Application-Driven Customization of Message-Oriented Middleware for Consumer Devices

Vivien Quéma[1], Luc Bellissard[2], Philippe Laumay[1,3]

[1]INRIA Rhône-Alpes ; Sardes Project
[2]Scalagent Distributed Technologies
[3]SchlumbergerSema-CP8
655, Avenue de l'Europe
F-38334 Saint-Ismier Cedex, France
Vivien.Quema@inria.fr

Abstract

There is a growing trend in developing applications on distributed heterogeneous devices. Message-Oriented Middleware (MOM) technology provides many solutions allowing to hide the management of the distribution of services and computations to the developers. On the other hand the middleware configuration becomes more and more complex. We propose to ease the configuration process using the configuration capability of a component-based MOM (the ScalAgent MOM). Our solution uses the application description (into an extended Architecture Description Language) to determine the set of middleware components required and their attributes to ensure non-functional properties required by the application. The customization process is controled by a customization algorithm that tries to minimize some non-functional property management costs. Our performance measurements clearly show the customization advantages.

1 Introduction

A new generation of intelligent and communicating devices such as consumer appliances, personal digital assistants, industrial automatons,... is emerging today. These devices embed increasing processing power and are connected to the outside world using Internet protocols on top of various types of networking technologies. This technological evolution enables the development of added-value services involving the devices, thus creating an opportunity for new markets and business models. Typical examples are : remote monitoring and maintenance systems ; collection and transfer of usage data from devices in order to feed entreprises information systems (e.g. billing, CRM, decisional systems, etc.). The main technical issues raised by these emerging environments are : scalability as thousands of devices can be involved in a given application, thus raising the problem of heterogeneity management and control of large volumes of data to be exchanged ; extensibility as new devices are joining and leaving the global system dynamically ; openess as devices are supposed to communicate with existing (and upcoming) business applications; flexibility and configurability as the overall system is facing the difficult problem of supporting simultaneously a wide spectrum of devices (with heterogeneous resources and functions) and changing operational conditions (e.g. disconnected mode due to mobility or transient faults, reduced bandwidth on wireless networks, etc.).

A typical example is a centralized monitoring management system for a set of UPS (uninterrupted power supply) systems distributed on a large-scale basis (up to hundred UPS boxes and thousands of devices protected by the UPS systems). Devices and UPS systems are heterogeneous and are accessible by various telecommunication means (i.e. with variable resource capabilities). Finally it should be noted that software components on devices and UPSs (e.g. shutdown procedures) are also heterogeneous according to the type of device and the type of usage. The problem to be faced by the designer of the remote monitoring system can be stated as follows : how would it be possible to design a global coherent system with local implementations customized to meet the specific local requirements ? It should be noted that customization involves both the application software components on UPSs and devices as well as run-time (middleware) components.

Today, the use of asynchronous communications (MOM for Message-Oriented Middleware) is recognized as the only means to achieve the aforementioned scalability-extensibility-openness objectives. On the other hand flexibility and configurability are addressed through the use of component-based approaches. This paper describes how these two key technologies are exploited in the ScalAgent distributed infrastructure to provide a high level of customization and configuration for applications concerned by the management of consumer devices.

The ScalAgent infrastructure has been designed to provide adequate support for the construction, deployment and operation of large-scale component-based distributed applications. This infrastructure is particularly adapted (but not restricted) to the support of distributed applications concerned by the management of consumer devices or any kind of internet appliances. The ScalAgent infrastructure is composed of :

- The ScalAgent MOM (Message-Oriented Middleware) provides an asynchronous messaging service that guarantees messages delivery and causal ordering of messages. Respective advantages and drawbacks of synchronous (client-server paradigm) and asynchronous (message passing paradigm) communication models are well-known today. Our experience gained from many years of experimentation with distributed systems has led to the choice of the asynchronous model to better meet the reliability and scalability issues in an wide-area network environment. The ScalAgent MOM is 100% java, so that it can run on a wide spectrum of stations and servers including embedded appliances. A lightweight version, dedicated to the smart card, is also available.

  This MOM can be used in two ways, using complementary application programmatic interfaces. One is compliant to the JMS standard (Java Messaging Service). A more sophisticated API, based on the agent paradigm, is also provided to application designers. Agents are distributed Java objects that conform to an atomic execution model and that use the underlying MOM to achieve a reliable event/reaction communication model.

- ScalAgent provides a comprehensive set of tools for the various types of users involved in the application life-cycle (designers, programmers, integrators, administrator). These tools are centered around the ADL concept (Architecture Description Languages [1]) to describe an application as an assembling of interacting software components. ADL allow components, their interfaces and properties to be specified, as well as links between them. This overall description is used during the whole application life-cycle to provide a complete and consistent view of what is the distributed application. It is thus possible to exploit this formal representation to pilot the deployment process, to monitor the application components and to later-on reconfigure part of the application in order to adapt it to evolving user

requirements or to specific changes of the run-time environment. It should be noted that theses tools are generic enough to handle any type of component model such as ScalAgent agents, EJB components, and also emerging Corba component model (CCM). These tools are used through a friendly graphical user interface and rely on a set of administration services provided as an extension of the MOM. The tools currently available include :

- a Component Builder Tool, to specify the application architecture : basic components (agents and/or legacy software components), and the composition and cooperation rules between components.
- a Component Assembling Tool (or Configuration Tool), to define and customize a particular instance of an application : instantiation and configuration of a set of appropriate application components
- an Application Deployment tool to install the actual components into their target environment and to set up the links between them. At the end ot the deployment stage, the application is ready for execution

As defined, the ScalAgent infrastructure allows component-based applications to be constructed, deployed and operated on large-scale distributed systems involving consumer devices. Available ADL-based tools allow a high level of customization and configurability at the application level. However, the current implementation of the ScalAgent run-time support is frozen for a given distributed system. This means that properties such as agent and/or message persistency, causal ordering of messages, etc. are globally pre-defined in the middleware layer, and thus are running on all nodes of the distributed system. Because of limited resources of consumer devices it is not possible nor desirable that the middleware responsible for the execution and communication of components provides systematically the support of all the non-functional properties for all of the functional components.

The overall objective is to overcome this limitation in order to be able to adapt the middleware properties on a given node to the resources available on the local equipment and also to the application requirements.

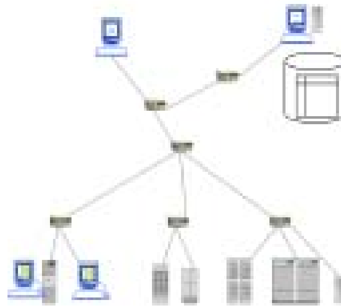

Figure 1: A monitoring application of a set of UPS systems.

To achieve this goal we modified the ScalAgent message oriented middleware [2] and the ScalAgent development environment based on an ADL [3] used to describe, configure and deploy distributed applications. The ADL has been extended to enable application developers to specify non-functional properties for both the components and their communication links (called connectors). On the other hand the ScalAgent MOM has been redesigned to integrate configurations capabilities internally (i.e. the ability to change a

system function). Finally an overall configuration system has been designed to allow the customization of the ScalAgent middleware so as to meet the application requirements specified with the ADL.

The paper is organized as follows. Section 2 is dedicated to the run-time environment, based on a MOM architecture. In a first step the main properties of the ScalAgent run-time environment are presented. Then the section describes how the componentization of the MOM architecture is achieved in order to allow MOM customization. Section 3 describes the extensions brought to the ADL (Architecture Description Language) in order to specify non-functional properties. Section 4 details the customization algorithm and section 5 gives some performance figures to illustrate the gain drawn from the customization process. Section 6 is a general conclusion.

2 Extending the ScalAgent platform

2.1 Application deployment process

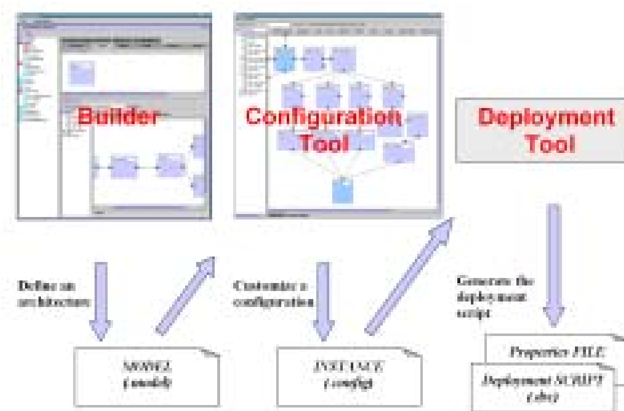

Figure 2: Application building process.

As depicted on figure 2, an application is built using the ScalAgent infrastructure in three phases. In the first step, the Component Builder is used to describe all the components used to build the application. This description called a *model* is stored in a XML format, following the Olan [4] syntax. In a second step, the Configuration tool is used to instantiate the model to build a custom application. This is achieved graphically by instantiating and assembling the components described in the model. The result of the configuration step is called a *configuration*. The builder and configuration tools are using an Architecture Description Language (ADL) formalism. Finally, the deployment tool is used to actually create the component execution structure, and to start and control the application. This tool allows a final level of application customization through the definition of environment variables.

The system is deployed on the ScalAgent run-time system which is said to be "homogeneous" since it has the same implementation[1] on all the nodes hosting application components. The middleware provides thus the same non-functional properties for all of the application components. The application developer has to modify the middleware by

[1]The word implementation refers here to a set of middleware components.

himself if he wants to provide only a subset of the non-functional properties to application components that really require them.

2.2 The ScalAgent MOM architecture

The ScalAgent run-time [5] is composed of a component-based message oriented middleware. Applications components are created and executed within a component server. A component server is a virtual machine which ensures components creation, execution, and communications. It embodies a local bus and a component factory. All the component servers are statically interconnected. In other words, the component servers supporting the execution of an application are all known before the application is started.

The local bus conveys the messages and makes the components react to messages. It handles the local communications internally (messages from and to components in the local component server). When a message is sent to a remote component, the local bus sends the message to the local bus of the remote component server. As depicted on figure 3, the local bus is composed of two main components :

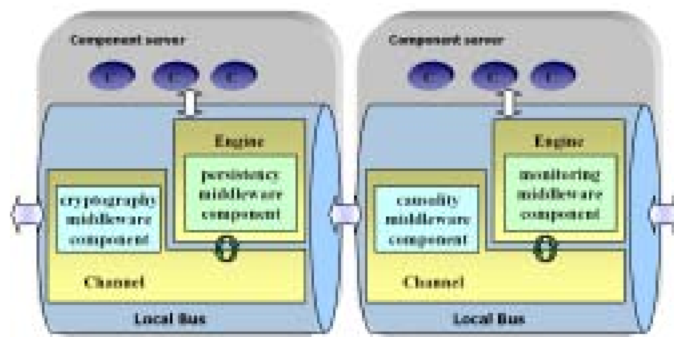

Figure 3: Component server architecture.

- The *channel* component that conveys the message reliably, and asynchronously. It uses a system of message queues. When a message cannot be sent to a remote node, the Channel suspects a failure in the network or in the remote node, so it periodically retries.
- The *engine* component that makes the components react to messages. The Engine is the main execution entity of the component server. It performs a set of instructions in a loop, getting the next message from the Channel, loading the proper component and making it react

These two entities allow the plugging of "middleware components". Each middleware component is responsible for a particular non-functional property. Two kinds of properties are distinguishable :

- properties related to application components : e.g. persistency — which consist in storing the state of some application components — and the monitoring of application components. Middleware components in charge of this kind of properties are located in the engine container.
- properties related to communication links : e.g. security — which consists in encrypting the messages exchanged between two application components — and the causal ordering of messages exchanged on a set of links. Middleware components responsible for this properties are interceptors plugged to the channel.

Each middleware component responsible for a non-functional properties owns a set of *attributes* required to provide the non functional property. The "persistency" middleware component has, for example, at least two attributes for each component it makes persistent : the first attribute gives information about the persistency technique (i.e. hard disk, floppy, cd,) while the second one specifies the file location on which data have to be stored (i.e. /save/data).

2.3 Toward an application-driven customization of middleware

As mentioned in section 2.1, the middleware deployed is homogeneous : all nodes host the same middleware components. For example, each node owns a middleware component that causally orders the messages it delivers. In practice each node maintains a matrix clock similar to [6], of size n^2, n being the number of nodes involved in the application. The problem resulting from such a building process is that the deployed middleware is often too "powerful". It enforces non functional properties for application components that do not necessarily require them. This is a major drawback for applications involving consumer, mobile or embedded devices. In fact such devices do not have enough internal resources to host the complete middleware. Our proposal is to use the configuration capability of the ScalAgent MOM resulting from its component-based architecture to generate a middleware that meets application requirements. In other words application components may have variable expectations with respect to non-functional properties. This is achieved by extending the Architecture Description Language specification capacity so as to enable the programmer to specify explicitly non-functional properties required by the application. Given this extended application description, a configuration algorithm is in charge of the middleware customization. The goal of the algorithm is to determine the set of middleware components required on each node to ensure properties required by the application. The new application building process is summarized on picture 4.

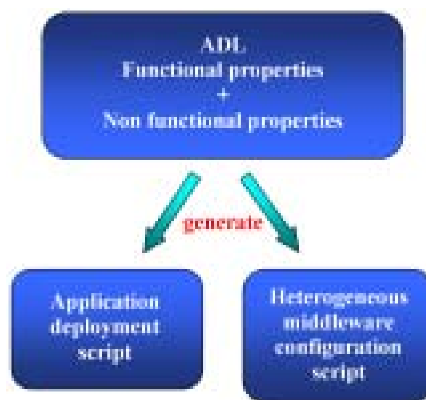

Figure 4: New Application building process.

The builder and the extended configuration tool help producing the application description which is processed both by the deployment tool and the configuration algorithm to deploy the application on a customized middleware. This middleware can be described as "heterogeneous" since its composition may differ from one node to another.

3 The extended Architecture Description Language

Architecture Description Languages have in common the objective of providing the structural view of a software system. ADL are based on the commonly accepted concepts of an architecture [1] :

- *components* that define computational units,
- *connectors* that define types of interactions between components,
- the *configuration* that defines an application structure in terms of the interconnection of components through connectors.

Existing ADLs differ on the exploitation of the application description. In fact, research work focuses on two distinct objectives :

- ADLs which provide assistance for the construction of the designed system. This is the case of ADLs like UniCon [7], Olan [3], Aster [8], C2 [9] that use the application description to automate the application deployment.
- ADLs which provide assistance for the analysis of the system. This is the case of ADLs like Rapide [10] or Wright [11] that allow application developers to specify the dynamic behavior of application components. These ADLs use the application description to model and analyse possible execution scenarios. They generate a model on which model checking techniques are applied to study, for example, liveness or safety properties.

The ADL used in the ScalAgent infrastructure is directly derived from Olan and thus belongs to the first category. To our knowledge, few ADLs of the first category permit the specification of non-functional properties associated with both the components and the connectors. The Aster ADL [8] has studied the specification of non-functional properties required by connectors using first order logic formulae. ACME [12], an interchange architecture description language which aims at enabling the combination of various ADLs, allows the application designer to specify non-functional properties. However, ACME does not interpret these definitions. They become useful only when a tool makes use of them for analysis, translation, and manipulation.

3.1 Specification of non-functional properties

We propose to extend the Olan syntax used by the ScalAgent tools to allow the application developer to specify non-functional properties required by components and connectors. Basically, Olan allows the programmer to describe these two kinds of entities using an XML syntax based on the Olan DTD [4]. Non-functional properties are expressed as sub-elements of a component or connector elements depending on the kind of property specified. The syntax used to specify a non-functional property is :

```
<property_name attribute_1="value" ...attribute_n="value"/>
```

To illustrate the use of non-functional properties, we consider below an example where the application developer wants to specify whether a component has to be persistent or not. As a matter of fact this example is related to the use of consumer devices that may not support persistent components, due to the lack of a flash file system. The syntax to be used to specify that an application component `C1` has to be persistent and that its state has to be stored on the file `save/C1` is the following :

```
<component name="C1">
        <persistency technique="fs" uri="save/C1"/>
</component>
```

The syntax to be used to specify that an application component C2 has to be persistent and that its state has to be stored in a LDAP data base is the following :

```
<component name="C1">
        <persistency technique="LDAP" uri="ldap://server/..."/>
</component>
```

Similarly, it is interesting to allow the developer to specify that messages exchanged between certain application components have to be encrypted. For example, to express that the communication link between the components C1 and C2 has to be secured, the syntax to be used is :

```
<connector componentIn="C1" componentOut="C2">
        <security protocol="SSL" rsa-key="X067..."/>
</connector>
```

3.2 Specification of the application components locations

The specification of application components location has also been extended. According to the Olan syntax, each application component location has to be specified by the application designer at configuration time. We allow the application designer to specify, for each application component, the (possibly empty) set of component servers that may host the application component. Note that specifying an empty set of component server means that the application component may be located on any component server. The next section presents how this extended application description — both in terms of application component location and non-functional properties — is used to determine the middleware configuration that best meets the application requirements.

4 The customization algorithm

4.1 Principles of the customization algorithm

The goal of the customization algorithm is to determine the location of application components and the set of middleware components required by each component server to fit with the application non-functional requirements. For each middleware component, the customization algorithm must also determine the attribute values. As we want the customized middleware to be as efficient as possible, assessments for the cost management of each non-functional property are required. By management cost, we mean the execution time overhead caused by non-functional property enforcement. Given all the non-functional properties management costs, the customization algorithm is able to determine the location of all application components that minimizes the overall cost.

4.2 Management cost evaluation

Each middleware component responsible for a non-functional property implements a function whose goal is to determine the corresponding property management cost. This function has the following signature : *int managementCost(deviceHWCharacteristics, numberOfComponents, numberOfRequiringComponents);*

The first parameter characterizes the device hosting the component server. Therefore the management costs differ from an embedded device to a workstation or a server. The second parameter is the number of application components hosted by the component server ; finally the third parameter is the number of application components hosted by the component server and requiring the non-functional property related to the given middleware component. The function returns an integer which evaluates the non-functional property management cost for the component server hosting the middleware component.

This function has to be implemented by each kind of middleware component as the cost depends on the non-functional property provided, and on the mechanisms used to enforce the property. As a matter of fact, providing the persistency property does not have the same cost as providing the security property. Moreover for a given non-functional property, several management policies can exist with different management costs. Let us take the example of the "persistency" property. Consider that a component server hosts five application components and let us assume that three of them require the persistency property. The middleware component responsible for this property has the choice between :

1. managing the different application components independently from each-other : i.e. it only makes persistent the three application components that require it,
2. managing the application components in the same way : i.e. if at least one of the hosted application components requires the persistency property, it makes persistent all the components it hosts. In our example, this means that it will make persistent the five application components.

When various management policies are supported by a middleware component, the choice between one of them is determined using management cost evaluation functions similar to the one described above. For example, the developer of the persistency middleware component must provide the following two functions :

- *independentManagementCost(deviceHWCharacteristics, numberOfComponents, numberOfPersistentComponents)* which evaluates the management cost for an independent management of the application components hosted by a component server,
- *commonManagementCost(deviceCharacteristics, numberOfComponents, numberOfPersistentComponents)* which evaluates the management cost for a common management of the application components hosted by a component server.

4.3 Choice of the application components locations

For each location, the algorithm is able to determine an overall customization cost based on the above described functions. The algorithm chooses the location that minimizes this cost. In the current implementation of the customization algorithm, the customization cost is simply computed by addition of the management costs of all the non-functional properties required on each component server. Future work will investigate other ways of computing this cost. One can imagine, for example, to give the application designer the way to specify non-functional properties that must be provided at lowest cost.

4.4 The Middleware description language

Similarly to ADL, the MDL (Middleware Description Language) is an XML-based formalism that is used to describe the structure of a run-time system in terms of its middleware components, their attributes and relationships between them. The customization

algorithm uses this formalism to generate a description of the customized version of the middleware, i.e. a description of the component servers. This description is used by the application deployment tool to install the right middleware configuration on a given node.

Let us take the example of an application component C1 which requires the persistency property. Figure 5 shows an example of what could be the description of the server hosting C1.

```
<!DOCTYPE "serverConfiguration.dtd">
    <serverConfiguration sid="1">
        <persistencyConfiguration management="independent">
            <component name="C1" technique="fs" file="c:/save/C1"/>
        </persistencyConfiguration>
        other non functional property
    </serverConfiguration>
```

Figure 5: Example of component server configuration file using the MDL.

This XML file indicates that the component server with sid 1 owns a middleware component responsible for persistency management. This middleware component must make the C1 component persistent as an independent management policy is specified and C1 has been specified as requiring the property. It also gives additional information about the support and file attributes required by the server 1 to handle the persistency property.

5 Performance evaluation

It should be reminded that the goal of the customization algorithm is twofold :

- to ease the development of distributed applications with variable non-functional requirements,
- to increase application performance by reducing the overhead involved in handling non-functional properties.

This section shows how the second goal is reached. [13] describes the implementation of a middleware component responsible for the causal ordering of messages delivered by application components. Measurements have been performed on applications involving various sets of component servers. The execution time has been compared for the same application running on the ScalAgent middleware in two scenarios : in the first case, the ScalAgent middleware is not customized. Messages exchanged in the system are causally ordered using a matrix clock based algorithm ; in the second case, the ScalAgent middleware is customized. Messages exchanged in the system are causally ordered using a middleware component automatically configured by the customization algorithm.

5.1 Protocol Description

The simplest performance indicator for the ScalAgent MOM is the turn-around time of a message between component servers, which is the sum of two terms : the first one is related to the transfer itself, the second one is caused by the management of causal ordering. The first term is nearly constant under our experimental conditions. Therefore the results are a good indicator of the efficiency of the causal ordering algorithm. Two causal ordering algorithms are compared. The first one does not use the customization algorithm. It consists in a matrix clock based management of causal ordering. Details on the protocol are given in [6]. The second causal ordering algorithm needs a pre-processing phase executed by the customization algorithm. This phase consists in determining the

application servers whose message exchanges require causal ordering due to the architecture of the application.

For the experiments, we have created a component on each component server, which sends back received messages (ping-pong). Messages are sent by a main component on component server 0, which computes the round-trip average time for 100 sends. We set up a network of thirty hosts in order to increase the number of servers. We used PC Pentium III 700MHz with 256 Mo, connected by a 100Mbit/s Ethernet adapter, running Linux kernel 2.4.6. We only present here the main results (complete results are available in [13]).

5.2 Experiments and results

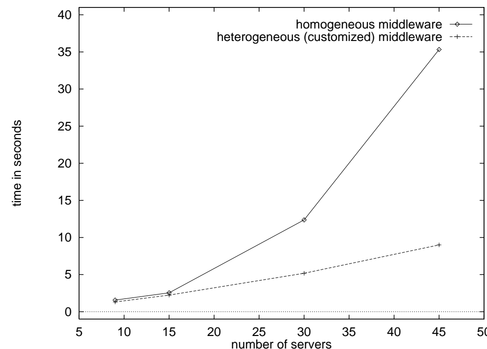

Figure 6: Performances for a application with slightly-connected components.

Figure 6 clearly shows the performance gain brought by the use of a customization algorithm. The measurements made without customization algorithm clearly show the quadratic increase of the message ordering cost with the number of servers. This quadratic increase is the direct consequence of the quadratic increase of the matrix size.

All these results slightly depend on our peculiar MOM test bed but nevertheless we believe that they can be considered as sufficiently general for MOMs using a MatrixClock-based causal ordering.

6 Conclusion

Today application designers are facing the problem of building complex distributed applications involving a large number of computer equipments ranging from traditional servers to mobile consumer devices. The design, deployment and operation of such applications remains a difficult task due to many factors such as : distribution, scalability, heterogeneity, reliability, security, adaptability and maintainability, etc. The ScalAgent infrastructure is aiming at providing a coherent set of solutions based on an asynchronous messaging middleware and on a component-based approach of distributed software development. The ADL approach adopted in the ScalAgent infrastructure provides a first level of customization at the application level that allows a set of components to be customized and assembled together to build a global application. Later on the application structure can be dynamically re-configured to integrate new types of equipment and to take into account evolving user requirements. However applications are more and more concerned with an increasing number of new embedded devices such as consumer appliances, personal devices, . . . which tend to be connected by any mean to any kind of networks. As a consequence, a given application should be able to accommodate a wide

range of equipments, with variable resource capabilities. A typical example is an energy supply management system where very simple automatons - with limited resources - have to cooperate with gateways - a PC with specific I/O interfaces - and with application servers. Several types of applications - remote monitoring, fault detection, billing, etc. - may also run concurrently on the same physical configuration. To face the lack of resources on some consumer devices and to meet variable application requirements, the middleware layer has also to be customizable so that it could be configured according to hardware limitations and application needs.

This paper has shown how the ScalAgent infrastructure has been extended to provide the support of heterogeneous devices and evolving application requirements. Basically this extension aims at extending the customization capability to the middleware layer in order to configure each node with appropriate middleware functions. This approach can be viewed as a joint development of the distributed application and its operational runtime support based on a common ADL philosophy. The ADL description of an application has been extended to allow the specification of non-functional requirements. From this overall description, a customized runtime execution platform is automatically built and deployed. This customization has been made available by adopting also the component-based approach for the design of the middleware internals. The customization process is controled by a customization algorithm that tries to minimize some non-functional property management costs. To achieve this goal, the customization algorithm determines the accurate execution node for application components.

Related work. Over the past decade, system customization has become an increasing area of interest. This is mainly due to the growing weight of non-functional properties in the overall system behavior. Many research project have thus addressed this issue. The Aster project [14] proposes a formal method for reasoning about matching of a customized software bus with some application requirements. The problem raised by such approaches is that formal specification of software components and application requirements remains a complex task. The Lasagne project [15, 16] proposes to compose aspect [17] at run-time to enable dynamic customization of systems. Composition is achieved by a run-time weaving mechanism that uses reflective techniques to perform a context-sensitive selection of aspects. According to Lasagne designers, the main drawback of this method is the run-time performance overhead.

Prior work had also been achieved in the programming language community in order to ease the construction of customized communication protocols [18]. However, to our knowledge, existing proposals do not address automatic configuration of middleware platforms from the specification of application non-functional requirements.

Implementation issues and perspectives. The management of non-functional properties and middleware customization raise complex implementation issues. Some of them are discussed here. One issue is related to the conjunction of different types of non-functional properties that may interfere. At this time, all the middleware components developed are in charge of compatible non-functional properties. Nevertheless, two non-functional properties relating to message ordering (e.g. causal ordering and messages priority on a link), for example, can be incompatible. We have not yet examined detailed consequences of such interferences upon our approach and this is an open issue for future work. Our premise is that the interdependency among non-functional properties can be made explicit through the definition of a relation over properties. Another issue is to study the configuration mechanism integration. Now the configuration is an ad hoc mechanism. It would be interesting to use a component model which integrate a configuration framework like the Fractal Composition Framework [19] which would ease the configuration and reconfiguration processes. Indeed, configurable middleware need to be reconfigurable to enable systems changes in the underlying systems infrastructure.

Finally, it would be of great interest to study the integration of more complex non-functional properties like real-time or QoS properties.

References

[1] V. Issarny, T. Saridakis, and A. Zarras. *A Survey of Architecture Description Languages.* C3DS Deliverable A3.1, ESPRIT LTR Project N24962, 1998.

[2] L. Bellissard, N. De Palma, A. Freyssinet, M. Herrmann, and S. Lacourte. *An Agent Platform for Reliable Asynchronous Distributed Programming.* Symposium on Reliable Distributed Systems, October 1999.

[3] L. Bellissard, S. Ben Atallah, F. Boyer, and M. Riveill. Component-Based Programming and Application Management with Olan. In *Proceedings of Workshop on Distributed Computing Systems*, pages 579–595, May 1996.

[4] L. Bellissard, N. De Palma, and D. Féliot. *The Olan Architecture Definition Language.* C3DS Technical Report, volume 24, 2000.

[5] L. Bellissard, N. De Palma, A. Freyssinet, M. Herrmann, and S. Lacourte. An Agent Plateform for Reliable Asynchronous Distributed Programming. In *Symposium on Reliable Distributed Systems (SRDS'99)*, Lausanne, Suisse, 1999.

[6] M. J. Fischer and A. Michael. Sacrifying Serializability to Attain High Availability of Data in an Unreliable Network. In *ACM Symposium on Principles of Database Systems*, pages 70–75, March 1982.

[7] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *Software Engineering*, 21(4):314–335, 1995.

[8] V. Issarny, C. Bidan, and T. Saridakis. Achieving Middleware Customization in a Configuration-Based Development Environment : Experience with the Aster Prototype. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 207–214, Annapolis, Maryland, USA, May 1998.

[9] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 44–53, 1999.

[10] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. In *IEEE Transactions on Software Engineering, Special Issue on Software Architecture, Vol. 21, No. 4*, pages 336–355, April 1995.

[11] R. Allen, D. Garlan, and R. Douence. Specifying Dynamism in Software Architectures. In *Proceedings of the Workshop on Foundations of Component-Based Software Engineering*, Zurich, Switzerland, September 1997.

[12] D. Garlan, R. T. Monroe, and D. Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[13] V. Quéma. Configuration d'un Middleware Dirigée par les Applications. Master's thesis, École Doctorale Mathématique et Informatique - Grenoble, June 2002. `http://sardes.inrialpes.fr/~quema/rapport_DEA.pdf`.

[14] V. Issarny, C. Kloukinas, and A. Zarras. Systematic Aid for Developing Middleware Architectures. In *Communications of the ACM, Issue on Adaptive Middleware, Volume 45, Issue 6*, pages 53–58, June 2002.

[15] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Jørgensen. Dynamic and Selective Combination of Extensions in Component-Based Applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, Toronto, Canada, May 2001.

[16] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Jørgensen. A Dynamic Customization Model for Distributed Component-Based Applications. In *International Workshop on Dynamic and Distributed Multiservice Architectures (DDMA'01)*, 2001.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, June 1997.

[18] M. Astley, D. C. Sturman, and G. A. Agha. Customizable Middleware for Modular Distributed Software. In *Communications of the ACM*, 1995.

[19] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal Composition Framework, *The ObjectWeb Consortium - Interface Specification*. `http://www.objectweb.org`, June, 2002.