba allows the dynamic modification of a remote invocation mechanism though a proxy class, which is its major reflective aspect. DynamicTAO is a reflective CORBA ORB written in C++ which extends TAO [8] to support runtime configuration already at the startup time of the ORB engine and non-CORBA applications OpenORB is a reflective middleware that has been implemented using Python and was designed to target configurable and dynamically reconfigurable platforms for applications that require dynamic requirements support. However, such platforms were based on standard middleware implementations and are therefore targeted to a wired distributed environment.

FlexiNet [19] is another CORBA compliant ORB implemented in Java that uses reflection to provide dynamic adaptation. Nevertheless, FlexiNet only supports static configuration of communication protocols stack layers at compile time.

OpenCOM [6] is a reflective middleware based on a component framework built atop a subset of Microsoft's COM, which can be specialized to application domains such as multimedia, real-time systems and mobile computing. However, OpenCOM only runs at Microsoft platforms. On the other hand, mobile devices such smart phones and personal digital assistants (PDA) is already coming with the Java Virtual Machine installed by default, which justifies our approach. Moreover, the Java language allows the development of applications that run in heterogeneous operating systems and machine architectures.

RECOM [22], like FlexiNet, has a reflective structure based on the Java platform that supports different transformations on a remote method invocation. RECOM supports dynamic configuration of the binding between the client and the server, such as inserting into the communication protocol stack some reflective layers of high level features which meet the needs of some nonfunctional properties required by adaptive middleware platforms. However, such platform only supports dynamic adaptation of the remote invocation mechanism (cache the results on client side and redirect the invocation to an alternative server when the initial server is down). Differently, our reflective architecture is extensible; thus, new attributes (computational resources) can be added and removed from the same and be managed in a coordinate manner.

CARISMA [5] is an adaptive middleware platform implemented in Java, which employs reflection and metadata to enable context-aware interactions between mobile applications. In such platform, the middleware can be seen by applications as a dynamically customizable service provider, where the customization takes place by means of application profiles. Each application profile defines associations between the services that the middleware delivers, the policies that can be applied to deliver the services and context configurations that must hold in order for a policy to be applied. Our abstraction of application profiles is based on this work.

9 Conclusions

Because conventional middleware technologies do not provide appropriate support for handling the dynamic aspects of mobile applications, the next generation applications will require a middleware platform that can be adapted to changes in the environment and customized to several computational devices.

Computational Reflection allows the creation of a middleware architecture that is flexible, adaptable and customizable. The architecture proposed in this work employs this concept in its design and implementation. The results appointed by the researchers in the reflective middleware field were validated with data collected from the prototype experimental evaluation.

About future work, we have to solve the problems below mentioned. The Adaptation Policy Manager does not have yet a module to resolve conflicts that can occur between the application policies. In addition, the statistical learning methods employed by this component to updating policies dynamically have to be designed and implemented and we still have to implement the others architecture components (Network, Memory and Security Managers).

References

1. Apache Software Foundation. Axis: A framework for constructing SOAP processors. In: http://ws.apache.org/axis, December 2005.
2. Blair, G. S., Coulson, G., Robin, P., Papathornas M. Architecture for Next Generation Middleware. In: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, the Lake District, England, September 1998.
3. Capra, L., Blair, G. S., Mascolo, C., Emmerich, W., Grace, P. Exploiting Reflection in Mobile Computing Middleware. ACM SIGMOBILE Mobile Computing and Communications Review, Vol. 6, No. 6, pp 34-44, 2002.
4. Capra, L., Emmerich, W., Mascolo, C. Middleware for Mobile Computing. In: Advanced Lectures on Networking - Networking 2002 Tutorials, Pisa, Italy. Volume 2497 of LNCS, pages 20-58, Springer Verlag. May 2002.
5. Capra, L., Emmerich, W., Mascolo, C. CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. IEEE Transactions on Software Engineering, 29(10):929-945, 2003.
6. Clarke, M., Blair, G., Coulson, G., Parlavantzas, N. An Efficient Component Model for the Construction of Adaptive Middleware. In: Proceedings of Middleware 2001, Heidelberg, Germany, November 2001.
7. CPUFreq. In: http://www.linux.org.uk/listinfo/cpufreq, December 2005.
8. Douglas, C. S., Cleeland, C. Applying Patterns to Develop Extensible ORB Middleware. IEEE Communications Magazine Special Issue on Design Patterns, 37(4), 54-63, May 1999.
9. Douglis, F., Krishnan, P., Marsh, B. Thwarting the Power-Hungry Disk. In: Proceedings of Winter USENIX Conference, California, 1994, pp. 292–306.
10. Efstratiou, C., Cheverst, K., Davies, N., Friday, A. Architectural Requirements for the Effective Support of Adaptive Mobile Applications. In: Proceedings of 2nd International Conference in Mobile Data Management. Hong Kong, Springer, Vol. Lecture Notes in Computer Science Volume 1987, pp. 15-26, January 2001.
11. Friday, A., Davies, N., Blair, G. S., Cheverst, K. W. J. Developing Adaptive Applications: The MOST Experience. Journal of Integrated Computer-Aided Engineering, Volume 6, Number 2, 1999, pp143-157.
12. Joseph, A., deLespinasse, A., Tauber, J., Gifford, D., and Kaashoek, M. Rover: A Toolkit for Mobile Information Access. In: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, December 1995.
13. Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhaes, L. C., R., Campbell, H. Monitoring, Security and Dynamic Configuration with the DynamicTAO Reflective ORB. In: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, New York, April 2000.
14. Kon, F., Gordon, B., Costa, F., Campbell, R. H. The Case for Reflective Middleware, CACM, Vol. 45, No. 6, pp 33-38, 2002.
15. Maes, P. Concepts and Experiments in Computational Reflection. In: Proceedings of the ACM Conference on Object-Oriented Languages, December 1987.
16. Noflushd. In: http://sourceforge.net/projects/noflushd, December 2005.

17. Pillai, P., Shin, K. G. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In: Proceedings of the Eighteenth ACM Symposium on Operating systems principles, Alberta, Canada, October 2001.
18. Pitoura, E., Bhargava, B. Maintaining Consistency of Data in Distributed Environments. In: Proceedings of Fifteenth International Conference on Distributed Computing Systems, Vancouver, Canada, May 1995.
19. R. Hayton, ANSA Team. FlexiNet Architecture. Architecture Report, Citrix Systems (Cambridge) Limited, February, 1999.
20. Satyanarayanan, M., Kistler, J. J., Mummert, L. B., Ebling, M. R., Kumar, P., Lu, Qi. Experience with Disconnected Operation in Mobile Computing Environment. In: Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing, Cambridge, MA, August 1993.
21. Satyanarayanan, M. Mobile Information Access. IEEE Personal Communications, February 1996.
22. Sizhong, Y., Jinde, L. RECOM: A Reflective Architecture of Middleware. In: Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, September 2001.
23. Smith, B. C. Reflection and Semantics in a Procedural Language. PhD thesis, MIT Laboratory of Computer Science, 1982, MIT Technical Report 272.
24. T. Ledoux. OpenCorba: A Reflective Open Broker. Lecture Notes in Computer Science, vol. 1616, 1999.
25. Weiser, M., Welch, B., Demers, A., Shenker, S. Scheduling for Reduced CPU Energy. In: Proceedings of Symposium on Operating Systems Design and Implementation, November 1994.
26. Welch, G.F. A Survey of Power Management Techniques. In Mobile Computing Operating Systems. Operating Systems Review, Volume 29, Number 4, October 1995.
27. Welling, G.S. Designing Adaptive Environmental-Aware Applications for Mobile Computing. PhD thesis, Rutgers University, New Brunswick, July 1999.
28. Zhang, C., Jacobsen, H. Aspectizing Middleware Platforms. Technical Report, Computer Systems Research Group, CSRG-466, University of Toronto, Canada, January 2003.