

Handling QoS in MDA: a discussion on availability and dynamic reconfiguration¹

João Paulo Almeida^a, Marten van Sinderen^a, Luís Ferreira Pires^a and Maarten Wegdam^{a, b}

^aCentre for Telematics and Information Technology, University of Twente
PO Box 217, 7500 AE, Enschede, The Netherlands

^bLucent Technologies, Bell Labs Advanced Technologies EMEA Twente
Capitool 5, 7521 PL, Enschede, The Netherlands

Abstract. In this paper, we discuss how Quality-of-Service (QoS) can be handled in the Model-Driven Architecture (MDA) approach. In order to illustrate our discussion, we consider the introduction of availability and dynamic reconfiguration QoS concepts at platform-independent level. We discuss the consequences of the introduction of these concepts in terms of the realizations of platform-independent models in different platforms. The platforms considered provide varying level of support for the QoS concepts introduced at the platform-independent level.

1 Introduction

There is a general agreement that distributed applications and services only achieve their desired impact if they properly cope with Quality-of-Service (QoS) issues such as performance and availability. In order to enable that, QoS issues should be addressed throughout a service's development life cycle. In this paper, we discuss how QoS can be handled in the Model-Driven Architecture (MDA) approach [9].

The concept of platform-independence plays a central role in MDA development. Platform-independence is a quality of a model that indicates the extent to which the model relies on characteristics of a particular platform. A consequence of the use of platform-independent models (PIMs) to specify a design is the ability to refine or implement a design on a number of target platforms. Platform-specific designs are specified through platform-specific models (PSMs).

For the purpose of this paper, we assume that services are ultimately realized in some specific object- or component-middleware technology, such as CORBA/CCM [10], .NET, and Web Services. Ideally one could strive for PIMs that are absolutely neutral with respect to all different classes of middleware technologies. However, at a certain point in the development trajectory, different sets of platform-independent modelling concepts may be used, each of which is needed only with respect to specific classes of target middleware platforms.

In this paper, we motivate the introduction of QoS concepts at the platform-independent level; we present *availability* as a QoS characteristic to be considered at the platform-independent level, as well as *dynamic reconfiguration* as a means to satisfy availability QoS constraints. We present some consequences of the introduction of these concepts in terms of the realizations of platform-independent models in different middleware platforms. For that, we consider platforms that provide varying levels of support for dynamic reconfiguration.

2 The Need for QoS Concepts for Platform-independent Design

Awareness of the qualitative aspects of a service starts in the initial phases of its design, when the service designer states the qualitative properties required from the service, e.g., that the service should support a certain level of availability and should perform according to certain temporal constraints. Since services should be specified at a level of abstraction at which the supporting infrastructure is not considered, service specifications are middleware-platform-independent by definition.

In order to illustrate our discussion, let us consider a groupware service that facilitates the interaction of users residing in different hosts. Initially, the service designer states QoS properties that are to be satisfied by the service. At subsequent stages of the design trajectory, the designer is

¹ This work is partly supported by the European Commission in context of the MODA-TEL IST project (<http://www.modatel.org>) and the Telematica Instituut in the context of the ArCo project (<http://arco.ctit.utwente.nl/>).

confronted with design decisions. In the design of the groupware service, we consider the following alternatives: (i) a centralized (server-based) design, and (ii) a distributed (peer-to-peer) design. Figure 1 depicts these two solutions. In solution (i), a server facilitates the interaction between users. In solution (ii), symmetric components facilitate the interaction without a centralized application-level component.

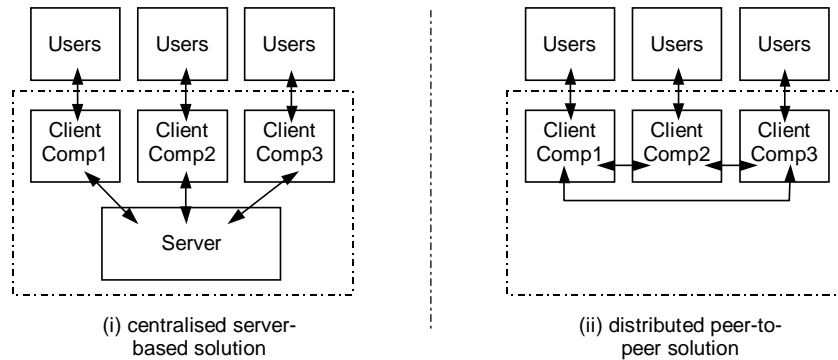


Figure 1 Alternative designs for the groupware service

Ideally, it should be possible to capture stable aspects of a system’s architecture in a platform-independent manner. Therefore, it would be desirable to select between alternative designs (i) and (ii) during platform-independent design. Nevertheless, platform-specific aspects such as the supported distribution transparencies (as defined in the Reference Model for Open Distributed Processing (RM-ODP) [6]) play an important role in the selection of an adequate architecture. For example, in case the platform provides support for replication transparency, solution (i) would not introduce a single point of failure in the architecture, and therefore would be acceptable as an alternative for the implementation of a highly available service.

Apparently, this places the designer in a dilemma, since platform selection would affect platform-independent design for the qualitative aspects. In order to solve this dilemma, QoS-aware MDA should allow the designer to express, at platform-independent level, (QoS) requirements on platform-specific realizations. These requirements should guide and justify design decisions at a platform-independent-level and provide input for platform selection.

3 Selection of Concepts for Platform-independent Design

QoS constraints may be satisfied by QoS mechanisms that may be implemented in the application and in target middleware platforms. Ideally, application developers should profit from the provision of distribution transparencies as a means to satisfy QoS constraints, shifting complexity from the application design to the platform.

This applies both at the platform-specific and platform-independent levels. At platform-independent level, these transparencies apply to what we call an *abstract platform*. The choice of abstract platform defines which (platform-independent) properties or aspects are actually considered and which (platform-specific) properties or aspects are abstracted from in a platform-independent design.

A platform-independent design relies on the (platform-independent) concepts provided by an abstract platform in an analogous way as a platform-specific design relies on platform concepts. In order to expose this relative notion of platform, we prefer the term abstract platform rather than the more general terms “meta-model” or “concept space”, as adopted in [4]. In order to define an abstract platform, one must carefully observe:

1. *Portability requirements for the platform-independent design.* The abstract platform should be generic enough to allow a mapping to different target platforms.
2. *The needs of application designers.* The abstract platform should provide concepts and facilities that ease platform-independent design, e.g., by providing required or desirable distribution transparencies at platform-independent level.
3. *The extent to which abstract platform and concrete target platforms differ.* The gap between abstract platform and concrete platforms has direct consequences for the mappings between platform-independent and platform-specific design.

Figure 2 illustrates the factors that influence the choice of abstract platform.

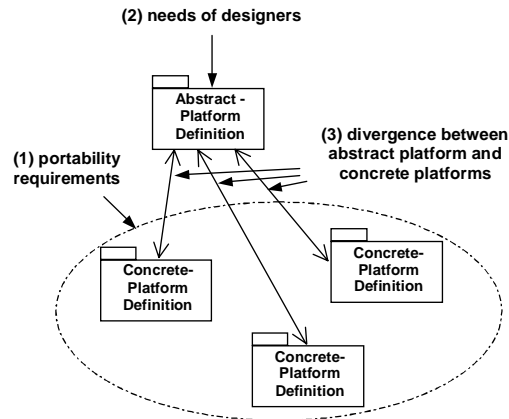


Figure 2 Forces in the choice of abstract platform

The forces exercised by factors (2) and (3) are often contradictory:

- Raising the provided support to observe the needs of designers may increase the gap between the abstract platform and concrete platforms. This is the case, for example, for the support of replication transparency [6] in the abstract platform, when a target platform has no support for the replication of components. By introducing replication transparency at platform-independent level, dealing with replication is deferred to platform-specific realization.
- Reducing the gap between support provided by the abstract platform and concrete platforms may lead to an abstract platform that handicaps the designer. This is the case, for example, for a “minimal” abstract platform that supports a common denominator of a broad class of middleware platforms such as point-to-point one-way message exchange. Patterns such as request/response and multicast message exchange, when necessary, are expected to be addressed by application developers in the platform-independent design of the application.

Differences in the architectural concepts used to build platform-independent designs and those concepts supported by the target platform may result in the use of intricate combinations of constructs in the platform-specific design. This may have an impact on the complexity of the mapping between platform-independent and platform-specific design and on some quality attributes of platform-specific design. It is questionable whether in case of really disparate abstract and concrete platforms, mappings are even feasible or can provide platform-specific designs with appropriate quality properties. For example, these mappings may sacrifice traceability from corresponding platform-independent designs, as well as intuitiveness for developers that are accustomed to a particular concrete platform.

Narrowing the gap between an abstract platform and concrete platforms is a challenging activity. Introducing new concrete platforms because of (unpredicted) changes in portability requirements may mean that the gap between the abstract platform and the newly introduced concrete platform is large. Besides that, narrowing the gap between an abstract platform and a particular concrete platform may enlarge the gap between the abstract platform and other concrete platforms.

4 Availability and Dynamic Reconfiguration

In the following, we consider availability as an example QoS characteristic, defined as the percentage of time that the system under consideration functions without disruptions (due to, e.g., faults or planned upgrades), the mean time between disruptions and the mean time to repair [14]. We also consider *dynamic configuration* as a means to satisfy availability QoS constraints. .

The aim of dynamic reconfiguration [3, 7, 8, 14] is to allow a system to evolve incrementally from one configuration to another at run-time. Dynamic reconfiguration exploits parallelism to improve a system’s overall availability. While certain activities of a system are affected during reconfiguration, other activities are left unaffected. Developing systems that can be dynamically reconfigured is a complex task, since a developer must ensure that dynamic reconfiguration results in a correct and useful system.

Reconfiguration is specified in terms of entities and operations on these entities. In this paper, we focus our attention on component replacement and migration. *Component replacement* allows one version of a component to be replaced by another version, while preserving component identity. We use the term version of a component to denote a set of implementation constructs that realizes the component. The new version of a component may have functional and QoS properties that differ from the old version. Nevertheless, the new version of the component should satisfy both the functional and QoS requirements of the environment in which the component is inserted. *Component migration* means that a component is moved from its current node to a destination node. Migrations of components can, e.g., be necessary when a certain node has to be taken offline.

A system evolves incrementally from its current configuration to a resulting configuration in a *reconfiguration step*. A reconfiguration step is perceived as an atomic action from the perspective of the application. We distinguish between simple and composite reconfiguration steps. A *simple reconfiguration step* consists of the execution of a reconfiguration operation that involves a single component. A *composite reconfiguration step* consists of the execution of reconfiguration operations involving several components. Composite steps are often required for reconfiguration of sets of related components. In a set of related components, a change to a component *A* may require changes to other components that depend on *A*'s behavior or other characteristics.

Support for dynamic reconfiguration can be clearly related to availability requirements when we consider disruptions (downtimes) due to upgrades of a system. A system without support for dynamic reconfiguration would typically be taken off-line a number of times during its lifetime for upgrades, causing downtimes. These downtimes could be avoided with dynamic reconfiguration.

4.1 Reconfiguration Transparency

Dynamic reconfiguration interferes with system activities and, therefore, requires special attention from the perspective of run-time reconfiguration management [7]. A system can become useless in case the preservation of consistency is ignored. The system under reconfiguration must be left in a "correct" state after reconfiguration. In order to support the notion of correctness of a distributed system, three aspects of correctness requirements are identified [8]. This notion of correctness is addressed in several dynamic reconfiguration approaches described in the literature (e.g., [7, 8, 14]).

From the perspective of application developers, platforms should ideally provide *reconfiguration transparency* [14]. The objective of reconfiguration transparency is to mask, from an application component and from its environment, the ability of a system to execute reconfiguration steps involving the component. Reconfiguration transparency hides from application developers the details and differences in mechanisms used to overcome the difficulties introduced by reconfiguration.

4.2 Dynamic Reconfiguration Concepts for Platform-independent Design

We introduce dynamic reconfiguration concepts in a platform-independent design by specializing the notion of a component to include the distinction between *reconfigurable* and *non-reconfigurable* components. Reconfigurable components can be *migrateable*, *replaceable* or both *migrateable* and *replaceable*. This allows a designer to establish these distinctions at a platform-independent level, specifying which components may be manipulated by reconfiguration operations in reconfiguration steps.

A (composite) reconfiguration step is specified by a set of simple reconfiguration steps. The definition of a replacement reconfiguration step identifies a component to be replaced and establishes its new version. The definition of a migration reconfiguration step identifies a component to be migrated and establishes its new location. Reconfiguration steps are committed to and handled by a reconfiguration manager component entailed by the abstract platform.

5 Platform-specific Realization

Platform-specific realization may be straightforward when the selected concrete platform corresponds (directly) to the abstract platform. When this is not the case, more effort has to be invested in platform-specific realization. In general, we distinguish two contrasting extreme approaches to proceeding with platform-specific realization:

1. *Adjust the concrete platform*, so that it corresponds directly to the abstract platform. This may imply the introduction of platform-specific (QoS) mechanisms, possibly defined in terms of internal components of the concrete platform. Since modifying a concrete platform is typically not feasible (e.g., re-implementing the CORBA ORB to match the abstract platform would be too expensive), extension of this platform in a non-intrusive manner is often the preferred way to adjust the concrete platform. In this approach, the boundary between abstract platform and platform-independent design is preserved in platform-specific design.
2. *Adjust the platform-specific design of the application*, to preserve requirements specified at platform-independent level. This may imply the introduction of (QoS) mechanisms in the platform-specific design of the application. This approach may be suitable in case it is impossible to adjust the concrete platform, e.g., due to the lack of extension mechanisms and/or the cost implications of these adjustments.

Approaches to realization that adjust both concrete platform and the platform-specific design of the application are positioned somewhere between these two extreme approaches.

Different middleware platforms provide varying levels of support for dynamic reconfiguration with varying levels of transparency. In order to discuss the consequences of the introduction of reconfiguration transparency at the platform-independent level for platform-specific realization, we consider the following platforms: CORBA enhanced by portable extensions with the Dynamic Reconfiguration Service (DRS) [1, 14]; and, CORBA with compliance to the Online Upgrades (Draft Adopted) Specification [11].

5.1 CORBA + DRS

The Dynamic Reconfiguration Service we have proposed in [1, 14] provides reconfiguration transparency for CORBA application objects, supporting both simple and composite reconfiguration steps. The DRS has been implemented by extending CORBA implementations through the use of portable interceptors, which are standardized extension mechanisms for CORBA ORB implementations [10]. In this realization, there is a direct correspondence between the DRS-enabled CORBA platform and the abstract platform, illustrating approach 1 to realization.

5.2 CORBA with compliance to the Online Upgrades

The Draft Adopted Specification for Online Upgrades [11] provides interfaces to manage the upgrading of the implementation of a single CORBA object instance. The specification allows these interfaces to be used to upgrade multiple CORBA object instances, but provides minimal mechanisms to coordinate the upgrading of multiple CORBA object instances.

This means that there is a compromise to be made with respect to support for composite reconfiguration steps. If composite reconfiguration steps are supported by the abstract platform, then realization on top of CORBA with online-upgrades is impaired. Nevertheless, a subset of the platform-independent designs can be mapped directly, namely the ones that do not use composite reconfiguration steps. Therefore, we could have considered an alternative abstract platform that provides support for simple reconfiguration steps only. This abstract platform would be defined as a “subset” of the platform that supports composite reconfiguration steps. This latter realization illustrates again approach 1.

The differences between the platforms could be reconciled by following approach 2, where transformations would introduce mechanisms at the application-level to coordinate the upgrading of multiple CORBA object instances. This would, however, sacrifice reconfiguration transparency at platform-specific level.

6 Conclusions

Since platform selection would affect platform-independent design for the qualitative aspects, QoS-aware MDA should allow designers to express, at platform-independent level, QoS requirements on platform-specific realizations. This is done through the definition of an abstract platform, which determines which (QoS) properties or aspects are actually considered and which are abstracted from in a platform-independent design.

Differently from the “traditional” discussion that couples levels of transparency to the support provided by middleware platforms, in MDA, platform-independent models are “decoupled” from their corresponding platform-specific counterparts by mappings. This adds a new dimension to the discussion on the level of support provided by a platform or located in the application. There is some degree of freedom between the selection of transparencies for the abstract platform and the provision of transparency for the concrete platform. In this paper, we identify three factors that should be observed when defining an abstract platform. We also discuss, in a general sense, the implications of choosing the level of transparency of the abstract platform for the mapping on the concrete target platforms.

As an example, we have considered availability as QoS characteristic, and dynamic reconfiguration as a means to satisfy availability QoS constraints. Dynamic reconfiguration concepts have been introduced in a platform-independent design and two contrasting approaches to platform-specific realization have been considered. These approaches are further illustrated using two target platforms (CORBA + DRS, CORBA + Online Upgrades) with different levels of reconfiguration transparency.

The work presented in this paper is related to the OMG-promoted work on MDA core technologies, such as UML and extensions, MOF, etc. These technologies include, more recently, QoS meta-modelling techniques and UML language extensions, such as in [5], QML and responses to OMG RFP on Modeling QoS and FT Characteristics and Mechanisms [12]. In this paper we have addressed the conceptual aspects, abstracting from language aspects. Platform-independent designs must be specified in suitable modelling languages, therefore these efforts are complementary to the conceptual discussion presented in this paper.

References

- [1] J. P. A. Almeida, M. Wegdam, M. van Sinderen, L. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA, *Proc. 3rd Intl. Symposium on Distributed Objects & Applications (DOA 2001)*, Rome, Italy, Sept. 2001.
- [2] J. P. Almeida, M. van Sinderen, L. Ferreira Pires, D. Quartel, “A systematic approach to platform-independent design based on the service concept”, *Proc. 7th Intl. Conf. on Enterprise Distributed Object Computing (EDOC 2003)*, Brisbane, Australia, Sept. 2003, to appear.
- [3] C. Bidan, V. Issarny, T. Saridakis, A. Zarras. A dynamic reconfiguration service for CORBA, *Proc. IEEE Intl. Conf. on Configurable Distributed Systems*, May 1998.
- [4] D. Exertier, O. Kath and B. Langois. PIMs Definition and Description to Model a Domain. MASTER IST-project D2.1, Dec. 2002.
- [5] A.T. van Halteren, *Towards an adaptable QoS aware middleware for distributed objects*, Ph.D. Thesis, University of Twente, Enschede, The Netherlands, 2003.
- [6] ITU-T / ISO, *Open Distributed Processing - Reference Model - Part 3: Architecture*, ITU-T X.903 | ISO/IEC 10746-3, Nov. 1995.
- [7] J. Kramer, J. Magee. Dynamic configuration for distributed systems. *IEEE Trans. on Software Engineering* 11(4), April 1985, pp. 424-436.
- [8] K. Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems*. Ph.D. thesis, Imperial College, London, UK, March 1999.
- [9] Object Management Group, *Model driven architecture (MDA)*, ormsc/01-07-01, July 2001.
- [10] Object Management Group, *Common Object Request Broker Architecture: Core Specification*, Version 3.0, formal/02-12-06, Dec. 2002.
- [11] Object Management Group, *Online Upgrades Draft Adopted Specification*, ptc/02-07-01, Jul. 2002.
- [12] Object Management Group. *UML Profile for Modeling QoS and FT Characteristics and Mechanisms RFP*, ad/02-01-07, February 2002.
- [13] L. A. Tewksbury, L. E. Moser, P. M. Melliar-Smith, Coordinating the Simultaneous Upgrade of Multiple CORBA Objects, *Proc. 3rd Intl. Symp. on Distributed Objects and Applications (DOA 2001)*, Rome, Italy, Sept., 2001.
- [14] M. Wegdam, *Dynamic Reconfiguration and Load Distribution in Component Middleware*, Ph.D. Thesis, University of Twente, The Netherlands, 2003.