On Reflexive and Dynamically Adaptable Environments for Distributed Computing

F. Ogel B. Folliot I. Piumarta

Regal group
INRIA Rocquencourt
Domaine de Voluceau,
78153 Le Chesnay, France

LIP6
Université Pierre et Marie Curie,
4, place Jussieu,
75252 Paris Cedex 05, France

Abstract

Current technologies for distributed application development, while being well-adapted for the client-server *model, do not adequately address emerging application domains, such as multimedia, embedded or mobile applications. The* black box *approach, used to achieve distribution and OS transparency, prevents application developers from adapting execution environments to the semantics and constraints of their application domain, thus leading to a growing number of* ad-hoc *solutions which are closed, static and poorly interoperable. As a solution, we present a reflexive and dynamically adaptable execution environment,*[1] *based on reification of hardware resources and a standalone dynamic code generator. This architecture allows dynamic construction of specialized OS or middleware components.*

1 Introduction

Nowadays solutions for distributed application development rely on the legacy of RPC and the use of middleware such as Corba or DCOM, to provide a set of high-level services and utilities. While being well-adapted to the traditional *client-server* model, they poorly support emerging application domains including, but not restricted to, multimedia, real-time or mobility. Middleware technologies have focused on distribution transparency and ease of use, providing application developers with high-level abstractions. Thus they tend to be *black boxes*, relying on a *one-size-fits-all* approach exporting well-defined, rigid, closed and static mechanisms; these mechanisms cannot accomodate constraints or semantics they were not designed for. For example, the replication and coherency strategies that a distributed application can use are restricted to the possibilities of the underlying ORB [BST99]. Emerging application domains, and their associated constraints and semantics have lead to *ad-hoc* solutions. These solutions are poorly interoperable and not even compatible with the technology they are based on. For example, both *Minimal Corba* and *Real-Time Corba* provide fixed solutions for given application domains, but in a completely *ad-hoc* way.

Several projects, based on reflexive ORBs propose solutions to increase flexibility, via dynamic loading of components and use of design patterns. Because their solutions are based on inflexible execution environments, they are obviously still limited: adaptation of system strategies and services is bound to the possibilities of the underlying systems, and adaptation of language aspects (such as component model, development paradigm, aspect composition, etc.) is limited by the reflexivity of the ORB's implementation.

In this paper we present a platform for dynamic construction of reflective and dynamically-adaptable ORBs. It relies on an execution environment (including both system and language aspects) that is minimal, completely open, reflexive and dynamically adaptable while remaining interoperable with existing code. This single execution engine, with a minimal common language substrate, *enforces* interoperability between environments and applications running on top of it. In the next section we describe the architecture of our execution environment. Some performance measurements are presented in section 3 and related projects in section 4. Section 5 draws some conclusions and perspectives.

2 Architecture

To address the issues raised by emerging application domains, in terms of flexibility and dynamism, we propose a novel execution environment, completely

[1]This work is partially founded by RNRT PHENIX and IST COACH projects.

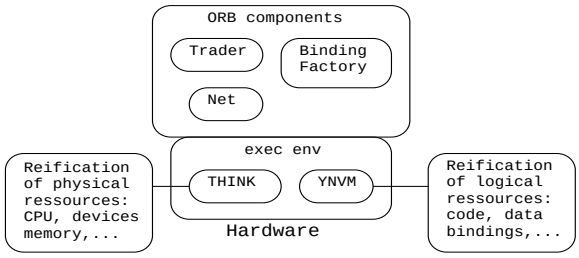

Figure 1: Architecture of the execution environment.

open, reflexive and flexible. Middleware built on top of it inherit these properties and thus are able to avoid limitations of current solutions. Applications can have complete control over their execution environment, and adapt it to their needs and semantics, from low-level mechanisms to high-level services or language features, such as component model or programing paradigms. Applications which do not have specific needs or constraints can obviously use standard high-level components and thus avoid the cost of re-implementing those services with low-level mechanisms.

2.1 Overview

Adaptation can take place at different levels in the execution environments and is therefore limited by the flexibility inherent in each of those levels. Most reflexive middleware is thus limited by the rigidity of the underlying OS. To allow complete dynamic adaptation of an application's execution environment, it should be possible to introspect at every level and change anything, according to the security policies if any. It seems obvious that not only the middleware layer has to be reflexive, but the whole execution environment: from the application down to the physical resources.

Our architecture, as illustrated in Figure 1, relies on a completely open, reflexive and dynamically-adaptable underlying execution environment, providing both system and language support without imposing any predefined model or abstraction. This environment permits the creation of reflexive and dynamically adaptable ORBs. Applications can therefore dynamically tailor an ORB to their needs, constraints and semantics, avoiding the cost of genericity without sacrificing portability and interoperability with existing platforms.

Since there's stronger and stronger coupling between language and system aspects, our execution environment aims at providing support for dynamic adaptation at both system and language levels. Pushing the Exokernel [Exo95] philosophy to the limit, it consists, for the system part, only of a bootstrap *Hardware Abstraction Layer* (*HAL*). This *HAL*, called *THINK*[2] [FS01], developed at France Telecom Research Lab, is responsible for reifying hardware components and therefore providing access to the *bare hardware*. Thus, no abstraction is (pre-)defined at this level. Just above *THINK*, comes a dynamic code generator, called *YNVM*[3]. The input language to the *YNVM* consists of syntax trees (often represented textually using a lisp-like prefix syntax). Our architecture is structured as a set of interfaces and components. The component model is based on the ODP Reference Model, but no composition model is enforced.[4] Each component exports one or more interface(s) that represent access points to it. Having different interfaces allows a component to separate its functionalities; for example, to represent different aspects of its semantics or functionalities, different security levels or QoS guarantees.

The communication channels (or bindings) between components are reified, through *binding-Factories*, to provide *"flexible bindings"*. Thus the semantic and the implementation of the interaction between two components can be dynamically adapted, for example to provide remote invocation after a migration or group remote invocation for dynamically-replicated components.

2.2 A Hardware Abstraction Layer

Whereas traditional kernels define *logical abstractions* like virtual memory or filesystems, *THINK* is just a library of *hardware abstractions*. Its purpose is to boot and reify hardware resources and their functionalities, such as memory or devices, without adding any *logical abstraction*. It is therefore structured as a set of components representing the hardware. Each component exports one or more interface(s), corresponding to the hardware's functionalities, without adding any semantic to them. Thus, components provide access to physical resources, without any management or control: they are "policy neutral". By exposing the hardware to higher-level software, *THINK* permits the construction of a completely open environment and therefore does not introduce any limitations to flexibility. *THINK* represents the main hardware-dependent part of the architecture and runs on PowerPC architectures.

For example the interface of the network driver looks like :

[2]for *THINK Is Not a Kernel*

[3]for *YNVM is Not a Virtual Machine*

[4]Thanks to dynamic flexibility, a contract-based approach can be dynamically added to the loading mechanism to ensure safe execution of components exporting contacts or specifications.

```
interface net {
int start(netif protocol);
int stop();
int transmit(packet msg, int res, int size);
void set_promiscuous(boolean promises);
char[] get_mac();
}
```

2.3 A Reflexive Dynamic Compiler

The *YNVM* is a dynamic code generator that provides both a complete, reflexive language, and an execution environment. It has been developed in the context of the *VVM* project [Fol00] to allow the dynamic generation of domain-specific virtual machines.

The main objectives of this environment are: (i) to maximize the amount of reflective access and intercession, at the lowest possible software level, while preserving simplicity and efficiency; (ii) to use a common language substrate to support multiple language and programming paradigms. To achieve this, the *YNVM* provides four *basic* services: (i) *code generation*: a fast, platform- and language-independent dynamic compiler producing efficient native code that adheres (by default) to the local platform's *C ABI* (*Application Binary Interface*); (ii) *meta-data* are kept between compilations, thus allowing higher-level software to reason about its implementation or that of the environment, and dynamically modify them; (iii) *introspection* on dynamically-compiled code, the application and the environment itself; (iv) *input methods*, giving access to the compilation and configuration processes at all levels.

The *YNVM* is structured as a set of interfaces and components, such as parser, tree optimizer/compiler, native instruction generator, etc. By combining components implementing those interfaces, applications in different execution format, from native ELF binaries to any bytecoded languages, can be loaded and executed. Moreover, having a single "root" execution environment allows complete interoperability and data sharing at both the execution environments and applications level.

The execution model is similar to *C* and the dynamically-compiled code has the same performance as a statically compiled and optimized *C* program.

2.4 A Dynamic and Reflexive ORB

On top of this "root" execution environment, we were able to build a dynamic reflexive ORB. This ORB relies on the underlying component model and a basic "flexible binding" model. A flexible binding can be seen as a communication channel between two components. Reifying the interactions between components allows the dynamic construction of specialized communication channels. Since anything built on top of the "root" execution environment inherits its reflexivity and dynamism, it is possible to add, remove or reconfigure any service in the ORB. Moreover, thanks to the complete reflexivity of the underlying environment, even the definition of the component model can be adapted at any time. Whereas flexibility is traditionally limited by rigid security rules, our architecture does not impose any protection or security. Rather it exports mechanisms such as MMU for memory isolation or the code loader for static and dynamic code verification, as in the Java VM. With such an environment, it becomes possible to tailor the middleware layer precisely to an application's needs.

For example, by reconfiguring the *binding factory*, communications between components can be dynamically adapted to execution conditions. An application can switch to an efficient lower-level protocol when using a reliable LAN connection, switch to a *mobility-aware* protocol when necessary, or even apply different security mechanisms to components based on their semantics. Thanks to the dynamic code generator, an application can *export* a component "on-the-fly"; that is, dynamically generate a stub/skeleton pair, possibly specialized for a given target environment or communication mode.

Adaptation is not limited to system aspect. For example, the basic component model can be dynamically enhanced by the addition of aggregation and composition operations (this can be used to enforce a contact-based component model). We also defined a basic *Aspect Oriented Programming* (*AOP*) extension, which can be used to dynamically wrap or weave different aspects into components. Moreover, because of the inherent reflexivity and flexibility of the global environment, the aspect weaver component itself can be enhanced with aspects or adapted. For example, adding a "history" aspect allow some "undo" operations on aspects wrapping and changing the weaver code allows adaptation of the wrapping strategy, and thus semantic: interposition, or inlining with dynamic recompilation. Since the underlying environment provides us with a common language substrate, we can preserve the basic properties of a middleware layer, portability and interoperability, while gaining dynamic flexibility and reflexivity.

2.5 Adaptation scenarios

Consider a host with a component A and two, or more, other hosts. When a remote component requires a binding to the component A, we can use the reflexivity of the environment to dynamically produce a

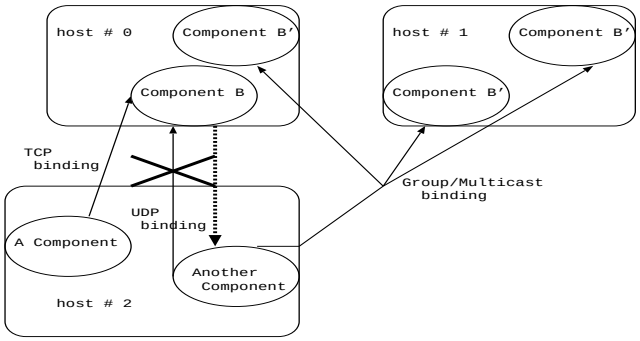

Figure 2: Reconfiguration of bindings.

binding (in this case a proxy) for component A.

However, because this is done "on-the-fly", the generated proxy does not have to pay the price of genericity and thus we can exploit all the information we might have, such as available quality of the connection or trust level, to choose the network protocol or the level of encryption to use. If another host requires a binding it will be given another proxy definition, dedicated to the link between the two components. For example, a component on the same LAN will receive a proxy that uses the network driver interface directly without any encryption, whereas another component, located in an "untrusted" domain, will receive a TCP-based proxy with data encryption.

This extreme level of flexibility and dynamism lets us deal with any QoS or security constraints on a "per-binding" basis: each binding between components can be dedicated, enforcing any requirements or specification of the application domains or component semantics.

Moreover, because the underlying execution environment does not impose any predefined logical abstractions, there are no limits to what can be done in terms of adaptation, within the security constraints, if any, defined by the administrator of the host.

Another example is represented in Figure 2 : a component A is bound to another component B. Because distributed environments can change rapidly, for performance or fault tolerance reasons, it may happen that component B needs to be dynamically replicated, as a set of B' components in the figure, for example if it has too many "clients". Thanks to the reification of bindings and to the dynamism of the *YNVM* we can change "on-the-fly" the bindings between component B and its "clients", again on a "per-bindings" basis. Thus "group communication" bindings, dedicated to the properties of the replication strategy used for component B and the specificities of each client, can be installed.

Configuration	Agent migration
G3 290MHz, YNVM	0.527 ms
G3 366MHz, YNVM	0.176 ms
P3 550MHz, Java	163.0 ms

Figure 3: Performance of agent migration.

As opposed to many distributed environments, which rely on a static compiler to "export" objects or components, with our architecture bindings between components are dynamically defined and therefore any component can be exported as and when necessary. Instead of having a fixed partition between local components and exported components defined at compile time, this partition is (re)defined dynamically and thus can exploit any runtime informations, such as identity or location of "client" components or available resources, to grant a binding request.

3 Performance evaluations

In order to evaluate our architecture, we are interested in measuring both the impact of the reflection and dynamic flexibility, compared to traditional static approach, and the performances of the dynamic adaptation, from switching between two existing components to dynamically generating new code.

To evaluate the performance of the resulting environment for distributed computing we consider two experiments that, from our point of view, are representative of the performance bottleneck of any middleware.

The first measurement we made was the migration of a component, since it represents a critical operation for mobile agent-based applications. It implies the following steps: (i) serialization of the expression; (ii) transmission over the network; (iii) deserialization; (iv) evaluation of the agent; (iv) serialization of the result; (v) transmission of the result and (vi) deserialization of the result.

As shown in Figure 3, using a Power-PC G3 290MHz, over UDP on a LAN, with a BMAC network card as the evaluation host, total time was 527μs, whereas the ping latency was about 475μs with LinuxPPC and more than 600μs with MacOS. Using a Power-PC G3 366MHz with a GMAC network card brings the total cost to less than 200μs. The agent consists of a few lines of code collecting information about the state of the host. In comparison, migra-

Configuration	Remote Invocation
G3 366MHz, YNVM	145 μs
P3 550MHz, ORBacus 4.0	1158 μs
P3 550MHz, TAO 1.1	1264 μs
P3 550MHz, Java	4551 μs

Figure 4: Performance of remote method invocation.

tion of a similar agent using Java[5] between 550MHz Pentium-III running Linux[6] required hundreds of milliseconds.

The second measurement was a classical remote method invocation, which is the basic service of any distributed computing environment. As opposed to traditional middleware, bound to the static RPC model, our environment allows us to test remote evaluation of dynamically-deployed code, like in the PLAN active network [HIC98].

Under the same conditions, as illustrated in figure 4, the total cost of a remote invocation was 145μs (on the G3 366MHz, which had a ping of 130μs), with serialization on both sides. In contrast, a remote method invocation, using different native Corba ORBs over Linux, range from 700μs to 1.5ms and several milliseconds with Java RMI.

This dynamic "remote evaluation" facility allows the fast dynamic deployment of components, as well as the construction of efficient distributed shared environment abstractions.

Another test consists of dynamically generating a binding factory for creating stub/skeleton pairs for components implementing some given interface. On the Power-PC G3 290MHz, with an interface comprising twelve methods, creation time is about 600μs, after which new stub/skeleton pairs, for any components implementing the interface, can be dynamically created in a couple of μs. That is, an application can decide to turn a simple local component into a server accessible from network in a few micro-seconds. A similar recompilation mechanism is used to define the aspect wrapper component mentioned earlier, on which it was possible to wrap aspects to adapt its wrapping strategy and semantics. The time needed to dynamically recompile a function containing several tens of lines of code is about a hundred of micro-seconds.

These first measurements show that (i) adaptation can be really efficient; (ii) dynamic reflexivity and flexibility do not prevent us from having good performances.

[5]jdk 1.1.8

[6]machines were connected with 100Mbits Ethernet.

4 Related Works

Our work can be compared to different kinds of projects, from extensible operating systems to reflexive middleware and MOP-based systems.

The *2K* system [KYH+01] is composed of a reflexive micro-kernel (*Off++*) and middleware (*Dynamic TAO*) that reifies dependencies among components. The objective is the management of these dependencies, but flexibility is still limited by the underlying micro-kernel and language aspects are not reified at all. One could say that we have tried to apply an *Exokernel approach* to this architecture, hence putting the middleware almost on bare hardware, and used language techniques in addition to allow complete flexibility.

Flexible operating systems, such as *Spin* [SP95] or *Exokernel* [Exo95], offer some flexibility, but only for the system aspects and therefore do not provide low-level mechanisms for reflection and interoperability. Moreover, projects based on *extension loading* such as *Spin* restrict the flexibility by imposing static, predefined security rules, where we argue that security is an application domain-specific concern: two different applications, even if they use common security mechanisms or algorithms, will not necessarily have the same requirements, especially considering the trade-off between performance and security level that has to be done.

Reflexive middleware such as *Open-ORB* [BC+98] and *Quality Objects* [LPS00], while offering a certain level of flexibility, are still limited: the framework itself can not be changed on-the-fly and reconfiguration possibilities are bound to those of the underlying OS. As an example, communication relies on the socket layer and nothing can be adapted beyond it (protocol parameters, buffer management, etc.).

To face the lack of support for real-time constraints in Corba middleware, [KSL99] proposes to enhance a Corba ORB (*TAO*) with a pluggable protocols framework to allow dynamic change of transport protocols. Once again, because the middleware relies on a rigid and closed operating system, adaptation is limited by the underlying execution environment's flexibility. Moreover, it is still an *ad-hoc* solution to a specific problem.

Microsoft *.Net* [MSN] is another project we could be compared to. Microsoft framework aims at responding to the needs of every possible user or application. Thus, it applies a "one-size-fits-all" approach which is known to (i) poorly face the evolution of application requirements and semantics; (ii) penalize performance; (iii) be closed, and thus to impose arti-

ficial constraints on developers.

In contrast, we want to give any user/application the ability to adapt the execution environment to their requirements and semantics, which results in (i) a better match with application needs; (ii) a more evolutionary solution, as each emerging application domain will not imply an update of the framework; (iii) better performance in the execution environment, as an application will never suffer from a "one-size-fits-all" service as often found in traditional operating systems.

5 Conclusions and perspectives

This paper presented the architecture of an execution environment for next-generation distributed computing environments. Our proposition follows a "less is better" philosophy and thus relies on two standalone sets of components representing the natural coupling between language and system aspects: the *THINK* exokernel, which reifies hardware resources in a "policy neutral" fashion (i.e., without any additional semantics) and the *YNVM* dynamic code generator, which is a fully-open, reflexive and flexible programming and execution environment.

Moreover, this basic execution environment does not impose any predefined *logical abstractions*, but it allows the dynamic construction of specialized higher-level execution environments including both system aspects, such as communications or scheduling, and language aspects, like programming paradigms.

The main contribution of this work is to propose a fully-open platform that reconciles reflection with performance. The resulting execution environments inherit the best of several worlds: flexibility, dynamicity, simplicity and performance.

We also argue that "ease of development", security and transparency should not justify rigid and closed execution environments. Because security, programming model or distribution aspects are part of application-domain semantics, an application must have the possibility to adapt them.

However we only provide a solution for bringing dynamic adaptability and performances together, therefore some support for expressing configuration and thus component composition is still needed. We believe an extensible configuration language following *Domain Specific Language* (*DSL*) principles can be used with a contract-based approach for component composition. It will ease the writing of adaptation scripts, while ensuring configuration/execution properties.

Dynamic construction and adaptation of a dedicated environment execution could be expressed in a configuration and deployment DSL and rely on a set of DSLs, one for each aspect of the system (communication, scheduling, security or language features).

We are currently in the process of porting our architecture to Linux, as a kernel module. We are also working on a standalone "Java-like" execution environment, defined on top of *THINK/YNVM*, in order to offer adaptable QoS for Java applications.

References

[BC+98] GS. Blair, G. Coulson, P. Robin and M. Papathomas, *An Architecture for Next Generation Middleware* in Proceedings of Middleware'98, September 1998.

[BST99] A. Bakker, M. van Steen, and A.S. Tanenbaum. *From Remote Objects to Physically Distributed Objects* in Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems, Cape Town, South Africa, December 1999, pp. 47-52.

[Exo95] F. Kaashoek, D. Engler, J. O'Toole, *Exokernel: an operating system architecture for application-level resource management* Proceedings of the 15th ACM SOSP, Copper Mountain, Colorado, December 1995.

[Fol00] B. Folliot, *The Virtual Virtual Machine Project*, Proceedings of IFIP Symposium on Computer Architecture and High Performance Computing, Sao Paulo, Brasil, October 2000.

[FS01] JP. Fassino and JB. Stephani, *THINK : un noyau d'infrastructure répartie adaptable*, Proceedings of CFSE 2, Paris, France, April 2001.

[MSN] see `http://www.microsoft.com/net/`.

[HIC98] M. Hicks and al. *PLAN: A Packet Language for Active Networks*, in Proceedings of the International Conference on Functional Programming, 1998.

[KSL99] F. Kuhns, DC. Schmidt and DL. Levine, *The Design and Performance of a Real-Time I/O Subsystem*, in Proceedings of the IEEE Real Time Technology and Applications Symposium, 1999, pp 154-163.

[KYH+01] F. Kon, T. Yamane, C. Hess, R. Campbell, MD. Mickunas, *Dynamic Resource Management and Automatic Configuration of Distributed Component Systems*, Proceedings of the 6th USENIX COOTS, San Antonio, Texas, January, 2001.

[LPS00] JP. Loyall, PP. Pal and RE. Schantz, *Building Adaptive and Agile Applications Using Intrusion Detection and Response*, in Proceedings of NDSS 2000, San Diego, CA, 2000.

[SP95] B. Bershad, S. Savage, P. Pardyack, E. Gun Sirer, D. Becker, M. Fiuczynski, C. Chambers and S. Eggers, *Extensibility, Safety and Performance in the SPIN Operating System* Proceedings of the 15th ACM SOSP, Copper Mountain, Colorado, December 1995.