A Step Towards Ubiquitous Computing : an Efficient Flexible micro-ORB

Frédéric Ogel*
INRIA Rocquencourt - Regal group
Domaine de Voluceau
78150 Le Chesnay, France
frederic.ogel@inria.fr

Bertil Folliot
LIP6, Universit´e Pierre et Marie Curie,
4, place Jussieu,
75252 Paris Cedex 05, France
bertil.folliot@lip6.fr

Ga el Thomas
LIP6, Universit´e Pierre et Marie Curie,
4, place Jussieu,
75252 Paris Cedex 05, France
gael.thomas@lip6.fr

ABSTRACT

Smart devices, such as personal assistants, mobile phone or smart cards, continuously spread and thus challenge every aspect of our lives. However, such environments exhibit specific constraints, such as mobility, high-level of dynamism and most often restricted resources. Traditional middlewares were not designed for such constraints and, because of their monolithic, static and rigid architectures, are not likely to become a fit.

In response, we propose a flexible micro-ORB, called FlexORB, that supports on demand export of services as well as their dynamic deployment and reconfiguration. FlexORB supports mobile code through an intermediate code representation. It is built on top of NEVERMIND, a flexible minimal execution environment, which uses a reflexive dynamic compiler as a central common language substrate upon which to achieve interoperability.

Preliminary performance measurements show that, while being relatively small (120 KB) and dynamically adaptable, FlexORB outperforms traditional middlewares such as RPC, CORBA and Java RMI.

1. INTRODUCTION

Recent developments in wireless and embedded technologies have led to the emergence of pervasive computing: a step towards ubiquitous computing. Active spaces and, more generally, ubiquitous computing seem to be no longer science fiction. As envisioned by Weiser in [17], ubiquitous computing relies on the interactions between *evanescent* systems vanishing into *active spaces*. It thus introduces issues like: context awareness and sensitivity, user-centrism, dynamic flexibility, interoperability and mobility [2]. Active spaces rely on the interactions of smart devices, such as personal assistants, and their environment, composed of projectors and printers for example. Realistic scenarios highlighting those issues can be found in [2, 7]. With respect to their limited resources, smart devices can not be provided in advance with every protocol or service they might need once and for all. In particular, such high-levels of heterogeneity and dynamism challenge traditional middleware architectures and raise severe interoperability and portability issues [16]. As previously stated in [7], mobile code is a solution to cope with such constraints.

Middlewares have emerged to hide issues related to distributed heterogeneous environments, such as network communications or interoperability. Nonetheless, they are still designed for server and workstation based environments with a monolithic architecture. Hence, they are not suitable for active spaces and smart objects. They lack flexibility, dynamism and are still too large to fit in resource-limited devices. Work around reflexive middlewares, as described in [4], represents a first step towards a solution to ubiquitous computing.

We propose a flexible micro-ORB based on the NEVERMIND dynamically adaptable minimal execution environment. It relies on reflection and dynamic compilation provided by NEVERMIND to support both mobile code, *on demand* export and deployment of components, as well as dynamic reconfiguration of inter-components bindings. Hence, it deals with interoperability and dynamic adaptation issues in such interaction-based environments as active spaces.

This paper makes the following contributions to the design of flexible middlewares for ubiquitous computing:

1. it describes the design and implementation of a small memory footprint flexible micro-ORB;
2. it shows that such a micro-ORB can outperform traditional middlewares on static remote invocations, while providing support for code mobility as well as *on demand* deployment and reconfiguration;
3. it indicates that dynamic flexibility itself can be very efficient.

*now with France Telecom R&D (DTL/ASR), 38-40, rue du général Leclerc, 92794 Issy Les Moulineaux Cedex 9, France.

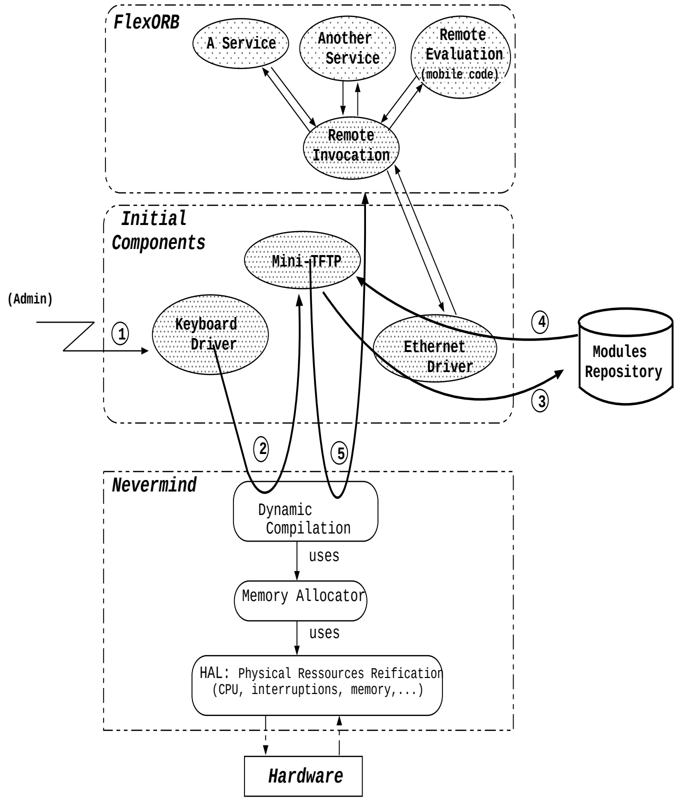

Figure 1: Dynamic construction of FlexORB.

The remainder of this paper starts with a presentation of the FlexORB flexible micro-ORB in Section 2. Section 3 presents some performance measurements, showing an improvement factor for remote invocations ranging from 7.5 to 63.7 compared with various CORBA implementations, followed by related work in Section 4. We conclude and give some perspectives in Section 5.

2. FlexORB: A FLEXIBLE MICRO-ORB

FlexORB is a flexible *minimal* middleware designed to support *on-demand* export of components, dynamic flexibility, interoperability and mobile computing. It is built on top of a dynamically adaptable minimal execution environment called NEVERMIND.[1] This section first introduces the NEVERMIND execution environment, followed by a presentaiton of FlexORB itself. With respect to traditional middleware architectures, we have implemented a naming service and a component trader, which are not described in this paper for sake of place.

2.1 Nevermind: a flexible execution environment

NEVERMIND offers support for interoperability, dynamic flexibility and mobile code while preserving performance. It is structured as a set of components and interfaces based on the ODP reference model [8]. NEVERMIND is based on a reflexive dynamic compiler [12] and an *Hardware Abstraction Level* (HAL) [6] (for more details on this architecture and the Virtual Vitrtual Machine project, see [11]). The HAL is responsible for reifying hardware resources in a policy neutral way, that is reifying access to physical resources without adding any semantic. The compiler defines a reflexive and open chain of dynamic compilation used to incrementally define higher-level abstractions and to execute arbitrary scripts (*i.e.* applications, extensions or reconfigurations).

NEVERMIND follows an *Exokernel-like* [5] approach: it defines a minimal environment that is extended or specialized by dynamically compiling appplication-level extensions. Moreover, the reflexivity of the compiler allows arbitrary adaptation of any language aspect, through the reification of its internals, as well as any application-level code, by keeping meta-data issued from the compilation phase. NEVERMIND defines a central common language substrate (and runtime), upon which interoperability can be achieved. In particular, it uses an intermediate data/code representation, based on *Lisp-like* syntax trees, as a front-end language.

In complement, this minimal execution environment includes some device drivers (mainly keyboard, framebuffer and network adapter) and a TFTP-like protocol used to incrementally load additional modules from a remote repository. Hence, dedicated execution environments are dynamically built and extended *on demand*.

Figure 1 represents the dynamic construction of such an environment upon NEVERMIND: a dynamically adaptable Micro-ORB called FlexORB. At the lowest level the HAL reifies physical resources through a set of components. The memory allocator is based on the component reifying access to physical memory. The dynamic compiler is in turn based on this minimal memory allocator. The administrator types *loading* commands (*step 1*), such as `(tftp-get a-repository a-package-name)`, in the console interface of NEVERMIND. Those commands are dynamically compiled (*step 2*) and executed: a request is sent to the specified repository through the TFTP-like protocol and the underlying network driver (*step 3*). This results in the download of the requested module using the intermediate representation (*step 4*). Once loaded, it is dynamically compiled, hence defining new components and services (*step 5*). Modules dependencies are checked after deployment (during the dynamic compilation), potentially resulting in additional requests to the repository. This minimal execution environment runs on bare hardware (currently PowerPC processors), with a memory footprint of 120 KB.

2.2 FlexORB

FlexORB defines a basic remote invocation protocol implemented directly upon the network adapter. It offers a checksum-based integrity control and supports message fragmentation. As with traditional middlewares, this remote invocation protocol is based on stub/skeleton pairs (or proxies) for accessing *statically* defined services. But, as opposed to traditional middlewares in which proxies are statically generated, FlexORB allows dynamic generation of proxies. Hence components can be exported *on demand*. Moreover, bindings between components can be dynamically reconfig-

[1]French acronym that stands for NEVERMIND *is an Extensible, Reflexive, Minimal and Dynamic Execution Environment.*

	Min	Max
FlexORB	27%	32%
RPC(UDP)	100%	100%
RPC(TCP)	111%	120%
ORBit2(IIOP)	216%	229%
Jonathan(IIOP)	535%	998%
RMI	769%	1430%
OpenORB(IIOP)	991%	1719%

Figure 2: Comparative[3] performance of remote invocations'response times.

ured: migration of a component can be addressed by dynamically generating a *redirection* proxy and an overloaded component can be replicated *on the fly* similarly. Moreover, those dynamic adaptations are completely transparent to the client.

Based on the intermediate code representation, we have implemented a remote evaluation facility to support mobile code. It is defined as a proxy for the `readEvalPrint` main function of the compiler that receives *serialized* code to execute as a parameter. Code serialization is a recursive transformation of the intermediate code representation into a sequence of bytes. For example, an expression being a *list* object containing *objects*, its serialized representation starts by the bytecode associated with the *list* type followed by the serialized representation of each *object* it contains.

Whereas *simple* bindings reconfigurations are based on dynamic compilation of *local* proxies, mobile code allows for more complex adaptations: both ends of a binding (server-side as well as client-side) are dynamically re-compiled and re-deployed.

3. PERFORMANCE MEASUREMENTS

In order to evaluate FlexORB, we developed several services performing classical simple tasks (arithmetics, table insert/lookup, string manipulations and file open/read/close) with various size of parameters (ranging from none up to 128 bytes) and results (ranging from none up to 4 KB). We used two 366 MHz G3 PowerPCs running Linux 2.4 and connected through a 100Mbits Ethernet. We implemented our test services with traditional SUN's RPC, a 100% C CORBA (ORBit 2.9.1), Java RMI (IBM 1.4 runtime, with Just-In-Time compilation enabled), a generic middleware providing a Java-based CORBA implementation (Jonathan 3.0) and a reflexive CORBA implementation (OpenORB 1.2.1).

3.1 Remote invocations

As illustrated in Figure 2, our lightweight FlexORB outperforms traditional distributed environments: using a high-level protocol introduces a heavier overhead, as well as using a generic approach. Based on the set of services we used, FlexORB is nearly 3.5 times faster than traditional RPC using UDP, 4 times faster than TCP-based RPC and more than 7.5 times faster than a 100% C implementation of CORBA. Although Java-based solutions (both RMI and

[3]RPC over UDP is used as the reference.

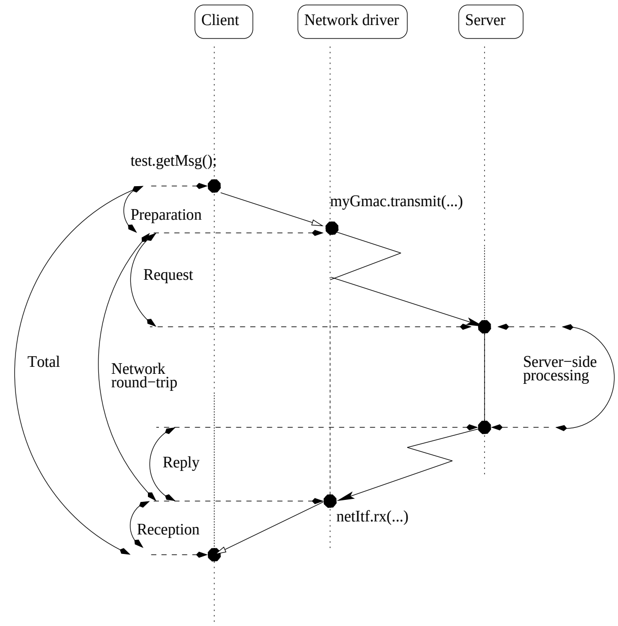

Figure 3: Decomposition of a remote invocation.

CORBA) used a dynamic (Just-In-Time) compiler too, FlexORB is still between 16.7 and 63.7 times faster. Hence, performance can be significantly improved by applying an *Exokernel-like* (minimalist) approach combined with a reflexive open environment. The overhead associated with genericity and high-levels protocols is no longer mandatory.

In order to present an in-depth evaluation of FlexORB, Figure 3 represents a decomposition of remote invocations response times, as experienced by the client, into four major steps: request preparation, physical round-trip over the network, reply reception and server-side processing. Figure 4 shows the detailed performance of several remote invocations. `empty` is a very simple method, without any parameter nor return value, that increments a local counter. `getMsg` has no parameter and returns a constant `string` (32 Bytes). `echoBytes` takes 128 bytes as a parameter and returns another 128 bytes array. `lookup` and `register` respectively correspond to search and insert operations on a relatively small table of structures ($\equiv$ 100 elements). `openFile`, `closeFile` and `readFile` are just wrappers over standard `open`, `close` and `read` system calls. We tested `readFile` without any I/O optimization, then with an I/O cache and read ahead enabled (to fairly compare with Linux-based solutions). The overall performance is clearly limited by the physical transport of messages over the network: even on a switched 100Mb/s Ethernet, it represents at least 91% of the response-time experienced by the client.

3.2 On-demand bindings

As mentioned in Section 2, the dynamism of FlexORB permits dynamic reconfiguration of components' bindings through dynamic recompilation of proxies. Such recompilations are rather cheap operations. For example, dynamically compiling a *proxy-factory* for an interface composed of a dozen of methods takes 474 μs. This *proxy-factory* is then used to

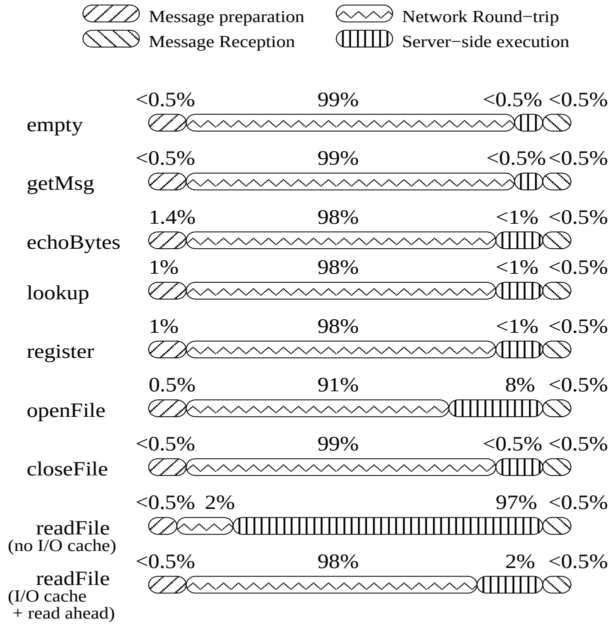

Figure 4: Detailed performance of remote invocations.

```
(module :myRPC)

(define echo
    (lambda(ptr)
        (:system.printf "received::%s\n" ptr)
        ))

(module :global)

(:rpc-register "myRPC.echo" myRPC.echo)
```

Figure 5: A simple deployment script.

produce proxies *on the fly* for any component implementing this interface in a couples of μs. Hence, components can export services *on demand*, in a completely transparent way for the client.

3.3 Mobile code

Nonetheless, such static RPC-like remote invocations are not sufficient to support the construction of active spaces [7]. Using the intermediate representation defined in NEVERMIND, FlexORB offers a direct support for mobile code. Services or protocols are deployed *on demand* and dynamically adapted, as and when needed. Figure 5 represents a script deploying a simple *HelloWorld-like* service. It is first serialized as approximately a hundred bytes long message and sent to the server. The `remote-eval` handler is called with the serialized code as parameter. It deserializes, compiles and executes the script: a new namespace `myRPC` and a handler function (`echo`) are defined, then the service is exported

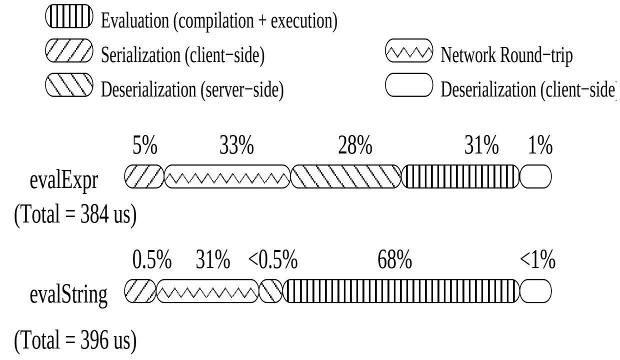

Figure 6: Detailed performance of mobile code.

through the call to *rpc-register*.

Figure 6 represents two detailed decompositions of the response time experienced by the client when invoking the remote evaluation of the previously presented deployment script: `evalExpr` uses a binary representation of syntax trees for code serialization and `evalString` uses a text-based representation for code serialization. In the former case, the code serialization and deserialization is heavier but it includes steps from the compilation phase: hence the larger dynamic compilation time with the second method. Since performance are quite similar, choosing between those two methods depends essentially on the relative processing power of both client and server as well as whether the serialized code is to be sent or executed more than once.

4. RELATED WORK

As distributed applications composed of heterogeneous interacting components became widely used, middlewares such as SUN's RPC, CORBA or Java RMI, have emerged as a solution to ease their development. They were designed to address interoperability and distribution issues by hiding network communications in server and workstation based environments and rely on large, static, rigid and monolithic architectures. Hence, they do not match constraints of emerging application-domains, such as multimedia [3] or ubiquitous computing [15].

Several projects propose to introduce reflection and design patterns at the ORB level to bring more flexibility. Zen [10] is a real-time ORB designed to support distributed embedded applications. It uses a design pattern known as *Virtual Component* to factor out rarely-used functionalities, resulting in a smaller memory footprint (around 60 KB for the root POA) [9]. Nevertheless, it relies on a standard monolithic JVM which should be factored into the results, both in term of memory footprint and flexibility.

DynamicTAO [13] is a CORBA ORB that uses reflection to manage dependencies among components and thus allows *safe* dynamic reconfigurations. It has evolved into a reflexive infrastructure dedicated to ubiquitous computing (UIC) [1]. It is composed of a core (a set of components implementing various behaviours) and a dynamic configuration tool used to specialize and reconfigure the generic core.

As they are not based on some kind of virtual machine with a portable code representation, they are unlikely to handle very heterogeneous dynamic environments requiring mobile code.

LegORB [14] is a small CORBA compliant ORB designed for ubiquitous computing. It has a rather small footprint, while supporting dynamic adaptation of core mecanisms: 140 KB for the dynamically adaptable version, running on top of an underlying OS, such as PalmOS or WindowsCE. LegORB, as DynamicTAO, does not offer any support for mobile code.

5. CONCLUSION AND PERSPECTIVES

This paper presents the design and implementation of FlexORB, a small footprint flexible micro-ORB based on the NEVERMIND dynamically adaptable minimal execution environment. It addresses heterogeneity and dynamism issues found in ubiquitous computing environments by providing support for mobile code and dynamic flexibility.

Preliminary results show that FlexORB significantly outperforms traditional middlewares on static remote invocations. Moreover, they demonstrate that both dynamic flexibility and mobile code are very efficient (hundreds of μs). Whereas flexibility traditionally comes at the cost of performance, our experience with FlexORB lead to reconsidering this belief.

However, more work needs to be done, in particular concerning consistency checks in reconfigurations and deployments. NEVERMIND provides support for code analysis and verification during the dynamic compilation phase, but a dedicated language for expressing consistency or security rules is still necessary. An approach based on *Domain Specific Languages* would certainly help to solve this issue. Moreover, since NEVERMIND can interoperate with compiled C and C++ codes, we investigate re-use of existing ODP-based components. We also plan to experiment further with FlexORB on several devices, like iPaqs, over wireless connections.

6. REFERENCES

[1] Universaly interoperable core. `http://ubi-core.com`.

[2] G. Banavar and A. Bernstein. Software infrastructure and design challenge for ubiquitous computing applications. *Communications of the ACM*, 45(12):92–96, December 2002.

[3] G. Blair. On the failure of middleware to support multimedia applications. In *Proceedings of the 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS'2000)*, Enschede, Netherlands, October 2000.

[4] G. Blair, G. Coulson, F. Costa, and H. Duran. On the design of reflective middleware platforms. In *Proceedings of the Workshop on Reflective Middleware (RM'2000)*, New York, April 2000.

[5] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: an operating system architecture for application-level ressource management. In *the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, Colorado, USA, December 1995.

[6] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A Software Framework for Component-based Operating System Kernels. In *the USENIX Annual Technical Conference*, pages 73–86, Monterey, CA, USA, June 2002.

[7] R. Grimm, T. Anderson, B. Bershad, and D. Wetherall. A system architecture for pervasive computing. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 177–182, Kolding, Denmark, 2000.

[8] ISO International Organization for Standardization. *Information Technology – Open Distributed Processing – Reference Model*, ISO/IEC 10746-(1-4) edition, 1996 — 1998.

[9] R. Klefstad, A. Krishna, and D. Schmidt. Design and performance of a modular portable object adapter for distributed, real-time, and embedded corba applications. In *Proceedings of CoopIS/DOA/ODBASE*, pages 549–567, 2002.

[10] R. Klefstad, D. Schmidt, and C. O'Ryan. Towards highly configurable real-time object request brokers. In *Proceedings of the Symposium on Object-Oriented Real-Time Distributed Computing*, pages 437–447, 2002.

[11] F. Ogel, G. Thomas, I. Piumarta, A. Galland, B. Folliot, and C. Baillarguet. Towards Active Applications: the Virtual Virtual Machine Approach. In *New Trends in Computer Science and Engineering.* A92 Publishing House, POLIROM Press, 2003.

[12] I. Piumarta, F. Ogel, and B. Folliot. YNVM: dynamic compilation in support of software evolution. In *Engineering Complex Object Oriented System for Evolution Workshop (OOPSLA)*, Tampa Bay, Florida, USA, October 2001.

[13] M. Roman, F. Kon, and R. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, July 2001. Special Issue on Reflective Middleware.

[14] M. Roman, D. Mickunas, F. Kon, and R. Campbell. Legorb and ubiquitous corba. In *Proceedings of the Workshop on Reflective Middleware (RM'2000)*, New York, April 2000.

[15] M. Roman, A. Singhai, D. Carvalho, C. Hess, and R. Campbell. Integrating pdas into distributed systems: 2k and palmorb. In *Proceedings of the 1st Symposium on Handheld and Ubiquitous Computing (HUC'99)*, Karlsruhe, Germany, September 1999.

[16] D. Saha and A. Mukherjee. Pervasive computing: A paradigm for the 21st century. *IEEE Computer*, pages 25–31, March 2003.

[17] M. Weiser. Some Computer Issues in Ubiquitous Computing. *Communications of the ACM*, 36(7):74–84, 1993.