Rapid Development Methodology for Customized Middleware

Thomas VERGNAUD, Jérôme HUGUES, Laurent PAUTET
GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
thomas.vergnaud@enst.fr, jerome.hugues@enst.fr, laurent.pautet@enst.fr

Fabrice KORDON
Université Pierre & Marie Curie, Laboratoire d'Informatique de Paris 6/SRC
4, place Jussieu, F-75252 Paris CEDEX 05, France
fabrice.kordon@lip6.fr

Abstract

Developing middleware for distributed application is a difficult challenge. Such software should be verifiable in order to help ensure its reliability; it also has to be configurable so that it can be tailored to the specific requirements of the target system. So there is a strong need for methodologies to manage numerous versions of such software. In this paper, we show the interest of architecture description languages (ADL) as a support for a development based on a prototyping process. We present our approach, which combines the Architecture Analysis & Design Language (AADL) and the schizophrenic middleware architecture, and show how those technologies can be used to design and configure verified middleware.

1 Introduction

Design and implementation of distributed applications rely more and more on middleware. As the middleware becomes a key component in such applications, there is a strong requirement to improve both performance and reliability.

The best way to achieve these performance and reliability objectives is to build a specifically designed middleware for each application. A general purpose middleware would drag numerous components that are not needed to perform the specific functions of a given application. Conversely, a dedicated middleware would only embed the useful mechanisms and components. However, it is impossible for cost and maintenance reasons to maintain one middleware per application.

Some projects such as Jonathan [1], TAO [3], or Quarterware [12] propose a partial solution to the problem by offering an approach to build a tailored middleware. These solutions rely on a well designed architecture that can be mapped onto several implementations. However, the architecture is built on top of complex design patterns, making it difficult to achieve reliability requirements and to detect useless components at a fine-grain level.

Figure 1 illustrates the new issues for middleware engineering. Once the middleware architecture is defined, engineers provide an implementation of the original middleware (step 1 in the figure). However, implementing the distribution mechanisms requires a very deep expertise in middleware engineering. Once this step is manually completed, there may be some glitches between the proposed architecture and the resulting implementation.

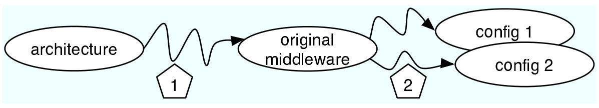

Figure 1. Steps to build middleware.

Then, to address the application requirements, a configured middleware may be derived from the original middleware (step 2 in the figure). Configurations may be maintained at the source level but components of the middleware are then selected using implementation criteria. This is not easy when application designers (and not the middleware designers) have to select a configuration. Moreover, there may be conflicts between configurations that are difficult to handle in the main source repository. So, when the middleware is tailored, the risk in having several source repositories reflecting different strategies is high. Reconciliation of these repositories is a very complex task.

This contradiction between the need to unify and the need to tailor middleware is often called the *middleware crisis*. This problem is a new key issue in middleware engineering since it prevents the use of middleware in application domains having strong implementation constraints such as memory footprints or performance execution.

The MDA (Model Driven Architecture) approach introduced by OMG [8] clearly suggests that most of the design should be performed at the model level. For middleware, we think that proposed solutions lack a notation to capture the architecture. Moreover, the use of a central model to describe the architecture requires the definition of a methodology. The objective is to reduce uncertainty generated by steps 1 and 2 in Figure 1 by using code generators.

Our paper presents a proposal for an integrated methodology to build domain specific middleware using a prototyping approach. The main idea is to replace traditional design and implementation techniques by an assisted one producing a specific middleware configuration from a set of data (model, components, implementation policies, etc.). We aim at easing the production of a specific configuration (tailored middleware) dedicated to the target application and embedding required services only. Thus, test and verification of the middleware only concerns appropriate code.

This paper is structured as follows. Section 2 shows the overall prototyping methodology. Then we present in section 3 the *schizophrenic architecture* that is a good candidate to support our methodology. The use of an architecture description language (ADL) is necessary; we selected AADL [10], presented in section 4. Section 5 is dedicated to the way automatic code generation is achieved to support our methodology.

2 Prototyping approach

The prototyping approach consists of a step-by-step process that allows periodic validatation of the project. Early problem identification during this process provides substantial cost savings.

2.1 Applying prototyping to middleware design

Our work aims at designing *a tailored middleware* for a specific system. A classical approach is to rely on a general architecture. This architecture is then adapted to the specific requirements of the system.

Middleware is often developed separately and verified by performing tests on the final version. However, performing verification on a middleware configuration is very difficult, since it is very versatile and has many execution cases.

In our approach, we use a prototyping methodology relying on code generation to help in the management of many variations (configurations) of a middleware.

2.2 Principles of prototyping

Prototypes are developed to verify some properties of the real system. They can be developed all over the project, in order to validate some concepts or to study particular properties that are expected from the final system. Prototypes can focus on a particular aspect of the problem or provide all the functionalities of the final product. As development goes on, prototypes are getting closer to the final system.

Under the word “prototyping” lie two different notions, corresponding to two different uses of the prototypes [4].

The first approach is called “throw-away”. It consists of creating prototypes in order to validate a concept, prior to implementing the real system. The throw-away approach is rarely used during the actual development: throw-away prototypes are not reused, the only feedback is a refinement of requirements.

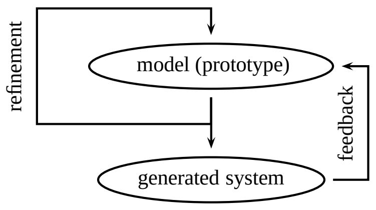

Figure 2. Evolutionary prototyping

The second approach is called “evolutionary”. Unlike the previous one, prototypes tend to become the final product. Thus, a prototype is a preview of the final system. Prototypes are refined to create more accurate ones. The last prototype actually corresponds to the final system.

Our methodology consists of starting from a general middleware architecture. This architecture is the first prototype of the final middleware implementation. We then perform evolutionary prototyping to adapt and specialize it. Prototypes are models; actual middleware implementations are generated from those models, as shown in figure 2. All the configuration elements should be placed in the model to ease system maintenance.

In order to achieve our development methodology, we have to first define a very versatile middleware architecture to potentially match a large panel of system requirements. This architecture must be formally described. All the configuration elements must be part of the description; this way we can use the model as a prototype and refine it. When the model fulfils all the required properties, we can generate an executable middleware implementation.

3 The schizophrenic architecture

Actual middleware has to fulfil the system requirements. Some solutions are based on “rigid” specifications; this is

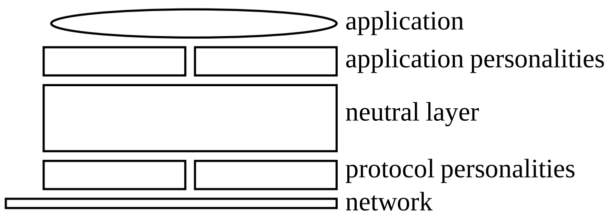

Figure 3. Outline of the schizophrenic architecture

the case for most CORBA implementations. Such middleware architectures are targeted to a certain application domain. They can be adapted to other fields of application (RT-CORBA, minimum CORBA...); yet, implementations are not as efficient as specifically designed middleware [5].

Thus, there is a need for middleware that can really fit many different systems. To do so, both a tailorable and verifiable architecture is required.

3.1 The need for a tailorable architecture

Solutions have been proposed to design tailorable middleware. *Configurable* middleware defines an architecture centered on a given distribution model [11] (e.g. distributed objects, message passing, etc.); this architecture can be tuned (tasking policy, etc.). *Generic* middleware [1] provides a canonical architecture, which has to be instantiated to create middleware implementations. Those implementations are called *personalities*. Generic middleware is not bound to a particular middleware model; however, various personalities seldom share a large amount of code.

Configurable and generic middleware architectures address the tailorability issue, as they ease middleware adaptation. However, they do not provide complete solutions, as they are either restricted to a class of distribution model, or their adaptations are too expensive.

3.2 Decoupling middleware functionalities

As an attempt to address those issues, we proposed the schizophrenic middleware architecture [14]. It separates concerns between distribution model APIs, communication protocols, and their implementations. Schizophrenic middleware refines the definition and role of personalities.

The schizophrenic architecture consists of three layers (as represented in figure 3): *application-level* and *protocol-level* personalities around a *neutral* core. The user application relies on the application personalities; the protocol personalities operate with the network.

Application personalities constitute the adaptation layer between application components and middleware through a dedicated API or code generator. They provide APIs to interface application components with the core middleware; they interact with the core layer in order to allow the exchange of requests between entities.

On the client side, they map requests made by client components from their personality-specific representation to a personality-independent one. Then they invoke the neutral layer, passing the neutral request. Results are translated back from neutral to personality-specific form.

On the server side, they receive requests for local objects from the core middleware, assign them to actual application components for evaluation, and return results.

Application personalities can instantiate middleware implementations such as CORBA, the Distributed System Annex of Ada 95 (DSA), the Java Message Service (JMS), etc.

Protocol personalities handle the mapping of personality-neutral requests (representing interactions between application entities) onto messages exchanged using a chosen communication network and protocol. Protocol personalities can instantiate middleware protocols such as IIOP (for CORBA), SOAP (for Web Services), etc.

The neutral core layer acts as an adaptation layer between application and protocol personalities. It manages execution resources and provides the necessary abstractions to transparently pass requests between protocol and application personalities in a neutral way. It is completely independent from both application and protocol personalities. This enables the selection of any combination of application and/or protocol personalities.

Several personalities can be collocated and cooperate in a given middleware instance. This is why we call it "schizophrenic" middleware.

3.3 The middleware core architecture

The middleware core provides neutral, model-independent functionalities. It requires a flexible implementation and the identification of the functionalities involved in request processing to ease the prototyping of new personalities and their interactions.

Figure 3.3 illustrates the architecture of the neutral layer.

The inner heart of the neutral layer is embodied by a component named "μBroker" [2]. The μBroker provides all the basic middleware mechanisms: I/O, task scheduling, etc. It is formally described, to support verification facilities. Hence it is possible to ensure its properties (no deadlock, no livelock, etc.).

The personalities do not directly interact with the μBroker. They are built on top of seven fundamental services that provide the client-server interactions found in most distribution models. Those services define the canonical operations performed in a middleware implementation.

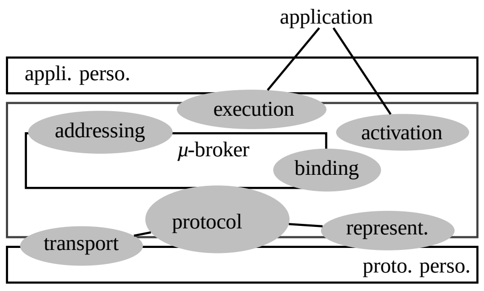

Figure 4. The core architecture

For example, upon the reception of a request on a server node, the *activation* service ensures the server entity exists in the application. The *execution* service is then invoked to actually process the request.

The composition of these fundamental services around the μBroker allows for the implementation of different distribution models.

3.4 Assessment

The seven fundamental services provide the basic middleware operations. The μBroker can be formally described to provide verification facilities, to ensure real-time properties. Those elements can be configured to create a middleware instance for particular system requirements.

The schizophrenic architecture is versatile enough to instantiate middleware supporting different distribution models. From an architectural point of view, the neutral core layer remains unchanged from a configuration to another; even if its behavior and properties may change.

4 Modeling the middleware architecture using the AADL

The schizophrenic architecture provides a clear structure to create tailorable middleware. A notation can be used to describe a schizophrenic middleware instance, it order to rapidly configure and deploy a tailored middleware that meets the application requirements. Architecture description languages [7] provide facilities for such description.

A few ADLs explicitly deal with real-time systems. Examples are ROOM [9] and AADL [6]. An AADL model can incorporate non-architectural elements: embedded real-time characteristics of the components (execution time, memory footprint…), behavioral descriptions, etc. Hence it is possible to use AADL as a backbone to describe all the aspects of a system.

4.1 Overview of AADL

“AADL” stands for Architecture Analysis & Design Language. It can be expressed using graphical and textual syntaxes; XML and UML representations are also defined.

AADL aims describe Distributed Real-Time Embedded (DRE) systems by assembling blocks separately developed. Thus it focuses on the definition of clear block interfaces [6], and separates the implementations from those interfaces. AADL allows for the description of both software and hardware parts of a system.

An AADL description is made of *components*. The AADL standard defines software components (data, threads, subprograms, processes…), execution platform components (memory, buses, processors…) and hybrid components (systems).

Components model well identified elements of the actual system. *Subprograms* model procedures such as those in C or Ada. *Threads* model the active part of an application (such as POSIX threads). *Processes* are memory spaces that contain the *threads*. *Processors* model microprocessors and a minimal operating system (mainly a scheduler). *Memories* model hard disks, RAMs, etc. *Buses* model all kinds of networks, wires, etc. Unlike other components, systems do not represent anything concrete; they actually create building blocks to help structure the description.

Component declarations have to be instantiated into subcomponents of other components in order to model an architecture. At the top-level, a system contains all the component instances.

Each component has an interface (called *component type*) that provides *features* (e.g. communication ports). Components communicate one with another by *connecting* their features.

To a given component type correspond zero or several implementations. Each of them describe the internals of the components: subcomponents, connections between those subcomponents, etc. An implementation of a thread or a subprogram can specify *call sequences* to other subprograms. This helps describe the whole execution flows in the architecture.

Most components can have subcomponents, so that an AADL description is hierarchical.

AADL defines a set of standard *properties* that can be attached to most elements (components, connections, features, etc.). Standard properties are used to specify things such as the clock frequency of a processor, the execution time of a thread, the bandwidth of a bus, etc. In addition, it is possible to add user-defined properties, to express specific description constraints.

By default, all elements of an AADL description are declared in a global namespace. To avoid possible name conflicts in the case of a large description, it is possible to gather components within *packages*.

A *package* can have a public part and a private part; only the elements of the package can have a visibility on the private part. *Packages* can contain *components* declarations. So, they can be used to structure the description from a logical point of view. Unlike systems, they do not impact the architecture.

4.2 Architectural description of the middleware components

Middleware is the lower part of an application; it can be viewed as a software component (or a set of software components) on which the user application rely. Given its modular structure, the schizophrenic architecture shall be modeled by a set of AADL software components.

4.2.1 Overall design

Middleware is a part of the application. Hence a middleware architecture shall be described using software components: a set of *subprograms* called by one or more *threads* (depending on the middleware configuration); *data* components model the data structures exchanged between the subprograms.

The subprograms should be organized so that they reflect the seven canonical services and the *μ*Broker of the schizophrenic architecture.

Subprograms cannot be subcomponents of a system, since they do not model “autonomous” components. Hence the schizophrenic architecture cannot be represented as a set of systems. Consequently, the description is to be organized as a collection of packages containing subprograms and data; the packages should reflect the logical organization of the architecture.

Basically, the model should then have seven packages containing the subprograms associated with the seven basic services; the components of the *μ*Broker, which constitutes the middleware “heart”, should also be materialized as a package. Finally, the different subprograms and data modeling the personalities should be placed into separate packages. Other “tools”, such as socket managers, could be placed into separate packages.

Each service can actually be modeled as a few main subprograms that are called from other parts of the architecture. Such subprograms shall be placed into the public sections of the packages, while more internal subprograms shall be placed into the private part.

4.2.2 Middleware configuration

The middleware configuration is either given by its architectural description, or by some properties associated to the components.

The personalities to use for a given configuration are materialized by the actual packages and components used to describe the architecture.

The actual number of threads to use is set by describing them in the architecture.

Some configuration elements such as the tasking policy actually deal with the behavioral description of the system, not its architecture; yet it is possible to specify them within the *μ*Broker, using user-defined properties.

The actual configuration of some services can be specified by providing a particular component implementation. For example, the activation service can either be a mere list associating references to procedures, or or more evolved mechanism with priorities, like CORBA’s POA. Those two possibilities correspond to two different implementations of the same subprogram type.

Since AADL does not allow dynamic configuration, the actual implementation of the components may depend on the middleware configuration.

5 Middleware generation

An AADL model can be a support to check the validity of a system: the execution time, the required memory, etc. can be specified for each component, using properties.

It is also possible to generate code from an AADL description. Hence we can create an executable system from its architectural description. The AADL model is then used to describe the components to integrate into the generated system.

In order to perform those tasks, an AADL model must provide a description with details enough regarding the properties of the system, and how it is to be implemented.

5.1 Evolutionary approach

AADL allows for the description of components at various levels of precision. One can either not specify the implementation of a component (provided that this is not required for a given processing purpose), or give very few architectural details of some parts of the description. On the contrary, one can precisely describe some parts of the architecture. It depends on what kind of process is intended for the model. Hence it facilitates an evolutionary approach in the model design.

We showed in 4.2.2 that some aspects of the architecture directly depend on the middleware configuration. For a first step, instead of giving all the architecture details, it would be better to describe some components from a higher level; for example without giving the implementation of

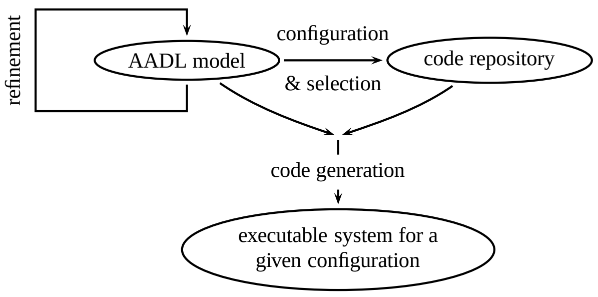

Figure 5. Middleware configuration & generation

some components. The exact architectural description can be described later, when the middleware outline is set.

A preliminary description of an architecture should be done by designing a collection of *subprograms* and the main *data* exchanged between the *subprograms*. Those components should be located in *packages*, in order to structure the description into main modules.

From this point, either we already have the architectural descriptions of some components we want to reuse, or not. Having pre-described components means that some parts of the systems are already fully described.

The overall methodology is summarized in figure 5. The AADL model for a middleware configuration is to be done step by step. The first step consists in assembling the components corresponding to the required personalities and services.

The model is iteratively refined: we can either describe the complete architecture of some subsystems, and then perform code generation or architecture validation in order to make unitary tests; we can also give a medium-level description of a part or the total of the global system, so that analysis can be done on the whole architecture.

The final model should contain all the required configuration elements to correctly configure the code templates. Then the source code is integrated into the structure generated from the AADL description.

5.2 Bindings with programming languages

In the end of the prototyping process, we obtain a model that fulfils all the system requirements. We must then generate an executable middleware implementation that corresponds to the AADL description.

AADL is an architectural language. It does not directly deal with the actual behavioral description of the components. AADL describes how the components interact with one another, and the data structures. The standard also defines execution templates for the threads. Generating code from an AADL description only consists of generating the "glue" between the components.

The standard defines a run-time to support the execution of the threads, subprograms, etc. It also defines rules to instantiate an AADL description using a programming language such as C or Ada [13].

Threads are handled by state automata provided by the AADL run-time. They define the thread execution cycle: initialization, dispatch, error states, etc. The AADL run-time also defines predeclared ports that are associated to standard actions (dispatch, job completion, etc.).

The source code describing the behavior of the components has to be provided as properties. Thus it can be integrated into the generated glue.

5.3 Achieving the middleware generation

Code generation requires a precise description: all component instances (i.e. subcomponents) must be designated with its implementation in the final model. In order to generate an executable system, we must have source code associated with all the components of the architecture.

To do so, a source code repository is to be associated with the AADL description. This repository shall contain two kinds of source codes. Static sources are for components implementations with no need for configuration. This is typically the case for protocol personalities: they always provide the same functions. Components, such as the μBroker, cannot be described by static source code, since their behavior depends on the middleware configuration. For such components, the repository must contain template source codes, that are configured according to the AADL model (e.g. number of threads that handle requests).

AADL properties are associated with every component implementation, specifying which part of the repository describes the component implementation. The source repository contains code for all available component implementations. The AADL description is likely to only deal with a subset of those implementations. For example, all personalities will not be included in a given configuration. Some services may have several possible implementations; only one will be used.

Building a middleware configuration manually may lead to including generic code where only a small portion is actually useful. From the AADL description, the generation process can include only the required sources from the repository. This facilitates the testing process since no dead code should be embedded in the corresponding configuration.

This methodology allows for a strict separation between the architectural design and the source code implementations. Thus it helps the maintenance: modifications in the configuration are made to the model; the generation process then propagates them to the executable system.

It is also possible to implement the configuration using different languages: we just have to change the code repository. The code repository could actually contain behavioral descriptions of the components using formal methods (e.g. Petri Nets), provided that we can generate source code from those descriptions. This would facilitate the verification process.

6 Conclusion

There is no "one size fits all" middleware. To achieve efficiency, middleware must match the specific requirements of a given system (memory size, tasking policy, etc.). For cost reasons, developing new middleware for each application is impossible. Our paper proposes a computer assisted methodology to elaborate customized middleware from a common basis: PolyORB, a schizophrenic middleware. Our development process ensures adequateness between the middleware implementation and the system requirements.

We presented the schizophrenic architecture, which synthesizes middleware adaptation techniques into a clear and modular architecture: personalities are built upon a neutral layer providing basic middleware operations. The middleware adaptation is performed through configuration of the neutral layer and personalities implementation.

We then presented AADL as a tool to describe the architectural of a complete system. AADL can serve as a backbone to aggregate the different aspects of a model (architectural issues, behavioral descriptions, execution times, etc.). AADL allows for the description of an architecture with various levels of details. A model can be described from a very high point of view; the components can then be described with more accuracy. This helps to apply an evolutionary prototyping approach.

In order to generate a middleware implementation, the AADL model must be associated with a code repository. This repository contains the source codes corresponding to the middleware components implementations. The code generated from AADL is a glue in which programs selected from the repository are included. The actual middleware generation consists of two operations: 1) the selection of the appropriate source code from the repository; 2) its integration of into the glue generated from the AADL description.

This provides a clear separation between the middleware architecture and its actual code. It helps include only the necessary components. It facilitates the middleware analysis and the use of different programming languages, since this is only related to the source repository; the structure remains unchanged.

We aim to apply, in the future, this development procedure to formal specification instead of code, as a extension of the work presented in [2]. This should help in introducing formal verification for middleware.

References

[1] B. Dumant, F. Horn, F. D. Tran, and J.-B. Stefani. Jonathan: an open distributed processing environment in java. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer-Verlag, 1998.

[2] J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Baarir, and T. Vergnaud. On the Formal Verification of Middleware Behavioral Properties. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, september 2004.

[3] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhes, and R. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *Proceedings of the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing (Middleware2000).*, april 2000.

[4] F. Kordon and Luqi. An introduction to rapid system prototyping. In *transaction on software engineering*, volume 28. IEEE, september 2002.

[5] F. Kordon and L. Pautet. Towards next generation middleware? *Distributed Systems online*, february 2005. available at `http://dsonline.computer.org/portal/site/dsonline/`.

[6] B. Lewis. architecture based model driven software and system development for real-time embedded systems, 2003. available at `http://la.sei.cmu.edu/aadlinfosite/LinkedDocuments/`.

[7] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*. Springer-Verlag, 1997.

[8] OMG. Model Driven Architecture (MDA), Document number ormsc/2001-07-01. Technical report, OMG, 2001.

[9] B. Rumpe, M. Schoenmakers, A. Radermacher, and A. Schürr. UML + ROOM as a standard ADL? In *Proc. ICECCS'99 Fifth IEEE International Conference on Engineering of Complex Computer Systems*, 1999.

[10] SAE. Architecture Analysis & Design Language (AS5506). available at `http://www.sae.org`, september 2004.

[11] D. Schmidt, D. Levine, and S. Mungee. The design and performance of real-time object request brokers. *Computer Communications*, 21, april 1998.

[12] A. Singhai. *QuarterWare: A middleware toolkit of software RISC components*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[13] J. Tokar. *Annex D: Language compliance and application program interface*, september 2004. Part of the AADL standard, available from SAE.

[14] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, volume LNCS 3063. Springer Verlag, june 2004.