Non-functional Oriented Dynamic Integration Of Distributed Components

Ahmed Saleh

Centre For Parallel Computing Computing, University of Westminster
London, UK

George R. R. Justo

Cap Gemini Ernst & Young
London, UK

Stephen C. Winter

Cavendish School of Computer Science, University of Westminster
London, UK

Abstract

Building distributed systems from scratch is known as one of the most difficult and time consuming process in software engineering. A great deal of progress has been made in the development of middleware and techniques to facilitate the process of distributed systems through integrating reusable components and customizing middleware. However, most of these solutions tend to focus on the functional aspects of the components and how to adapt their behavior according to their needs. Although the non-functional properties of the services are equally important, because of the difficulties in representing and controlling them, they are usually neglected. This paper investigates the problem of distributed systems’ non-functional properties and describes a framework that presents a solution to the problem. The framework is based on the idea of defining the non-functional properties of the services at the early stages of the design and incorporating such properties within the architecture to ease the process of managing the system at run time.

1 Email:saleha@wmin.ac.uk
2 Email:George.Justo@capgemini.co.uk
3 The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied of Cap Gemini Ernst and Young
4 Email:wintersc@wmin.ac.uk

©2003 Published by Elsevier Science B. V.
1 Introduction

Using traditional development techniques in developing distributed applications often result in static and difficult to understand applications that do not address the user requirements. Also, due to the evolving nature of the user requirements and the constant change of the environment, applications that can tolerate the continuous upgrade of such environment are often developed on a per application basis. Object-oriented paradigms have emerged as the new technology that can facilitate the development of distributed components through inheritance and object composition of pre-developed software components (objects).

Despite its wide recognition and many advantages, OO technology proved that it does not offer the ultimate solution for the development of reusable software artifacts. The reason being that it requires in-depth understanding of the classes’ internal structure, full consideration of all functionalities before overriding any method, and the excessive use of inheritance. As a consequence, the Component technology has appeared as the new effective tool to alleviate those problems.

The idea behind component technology is to make (software components) commodities that can be easily customized and assembled together to provide cheaper and quicker way of developing sophisticated software systems. Despite much progress in creating component-based frameworks and customization techniques, the problem of addressing the non-functional properties of distributed applications is far from being solved.

Research has been carried out in improving component-based middlewares in order to support QoS sensitive applications [12] and in some cases even using dynamic resource management [9] and customization [1]. However, this transparent approach is not suitable for most applications, which explicitly require non-functional properties (NFPs) to be met. Secondly, most non-functional properties do not correspond to QoS. Thirdly, transparent support makes validation quite difficult and with the widespread use of Web services [10] and E-business technology, architects and designers will certainly require an explicit definition of the NFPs that their components and architectures should satisfy. The reason being the platform or middleware on which they run may not be able to provide the required support for managing NFPs and the application architecture may have to do the checking whether the properties are satisfied.

This paper describes a framework to support the definition and control of NFPs of distributed component architectures. The essence of the proposed approach is to demonstrate how the existing concepts of components services and architectures can be extended to support the NFPs description and control at run-time.
2 The Framework

In this section, we describe a Java-based framework to support the management of NFPs of distributed applications. The framework is made of three main elements: An Interface Definition Language (NIDL) to define components interfaces, an Architecture Description Language to specify applications architecture and configuration, and a run-time management system to monitor and control the non-functional properties.

2.1 A Non-Functional Interface Definition Language (NIDL)

NIDL is an extended interface definition language for defining components interfaces and the non-functional properties of their services. NIDL is based on java, where components are defined in terms of the services they provide/require, together with the NFPs supported by each service or expected to be provided by invoked services. A service corresponds to a method in Java. Each service can support a set of NFPs. The NIDL compiler generates both Java and XML files. The interface repository stores XML information about the interfaces to be used by both the ADL compiler and the runtime system. NIDL currently supports three types of non-functional attributes:

- Performance: The performance is defined in terms of the average time to perform a service, and it is measured in Kbytes per second.
- Reliability: The reliability is measured in terms of the MTTF (mean time between failures), and it is calculated in milliseconds.
- Availability: The availability is measured in terms of the average time to restore (MTTR-mean time to restore) a service after a failure. It is a function of MTTF and MTTR, which is calculated in milliseconds.

The above words are regarded as keywords in NIDL. NIDL also provides the concept of NFP expressions that are Boolean and conditional expressions combining non-functional attribute keywords and their values. For example, a service is required to provide a 'performance = 500 kb/sec and reliability > 5000 msec’. A detailed example will be presented in Section 3.

To reduce overheads, a component is not required to compute all non-functional attributes, when they are not related to any NFP, but only those critical ones. In this case, NIDL contains a ‘support’ clause that indicates which non-functional attributes are computed by the component. Methods that handle the implementation and control of NFPs are generic and can be inherited by components of any type. Remember that the interface corresponds to a contract, therefore if a component supports a non-functional attribute, as described in more details later, the environment and an ADL script can query the value of that non-functional attribute at runtime.
2.2 The Non-Functional ADL

The non-functional architecture description language is referred to as NADL. Most constructors of NADL are similar to those of an ADL supporting planned reconfiguration such as C++CL [8]. It enables the instantiation and connection of components and connectors, and similarly it provides constructors to reverse a configuration operation, such as disconnect or interrupt a component instance.

A key property of NADL is that in order to support service-based architectures, component instances offering a service may run longer than a particular application. This means that existing component instances can be shared by different configurations. The architect may decide whether to use a fresh instance of a service or an existing service.

As language that supports the description of re-configurable distributed system according to both their functional and non-functional properties, NADL provides special constructs to deal with NFP description and management. An NADL description is made of two main sections: a configuration section where components are selected according to their services and their NFPs, and a re-configuration section where reconfiguration actions are taken, depending on the failure or changes of NFPs.

NADL also allows the architect to define environment specific properties that a component supporting a service must satisfy. For example, a component must run on a certain type of operating system or a machine with a certain minimum amount of memory. These properties enable the architect to refine the selection to identify components that are more specific.

The key constructor of NADL is the concept of NFP expressions that are extensions of those used in NIDL. In NADL, NFP expressions may contain services from different components while in NIDL they refer to the non-functional attributes of a specific service. For example, the expression below defines that the service video provided by component comp1 should support availability above 5000 msec and at the same time, the sound provided by component comp2 should perform above 600 kb/sec:

\[
\begin{align*}
\text{comp1.video.availability} & > 5000 \text{ msec} \quad \text{and} \\
\text{comp2.sound.performance} & > 600 \text{ kb/sec}
\end{align*}
\]

The first section of the NADL description contains the selection criteria for the components. The components can be selected according to their interfaces, which already specify their NFPs. After the system identifies possible candidates components, the configuration can be defined. In general, the selection should be the minimum requirement of the system. During the configuration, it is then possible to define further constraints depending on the candidate components that have been selected. In addition, the architect can specify global constraints relating the various components.

The configuration is built by using the typical ADL constructs such as connect, and start. Observe that NADL uses the concept of connector. A
connector is similar to a component but it is only used to plug components together. NADL also uses the concept of default connectors, which are implemented by the supporting middleware. For instance, in the case of Java components communicating using RMI (Remote Method Invocation), it is possible to connect the components directly by using an RMIConnector default connector.

After the configuration has been successfully built, the reconfiguration section specifies conditions for monitoring and managing the configuration. This is done using when clauses similar to those used during the configuration. The when clauses are evaluated sequentially and the first one that satisfies the corresponding reconfiguration block is triggered. During the reconfiguration, components and connectors can be connected or disconnected, and new components and connectors can be selected to satisfy the architecture NFPs.

NADL also provides the concept of (global) constraints, which define an NFP invariant for the architecture. The constraint is reevaluated after every reconfiguration. Observe that, since NADL is service-driven, reconfiguration is carried out at service level, which means that during reconfiguration the whole component is not affected but only those services involved. A detailed example is presented in Section 3.

2.3 The Management System

The overall architecture of the management system is shown in Figure 1. The architecture is a federated architecture of local manager components that function like containers for components and connectors.

Figure 1: The management system architecture
The key element for the monitoring and management of NFPs is the event system. The first level of event distribution happens at the local manager level, which is not shown in Figure 1 for simplicity. Components usually generate events to indicate failure of NFPs or change of non-functional attribute states. Observe that change of state is important because NADL can specify constraint expressions finer than those defined by the component NFPs. In order to avoid excessive numbers of events being generated, the local managers can filter some of these events. The events that are not filtered, such as NFP failures, are forwarded to the (global) event system.

Events are mainly consumed by the configuration manager, which is managing a particular configuration defined as an XML file in the NADL repository. The configuration manager interprets the NADL XML description and executes the necessary actions. All requests from the configuration manager are sent to the local manager, which then interacts with the global manager or other local managers. Requests for selection of components are usually processed by the global manager, which, like a CORBA trader [16], that keeps track of all components and their states. It can also apply some form of intelligence to select components. In order to do this, it receives events about NFP failures and how components are behaving. This information is used to compute statistics about components and services. For instance, which components are most popular, and those that have the least or the greatest NFP failures.

The current implementation of the management system is based on FRODICA (Framework for Distributed Configurable Applications), previously referred to as COREMEDIA [15]. The overall structured of the framework is presented in Figure 2. It is a typical multi-layered object-oriented architectural framework [8]. It offers a set of media components but it is not the aim of the framework to provide full support for distributed multimedia. It only supports media content messages but no distributed media synchronization and control.

Components and connectors implement the framework interfaces that enable the local managers to control their lifecycle. Most of the communication between components is based on Java RMI but the framework isolates the communication as much as possible.

Components and connectors register to their local manager, which can then interrupt and control their execution. Information about components and connectors, as well as about the local managers themselves, is also stored in the global manager, which functions like a global information repository or CORBA trader [16]. This simplifies the location of components within the systems.
3 A BANKING SYSTEM WITH NFPs

This section presents an example of a simple banking system, which illustrates the use of the proposed framework. The case study concentrates on the use of the IDL and ADL.

The banking system consists of a main server, which supports a set of branches. Each branch manages a set of ATMs but for simplicity, they are not modeled. The branches provide the same types of services such as (check balance, withdraw, print statement, etc.) but they are categorized according to the QoS they are supported to provide to the customers. A gold branch provides high QoS to premium customers, while a silver branch provides a lower QoS. The QoS is characterized by NF properties. In this example, they correspond to performance and availability. This means that gold branches should support higher performance and availability than silver branches. These properties are informally defined as follows:

GoldBranch: performance > 700 kb/sec and availability >= 7500 msec
SilverBranch: performance > 600 kb/sec and availability >= 6000 msec

Each one of the branches is represented by a single component connected to an ATM machine. Clients using ATMs select different services that invoke methods within branch components, consequently these methods will either be executed locally within the branch component of will invoke remote services.
based in the main bank component and forward their replies to the clients.

In order to achieve its goal in providing good service to premium customers, a reconfiguration strategy based on variation of NF properties has been defined. If the performance or availability of the main server degrades below a certain threshold, services of silver branches are suspended. When the performance and availability is above the defined threshold, the suspended services can resume. Observe that these properties could be implemented in the software component themselves but by placing them in the ADL, the reconfiguration strategy can be easily changed without modifying the components. This makes the components more reusable and the system more flexible. The reconfiguration thresholds are defined as follows:

MainServer : performance $\geq$ 500 kb/sec and availability $\geq$ 500 msec

3.1 The NIDL of the Banking System Components

Figure 3 shows a brief description of the NIDL specifications for the GoldBranch component, which have a group of provided and required services associated with their NFPs.

```java
interface GoldBranch {
   // provided services //
   provide float checkBalance (int customerID, int customer PID) ;
   support { performance $\geq$ 66 // supported NFPs
              availability } ;

   // required services //
   require float getBalance (int customerID, string customer name) ;
   with { performance $\geq$ 500 kb/sec $\&\&$ // required NFPs
           availability $\geq$ 5000 msec } ;
}
```

Figure 3: The NIDL specifications for the GoldBranch component

The NFPs of each service are defined by the keywords support and with. The properties defined by the word 'support' are supposed to be automatically calculated by the system (at run time), while the ones defined by 'with' must be explicitly defined by the system developer or the component prior to starting up. The Boolean expressions denoted by the '$\&\&$' and '$——$' signs provides additional flexibility to application programmers to express/merge the NFPs of each service.

The GoldBranch component implementing this interface has to incorporate the appropriate methods for computing the NFPs adopted by the interface. To illustrate the implementation of different components having the same functionality but with different NFPs, we can use the above NIDL to create a SilverBranch component interface with the relevant NFPs specifications.
3.2 The Architecture Description

The NADL specifications for the banking system Figure 4 represents the main sections of the NADL used to create this application. Generally, NADL files are parsed using a special kind of parser that scans its contents in order and detects any clashes in the specifications before execution. The first three main parts to be checked are the select, constraints and implementation sections, where the system has to search for the appropriate components that satisfies the conditions specified by these three sections.

```plaintext
Application : Bank {

select {

component: Comp1 { interface: MainBank ;
                location: remote [0x163000, MainBank] ;
                properties: { { getBalance.performance >= 500 KB/sec }||
                              getBalance.availability >= 5000 msec };
                updateBalance.performance >= 500 KB/sec };
        } ; // End Comp1 //

connector: Conn1 { interface: GoldConnector ;
                  properties: { dataStream.availability >= 8000 msec
                                dataStream.reliability > 7500 msec
                          } ; // End Properties //
          } ; // End Select //

constraints: { Comp1.averagePerformance := 400 KB/sec ;
               propertyCheckRate := 4000 msec; };
               // Rate of checking NFPs in msec

implementation: { Bank.Platform := java: //platform of the entire system//
                 Bank.OS := Unix ;}; //OS where the system is running //

configuration: { comp1: when (===letc) ;
                do { connect Comp1.getBalance To Conn1.dataStream;
                    connect Comp1.getBalance To Conn2.dataStream;
                    connect Comp2.checkBalance To Conn1.dataStream;
                    connect Comp2.withdrawCash To Conn2.dataStream ;
                } ; // End Comp1 //

reconfiguration: { when ( Comp1.getBalance.performance < 500 KB/sec ||
                     Comp1.getBalance.availability < 5000 msec ) ;
                do { (.........);
                        stop Comp3.checkBalance ;
                        stop Comp3.withdrawCash ;
                        (.........) ;
                } ; // End reconfiguration //

} ; // End Application //
```

Figure 4: The NADL specifications for the Banking System

The NFPs of each service is defined by the properties section, which concentrates on the properties of the application rather than the individual properties of each component (which is specified by the NIDL). In addition, the
Banking system components must obey the conditions specified by the general constraints and implementation sections.

After components are being selected, the configuration section will be executed whereby conditions preceding each sub-configuration (i.e. conf1, conf2, etc.) have to be validated before executing such a sub-configuration. Observe that the configuration section is scanned sequentially, where the first condition found to be true would cause its configuration to be executed, otherwise the system will continue searching until it finds a satisfied condition or a default configuration.

Once the configuration section is executed, the system will start running and the parser will move to the reconfiguration section, where new conditions will be continuously scanned at runtime. Unlike configuration, the reconfiguration section is scanned in parallel using multiple threads, where all conditions are validated simultaneously and their reconfiguration is executed once they are satisfied. For example, if the constraints of the system are violated during execution or the performance/availability of the MainBank getBalance service declined to 500 kb/sec or 5000 msec respectively, the application will either be stopped or the SilverBranch services will be suspended.

4 RELATED WORK

The functional properties define what features a software system is expected to have while the non-functional properties relate to the quality properties or abilities that specify how the functionality is exhibited or the constraint on how the systems operates [4]. Functional properties usually have localized effects, as they only affect the part of system addressed by the specific requirements. NFPs, however, have a global nature, which means that they affect several components. The problem with NFPs is that they are difficult to measure. As we are interested in monitoring and controlling NFPs during the system execution, we will focus on measurable run-time NFPs.

In order to define NFPs for applications, it is important to define which services will support such NFPs and their expected reactions towards the changes of their NFPs. One way of defining NFPs is through IDLs (Interface Definition Languages). Such an approach has been adopted by the Xelha language [3] that provides support for the specification of QoS management. This language is mainly concerned with the QoS properties provided by the service and the actions taken in order to attain the contracted level of service. However, it focuses on multimedia applications and QoS rather than the global properties of the application defined by its NFPs.

Quality Description Language (QDL) is another example of a language to specify QoS of multimedia applications [12]. QDL is a set of two languages (a contract language and a structure description language) used to describe the QoS contract and the structure of the application. It defines QoS regions and the actions to be taken when these QoS are not met.
There have been new developments in the standardization of interface definition to simplify integration and interoperability [11, 14]. XML and its extensions such as ebXML and WSDL [8] are good examples of this trend. However, they tend to focus on data formats and interoperability and ignore the implementation and QoS issues. Although WSDL is trying to avoid such limitations but it is still in its infancy.

We believe that the description of the NFPs of each service in the interface should be sufficient to enable the selection and use of the service. Although the definition of the actions to be taken if a service does not satisfy its NFPs is important, they should not be defined in the component interface, as it may reduce reusability. As described in the next section, decisions about failures of QoS or NFPs are application dependent, and by describing them in the interface, the component becomes attached to the application context. For example, an NFP should specify that a service should provide 100 transactions per second. If, however, it is specified that if the component fails to satisfy that requirement it should not support a certain type of resource consuming transaction, then the component becomes more specific and less reusable. The definition of NFPs of the application could be given to the middleware that could then manage them, as proposed in [9].

In order to allow the incorporation of NFPs within a middleware, NFPs have to be defined at the early stages of the design, in particular during the architecture description stage of the application. Since the architecture description is the most suitable place for describing the global properties of the application and its structure in terms of the components, their relationships and global constraints.

Many architecture description languages (ADLs) [13, 8] have been proposed to specifically support the definition of the architecture elements and their associated NFPs. For example, the Aster language [6] is a combination of an IDL and an MIL (Module Interconnection language) that describes the software architecture of an application together with its components’ NFPs. The Aster language is based on matching the NFPs of an application with the NFPs of selected components and connectors manipulated by the Aster framework [7]. This matching result in generating a customized middleware that provides the NFPs of the application. Although the Aster framework proved to be efficient in implementing several transactional and non-functional properties [17, 18], it does not cover all concepts of software architecture (e.g. connectors, ports, etc.) that describes application’s architecture, only components and some basic connectors are supported. In addition, it does not address the problem of managing NFPs during run time.

Frolund and Koistinen have extended UML (Unified Modeling Language)[2] to incorporate QoS issues in designing distributed systems. They created QML (QoS Modeling Language)[5] that relies on the concept of contract types, contracts and profiles in specifying QoS specifications. Although QML is a general-purpose QoS specification language, it does not allow more than one
QoS dimension to be specified in each contract type. In addition, it relies heavily on object-oriented techniques to facilitate the specification of QoS requirements at run-time as well as at design time.

Loyall, Bakken and Schantz’s research goes further by creating a framework called QuO (Quality Objects) [12] for developing distributed applications with QoS requirements. Their idea relies on the notion of contracts, delegates and system condition objects that negotiate an acceptable region of QoS prior to establishing a connection between a client and a server. When both client and server agree upon a specific region, the connection is established and the QoS level is monitored for further developments. Although QuO offers more flexibility than QML, it tends to concentrate on the structure of its components and their QoS without paying attention to the global structure of the application and its non-functional properties. In addition, QuO depends on the interface provided by the CORBA IDL, which restricts its portability in case if programmers want to use other platforms like DCOM or JAVA RMI.

5 CONCLUDING REMARKS

Service-based component architectures are becoming increasingly popular, especially with the appearance of the concept of Web services. Another important recent concept that is becoming popular is that of ASP (application service provider). These concepts are in fact the natural evolution of component technology but one of the most important challenges of the future component technology is still effective integration.

Standardization has been very important in reducing the problems of integration. However, most standards are still focusing on syntactical or protocol level integration. High-level integration is still evolving very slowly. QoS management has recently received a lot of attention from the middleware community. There is no doubt that the middleware plays an essential role in the integration and management of components, and their QoS. However, QoS and more importantly NFP management involves several layers of abstractions and cannot be seen simply from the middleware point of view, especially because, the management should occur at the application architecture level and not solely at the component level.

The paper has shown how it is possible to extend existing IDLs and ADLs to support the management of NFPs. It has also demonstrated how existing middleware platforms can be used to provide the necessary support for NFP runtime management. We have decided to build our own management service but there is no reason why the management system could not use services of a middleware such as [9], which supports QoS management. We see these two technologies as complementary rather than competing.
References

[1] Astley, M., D. C. Sturmman, and G. A. Agha, *Customisable middleware for modular distributed software*, CACM, Vol. 44, No. 5, (2001).

[2] Booch, G., I. Jacobson and J. Rumbaugh, ”Unified Modelling Language. Rational Software Corporation”, version 10, 1997.

[3] Duran-Limon, H., and G. Blair, *Specifying real-time behaviour in distributed software architectures*, In Proceedings of the 3rd Australasian Workshop on Software Systems Architectures. Sydney, Nov 2000.

[4] Filman, R. E., S. Barret, D. Lee and T. Linden, *Inserting Illities by Controlling Communications*, Communications of the ACM, August 1999.

[5] Frolund, S., and J. Koistinen, *Quality of Service Specification in Distributed Object Systems Design* In Proceedings of the 4th USENIX conference on Object-oriented Technologies and Systems (COOTS). New Mexico, April, 1998.

[6] Issarny, V., and C. Bidan, *Aster: A Framework for Sound Customization of Distributed Runtime Systems*, In Proceedings of the 16th International Conference on Distributed Computing Systems, Hong Kong, May 1996.

[7] Issarny, V., and C. Bidan, *Aster: A CORBA-Based Software Interconnection System Supporting Distributed System Customization*, In Proceedings of the 3rd International Conference on Configurable Distributed Systems (ICCDS’96). Mayland, USA, May 1996.

[8] Justo, G. R. R., and P. R. F. Cunha, *An architectural application framework for evolving distributed systems*, Journal of Systems Architecture: Special Issue on New Trends in Programming and Execution Model for Parallel Architectures, Heterogeneous Distributed Systems and Mobile Computing, H. Zima and B. Di Martino Eds. Vol. 45, No 12, (1999), pp. 1375-1384.

[9] Koh, F., and T. Yamane, *Dynamic resource management and automatic configuration of distributed component system*, In Proceedings of the Sixth USENIX COOTS, Jan 2001.

[10] Kreger, H., Web Services Conceptual Architecture (WSCA 1.0), IBM S/W Group, May 01.

[11] Linthiccum, D. S., ”B2B Application Integration”, Addison-Wesley, 2000.

[12] Loyall, J., D. Bakken, R. Schantz, J. Zinky, R. Vanegas and K. Anderson, *QoS Aspect languages and Their Run-time Integration*, Proceedings of the 4th Workshop on Languages, Compilers and Run-time Systems for Scalable computers (LCR), Pennsylvania, USA, 1998.

[13] Medvidovic, N., and Richard N. Taylor *A framework for classifying and comparing architecture description languages*, In Proceedings of the 6th European Software Engineering Conference. Sep 97.
[14] Olsen, G., An overview of B2B integration. EAI Journal, May 2000.

[15] Saleh, A., and G. R. Ribeiro Justo, A configuration-oriented framework for distributed multimedia applications, In Proceedings of the Fifteenth Symposium on Applied Computing (SAC200), Como, Italy, ACM Press, March 2000.

[16] Zahavi, R.,”Enterprise Application Integration with CORBA”, OMG Press, 2000.

[17] Zarras, A., and V. Issarny, Assessing Software Reliability at the Architectural Level, In Proceedings of the 4th International Software Architecture Workshop (ISAW4), Ireland, June, 2000.

[18] Zarras, A., and V. Issarny, Imposing Transactional Properties on Distributed Software Architecture, In the Proceedings of the 8th ACM SIGOPS European Workshop, Portugal, September 1998.