Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA
in Ingegneria Elettronica, Informatica e delle Telecomunicazioni

Ciclo XXI

Settore scientifico disciplinare di afferenza: ING-INF/05

MIDDLEWARE PRINCIPLES AND DESIGN FOR THE INTEGRATION OF UBIQUITOUS MOBILE SERVICES

Presentata da: Stefano Monti

Coordinatore Dottorato
Chiar.ma Prof. Ing. Paola Mello

Relatori
Chiar.mo Prof. Ing. Maurelio Boari
Chiar.mo Prof. Ing. Antonio Corradi

Esame finale anno 2009

To my parents, Liviana and Cesare,
I thank them for having made me become as I am

To Silvia,
I thank her for standing me as I am

"But is not an event in fact more significant and noteworthy the greater the number of fortuities necessary to bring it about?"
The unbearable essence of being,
M. Kundera

Table of Contents

ABSTRACT 7

1 INTRODUCTION 8

2 UBIQUITOUS SERVICE PROVISIONING: BACKGROUND 11

2.1 Distributed architectures 12

2.1.1 Middleware 12

2.1.2 Service-oriented architecture 13

2.2 Ubiquitous computing scenarios 15

2.2.1 Mobility 15

2.2.2 Context-awareness 16

2.2.3 Multimodal multichannel multipattern user interaction 18

2.2.3.1 Multimodal access 18

2.2.3.2 Multichannel access 19

2.2.3.3 Multipattern access 20

3 MIDDLEWARE DESIGN 21

3.1 Requirements 21

3.2 Design principles 23

3.2.1 Delegation: a disappearing middleware approach 25

3.2.2 Decoupling: service-oriented computing 26

3.2.3 Layered semantics 27

3.3 Architecture 28

3.4 Related work 30

3.4.1 Web mashups 31

3.4.2 Semantic ubiquitous service composition 31

4 SERVICE COMPOSITION 33

4.1 Service composition principles 33

4.2 Composition model....37
4.2.1 Business logic layer....40
4.2.1.1 Services....40
4.2.1.2 Workflows....40
4.2.2 Semantics fusion layer....41
4.2.2.1 Semantic domains....41
4.2.2.2 Roles....43
4.2.2.3 Rules....44
4.2.3 User semantics layer....45
4.2.3.1 Service metadata....45
4.2.3.2 Templates....46
4.2.3.3 Interaction patterns....47
4.2.4 Usage scenario – user requirements....48
4.3 Composition reification....51
4.3.1 Evaluation facilities....51
4.3.1.1 Consistency....51
4.3.1.2 Scoring....52
4.3.1.3 Substitution....52
4.3.2 Service and template rules....52
4.3.3 Rule evaluation....55
4.3.4 Template reification....57
4.3.5 Syntactical consistency....59
4.3.6 Reification process principles....59
4.3.7 Workflow ranking....61
4.3.8 Usage scenario - template reification....61

5 KERNEL SUPPORT FEATURES....64

5.1 Service composition....64
5.2 Service support....65
5.2.1 Service lifecycle management....65

5.2.2 Operational parameter mapping.....*68*
5.2.2.1 Parameter mapping - responsibility.....68
5.2.2.2 Parameter mapping - choice policies.....69
5.2.2.3 Parameter mapping - example excerpt code.....70
5.3 Workflow management.....71
5.3.1 Workflow lifecycle management.....*71*
5.3.2 Workflow Execution.....*72*
5.4 User proxy.....73
5.5 Interaction management.....73
5.5.1 Interceptors.....*74*
5.5.2 Interaction Managers.....*75*
5.6 Session Manager.....75
5.7 Message broker.....77
5.8 Support features.....78

6 IMPLEMENTATION.....80
6.1 Service layer.....82
6.1.1 Operational and semantic interfaces.....*83*
6.1.2 Service invocation.....*84*
6.1.3 Service provider standpoint.....*86*
6.2 Kernel layer.....87

7 CASE STUDIES.....90
7.1 Push-based interaction: news-on-SMS.....92
7.2 Pull-based interaction: adaptedHTML.....96
7.3 Multi-output interaction.....99

8 DESIGNING MIDDLEWARE RECONFIGURABILITY.....105
8.1 Related work.....106
8.1.1 Reconfigurable systems.....*106*
8.1.2 Reconfiguring Ubiquitous middleware.....*107*

8.2 Design principles....108
8.2.1 Layered architecture....108
8.2.2 Delegating reconfiguration responsibility....109
8.2.3 Decoupling non-functional logic....110
8.3 Architecture....111
8.4 Reconfiguration details....112
8.4.1 Applicative layer reconfiguration....112
8.4.2 Non-functional layer reconfiguration....113
8.5 Implementation....115
8.6 Case Study....115

9 CONCLUSIONS....117

PUBLICATIONS....120

REFERENCES....122

ACKNOWLEDGMENTS....132

Abstract

Nowadays, in Ubiquitous computing scenarios users more and more require to exploit online contents and services by means of any device at hand, no matter their physical location, and by personalizing and tailoring content and service access to their own requirements. The coordinated provisioning of content tailored to user context and preferences, and the support for mobile multimodal and multichannel interactions are of paramount importance in providing users with a truly effective Ubiquitous support.

However, so far the intrinsic heterogeneity and the lack of an integrated approach led to several either too vertical, or practically unusable proposals, thus resulting in poor and non-versatile support platforms for Ubiquitous computing.

This work investigates and promotes design principles to help cope with these ever-changing and inherently dynamic scenarios. By following the outlined principles, we have designed and implemented a middleware support platform to support the provisioning of Ubiquitous mobile services and contents. To prove the viability of our approach, we have realized and stressed on top of our support platform a number of different, extremely complex and heterogeneous content and service provisioning scenarios.

The encouraging results obtained are pushing our research work further, in order to provide a dynamic platform that is able to not only dynamically support novel Ubiquitous applicative scenarios by tailoring extremely diverse services and contents to heterogeneous user needs, but is also able to reconfigure and adapt itself in order to provide a truly optimized and tailored support for Ubiquitous service provisioning.

Keywords: Ubiquitous computing, Mobility, Context-awareness, Middleware, Service-oriented computing, Service composition.

1 Introduction

Ubiquitous computing envisions a landscape where Information Technology so intimately permeates everyday user life that it becomes a commodity final users access almost unconsciously. From the Ubiquitous computing first proposal in early 90's, in effect, technology advances have been astonishing and have begun revolutionize people everyday life. Novel wired and wireless connectivity channels as well as increasingly sophisticated, heterogeneous and powerful user devices are making users more and more eager to access services and contents while moving, no matter the device they use, and according to their own preferences.

Nevertheless we can not say the Ubiquitous computing scenario has become really pervasive and ultimately available. Ironically enough, users are more than ready to it, but technology is still a step behind. So far, the major barrier toward Ubiquitous computing is the lack of integration: technologies (both for connectivity and for end user devices) have grown so rapidly that they have missed to evolve in a coordinate and integrated way.

As an example, users are becoming more and more skilled in handling mobile portable devices such as smartphones and palmtops that enable an almost always-on connection to the Internet (by means of, say, 3G mobile networks or WiFi connections). But, the contents on the Web hardly suit these mobile devices and ready to use devices are still to come. As another example, voice synthesis technologies have fostered tools such as screen readers that allow impaired users to have the plain text or web content read by their fixed PC browser. Imagine that while driving a car we can seamlessly exploit the same technology to receive phone calls on our mobile phone with a vocal reading of traffic news HTML portal content and newly arrived e-mails.

In our vision, to make Ubiquitous computing scenarios ultimately concrete and widespread, novel connectivity and computational technologies need to coordinate, cooperate, and integrate seamlessly, transparently and with no effort for final users. However, the intrinsic heterogeneity of Ubiquitous computing scenarios and the need for a user-transparent approach make the design of such integration platforms still extremely challenging.

Clear identification and separation of concerns related to integration is the key for providing an integration support platform able to deal with heterogeneity and to provide users with the right abstraction level, so as to result extremely seamless and easy to use. In fact, currently proposed integration platforms tend to lack the desired clean separation, and results are either too vertical and tightly bound to ad-hoc and specific applicative domains (i.e., by dealing with a limited set of communication channels and formats or with specific services and contents), or too generic and overly complex for final general knowledge users.

This work proposes to overcome the lack of technology integration by means of a middleware platform to support cooperation of very heterogeneous services and contents on the Internet, and to enable users to access them while moving, by means of any device and in any format. Our design aims at neatly identifying and separating concerns that emerge in the integration of Ubiquitous scenarios, thus

providing an extremely flexible yet user-friendly platform for the integration of Ubiquitous services and contents.

This dissertation is organized as follows. Section 2 provides background knowledge of middleware platforms and of the main issues in Ubiquitous computing. Section 3 delineates the design requirements and principles that animated our work and ends sketching out the overall architecture of our middleware platform. Section 4 describes the key architectural element in our proposal, namely the service composition model that grants our platform both flexibility and support for heterogeneity, and user transparency and ease of use. Section 5 describes support features our platform relies on to cope with some relevant ubiquity aspects. Section 6 provides an overview of the most notable implementation technologies we adopted in the realization of our platform. Section 7 describes some relevant case studies that our platform has successfully realized in several different deployment. Finally, section 8 reports some considerations related to the extension of our platform toward autonomous reconfiguration, in order for the platform to better and more efficiently cope with changes in Ubiquitous scenarios. Section 9 concludes the dissertation, by providing the most relevant remarks in our research experience in the field of Ubiquitous computing.

2 Ubiquitous service provisioning: background

In Ubiquitous computing scenarios, users require to access services and contents from anywhere, at anytime and with any device at hand. This forces service provisioning support platforms to address several challenging and debated research areas, such as mobility management, or multimodality and context-awareness support. Middleware-based approaches are emerging in order to face this issue, however current solutions only partially support Ubiquitous service provisioning and tend to focus only on specific research areas, thus providing vertical and ad-hoc support.

This section describes some relevant architectural approaches in designing large heterogeneous distributed applications, then deepens the description of state-of-the-art research in Ubiquitous Computing by also reviewing some preliminary and partial proposals in the field of Ubiquitous service support and provisioning.

2.1 Distributed architectures

In distributed systems, different pieces of business logic spread over different network nodes cooperate with the goal of realizing a certain application or business case. Heterogeneity in these scenarios is a key characteristic: network nodes can be extremely different in terms of hardware resources and of software support (e.g., operating systems, programming languages, ...). A number of different architectural approaches have emerged to help distributed applications cope with these heterogeneity issues; in the following sections we describe some of the most notable ones.

2.1.1 Middleware

Distributed applications typically reside on a number of different nodes, each with its on peculiarities (e.g., hardware, operating system, ...); in order to cooperate, they need to overcome this intrinsic heterogeneity. Middleware is emerging as an architectural approach to facilitate the interaction between applications distributed across heterogeneous network nodes [1].

Middleware offers an abstraction layer that relieves application designers of some of the burden of realizing distributed applications from scratch. In recent years, different kinds of middleware have emerged, each providing a different category of features and abstraction level. In the following we report some of the most notable ones.

RPC-based middleware [2] is one of the most basic forms of middleware and aims at providing programmers of distributed applications with an intuitive and powerful abstraction: pieces of distributed software that need to cooperate can invoke procedures of each other transparently from their physical locations, as if they were on the same network node. *TP Monitors* [3] enable the abstraction of distributed transactions: in a typical distributed interaction, in order to realize their business logic, pieces of distributed software typically need interact with physical

or logical resources either locally, or remotely. These interactions typically are strictly interconnected, so, for instance, the failure of one piece of business logic can invalidate the overall process. Distributed transactions provide a way to master the overall execution of a series of complex interdependent distributed tasks in a consistent way. *Object brokers* appeared as an evolution of RPC-based middleware to allow for the remote interaction and cooperation of distributed objects; in time, their specification has grown to encompass much more complex features than simple remote invocation of business logic, for instance by providing naming, discovery, or event management services, Quality of Service management and so on. The most notable class of Object Broker middleware is the *Common Object Request Broker Architecture (CORBA)* [4], promoted and standardized by the Object Management Group (OMG). *Object Monitors* [5] basically resulted from the convergence and fusion of Object brokers with TP monitors to extend remote object brokering with transaction support. *Message-oriented middleware* (MOM) has raised as a proposal to overcome the intrinsically synchronous remote interaction style proposed by RPC-based mechanisms and has promoted asynchronous messaging as a way to coordinate distributed pieces of business logic. Some of the most notable proposals in this area are IBM Websphere MQ [6], Microsoft's MSMQ [7], and the Java API standard Java Message Service [8].

2.1.2 Service-oriented architecture

Service Oriented Architecture (SOA) [9] is an architectural approach in building large heterogeneous distributed systems that leverages the abstraction of service to encapsulate and easily manage heterogeneous pieces of distributed business logic. Service clients (either end users or machines, e.g., other services) access services by means of a standardized interface that decouples concrete service implementation from description of features and the way to access them. Service providers can publish services (and their interfaces) to publicly available service registries that service client in turn can query. The service interface is the key in mastering heterogeneity: it represents a standardized contract that hides

service realization details and by means of which users can automate access to the service itself. SOA is not a novel concept and has proven to be a suitable approach to hide service heterogeneity, but in recent years it has gained momentum and renewed interest thanks to the Web Services [10] revolution. Web Services initiative is an implementation of a SOA that leverages widespread Web-related technologies for both the description and the enactment of services. Web Services do not formally mandate any specific standard, but typically they exploit a set of widespread specifications, like the HTTP protocol for communication of exchanged data, and XML-based grammars for exchanged data format definition (SOAP protocol [11]), service interface description (WSDL protocol [12]), and service registry standardization (UDDI protocol [13]).

Service-oriented Computing (SOC) [14] is a more and more emerging paradigm in the realization of distributed software that extends Service Oriented Architectures by proposing a much more complex and feature-rich layered vision in which services are just the bare low-level of the architecture. Higher levels of the proposed architecture promote features to, for instance,

- compose and coordinate services into more complex and value-added aggregates;
- monitor services and Quality of service characteristics
- establish and enforce Service Level Agreement policies between producers of services and consumers
- rate and certificate services or compositions of service

This vision is becoming more and more important in that it envisions a comprehensive and more structured approach to service oriented architectures, specially by intimately promoting service composition and aggregation to build more complex services by simply assembling other *off-the-shelf* services.

2.2 Ubiquitous computing scenarios

Ubiquitous computing scenarios were first envisioned in early '90s by some work [15] at Xerox PARC and promoted an ecosystem where mobile devices and network connectivity intimately permeates end users everyday life.

Recent technology advances in wired and wireless network connectivity and the availability of increasingly powerful and feature-rich mobile end user devices are more and more fostering and making Ubiquitous computing scenarios concrete. Users are more and more requiring to exploit services and contents anytime, anywhere and by means of any device. Furthermore, services and contents much more need to tailor to fit user needs and characteristics (e.g., device in use) as well as to adapt to environmental conditions (e.g., network connectivity type and status, user location,...). Ubiquitous computing scenarios stress many debated research fields, from mobility to context-awareness and multimodal multichannel content access. Though preliminary works exist that try to cope with the aforementioned issues, currently, they are not really adopted on a large scale; the main reason is that most approaches tend to face only a limited set of the previous properties, thus producing solutions that, for instance, provide support for content adaptation but miss to support mobility and multimodality.

In the following we analyze state of the art in the most relevant fields of Ubiquitous computing and we provide an insight into the most relevant middleware support proposals for each one of them.

2.2.1 Mobility

The increasing availability and mass-market adoption of novel wireless connectivities (e.g., 3G mobile networks, IEEE 802.11 standards, ...) and much more powerful devices able to exploit them promotes novel scenarios for the end users. Access to services and computation more and more becomes free from fixed positions and enables users to roam and move while still performing their tasks.

Typically, state of the art of research identifies different categories of mobility, each one with well-defined characteristics [16]. User mobility concerns problems in supporting user activities while they move across different locations; in this situation, users require to access a uniform and consistent view of their specific working environment (e.g., user preferences or profile information) independently of their current location. Terminal mobility allows end user devices to move and connect or reconnect to different communication networks while remaining reachable and keeping communication sessions consistent. Finally, resource mobility allows resources to move across different location by still remaining available, independently of their physical location and the current position of their (possibly mobile) clients.

In recent years, some proposals tried to face mobility issues by means of Mobile Agent platforms. Mobile Agents platforms [17] provide a support layer that allows software components to migrate between different network nodes during execution, by carrying their code and the reached execution state. Solutions basing on this approach are currently adopted not only to support user and terminal mobility (e.g., [18]), but also to realize multimedia content adaptation for both fixed and mobile users (e.g., [19]). Agents are also used to convey context information (e.g., [20]) while effectively integrating services.

2.2.2 Context-awareness

Mobile computing opens up novel scenarios in which computation can occur at different physical locations and by spanning a multitude of different environmental conditions. *Context* is a rather generic term used to indicate a broad category of information that relates to specific characteristics of both users and devices operating in a certain applicative domain [21]. A typical example of context information is the current location of devices and users; in fact, preliminary research work in the area of context-aware computing highly focused on location information. However, other relevant context information exist and

they typically relate to user activities and/or preferences, user interactions and interrelationships with other users and/or devices, as well as device capabilities and/or their current state and operating conditions.

Context-awareness refers to the ability of a computing system to provide services and contents that are adapted and tailored to the specific conditions in which users and devices are currently operating [22]. Context-aware systems need to face a number of non-trivial tasks, some of which do not yet have a clear and commonly agreed upon solution. The most intuitive task in designing context-aware systems relates to context information retrieval and basically requires context-aware systems to provide convenient and effective ways to both gather context information from a wide variety of sources, such as user profiles held in a database, sensors that monitor environmental conditions, status and operative conditions of user device and/or other devices operating in the same area. Another crucial task relates to reasoning and reaction to context information changes: variations of context can force the system to re-adapt and reconfigure in order to provide a much more tailored system.

As heterogeneity of context-aware scenarios increases, different sources of context information may be involved, possibly exploiting different formats for conveying such information; the need for common formats and models for context information thus becomes a compelling issue. Furthermore, as context information becomes very large, reasoning and reacting to context variations may lead to inconsistencies or conflicts in the actions to be taken; therefore conflict resolution in context-aware adaptation process becomes a non negligible task. Finally, other relevant aspects context-aware systems may need to cope with relate to efficient and distributed context information storage and dissemination.

State-of-the-art in context-awareness support tends to focus on positioning-based service provisioning and on the development of toolkits and frameworks to create new context-driven applications. As for location-awareness, most widespread applications so far have been GPS-based car navigation systems and

handheld (sometimes wearable) tourist guide systems (e.g., [23]). Despite the success in this field, location-aware applications are often dedicated to precise scenarios (e.g., museum locations, car driving, ...) and it is still difficult to integrate heterogeneous positioning systems (e.g., GPS does not work indoor). Works are being published to address the issue of integrating different positioning information (e.g., [24]). As for toolkit-solutions, some frameworks exist (e.g., [25]) that offer tools and libraries to easily develop services that leverage context-related information such as user location, connection type, device features and so on. We do not disregard these approaches, but claim the importance of a much more comprehensive view that takes into account a wider range of both context information and, more generally, of Ubiquitous issues.

2.2.3 Multimodal multichannel multipattern user interaction

Device heterogeneity opens up novel ways for the users to exploit contents: users are no longer bound to traditional fixed PC workstations with Web browsers but can access content or applications on the Internet by means of different user interfaces, via different communication channels and according to their preferences or device features.

2.2.3.1 Multimodal access

Multimodal access relates to the coordination of different natural input modalities (such as speech, touch, hand gestures, eye gaze and body movements) with different multimedia output modalities (text-only documents, images or vocal readings are typical output formats). This aspect is becoming important not only to provide users with multiple media access channels but also to promote and extend content accessibility to impaired users. The ‘‘eEurope 2005 Action Plan’’ from the Commission of the European Communities [26] witnesses the importance of this issue for e-government stakeholders. Though compelling requirements for integration of different natural input/output modalities are evident, the proposed

solutions and frameworks tend to have vertical approaches and focus only on specific and fixed sets of interaction modalities or application domains. Typical solutions address, for instance, e-learning [27], medical consultation [28] or crisis management [29]. Similarly, some general purpose multimodal frameworks [30-34] have been proposed, but, again, they tend to be limited to sets of predefined interaction modes (specially auditive ones) and therefore still lack a concrete and widespread adoption.

2.2.3.2 Multichannel access

We refer to *multichannel content access* as the ability of providing services or information content through different media channels and platforms [35]. Typically, different heterogeneous communication channels can be involved in service/content provisioning, from traditional fixed Ethernet or DSL connections, to wireless technologies (e.g., WiFi, 3G mobile phone networks, Bluetooth PANs, ...), and also GSM SMS technology or DVB-T broadcasting. By supporting multichannel access, heterogeneous devices access contents in a consistent manner and receive them in different forms, depending on the particular channel being exploited. For instance, TV news can come as video streaming on DVB-T channels and broadband networks, perhaps together with useful MHP applications; on limited devices or GPRS connections, instead, they should be converted to snapshot images surrounded by plain text to save bandwidth. Finally, users willing to exploit older legacy technologies such as SMS and/or GSM standard can receive plain text short messages or phone calls with a synthesized voice reading news content. Traditional multichannel content access platforms, anyway, are usually built with a restricted number of delivery channels in mind and need re-engineering to enable access via multiple channels. Typically, this is achieved by exposing functionalities as software services and adopting SOA strategies to compose them [36], either implementing a channel-agnostic communication system [37] or channel-adaptive information systems [38].

2.2.3.3 Multipattern access

Support for multimodal and multichannel access allows users to remodel the *interaction patterns* [39] to exploit services and contents. Indeed, different interaction forms and channels could render the typical pull-type request/response interaction pattern quite limiting; it becomes more and more necessary to support also push-based, conversational or even mixed communication patterns. By mixing different interaction styles and channels it is possible, for instance, to realize complex single-request/multiple-response patterns: a user may ask (say, by means of an SMS) for traffic information related to a certain path. In response she could receive a concise resume by an SMS text message and a detailed mail that, along with textual content, provides user with maps of alternative paths. State of the art research in this field focuses on generically modeling human/services interaction by means of coordination/orchestration platforms: BPEL4People [40] and WS-HumanTask [41] proposals try to model human participation in process orchestration by providing extensions to BPEL that integrate human resources and coordinate with human tasks. However, these approaches are controversial: some recent work criticizes the richness and quality of offered features [42], others [43] argue that these approaches are too technology-dependent and suggest to raise the abstraction level to provide a much more user-friendly model-driven approach.

3 Middleware Design

"The most profound technologies are those that disappear.
They weave themselves into the fabric of everyday life
until they are indistinguishable from it."

Mark Weiser,
The Computer for the 21st Century

This section describes the main requirements we identified in building a Ubiquitous support middleware. By analyzing them, and by evaluating state of the art, we identified some key design principles in the realization of a truly Ubiquitous support middleware.

3.1 Requirements

Heterogeneity is a key characteristic of Ubiquitous computing scenarios. Users can exploit a plethora of different devices and connectivities, and need to access a virtually unlimited set of services and contents. Furthermore, environmental conditions in which both users and services operate may be extremely different and can vary in time, thus requiring to cope with such changes. A platform that supports Ubiquitous Pervasive computing should therefore be extremely *flexible* and *extensible* in order to enrich and tailor itself, by adding novel features and

support for novel scenarios.

Furthermore, end users are the primary audience for such a kind of platform: users should be able to easily arrange, access and share contents and services, without having to cope with technical details. *User-friendliness* is therefore a key element in providing a really usable and pervasive platform to support Ubiquitous computing. Weiser [15] himself recognized that a truly effective and widespread technology needs to permeate everyday life so intimately that it disappears at all, and it is so simple to use that users exploit it unawarely.

Middleware approaches typically tame heterogeneity of distributed applications by providing a uniform layer of support functionalities that hides heterogeneity (hardware, operating systems, network connectivity) of network nodes involved in the realization of a distributed application. In fact, some recent work in the literature propose middleware infrastructures to provide ubiquity support features to help build Ubiquitous scenarios. As described in section 2, current middleware solutions typically tend to face only a limited set of Ubiquitous issues, and, as Ubiquitous scenarios become more and more mature, they tend to enrich with novel, more complex and extremely interconnected features. However, this collides with a basic middleware principle: in order to be really effective, middleware should be extremely essential and tailored, and provide exactly the needed features and no more [44].

Platform design needs therefore to cope with apparently strongly diverging driving forces: support for heterogeneity, flexibility and extensibility calls for the dynamic addition and enrichment of middleware with novel features whereas the need for essentiality and tailoring pushes the platform to provide the sole features needed to realize a certain scenario.

Dealing with increasing complexity, frequent changes and tailoring needs have always been compelling concerns in the design of large complex software systems; however some software design principles have proven to help master these requirements. The following sections describe key principles in designing

and engineering a truly effective middleware for Ubiquitous computing.

3.2 Design principles

> *"... But nothing is gained --on the contrary!-- by tackling these various aspects simultaneously. It is what I sometimes have called 'the separation of concerns', which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by 'focussing one's attention upon some aspect': it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously. ..."* (E.W. Dijkstra) [45].

Though many valuable architectural design approaches in modern software development help software architects in building large heterogeneous distributed systems, *separation of concerns* has proven to the the key in approaching problems with the right abstraction level and in a manageable way [46, 47]. Separation of concerns refers to the process of identifying and decomposing software logic into parts that are relevant to a particular concern (concept, goal, purpose, etc.), with the goal of addressing each problem separately, still with a unifying approach that ultimately aims at integrating them into a coherent view.

We claim that separation of concerns is the key in providing an intuitive, user oriented platform for ubiquity support.

The first essential separation of concerns stage in the design of this kind of platform calls for a neat distinction of the features our platform exposes to final users: in our opinion, users need to access a restricted number of simply and

explicitly defined facilities. Hence, from a user perspective, we adopt separation of concerns to clearly classify the major different kinds of tasks users are able to perform by means of our platform. In our opinion, no matter the complexity of the Ubiquitous scenarios, users will always need (and are restricted) to cope with three main concerns: *service/content choice logic* grants users a convenient and intuitive way to search for and choose services and contents they are interested in among the currently available ones; *service coordination logic* provides high-level service coordination features, such as facilities for aggregating contents from a bunch of services rather than using them standalone; finally *user interaction logic* lets users choose how to exploit services or groups of coordinated services, by choosing the interaction style/pattern as well as the input/output channels and formats.

This broad separation is a first step toward user-friendliness in that it clearly drives what users can generally do with the platform; however, so far, this does not fill the gap between high-level user requisites and concrete pieces of ubiquitous business logic.

To face this gap by still supporting heterogeneity and flexibility, the middleware platform itself should undergo a design intimately inspired by separation of concerns. Many different software design architectural principles incarnate the concept of separation of concerns and have emerged as patterns and approaches that help in the realization of complex heterogeneous software systems. The *delegation* principle aims at keeping software component logic simple: each software component should cope with a specific concern (or a limited set thereof) and delegate responsibility of other concerns to other suitable components. The *decoupling* principle aims at keeping interacting software components as much reciprocally loosely coupled as possible. *Semantics*-based software description allows to separate and abstract software high level features and characteristics from concrete, low-level operational details. Finally, the *layering principle* promotes to stratify software functions into different levels (layers), each one at a different abstraction level. Hence, lower levels typically

target practical operational concerns, whereas higher levels typically address high abstraction level concerns that stress the inherent principles of the software being realized. The following sections describe how design principles that directly stem from separation of concerns can help in the design of our middleware for Ubiquitous computing.

3.2.1 Delegation: a disappearing middleware approach

The *delegation* principle [48] pushes a software component receiving requests of a certain feature to delegate their fulfillment to another piece of software. This design principle is extremely helpful to tame the growth in software complexity: increasingly elaborate software components may decide to delegate and modularize software logic to other components, thus keeping inner logic simple. The delegation principle naturally fits the inherently dynamic and ever-growing Ubiquitous scenarios and is the key in mastering the diverging forces that drive Ubiquitous middleware design.

By following the delegation principle, we propose a Ubiquitous middleware design that delegates all of the concrete Ubiquity support features outside of the middleware itself and that leaves middleware only a limited set of basic support functionalities. Thus novel ubiquity support features (e.g., different kinds of content retrieval/transcoding/adaptation or novel communication channels) can be added/removed with little or no effort; middleware then somehow tends to *disappear* behind an increasingly heterogeneous and varying set of features it is able to offer.

In our opinion, by keeping middleware logic simple and lightweight, this *disappearing middleware* approach is able to perfectly tailor to a wide variety of Ubiquitous scenarios, providing the sole needed features and thus remaining extremely effective.

3.2.2 Decoupling: service-oriented computing

The *decoupling* principle refers to the practice of keeping pieces of software logic as independent and unaware as possible from technical and operational details of other software artifacts they collaborate with. This sort of *divide and conquer* approach aims at maximizing software maintainability and manageability and at minimizing the impact of changes, additions or removals of software artifacts. Service-oriented computing pushes decoupling principles to the extremes, calling for a distributed landscape where software functionalities are modeled by means of the abstraction of services. Services know and cooperate with each other only by means of service interfaces that completely hide implementation details. Service providers can publish novel services, hence allowing old existing services to exploit novel ones.

By following a decoupling principle we therefore model ubiquity support logic by means of the abstraction of service, hence allowing for easy addition and removal of novel features in the form of services. Services ultimately are the pieces of business logic our platform exploits to satisfy user needs by choosing among the currently available ones and by arranging and making them cooperate. As a consequence, our middleware platform provides only features to help users select, compose and coordinate services, thus remaining extremely lightweight and application-unaware.

Services can be implemented by exploiting a vast heterogeneity of different programming languages, operating systems or physical resources; however, each service provides a standardized interface that completely, in detail and in a standardized way describes all of the features the service offers. This kind of description is typically targeted at operational description (input/output parameters, methods/procedure names), hence determining *how* to interact with a service and allowing automated tools to autonomously generate logic to interact. On the contrary, it features poor user-friendliness and typically is not suitable to provide a high-level, user comprehensible description of *what* a service does. End

users should be able to choose, arrange and exploit service logic according to their needs in an easily understandable, intuitive way.

3.2.3 Layered semantics

In recent year, semantics [49] has emerged as a means to "decorate" concrete pieces of business logic with high-abstraction level information that helps in describing, reasoning on and managing application logic from a much higher standpoint than operational details. Various examples of semantics are nowadays widespread: metadata annotations of modern programming languages (e.g., Java) allow compilers to manage and more thoroughly and deeply reason on code correctness than simply enforcing syntax checks, and at the same time provide users with high-level description of certain characteristics of the software they are writing. As another example, the Semantic Web initiatives [50, 51] aim at enriching contents on the Web with descriptions that end users (or machines) can exploit to more naturally and intuitively search for and establish correlations between contents from different content sources on the Web.

In our opinion, semantics is the keystone that prevents users from having to cope with low-level operational details and allows middleware to automatically handle and translate user requirements into concrete arrangements of business logic. To allow for this, semantics needs to face three distinct concerns, each one at a distinct abstraction layer: the first one relates to providing users with a set of high-level abstractions that help them easily expressing their requirements; the second one relates to providing low-level operational instruments for the middleware to concretely arrange business logic to fulfill user needs; finally, the third one relates to mechanisms and formal tools to translate high-level user requirements into concrete arrangements of business logic.

The *Layers* architectural pattern [52] promotes separation of concerns into a stratified view where each layer groups concerns at a specific abstraction level and hides details of the underlying layers to the upper ones. By following this

principle, we stratify semantics into a layered structure that features the following levels: *user semantics layer* provides the high level user requirement description facilities; the *semantics fusion layer* is in charge of interpreting and translating details of the user semantics layer into concrete arrangements of pieces of business logic; finally, the *business logic layer* provides the low level commodities to manage concrete business logic and arrange it according to user requirements.

3.3 Architecture

By following the principles described in the previous section, we can devise and put together a unifying and integrated architecture, depicted in Figure **1**.

In our opinion, the separation of concerns approach is the key in both smoothing and making users experience easier, and in designing an extremely flexible, open and heterogeneous middleware platform for Ubiquitous service and content provisioning.

The final architecture reflects an intimate adoption of the separation of concerns approach for what concerns both users and the middleware layer. In fact, users approach our middleware with a clear and neatly distincted view of what they can do; in our opinion, no matter the complexity and heterogeneity of the Ubiquitous scenarios and applications, users basically will always have to choose one or more services (or contents) of interest, to arrange them according to their preferences, and to define how to interact with and exploit them.

Similarly, middleware clearly separates a minimal, almost disappearing, kernel layer that provides support functionalities and delegates the responsibility of concrete Ubiquitous logic to a layer of services that can be added (and removed) by need.

To fill the gap between high-level user requirements and low-level details of building correct and sound arrangements of services, we introduce semantics mechanisms and we separate them into different levels of abstraction. The

business logic layer encompasses all of the concrete logic to realize Ubiquitous Pervasive scenarios; content retrieval and adaptation services or channel management and delivery services typically reside at this level. This level is also responsible of managing the concrete operational details of services that determine whether they can concretely cooperate and interact with each other, e.g., by checking that input/output messages of cooperating services are compatible and expressed in the same format. The *user semantics layer*, on the contrary, provides a restricted set of tools and high-level abstractions that easily allow users to accomplish the tasks of choosing services, arranging them and interacting with such arrangements, without having to delve into the hard to manage service operational details. The *semantics fusion layer* provides a set of instruments to help the platform interpret, merge, and translate distinct high-level user requirements (service choice, service coordination and user interaction) into a unified concrete set of services, suitably arranged to fulfill user needs.

User semantics layer, semantics fusion layer and business logic layer constitute the *composition model* of our platform and their elements (described in section 4) drive the concrete process of composing services according to user needs. The *service composition engine* is the key middleware element that manages the composition model (i.e., all of the above mentioned abstraction tools) and concretely enacts routines and algorithms to realize service composition.

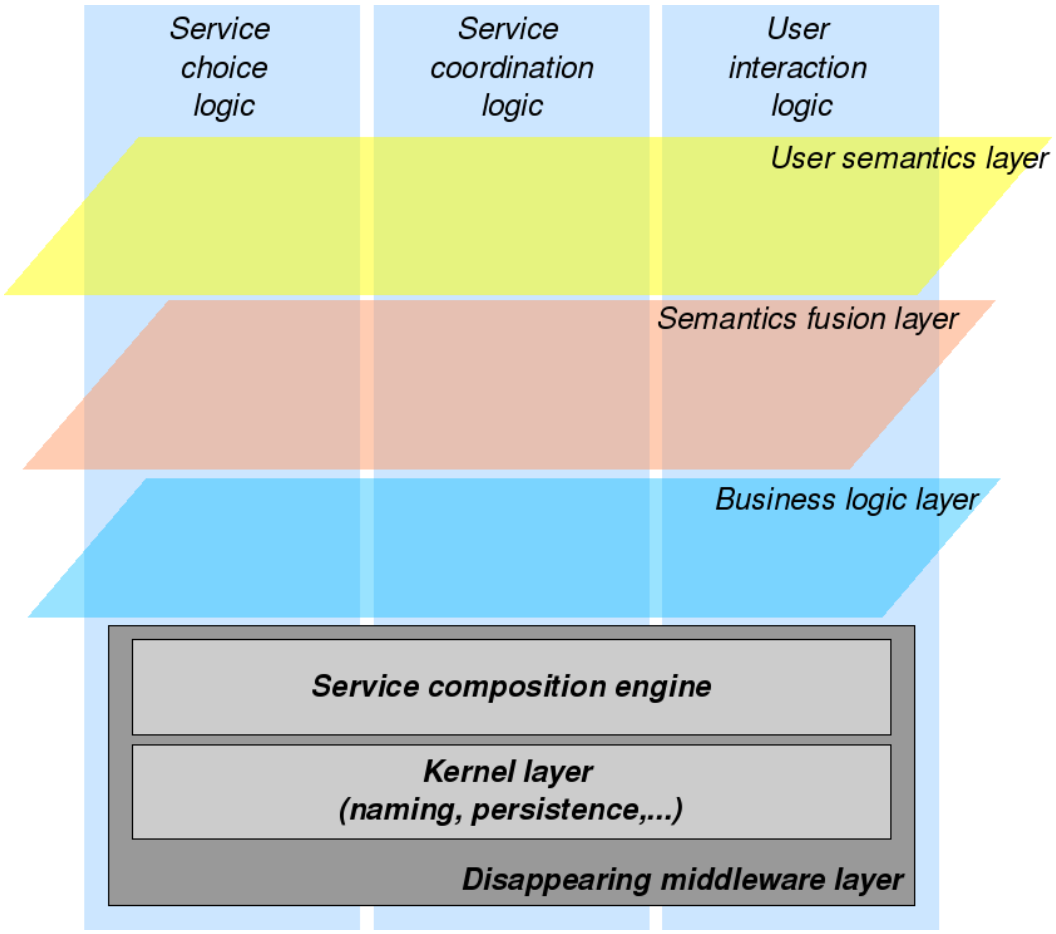


Figure 1. Architecture

3.4 Related work

Aggregating and composing pieces of business logic/services is gaining momentum as a way to tame the inherently increasing complexity of Ubiquitous Pervasive Internet scenarios. An SoC approach allows to easily plug in support for novel features (communication channels/patterns, media format, and so on) by simply adding new services and by arranging (more or less complex) service aggregates. State of the art highlights two main tendencies about adopting service composition to provide users with ubiquity and pervasivity support platforms. On the one hand, Web mashup platforms leverage user-friendly Web techniques to let

users graphically arrange compositions of contents from a (usually restricted) catalogue of available ones. On the other hand, more formal approaches leverage semantics to propose automatic composition platforms able to reason on and interpret user requirements expressed by means of a certain semantic agreement.

3.4.1 Web mashups

An interesting trend in service composition directly relates to the emerging Web Mashup scenarios: users more and more are provided with Web-enabled user-friendly appealing tools to aggregate contents over the Web [53]. Yahoo Pipes [54], Intel Mashmaker [55] and Google Maps-based [56] mashups allow users to directly aggregate and interconnect Web-based contents by means of easily exploitable visual tools.

These tools let users participate more and more in the process of content creation and aggregation and generally propose an effective way to help and guide them throughout such a non-trivial task. However, these solutions are typically vertical and ad-hoc: allowed contents and services are usually Web pages (or XML-based formats such as RSS) and users are allowed to exploit such contents basically by means of the sole Web browser. As a consequence, flexibility and extensibility are still open issues of this kind of approach and research [57] starts perceiving the SoC model as a promising way to extending and broadening mashup platform support.

3.4.2 Semantic ubiquitous service composition

The semantic service composition tries to overcome heterogeneity and complexity of Ubiquitous scenarios by modeling business logic into semantic-enabled services and composing them into value-added aggregates. Some current work [58] propose composition models based on a fixed stack of semantic description layers; this clearly evidences the main different abstraction levels involved in semantic service description, but, being fixed, it inherently suffers

from the lack of extendibility: service providers willing to plug in new services need to conform to such fixed model and are not able to provide newer or different semantic metadata to capture novel service features. The *Scooby* middleware platform [59] aims at providing a user-oriented service description and composition enactment middleware; even if this approach seems promising, the chosen model for service description limits service modularity and reuse; as an example, if a service needs to interact with other services, its description needs to explicitly define bindings with the other required services. Other works [60] propose a semantic-enabled framework for dynamic service composition where users can exploit natural language to express their requirements; platform is then in charge of translating natural language requests into concrete service compositions. In our opinion, natural language requirement specification is potentially extremely flexible but offers no help to the average end user in the process of service choice; by allowing for natural language expression, the platform gives no perception or feedback to users about, for instance, what kind of compositions the platform can cope with, what kind of services are available and so on. As a consequence, so far semantic composition of ubiquitous pervasive services seems to be a promising, powerful and flexible way to realize and automate service composition, but current approaches miss the right abstraction level and result either overly complex for average end users (in reason of a lower abstraction level that lets emerge large part of the operational details), or too expressive and free (hence at a higher abstraction layer) but practically poorly usable in real world scenarios.

4 Service Composition

"*All models are wrong. Some are userful*"

(George E.P. Box)

Application and service composition is at the heart of our model and is the key in providing an extremely flexible and heterogeneous Ubiquitous support platform. Novel services (e.g., logic to handle novel communication channels or novel content kinds) can be plugged in by need and composed with other services to face novel ubiquity scenarios. This section describes the composition principles that drove the design of our composition model, then deepens the description of the composition model itself and of the concrete process to translate user requirements into concrete service aggregates.

4.1 Service composition principles

Service-oriented Computing strongly promotes aggregation and reuse of software artifacts (services) to increase modularity and flexibility of distributed systems. Service composition is rapidly gaining momentum as a way to fuse

existing services to realize novel value-added service aggregates.

The extremely vast and heterogeneous landscape of service composition proposes a number of different approaches and proposals that target extremely different scenarios.

Early service composition platforms focused on rather static scenarios (especially Enterprise Application Integration) that required to coordinate a (usually limited) number of services in a well-defined and deterministic way. First proposals therefore aimed at providing methods and tools to clearly define static and immutable compositions of services by explicitly expressing how services had to cooperate, e.g., the order in which they needed to be invoked and the operational parameters (e.g., input/output) involved. BPEL4WS [61] is one of the most widespread standards for service composition and proposes an XML-based grammar to define compositions of Web Services; a number of tools currently exist to both easily and graphically sketch out service compositions and to manage the concrete execution of BPEL4WS-based service compositions.

However, this kind of approach has proven to be very limited for some compelling reasons. The first crucial one relates to the fact that designing a service composition in such a way is typically a completely user-dependent process: a human is in charge of finding useful services and of manually defining interconnections between them to realize the required task. This obviously requires the composition designer to have a wide and high level expertise in both the applicative domain the task relates to and in the formal grammar used to express the composition.

The second problem with early static approaches relies in the fact that they inherently fall short in more dynamic scenarios. For instance, the initial set of available services may vary in time (by either growing or shrinking), an exact match between a specific subtask and a concrete service may not be available, or the overall final task can not be expressed in a precise and unambiguous way, either because the final service composition user has little expertise of the

applicative domain or of the composition model, or because the requirements themselves are unclear.

Many different approaches tried to face the problems that arise in such dynamic scenarios; basically, two main tendencies outstand and sometimes even coexist.

The adoption of a *semantic* description allows to capture service/service composition features that go beyond traditional basic operational features (such as input/output parameters) and provides a higher level description both of requirements the composition need to fulfill, and of service features such as behavior and/or interoperability constraints. WSDL-S [62] and OWL-S [63] are two of the most notable XML-based proposals in the field of semantic metadata service description and enforcement. The semantic approach provides users with richer and more detailed descriptions of services. This has the obvious benefit of being much more clear to unexperienced users. However, a richer service description allows also to capture details such as what a service is able to do rather than how it does it; this information can be used to automate (e.g., by inference) compositions of suitable services each time no clear solution is evidently achievable.

Other proposals aim at providing much more theoretical formal service composition models to not only describe service compositions but also to help reasoning on them, for instance to detect inconsistencies and/or possible deadlock conditions or to infer novel and/or better compositions from previous ones. Typical approaches that fall in this category model service compositions by means of Petri Nets [64] or of some variants of process algebras (e.g., Calculus of Communicating Systems [65] or Calculus of Sequential Processes [66]). Other approaches [67] define semantics in terms of a first-order logic, namely the situation calculus [68] and, based on this semantics, they describe service compositions by means of a Petri Nets model. Formal approaches, such as Petri Nets or first-order logic ones, have proven to be extremely powerful, especially when it comes to reason on a certain applicative domain and/or set of service

compositions. Some models are able to determine whether a composition not only satisfies initial requirements but also if it is *secure,* e.g., provides no deadlock conditions or unreachable states. Other models allow to automatically infer novel service compositions from existing ones in order, for instance, to provide optimized compositions (e.g., service composition with equivalent overall behavior but with less services involved) or alternative versions.

In our opinion, the main features a composition model should provide in order to help realizing ubiquity support scenarios relate to *user-friendliness, automation* of service composition process, *scalability* of the process itself, and *extensibility*.

User-friendliness requires to lower the level of required expertise of the final user, by hiding service connectivity details and by rather conveying high-level features description. Service composition *automation* requires the concrete process of choosing suitable services and arranging them into suitable service compositions to not involve users, apart from initial requirements specification. S*calability* requires to build a service composition model that can scale as the number of available services and/or templates grow, by finding out a reasonable amount of compositions in a reasonable amount of time. On the contrary, we are not interested in building an intelligent composition system that can infer novel optimal solutions by, for instance, recursively applying previous solution patterns (such as previous formal models), since this approach can quickly become unmanageable as composition elements (e.g., services and templates) number grows.

Finally, in a highly dynamic and flexible scenario where novel applicative services can be plugged into the platform by need and therefore can be employed to build novel compositions, *extensibility* forces the composition model itself to be able to cope with novel services and novel scenarios in a flexible and extensible way.

We acknowledge that automatic composition of services needs both a formal model to represent compositions and semantics to give meaning to the formal

representation itself. However, we aim at tackling this problem from a different perspective with regards to, say, first-order logic or Petri nets modeling approaches. Even though extremely extensible and intelligent, these formal models in fact typically result extremely difficult for unexperienced users (since the elements of the model typically are mathematical or logical entities) and they seriously affect and compromise scalability.

We therefore claim that semantics provides a convenient means to fill in the gap between concrete service arrangements and unexperienced users, and therefore we adopt semantics to convey high-level description of services and service compositions; similarly, we acknowledge the need for a strong and rigorous formal model to help automating service composition. However we want our formal model to explicitly provide a set of clear abstractions the users can exploit to accomplish the composition process in a more intuitive way rather than being so extremely powerful to be able to reason, infer and extend itself.

By using our composition model, an average user is able to express high-level requirements about the overall task he is interested in by means of a set of intuitive semantic notions and abstract modeling facilities. Then, it is up to the platform to decide whether suitable compositions can be arranged out of existing services, and if more than one exist, possibly to rank them by a certain criteria.

4.2 Composition model

Figure 2 reports an overall view of the architecture, with a specific focus on the main components of the composition model.

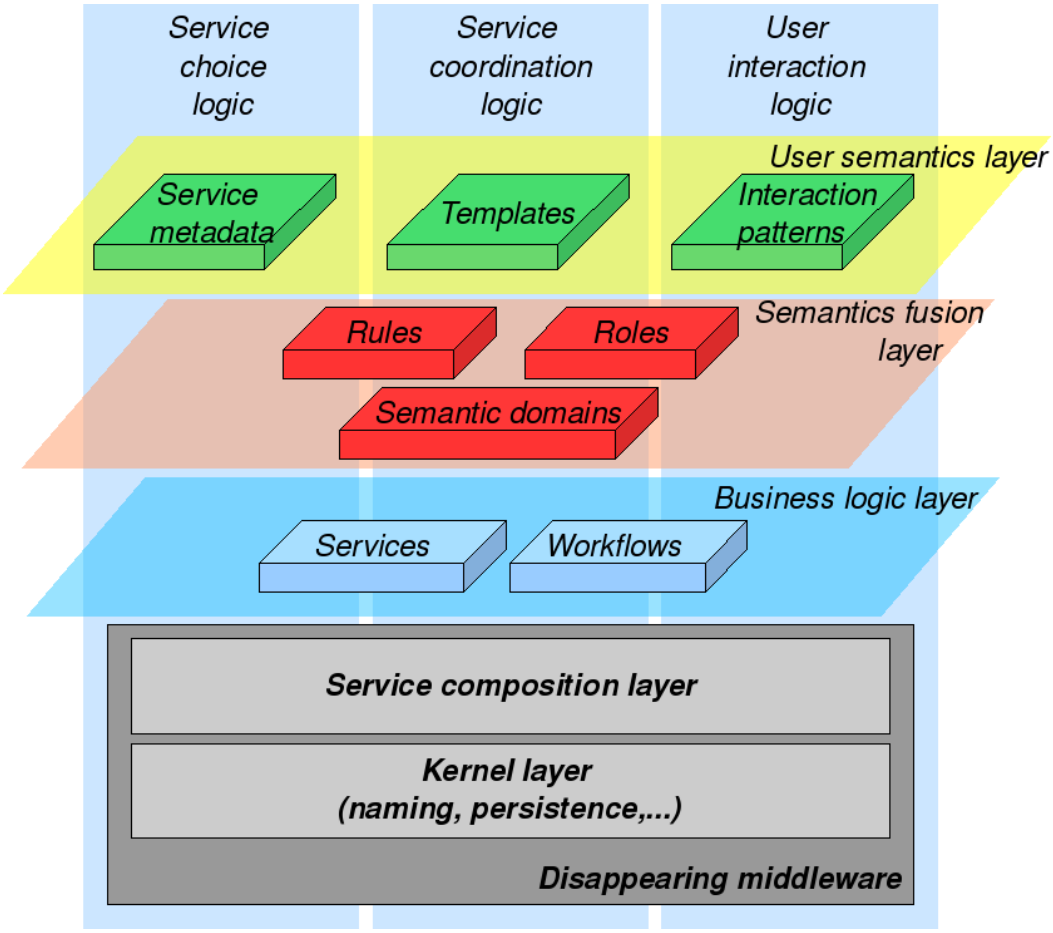


Figure 2. Architecture

Services and *workflows* represent the lowest level of our model and basically involve syntactical elements: services are the basic building blocks of our applicative system and workflow [69] is the concrete means to make them cooperate. Workflows describe structured activities and their complexity can range from simple sequences of services activated after one another, to complex compositions of both services and control blocks, such as conditional branches, forks, joins and so on.

At the opposite highest level, *service metadata* conveys high-level information about semantic features of services, e.g., their typology (content generation and retrieval, transcoding, etc...) or QoS-related aspects (average computational load,

etc) that can be used to drive service choice for users. *Templates* model abstract flows of activities, i.e. flow definitions whose nodes need to be partially of fully filled in with concrete business logic; hence templates are a suitable abstraction to help users in sketching out arrangements of services (so to express service coordination logic). Finally, *Interaction patterns* allow users to model their preferred service (or service aggregates) interaction styles, hence providing suitable abstractions to drive user interaction logic specification.

In between, the *semantics fusion layer* provides features that allow to translate abstract templates into concrete workflows. *Rules* express constraints on pieces of business logic that participate in the realization of a template, whereas *roles* allow to express such constraints not on a specific business logic element (e.g., a service or a workflow node) but to abstract, share and reuse them across different elements of the template. Finally, *semantic domains* convey a useful means to partition semantic features into distinct spaces, so as to avoid providing a fixed and immutable semantic knowledge base, but rather to foster insertion of novel semantic concepts while still keeping older ones consistent.

It is important here to notice that we choose to realize an inherently non-recursive service composition model; basically, consistently with principles described in the previous section, we do not want our system to find out all possible solutions or even novel ones, by automatically composing templates into much more complex templates. In fact, when available templates, services and semantic domain number increases, intelligent approaches that recursively explore all possible solutions may become practically unusable and too much expensive in terms of computational cost. Rather, in an average system condition, we prefer arranging only the most minimal workflows, without having to guess whether more complex ones can be arranged with similar functionalities.

Nevertheless, we acknowledge that under certain conditions, recursive compositions may help realizing infeasible solutions. The most typical case relates to a piece of business logic that can not be carried out by a single service but can

be realized by, say, building up a sequence of several available services.

In ourmodel, we want to explicitly control the adoption of recursion, for instance by limiting it to specific cases (e.g., when no other compositions can be arranged) and by carefully selecting the most meaningful templates that can act as sub-pieces of other templates. This is why we relegated recursive mechanisms to the concrete service composition algorithm, as described in Section 4.3.6.

Finally, to foster model extensibility and by following a SoC approach, novel service metadata, templates and interaction patterns can be plugged in by need; therefore, since average end users will typically exploit already available ones, administrators and/or smart users can build and share novel metadata, templates and patterns, hence extending platform facilities.

4.2.1 Business logic layer

Business logic layer provides the low level facilities that concretely realize applicative scenarios. Entities of this layer should be completely invisible to final users: it is up to our middleware to concretely manage business logic implementation details to realize user requirements.

4.2.1.1 Services

Following a SoC paradigm, we model pieces of application logic as services that can be plugged in by need to extend middleware ubiquitous features support. Hence, support for novel content types as well as novel formats (and consequent adaptation/transcoding logic) or novel user interaction channels can be easily added by simply adding new services.

4.2.1.2 Workflows

In traditional SoC approaches, aggregation and coordination of services help realizing more complex value-added applicative scenarios out of basic building blocks, thus promoting business logic reuse and modularity. Workflows can range

from simple sequences of services to more complex aggregates with conditional branches, fork/join nodes and so on. Managing execution of logic entails concretely invoking services after one another, hence workflows are in charge of tasks such as parameter passing between subsequent stages and exception handling.

Definition 1. *We model workflows as directed graphs* $WF := (WFN, WFL)$ *Workflow nodes (WFN) can be concrete services or control blocks (e.g., fork, join or conditional nodes). Workflow links (WFL) are directed connections that interconnect two workflow nodes.*

Definition 2. *Two workflow nodes connected by a link are adjacent.*

Services and workflows are concrete entities of the system and are in charge of concretely realizing user-driven ubiquitous scenarios. Once established, workflows and services need no semantic interpretation; on the contrary, semantics is used to decide whether a given (more or less formal) description of requirements can be satisfied and translated into a concrete workflow of services.

4.2.2 Semantics fusion layer

Semantics fusion layer realizes the glue that helps translate high level user requirements into concrete workflows of available services.

4.2.2.1 Semantic domains

A number of different proposals exist to specify semantic information on services; some approaches are extremely tailored to specific areas of interest or applicative domains, whereas other proposals aim at giving generic purpose models and languages to describe any kind of semantic feature. As an example, Web Service Semantics [62] or OWL-S [63] promote standard XML formats to describe semantics.

The model we propose does not rely on a specific service semantic description, but rather can be reused with any standard, thus improving flexibility and reuse.

Furthermore, a monolithic and predetermined set of semantic notions does not fit well with intrinsically dynamical scenarios where semantic itself may need to grow and adapt to ever-changing scenarios.

We therefore prefer providing our system with a way to conveniently add novel semantic information and make it coexist with already existing one. To cope with such intrinsic heterogeneity and openness, we propose the notion of semantic domains to conveniently group semantic information on the basis of, for instance, metadata area of interest (e.g., metadata regarding service quality rather than binding features) or even metadata format. Novel semantic domains can be introduced to capture novel aspects or give novel and different interpretations to pieces of business logic.

Definition 3. *We define* $\boldsymbol{D}$ *as the set of available semantic domains. Each domain can carry in semantic attributes (i.e. named properties that describe specific features) and values related to such attributes. We define* $\boldsymbol{A_d}$ *the set of available semantic attributes over semantic domain d and* $\boldsymbol{V_{ad}}$ *is the set of available semantic values for semantic attribute a of domain d.*

As an example, given the *Syntax* semantic domain, possible attributes could be

$$A_{syntax} = \{input, output\}$$

and possible values for attribute *input* could be:

$$V_{input,\,syntax} = \{application/xml, text/plain, \ldots\}$$

Typical attributes for a *QoS* semantic domain could be

$$A_{QoS} = \{estimatedComputationLoad, billing, \ldots\}$$

and possible values for attributes could be numerical values representing the average estimated computational load or the cost of the service if its use is not free-of-charge.

Semantic attributes and values can be associated to any kind of element in our model. Associations between an element of our model and an attribute or value can be either *direct* or *indirect*.

Directly associating an attribute or value to an element means describing an element with a certain semantic meaning. Even if this is a perfectly viable approach (and we will use this approach in the following for service semantic metadata), sometimes it is much more helpful to provide a way to express a certain semantic feature for an entire *class* or *group* of elements without having to explicitly bind each one of them to that feature. Furthermore, sometimes it could be impossible at all to specify semantics for an element since this element is not a concrete one but rather is an abstract element our service composition engine needs to concretely substitute with pieces of business logic.

To overcome these problems and provide indirect attribute associations, we introduce the notion of role.

4.2.2.2 Roles

Roles allow to create classes of model elements that share common semantic features. Adding a semantic feature (attribute and/or value) to a role means each element that wants to play that role has the specified attribute. Roles are a convenient means to realize *indirect semantic association,* hence they can be used to express semantic on elements that are still not concrete (e.g., template elements).

Definition 4. *We define* $\boldsymbol{R}$ *as the set of available roles and* A^r_d *as the set of available semantic attributes of domain d for role r.*

For instance, given the *contentGenerator* role, the attribute

$$output_{syntax}^{contentGenerator} = \{contentGenerator, syntax, output\}$$

identifies the semantic attribute *output* (of semantic domain *syntax*) for business logic willing to play the role of *contentGenerator*.

4.2.2.3 Rules

Rules are the concrete means to drive selection and arrangement of concrete services into workflows that realize user requirements.

Rules provide semantic composition constraints by comparing semantic attributes and/or values of a specific semantic domain for one or more pieces of business logic; hence they are used to concretely evaluate whether a real composition of services can be arranged to fulfill user requirements.

We distinguish *consistency rules* and *scoring rules* as follows.

Definition 5. *Consistency rules (cr) evaluate whether a certain set of semantic attributes and/or values are compatible with each other.*

$$cr := [A_D^R \cup V_D]^n \rightarrow \{0,1\}$$

Definition 6. *Scoring rules (sr) evaluate the degree of compatibility of a certain set of semantic attributes and/or values. We indicate the degree of compatibility with a real value*

$$sr := [A_D^R \cup V_D]^n \rightarrow \mathbb{R}$$

As an example, we provide the following rules.

$$output_{syntax}^{generator} = input_{syntax}^{deliverer}$$

$$\sum estimatedComputationLoad_{QoS}^{role_i}, role_i \in \{generator, transcoder, deliverer\}$$

The former one is a consistency rule and determines whether the semantic attribute *output* (of semantic domain *syntax*) of role *generator* and semantic attribute *input* of role *deliverer* are compatible; the latter one sums up values of attribute *estimatedComputationLoad* (semantic domain *QoS*) for roles *generator*, *transcoder*, and *deliverer*, in order to evaluate the overall estimated computational cost for each piece of business logic that plays one of the aforementioned roles.

4.2.3 User semantics layer

User semantics layer provides facilities that can easily assist users in choosing the right services, in arranging them, and in deciding how to exploit them.

4.2.3.1 Service metadata

Services represent atomic pieces of business logic related to content production, transcoding, adaptation and so on, and are described by means of semantic *service metadata*, to express both low-level grounding connection features and high level semantic information.

Definition 7. *Given **S** the set of available services, we define service metadata property*

$$p_{d,a}^{s} = (s, a_d, v_{ad}) \quad where \quad s \in S, a_d \in A_d, v_{ad} \in V_{ad}$$

Service metadata property (or simply property) is the value v_{ad} of semantic attribute a on semantic domain d for service s.

Similarly, P_d^s denotes the set of properties of service s on semantic domain d and P^s the set of properties of service s.

4.2.3.2 Templates

Templates are modeled as a directed graph and represent abstract workflows of business logic: they are made up of nodes that can represent both concrete service logic and abstract *placeholders* with some semantics associated.

By adopting a graph-based description, we are able to easily and graphically convey information of what a template does to final users; in fact, graph-based representations easily allow users to perceive the flow of control between subsequent stages of a complex aggregate of business logic. Not surprisingly, intuitive and user-oriented Web 2.0 mashup tools such as Yahoo Pipes [54] exploit the same approach and provide a *drag-n-drop* graphical interface that allows to arrange blocks (services) into more or less complex graphs.

Definition 8. *We define* $N := S \cup CB \cup PL$ *as the set of available template nodes. Thus each node in a template can be a concrete service*, a *control block (CB)* or a *placeholder (PL)*.

Definition 9. *Nodes are connected by links that represent directed connections between two nodes. We define* $L := (N \times N)$ *as the set of links connecting available nodes.*

Control blocks (*CB* set) can be nodes such as *fork, join, condition*, and so on, and they are typically used to manage and control the flow of execution among successive stages.

Placeholders (*PH* set) are the key elements in templates since they are the abstract nodes our platform must substitute with concrete business logic in order to fulfill user requirements. In order to do so, we typically put consistency rules on placeholders, thus expressing semantic constraints on the concrete business logic

that will replace placeholders. Typically, consistency rules may involve different placeholders and can be also shared and reused for different sets of placeholders in the same template. A typical example would be a rule to constrain each service willing to replace any of the placeholders to have a computation load (e.g., *estimatedComputationLoad* semantic attribute) below a certain threshold value. *Indirect semantic association* by means of roles is a straightforward method to avoid having to specify such a rule for each placeholder.

As a consequence, we provide a way to explicitly associate roles to placeholders, hence allowing for the sharing and reuse of rules across the template.

Definition 10. *We define* $PRR := \{prr : PH \rightarrow R\}$ *as the set of Placeholder-Role Relations and* PRR_p *as the set of Placeholder-Role Relations for placeholder p.*

Finally each template carries a set of rules *RU* that drive the process of filling placeholders by evaluating semantic attributes over placeholders roles they declare.

Definition 11. *We define a template as follows:* $T := \{N, L, PRR, RU\}$ *Given a template t,* N_t, L_t, PRR_t, RU_t *identify respectively the nodes, links, Placeholder-Role Relations and rules of template t.*

4.2.3.3 Interaction patterns

Interaction patterns provide convenient facilities to help users easily specify how to interact with a given template (more precisely, with the corresponding

workflow, if one can be generated out of the given template and existing services).

Interaction roles (*IR* set) are a subset of roles used to mark template nodes as, for instance, user input (*userInput*), user output (*userOutput*) or event-driven nodes (*eventInput*).

Each interaction role is associated to specific consistency rules that drive the selection of business logic suitable for playing interaction roles. As an example, a specific rule

$$typology_{behavior}^{userOutput} = delivery$$

constrains each service willing to play the *userOutput* role to provide a certain value ("delivery") for the property *typology* of semantic domain *behavior*.

Definition 12. *Given a template t, we model an interaction pattern*

$$IP := \{IPRR, IRU\}$$

where IPRR is a subset of PRR relations that mark template nodes with interaction roles and IRU (a subset of RU) is a set of interaction rules on interaction roles to drive concrete interaction service choice.

4.2.4 Usage scenario – user requirements

In the following we will describe a typical ubiquitous content aggregation scenario from the user standpoint. User requires to gather information from different content sources (e.g., an RSS feed, a newsletter and a plain HTML portal); furthermore, user requires to receive aggregated content via an SMS message on her mobile phone at a certain hour every day.

In our vision, the average end user should provide no deeper or more technical information about her requirements and it is up to the platform to arrange available business logic components to satisfy user needs (if possible).

Our platform provides a *content aggregation template* that features a couple of initial and final placeholders and a variable number of placeholders in between (in the following we will consider three generator nodes), each one of them playing a *generator* role.

This template already comes with a rule that constrains services willing to play the g*enerator* role to provide the value "*generation"* for semantic attribute *typology* of domain *behavior*.

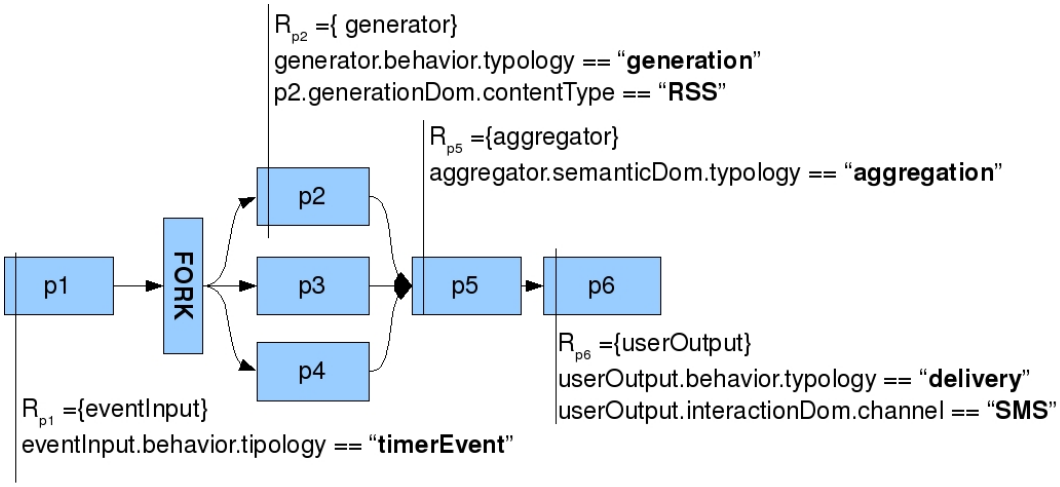


Figure 3. User requirements

The user marks the initial placeholder (*p1*) as an *eventInput* node to tell the system she wants the composition be activated asynchronously by means of an event. This action brings into the template a novel rule (associated to the role *eventInput*) that constrains services willing to play the *eventInput* role to provide the value "timerEvent*"* for semantic attribute *typology* of domain *behavior*.

Similarly, she marks the final placeholder (*p6*) as an *userOutput* node to tell the system she wants the composition to send its output via an SMS message. This brings into the template a novel rule (associated to the role *userOutput*) that constrains services willing to play the *userOutput* role to provide the value "delivery*"* for semantic attribute *typology* of domain *behavior*.

By performing these simple choices, user has constrained the template to

behave and interact with the user in a well-defined way, i.e. by asynchronously reacting to an event and by notifying the user of the elaboration result via an SMS message. In a similarly simple way, the user could have required a synchronous direct pull-based interaction, for instance by configuring both input and output on an HTTP channel.

Available semantic attributes over the *generationDomain* semantic domain relate, for instance, to content type (*contentType*). Users can therefore select semantic values (e.g., by means of convenient web user interfaces) for such attributes, to impose constraints on each placeholder. Our platform therefore adds a rule to the template that forces service (or service aggregates) willing to replace node *p2* to provide the semantic value "RSS" for attribute *contentType*. By following the same approach, user configures nodes *p3* and *p4* to produce newsletter- and HTML-related content. Note that rules that can (or need to) be shared among different placeholders (e.g, rules on the generator role) should be expressed indirectly by means of attributes over a role that marks more than one placeholder. To force placeholder-specific semantic values, we use roles specific to each placeholder (e.g., by convention, a role with the same name as the placeholder). This is the case with "contentType" attribute for *generator* nodes: each generator placeholder should feature a different value, hence a different rule, to force the platform select different kinds of contents.

Finally, user drives the interaction pattern choice by requiring output to be of type SMS.

The service composition layer is now in charge of deciding whether currently available services (or service aggregates) can satisfy user needs.

Notice that a skilled user may access a more sophisticated interface by means of which she can modify the template graph (e.g., by inserting and/or removing templates) in order, for instance to provide two alternative input or output placeholders.

4.3 Composition reification

The main goal of the composition layer is to translate abstract templates into concrete workflows (we call this process *reification*) made up of available business logic. This basically entails filling template placeholders with services (or service workflows, in cases of limited recursiveness, as explained in the following) that are suitable to play the roles declared by the placeholder. Service suitability is determined by evaluating all of the rules that involve roles of the placeholder to be filled.

In section 4.2 we introduced rules as a means to compare semantic attributes and values. In this section we will deepen the definition of rules, and show how to use them to enforce user requirements.

4.3.1 Evaluation facilities

This section describes basic tasks at the heart of the template reification process, namely service substitution, consistency and scoring evaluation.

4.3.1.1 Consistency

Consistency evaluation refers to the process of determining whether semantic values are consistent with each other under a certain meaning.

Definition 13. *We define consistency as a function that compares semantic values to check whether they are consistent.*

$$f_{consistency} = [V_{AD}]^n \rightarrow \{0,1\}, n \geqslant 2$$

The most common consistency function imposes that two or more semantic values have to be equal, nevertheless, our platform is able to deal with any kind of consistency function, thus providing a convenient way to model complex

relationships. For instance, in a typical heterogeneous content format scenario, some kind of business logic (e.g., audio transcoding) can be compatible with each type of MIME audio input type (*"audio/*"*).

4.3.1.2 Scoring

Scoring evaluation refers to the process of determining the degree of consistency of semantic values with each other under a certain meaning.

Definition 14. *We define scoring as a function that scores the degree of consistency of two or more semantic values.*

$$f_{score} = [V_{AD}]^n \rightarrow \mathbb{R}, n \geqslant 2$$

4.3.1.3 Substitution

Concrete services are meant to substitute placeholders by playing certain roles. Since each role may be associated with semantic attributes, the *substitution* function is in charge of extracting the service semantic property whose attribute matches with the one of the role. This value is then used to either concretely verify whether consistency rules are satisfied, or to evaluate scoring rules.

Definition 15. *We define substitution as a function*

$$f_{\text{sub}} = [A^R \times S] \rightarrow V_{AD} \cup \emptyset$$

4.3.2 Service and template rules

Rules usually do not tie to a particular service, instead, they are expressed in terms of roles; hence roles allow to abstract and reuse rules across services.

Indeed, each service willing to play a specific role must satisfy each rule that involves such roles.

Rule definitions given in the previous section are extremely generic; in this section we refine their definition and we identify significant subsets of both *consistency and scoring rules.*

Service rules bind a semantic attribute of a candidate service to a concrete semantic value; hence service rules constrain the choice of a single service.

Template rules compare semantic attributes of candidate services to semantic attributes of other candidate services; hence template rules establish relationships among different service candidates.

By following the previous considerations (and by explicitly including consistency and scoring functions), we refine consistency and scoring rules as follows.

Definition 16. *We define SCR as the set of service consistency rules (scr) defined as follows:*

$$SCR := \{scr_r : (a_d^r, v_{ad}, f_{consistency}) | a_d^r \in A_d^r, v_{ad} \in V_{ad}\}$$

A service consistency rule therefore binds a specific attribute of a role to a specific semantic value. Each service willing to play role *r* needs to provide a semantic property whose value is consistent (by verification with a consistency function $f_{consistency}$) with v_{ad}.

Definition 17. We define *TCR as the set of template consistency rules* (*tcr*) *defined as follows:*

$$TCR := \{tcr_{r_1,\ldots,r_m} : (a1_d^{r_1}, \ldots, an_d^{r_m}, f_{consistency}) | a1_d^{r_1}, \ldots, an_d^{r_m} \in A_d\}$$

A template consistency rule therefore binds *n* attributes of *m* roles to each other (m<=n since more attributes of the same role can participate in the rule).

Similarly, we impose the same distinction on scoring rules and we define *service scoring rules* (*ssr*) and *template scoring rules* (*tsr*):

Definition 18. *We define SSR as the set of service scoring rules (ssr) defined as follows:*

$$SSR := \{ssr_r : (a_d^r, v_{ad}, f_{score}) | a_d^r \in A_d^r, v_{ad} \in V_{ad}\}$$

Definition 19. *We define TSR as the set of template scoring rules* (*tsr*) *defined as follows:*

$$TSR := \{tsr_{r_1,\ldots,r_m} : (a1_d^{r_1}, \ldots, an_d^{r_m}, f_{score}) | a1_d^{r_1}, \ldots, an_d^{r_m} \in A_d\}$$

Even though in section 4.2 we modeled rules in a more generic way, in practice from an operational standpoint, we claim that the only interesting rules for a template are the ones defined in Definition 16-19. As a consequence we make the following operational hypothesis.

Hypothesis 1. *Each template declares only service consistency, template consistency, service scoring, and/or template scoring rules.*

$$\forall t \in T, RU_t \subset (SCR \cup TCR \cup SSR \cup TSR)$$

4.3.3 Rule evaluation

In order for a placeholder to be filled with a candidate service, rules related to the placeholder roles (PRR relations) must be evaluated. Consistency rule evaluation determines whether a service (or a set of services) can play the required role(s), whereas scoring rule evaluation determines "how well" the candidate service can play the required role(s).

Definition 20. *We define service rule consistency evaluation as a function that determines whether a given service can play a given role according to a given scr.*

$$eval_{scr}:[SCR\times R\times S]\rightarrow\{0,1\}$$

Specifically, given a service *s*, a role *r,* and an $scr_r:(a_d^r, v_{ad}, f_{consistency})$ consistency evaluation takes place by substituting service *s* to the corresponding roles *r* in the rule, and then by applying the consistency function declared by the rule itself.

$$eval_{scr}(scr_r, r, s)=f_{consistency}(f_{\text{sub}}(a_d^r, s), v_{ad})$$

Definition 21. *We define template rule consistency evaluation as a function that determines whether a given set of services can play a given set of roles according to a given tcr.*

$$eval_{tcr}:[TCR\times[R\times S]^n]\rightarrow\{0,1\}$$

Specifically, given a template consistency rule

$$tcr_{r_1,\dots,r_m}:(a1_d^{r_1},\dots,an_d^{r_m},f_{consistency})$$

and a set of role-service substitutions[1] (r_j ,s_k), $j\in(1,m),k\in(1,p)$ evaluation takes place by substituting services to the corresponding roles in the rule, and the by applying the consistency function declared by the rule itself.

$$eval_{tcr}(tcr_{r_1,..r_m},(r_1,s_1),\dots,(r_m,s_p))=f_{consistency}(f_{\text{sub}}(a1_d^{r_1},s_1),\dots,f_{\text{sub}}(an_d^{r_m},s_p))$$

Definition 22. *We define service rule scoring as a function that evaluates "how well" a given service can play role according to a given ssr.*

$$score_{ssr}:[SSR\times R\times S]\rightarrow\mathbb{R}$$

Specifically, given a service *s*, and a service scoring rule $ssr_r:(a_d^r,v_{ad},f_{score})$ scoring takes place by substituting service *s* to the corresponding roles *r* in the rule, and then by applying the scoring function declared by the rule itself.

$$score_{ssr}(ssr_r,r,s)=f_{score}(f_{\text{sub}}(a_d^r,s),v_{ad})$$

Definition 23. *We define template rule scoring as a function that evaluates "how well" a given set of services can play a given set of roles according to a given tsr.*

$$score_{tsr}:[TSR\times[R\times S]^n]\rightarrow\mathbb{R}$$

1 *Notice that it is allowed that a service plays more than one role.*

Specifically, given a template scoring rule $tsr_{r_1,\ldots,r_m}:(a1_d^{r_1},\ldots,an_d^{r_m},f_{score})$ and a set of role-service substitutions[2] $(r_j,s_k), j\in[1,m], k\in[1,p]$ scoring takes place by substituting services to the corresponding roles in the rule, and then by applying the scoring function declared by the rule itself.

$$score_{tsr}(tsr_{r_1,..r_m},(r_1,s_1),\ldots,(r_m,s_p))=f_{score}(f_{\text{sub}}(a1_d^{r_1},s_1),\ldots,f_{\text{sub}}(an_d^{r_m},s_p))$$

4.3.4 Template reification

We call *template reification* the process of filling each placeholder node in a template with a suitable service. A *reifiable template* is a template whose placeholders can be substituted by at least a set of services that satisfy the following two consistency properties, namely *service consistency* and *template consistency.*

Definition 24. *Service consistency requires that each service willing to replace a placeholder should satisfy all of the service consistency rules associated with each one of the roles associated with the placeholder.*

Given a placeholder p, a template t, and a candidate service (for placeholder p) $c_p\in S$ *c_p is service-consistent for placeholder p iff:*

$\forall r|\exists prr_p\rightarrow\{r\}$,

$\forall scr_r\in RU_t$,

$eval_{scr}(scr_p,r,c_p)=1$

Definition 25. *Template consistency requires that each set of services willing to replace a set of placeholders should satisfy all of the template consistency rules*

2 *Notice that it is allowed that a service plays more than one role.*

associated with each one of the roles associated with each placeholder.

Given a placeholder p, a service candidate service s_p to substitute p, and a set of other candidate services CS, s_p is template consistent in CS iff

$$\forall r | \exists\, prr_p \rightarrow \{r\},$$

$$\forall\, tcr_{r_1,\ldots,r_m} \in RU\, t | \exists r_i = r$$

$$\exists \{s_1, \ldots s_{m-1}\} | s_i \, service\ consistent \text{ in } p_x \in PH_t \forall\, i$$

$$eval_{tcr}(tcr_{r_1,\ldots r_m}, (r_1, s_1), \ldots, (r_i, s_p), \ldots, (r_m, s_{m-1})) = 1$$

Template consistency verifies that a service willing to replace a placeholder can satisfy all of the template consistency rules that involve one (or more) role of the placeholder to be replaced.

Definition 26. *Given a template t and a set of candidate services CS={s_1,...s_n}, template t is reifiable in {s_1,...s_n} iff*

$$\forall\, s_i \in CS$$

s_i is service consistent in $p_i \in PH_t$

s_i is template consistent in {s_1,...s_n}

Each service in *{s_1,...s_n}* can therefore be used to substitute a corresponding placeholder in a way that guarantees satisfaction of all the consistency rules. So, the composition platform can build a concrete workflow out of the template by consistently replacing its abstract placeholders with existing services (set *{s_1,...s_n}*).

Definition 27.*Given a template t that is reifiable in {s_1,...s_n}, we call {s_1,...s_n}* ***reification set****.*

4.3.5 Syntactical consistency

Our model easily and flexibly allows to express syntactical consistency, e.g., to check whether services can interoperate in terms of basic interconnection features such as input/output parameters or pre- and post-conditions satisfaction.

In our approach we define a specific *syntaxDomain* semantic domain to express service input/output features. We also provide a *link consistency rule* (a template consistency rule) in this generic form:

$$lcr := ((producer, syntaxDom, output), (consumer, syntaxDom, input)) \rightarrow \{0,1\}$$

where *producer* and *consumer* are example roles that mark subsequent nodes.

So, basically, a service willing to play the producer role should declare a semantic property *output* (in the semantic domain *syntaxDomain*) whose value is consistent with the value of semantic property *input* of the service willing to play the role of consumer.

To guarantee that each preceding service in a template has output compatible with the following service input, it is sufficient to add a link consistency rule to each adjacent couple of services.

4.3.6 Reification process principles

The reification process is in charge of determining whether a specific template can be reified and by means of which reification set(s), if any.

Different kinds of techniques can be used to determine whether one or more set of services can reify the required template. The most naïve solutions could provide an imperative brute-force-like approach that randomly selects a subset of services (a candidate reification set) and checks whether template t is reifiable in the

candidate reification set (e.g., checks whether template and service consistency rules are satisfied). On the contrary, much smarter solutions might exploit *Artificial Intelligence* techniques (such as first-order logic approaches) to logically determine reification sets out of existing rules and available services. Such techniques could also allow for inference-based or recursive approaches to build a sub-workflow that fulfills a specific task no currently available service is able to satisfy.

Our approach is much more operational and exploits a Constraints Satisfaction approach [70] in order to realize an efficient algorithm that iteratively reduces the set of available candidates by using rules as constraints on suitable services. More precisely, at the first stage of our algorithm we apply service consistency rules to each role associated to each placeholder. This step helps creating finite sets of candidate services (we indicate them as CS_r) for each role.

Once got a finite number of candidate services for each role, template consistency rules now define typical CSP constraints, where roles are variables of the CSP and CS_{rn} is the domain of the n-th variable (role). We therefore adopt CSP solution techniques to further shrink CS_{rn} sets, by eliminating services that can not cooperate with other services according to a certain *tcr*. At the end of this process, each CS_{rn} contains services that can safely play the specified role (r_n). Obviously, if no service satisfies a given role, composition is infeasible.

Finally, to determine whether a suitable composition exists, for each placeholder *p* we determine the set of replaceable services (RS_p) by intersecting CS for each role declared by the placeholder *p*. Again, if any of the RS_p is an empty set, no reification set is available. Otherwise, each service in each RS_p can be used to build a valid reification set.

In case the algorithm detects an infeasible composition (either an empty CS_r or an empty RS_p), it tries to recursively generate sub-compositions that can provide aggregates of services compatible with rules of the initial template. According to scalability requirements, and in order to not provide potentially unmanageable

recursive algorithms, this process is sub-template dependent, and only a limited set of templates is actually available for sub-process composition. As an example, a transcoding template is a template made up of a sequence of placeholders, each one forcing link consistency with preceding/following placeholders; moreover, first/last placeholders need to enforce link consistency with the preceding/following placeholders in the initial template.

4.3.7 Workflow ranking

Template reification process allows to translate user requirements into concrete workflows; as the number of available services grow, more reification sets can satisfy user needs and could be translated into workflows. Scoring rules (both service and template ones) allow to establish metrics to evaluate workflows and eventually rank them in order to automatically provide users with the most suitable composition. Service and template rules can be used to enforce any kind of workflow ranking. One typical example relates to QoS policies enforcement: services specify service metadata to describe features such as average computational time, cost and so on. Template scoring rules can evaluate these values, for instance by simply summing them: workflow with the best value is therefore the preferred candidate the platform suggests to the user.

4.3.8 Usage scenario - template reification

In section 4.2 we showed how user selected a *content aggregation* template and how user choices translated into concrete template placeholders by means of consistency rules. Service composition layer now inspects available services to determine whether services exist whose semantic properties can cope with the specified rules.

The available “generation” *RSSReader* service provides the “RSS” value for

attribute *contentType,* therefore it can play the role *generator* and thus fill in placeholder p2. The same applies to placeholders p3 and p4 and *NewsReader* and *HTMLReader* services. By providing "*aggregation*" as *typology* attribute, the *Aggregator* service is suitable to fill in placeholder p5. Finally, the *SMSSender* service metadata allow *SMSSender* to fill in placeholder p6.

However, suitable link consistency rules enforce syntactical correctness by means of link consistency checks on each couple of adjacent nodes.

The *Aggregator* and *SMSSender* violate link consistency check, since the latter one requires input as plain text but the former one provides an XML content.

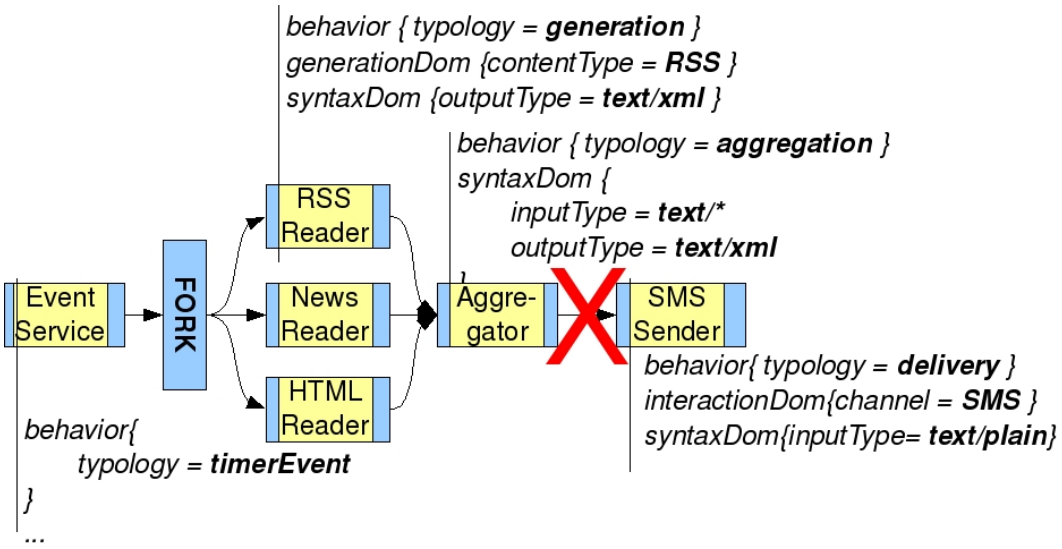


Figure 4. Link consistency violation

The algorithm then detects that no suitable reification set exists; as a consequence, it tries to adopt a recursive approach to *p5* and *p6* nodes. The platform provides an *adaptation template* that is able to transcode data from one format into another. This template is made up of three placeholders, respectively marked with *source*, *transcoder*, and *destination* roles. Finally, template bounds to a single service consistency rule that forces the service willing to play the *transcoder* role to explicitly provide the value *transcoding* for semantic attribute

$typology_{behavior}$.

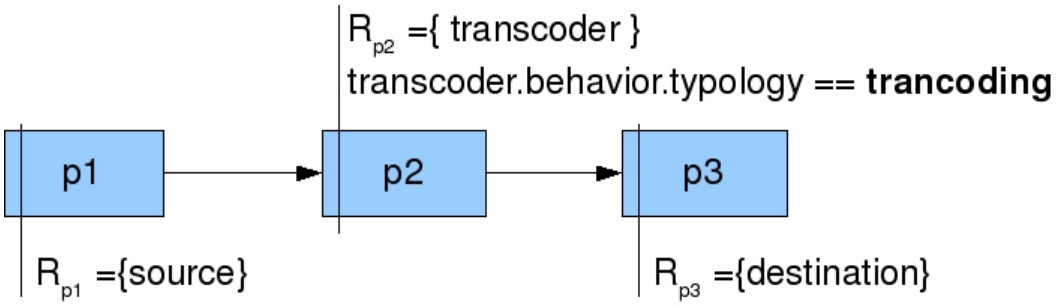


Figure 5. Adaptation template

By recursively applying the adaptation template to the original *content aggregation* one, the service composition layer is able to determine that a convenient *XML-to-text* transcoding service is available.

The composition layer therefore arranges the final workflow as reported in Figure 6.

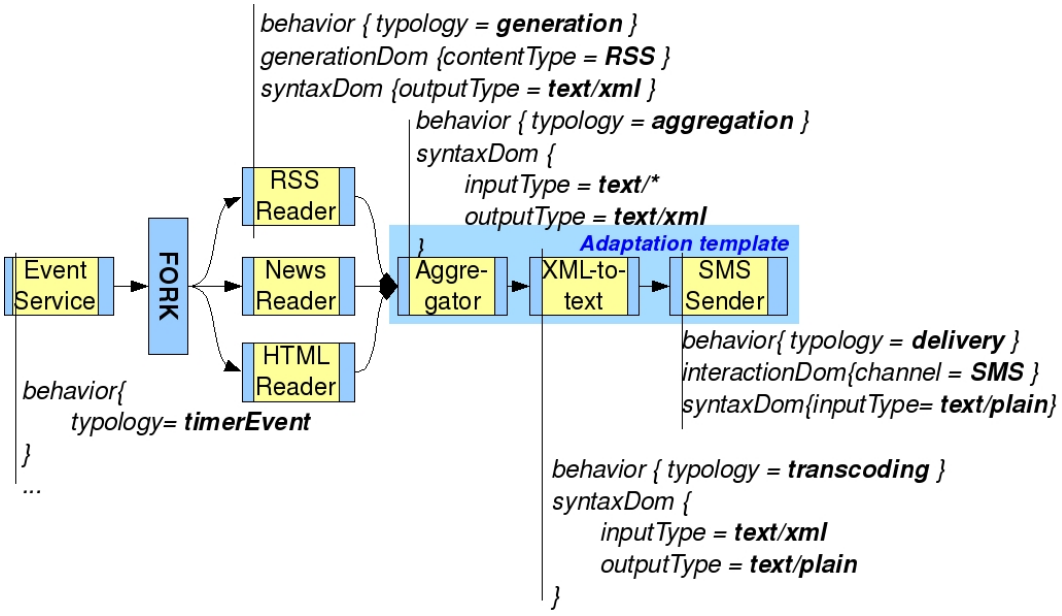


Figure 6. Final workflow

5 Kernel support features

This section describes kernel-level features of our middleware. These features range from basic persistence and naming support to more complex workflow management and execution or user context management. However, no matter their degree of complexity, functionalities of this layer all act as support facilities that can be exploited by other entities in the platform, both at the applicative level (services) and at the kernel level itself (i.e., by other kernel components).

5.1 Service composition

The *Service Composition Engine* is the concrete kernel component that manages the composition model and the reification process described in section 4. The main tasks of the Composition engine component relate to gathering user requirements (e.g., by means of a convenient graphical user interface), and to translating them into concrete workflows (if possible), by searching the service catalogue for currently available services.

5.2 Service support

As stated in the previous chapter, each service provides a semantic description to help the composition engine translate user requirements into concrete workflows. However, a part from semantic high-level descriptions, concrete service entities feature a lot more interesting operational details that our platform almost completely hides to final users.

Specifically, two main aspects need to be taken into account when concretely managing services from an operational standpoint:

- services seldom are *stateless* entities that operate with no side-effects on external resources, no notion of status or no need for pre- or post-conditions check;
- services seldom operate only on single input parameter (e.g., data coming from previous services in a workflow) and produce a single result; far more frequently, services operate also by exploiting (and by modifying) external resources (e.g., file system, other network resources and external services, and so on).

To overcome these limitations, in our proposal services undergo a specific *lifecycle* that manages service status across executions and they exploit a *resource mapping management* to bind execution to external environment.

To convey information about lifecycle and resource mapping, in a typical SoC style, services provide an *operational interface* that provides the platform with useful configuration information.

5.2.1 Service lifecycle management

When executing standalone or within a workflow, services rarely are stateless; instead, some notion of state between executions may be required. The most basic

notion of state we support relates to configuration and de-configuration of services. Specifically, each service instance can be configured once (at the beginning of its lifecycle) and keep this configuration across all executions of the workflow. More complex and different notions of state could apply to services: as an example, some could exploit state for optimization purposes (e.g., to cache previous executions in order to reason on and/or speedup subsequent ones). However, in our opinion these kinds of state are extremely application-dependent and therefore should be mastered by the service logic itself.

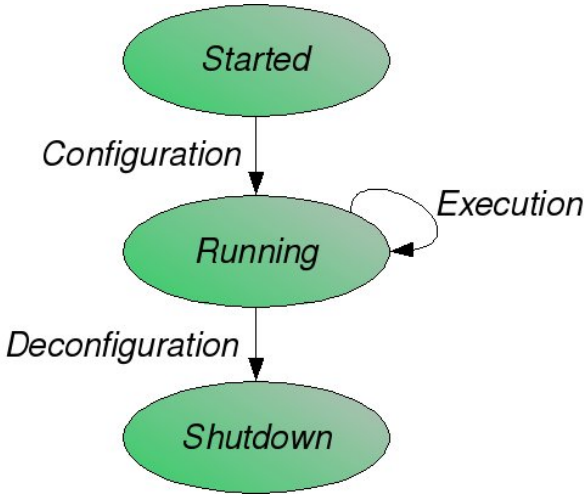


Figure 7. Service lifecycle

In our platform, service lifecycle undergoes three main states:

- a *started* state: services are in this state immediately after the template reification process; at this stage, an instance of the service has been put into the workflow. In this phase, the service composition engine can call suitable service functionalities (*configuration* activity) to configure initial parameters and/or resources of services (e.g., instantiate database connections);
- a *running* state: service instances in this state are configured and fully operational and can be executed each time a workflow needs them; services remain in this state as long as the corresponding workflow exists;

- a *shutdown* state: when a workflow instance is no longer required into the system, it can be removed; before entering the *shutdown* state, some convenient logic (*deconfiguration* activity) can be called to shutdown services (e.g., to deallocate resources).

Figure 7 depicts the main states and the associated transitions.

During transitions between states, service-specific logic can execute to *configure*, concretely *execute* and *deconfigure* services;

```
public interface RSS {
    @Configuration( { "urlToWatch"} )
    public boolean activateChannel(String url);
    @Execution( { "url"} )
    public Object pollChannel(String url );
    @Deconfiguration( { "urlToUnwatch"} )
    public boolean deactivateChannel(String url);
    ...
}
```

Figure 8. Declaring lifecycle methods

To provide this information, services exploit the operational interface and can declare methods of this interface to be *configuration, execution,* and *deconfiguration* methods to specify how to configure, execute and/or deconfigure themselves. None of these methods is strictly required, so, as an example, services declaring only an execution method typically require no configuration/deconfiguration logic.

Figure 8 reports a Java-style code excerpt describing an RSS service operational interface. This service provides a method (*activateChannel*), marked as *Configuration* to initially configure the service (e.g., perform some initial RSS content pre-fetching for performance reasons); another method (*pollChannel*), marked as *Execution* concretely executes service logic (e.g., retrieve novel

content from and RSS feed). Finally, the *deactivateChannel* method, marked as *Deconfiguration,* deconfigures the service (e.g., frees cache from pre-fetched content).

5.2.2 Operational parameter mapping

During their lifecycle, services can exploit different resources (other than parameters received by preceding services in the workflow) to both configure/deconfigure themselves and to execute their business logic.

The needed resources are declared as parameters of service lifecycle methods of the operational interface. The platform is responsible of dynamically binding each parameter to a concrete resource.

To perform this task, the platform needs to determine which element in the platform is concretely *responsible* of parameter resolution and, possibly, how to automatically *choose* the correct parameter value when the responsible component can not provide an unambiguous one (e.g., in case more valid values are available).

5.2.2.1 Parameter mapping - responsibility

Some relevant built-in responsibility levels are:

- *ServiceInstance* responsibility: parameters of this kind are determined on a per-service-instance base;
- *Session* responsibility: a session-level parameter is currently stored and managed by the SessionManager component;
- *UserProfile* responsibility: a user-profile-level parameter is currently stored and managed by the *UserProxy* component;
- *ExecutionContext* state: an execution-context level parameter comes from previous stages in the workflow execution;
- *CoreProperties* state: parameters of this level reference system-wide

defined properties, such as deployment network nodes IP address.

Our platform is responsible for parameter resolution, hence to runtime determine the current value of the declared parameter by querying the specific kernel component.

5.2.2.2 Parameter mapping - choice policies

Some of the responsibility levels may not be able to directly provide an unambiguously determined value; a typical example is a parameter that maps to a phone number (e.g., for an SMS delivery service) of a user profile (*UserProfile* responsibility): the corresponding information in the *user proxy* may be bound to more than one phone number, thus impeding automatic selection of a suitable value.

To overcome this and similar limitations, we introduce the notion of *choice policies*: each parameter value can be determined according to a choice policy defined in the operational interface itself.

Relevant choice policies are in the following:

- *AskUser*: the user himself is responsible of resolving the ambiguity, by specifying a value;
- *SuggestUser*: the same as the previous one, but user is prompted with a choice of already available candidate values (if any exist);
- *PickFirst*: the platform autonomously selects the first value out of a list of available ones;
- *Random*: the platform autonomously selects a random value out of a list of available ones.

5.2.2.3 Parameter mapping - example excerpt code

Figure 9 reports an excerpt code of parameter mapping for the previously mentioned *Configuration* method for the RSS service. Since instances of this service can be used by different users, hard-coding the concrete RSS feed from which to retrieve contents is not a suitable approach; on the contrary, users should be able to specify their own configuration. To capture this, the RSS service declares responsibility of the parameter (*urlToWatch*) to be of type *ServiceInstance*, hence, once specified, its value is bound to the service instance.

```
public interface RSS {
   @Configuration( { "urlToWatch"} )
   public boolean activateChannel(
          @Declaration(
              name = "urlToWatch",
              description = "the rss url to monitor",
              responsibility = ResponsibilityLevels.ServiceInstance,
          )
          @MappingStrategy(
              choice = ChoicePolicies.AskUser
          )
          String url,
   );
   ...
}
```

Figure 9. Parameter mapping

However, initially no suitable value exists, and it should be up to the user to specify the RSS feed of interest: to capture this, this parameter is bound to a choice policy of type *AskUser*. This basically drives the service composition engine to explicitly ask user for a suitable value during the process of service and workflow setup (especially during the *configuration* activity).

5.3 Workflow management

The *Workflow Manager* kernel component is in charge of concretely managing workflows of services. This basically entails two main activities, namely *lifecycle management* and *workflow execution*. Following sections will deepen their description.

5.3.1 Workflow lifecycle management

Composition engine is in charge of translating user requirements into concrete workflows made up of available services. Once created, each workflow undergoes a lifecycle made up of three main states:

- *started* state: this is the state of newly created workflows;
- *running* state: workflows in this state are fully functional and can execute;
- *shutdown* state: workflows in this state are no longer runnable and can be deallocated.

Workflows in the *started* state are not ready to run, since services may need to be configured (see previous section). Hence, transition (namely, *workflow configuration*) from *started* state to *running* state requires to run configuration activity of each service of the workflow. Similarly, when a workflow is no longer needed in the system, it can be shut down; to do this, every service should be deconfigured first. Hence, transition (*workflow deconfiguration* activity) from *running* state to *shutdown* state entails invoking the deconfiguration activity on each service that constitutes the workflow.

Figure 10 illustrates the aforementioned states.

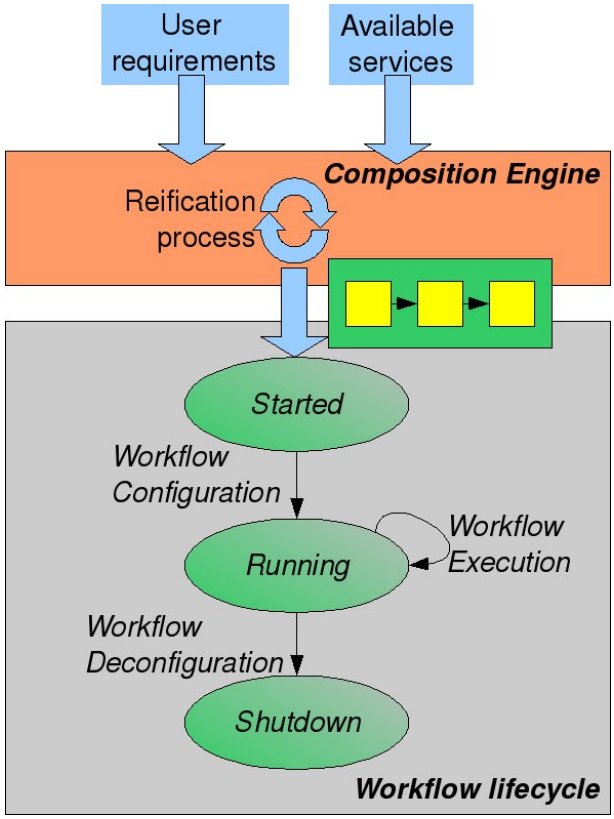


Figure 10. Workflow lifecycle

5.3.2 Workflow Execution

Workflow Manager is in charge of concretely executing workflows. This basically entails invoking *execution* methods of each involved service, coordinating *parameter passing* between subsequent stages (services) of the flow and, finally, *managing execution flow* in case of control blocks (e.g., by choosing the valid branch in a conditional execution flow).

Workflow executions can be triggered by any kind of kernel component; however, in practice, workflow executions happen to fulfill explicit user requests or in reaction to specific events. As a consequence, typically workflow executions are issued by Interaction Managers (for user request) and by the Message Broker (for asynchronous events and messaging) kernel components.

5.4 User proxy

In a typical Ubiquitous scenario, users freely move across different locations and change the devices in use. This poses non-trivial issues when it comes to keeping user-related information (e.g., device in use) consistent, especially when available networks force to frequent disconnections or are limited in terms of bandwidth.

The *user proxy* kernel component is responsible for keeping context information consistent and for suitably reacting to changes. User proxies (one for each user in the system) collaborate with other kernel components in order, for instance, to keep session up-to-date (by interacting with session manager) or to trigger workflow reconfiguration in case context variations invalidated previous ones. A notable piece of context information held by the proxy is the catalogue of the workflows currently *running* for each user.

Finally, to better follow mobile users, user proxies can exploit mobile code techniques (especially Mobile Agent-based ones) to move across network nodes and keep proximity with current user device.

5.5 Interaction management

As Ubiquitous scenarios become more and more complex, users may want to configure rich heterogeneous interaction patterns; specifically, users want to access the system (e.g., require execution of specified workflows) no matter the device and the media channel they exploit. This basically concerns two main aspects that need to be addressed.

On the one hand, it is necessary to physically capture interactions coming from different communication channels, with different data exchange formats. As an example, in order to provide users with a way to request interactions by means of an SMS message, a suitable SMS gateway logic is in charge of physically

receiving messages on a given PSTN network. Since this task is intrinsically channel-dependent and obviously calls for extensibility, the abstraction of service is the most intuitive and viable approach to deal with this issue. We call *Interceptors* specific kinds of services whose main task is to capture user requests coming from different communication channels.

On the other hand, it is necessary to provide an abstraction level that helps deciding which actions to take in response to a specific interaction, no matter the communication channel or format interactions come from. This task is responsibility of the *Interaction Managers* kernel components.

5.5.1 Interceptors

Interceptors 'physically' intercept user requests from specific communication devices and channels (e.g., HTTP, SMS, e-mail, ...) and expressed in a certain channel-dependent syntax. Requests typically contain the following pieces of information:

- *user identity*: any kind of information that helps determining user identity; as an example, for an SMS input channel, the incoming message sender number or a session token for an HTTP request;
- *required action*: information that helps determining what the user wants the platform to do in response to the current interaction;
- *action parameters*: optional parameters of the action to perform.

For instance, along with user sending number, an SMS message containing the text "RSS http://rss.url/... 5" could express the will to obtain the five latest RSS feeds from URL "http://rss.url/...". Each request, along with its syntax indication, is forwarded to the appropriate interaction manager. Some interceptors are also responsible for returning activity results to the users: HTTP interceptors, for instance, are used both to receive an HTTP request and to convey its corresponding response.

5.5.2 Interaction Managers

Interaction managers receive user request messages (from interceptors) along with the indication of the syntax they refer to (so as to determine the suitable request processing algorithm) and are responsible for the following activities:

- they identify users by means of syntax-dependent identification information (e.g., a session cookie for an HTTP channel, the sender phone number for an SMS channel, ...);
- they translate user-friendly information conveyed within requests into a middleware-interpretable command and extract possible parameters;
- they use these pieces of information to enact user requirements such as activating previously configured workflows, performing common predefined middleware tasks, and so on.

Among the others, our platform provides pull-based symmetric (request/response) and pull-based asymmetric (request-only) one-shot interaction managers; the former one returns a result through the same interceptor from which the request came, while the latter one does not return results at all, meaning that request result(s) will be delivered to user through different channel(s) than the request one.

5.6 Session Manager

The *Session Manager* component is in charge of managing session information.

The notion of *session* is generally quite ambiguous and depends on the applicative or technological domain. As an example, HTTP session is an abstraction that captures the status of an ongoing interaction between a Web server and a generic client; once the client first contacts a server, the latter can initiate a session to keep track of what a specific user has done during the interaction with the server (e.g., HTML pages viewed or parameter submitted by means of forms).

Session then can end either by an explicit user action (e.g., logout actions) or by server initiative (e.g., session timeout). Generally speaking, a session captures the notion of an interaction between a user and a service; this is obviously of paramount importance in Ubiquitous Computing scenarios where user interactions are much more heterogeneous and dynamic.

In a typical example, a mobile user begins exploiting a service, e.g., an Instant Messaging (IM) application, by means of her smartphone. When user arrives at office, she would like to seamlessly switch interaction with the IM application to her fixed workstation. Keeping session consistent here requires the user to continue exploiting the application by the fixed workstation without losing previous conversation messages and information about other online users.

In our opinion, no unambiguous and monolithic concept of session can capture the heterogeneity of Ubiquitous Computing applications; on the contrary our SessionManager adopts more different levels of granularity.

User lifetime session level is the coarsest-grained level of session our system supports; as long as our platform "knows" a user (e.g by registration), user lifetime session information is guaranteed to be consistent.

User interaction level holds session information as long as a user explicitly interacts with our system, either by arranging compositions or by exploiting workflows; a typical case relates to users logged into the web-based interface of our platform for composition arrangement.

Workflow lifetime session level holds session information as long as a specific workflow exists; when a workflow gets deallocated, Session Manager discards its corresponding workflow lifetime session information.

Workflow execution session level holds session information during a single executions of a specific workflow; when a workflow execution ends, Session Manager discards its corresponding workflow execution session information.

5.7 Message broker

The *Message Broker* kernel component provides messaging facilities to help coordinate components of the platform via asynchronous message exchange.

We model the Message Broker as a *Publish-Subscribe*[3] [71] messaging component: *publisher* components can produce (e.g., send to the Message Broker) messages for a certain *topic,* whereas *subscriber* components register to the Message Broker for a given topic. Each time a publisher sends a message for a given topic, Message Broker dispatches that message to every subscriber who has previously registered for that specific topic.

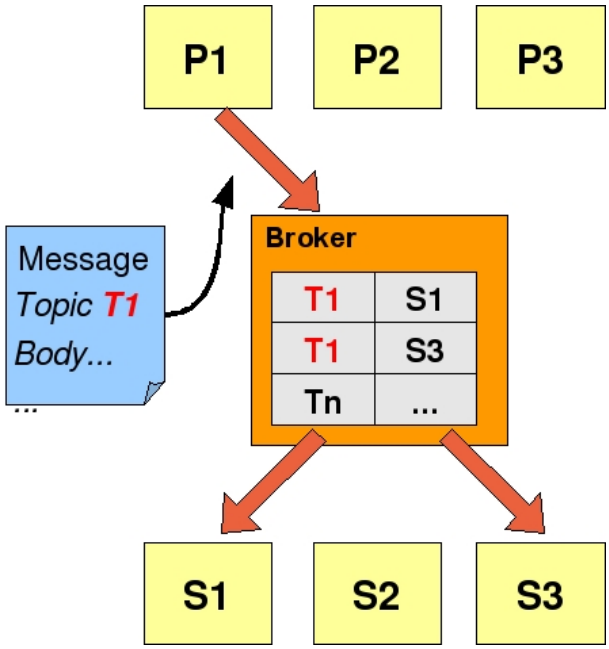


Figure 11. Message Broker behavior

A typical Message Broker scenario relates to inter-workflow interactions. Specifically, workflows typically execute autonomously and separated from each

3 A Publish-Subscribe messaging infrastructure allows for the asynchronous one-to-many exchange of messages between two kinds of entities. *Publishers* produce messages whereas *subscribers* register to the infrastructure in order to get notified each time a publisher produces a message.

other; however, in some cases it is helpful or even necessary to make existing workflows cooperate. For instance, suppose (see Figure 12) a workflow gathers content from different sources (e.g., HTML portals) and dynamically arranges (and publishes to a given website) a personalized homepage for a user. In case other users are interested in variations of user homepage, it would be extremely more efficient to have the first workflow asynchronously (at the end of execution) notify other workflows rather than having them to explicitly monitor homepage content to check for differences.

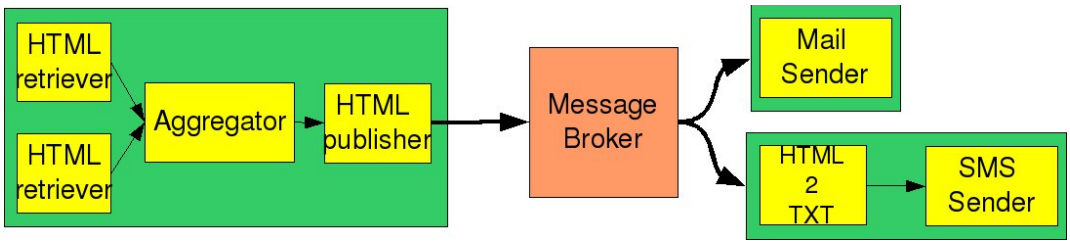


Figure 12. Inter-workflow messaging example

This approach also allows to optimize workflow creation, for instance by splitting monolithic workflows into smaller ones that share common fragments and communicate asynchronously.

5.8 Support features

Our platform provides also other basic features as reported in the following.

The *Service catalogue* component provides basic features to register new services to the platform and to search for them by following different criteria. Search criteria obviously encompass semantic properties, hence providing a sound basis for the composition engine to retrieve suitable services.

The *Naming* component provides a naming system for elements of the platform, so as to facilitate interoperation of platform components.

Finally, *Persistence* component provides the basic layer other components

exploit when they need to persistently store information, e.g., by means of a relational database.

6 Implementation

Implementation of a distributed middleware platform entails a number of non-trivial tasks, that range from realizing and exposing application-driven logic (i.e., the logic middleware is able to provide, e.g., by means of a web interface, to its users or clients) to coping with lower-level details such as performing remote intercommunication between elements of the platform, or managing distributed information storage.

To overcome these issues, in recent years *Application Servers* [44] have emerged as convenient means to help developers realize Web-oriented distributed applications. Application servers usually provide a layered stack of support functionalities that are typically required in building distributed applications.

Figure 13 sketches out the main levels of an application server. At the lowest level, *resource management layer* provides basic features to access to physical or logical resources applications need to access to perform their tasks. Typical examples of resources relate to data stored in databases or file systems. The *application logic* layer is in charge of providing facilities to help develop the concrete business logic; typical examples include for instance management of

remote communications between software elements or naming functionalities. Finally, the *presentation* layer exposes application logic functionalities to final clients via different formats and interaction protocols or styles. As an example, typical human interactions happen by providing a suitable Web interface, whereas machine-to-machine integration can exploit Web Services-related protocols such as SOAP.

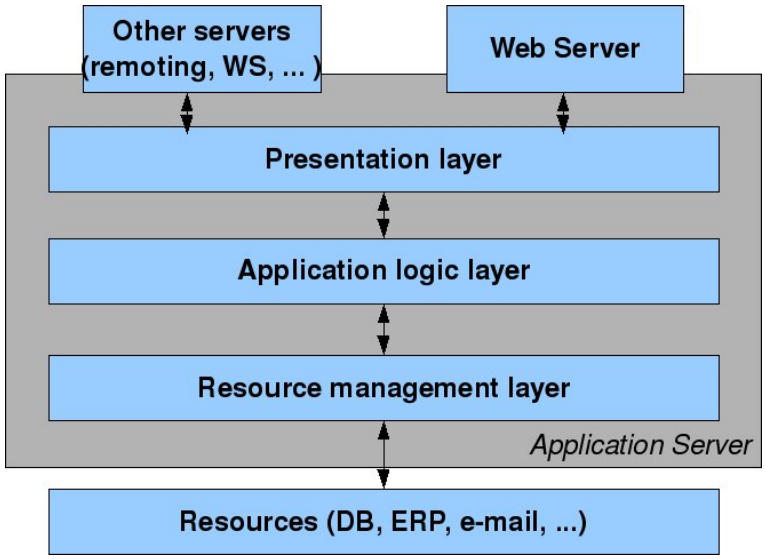


Figure 13. Application server layers

The most preeminent proposals among application servers rely on two main frameworks, the Sun's Java 2 Enterprise Edition (J2EE) [72] and the Microsoft's .NET. Both approaches outline a (similar) set of functionalities that an application server needs to implement in order to provide a useful framework for developers of distributed applications. In a traditional Ubiquitous computing scenario, network nodes where distributed applications (or fragments of distributed applications) reside are inherently heterogeneous and can rely on a plethora of different hardware and/or software configurations. The intrinsic portability of the J2EE platform has naturally led our implementation choices, and we adopted the JBoss [73] open source implementation of this framework to build

our middleware platform.

The J2EE proposal mandates the adoption of several relevant technologies at each layer of the application server. Specifically, the resource management layer relies on the Java DataBase Connectivity (JDBC) [74] functionalities to manage access to databases and in the Java Connector Architecture (JCA) [75] to access generic Enterprise Information System resources (such as ERPs or legacy non-Java applications). The application logic layer proposes standard tools to realize applicative logic; the Enterprise Java Beans (EJB) 3.0 [76] specification is at the heart of this layer and provides a component-oriented framework to build applicative logic in a modular fashion. EJB components live inside an EJB container which provides support features for, as an example, remote inter-component communication (by remote method invocation), component naming and security. Other relevant standards in the application logic layer relate to, for instance, to asynchronous messaging (Java Message Service, namely JMS [8]), transaction management (the Java Transaction API specification, namely JTA [77]) and naming/directory (Java Naming and Directory Interface, namely JNDI [78]). Finally, the presentation layer offers features to help realize client Web interfaces, such as Java Server Pages (JSP) [79] and Java Server Faces (JSF) [80] specifications.

Main kernel components have been realized as J2EE EJB 3.0 components; this approach has extremely fastened realization and deployment times, since the modular approach of the EJB architecture easily allowed us to separate applicative concerns into (relatively) small interacting components.

In the following we deepen the description of some relevant implementation aspects of some kernel components.

6.1 Service layer

In our platform, virtually any kind of business logic can act as an applicative

service, and hence be composed with other services and participate in the fulfillment of user requirements. So, even if realizing service logic by means of Web Services technologies or EJB components is probably the most straightforward way (due to the inherent support the application server provides for these technologies), we do not want to limit our platform to these kinds of technologies, and rather we prefer opening up to a wider landscape of service providers with different implementation technologies.

Another important aspect to take into account relates to the concrete ownership and physical location of service implementations. In the most intuitive case, our platform receives service implementations from service providers and moves them into our platform to manage and run it locally. Even if, to some extent (e.g., for services realized in J2EE-compatible technologies), our platform is able to do that for performance reasons (local invocations are far more efficient than remote invocations), this is not a generally feasible approach. Service implementation logic, in fact, can be strictly resource-dependent (e.g., rely on certain operating system or hardware features) and therefore can need to operate only on a specific network node. In this case, our platform needs a way to transparently manage remote invocations.

The aforementioned requirements pose two main issues; on the one hand, even if realized in any technology, services need to provide middleware with metadata about both operational aspects (the *operational interface*) and semantic (the *semantic metadata* exploited by the composition model); on the other hand our platform needs a way to transparently invoke services, no matter the technology (e.g., programming language, remote communication facilities) they rely upon or their physical location.

6.1.1 Operational and semantic interfaces

We chose to realize both the *operational interface* and the *semantic metadata* as plain Java interfaces, and to exploit Java Annotations to convey additional

metadata on them. As an example, in section 5, we already showed (see Figure 9) excerpts of the operational interface to describe how to identify *activity* methods (i.e., configuration, execution, and deconfiguration), and how to describe parameter responsibility and resolution policies.

```
@Composability(
            domains = "behaviorDomain;generationDomain;syntaxDomain" ;
)
public interface RSSSemantics {
    @ComposabilityAttributes( domainName = "behaviorDomain" )
    public String[] behavior_attributes = {
            "typology=generation"
    };
    @ComposabilityAttributes( domainName = "generationDomain" )
    public String[] generation_attributes = {
            "contentType=RSS"
    };
    @ComposabilityAttributes( domainName = "syntaxDomain" )
    public String[] syntax_attributes = {
            "output=text/xml"
    };
    ...
}
```

Figure 14. Semantic metadata interface

The code excerpt in Figure 14 presents semantic metadata information for an RSS reader service. This service basically conveys semantic metadata properties to tell that its behavior relates to generating content (*behaviorDomain*), that the kind of generated content is of type RSS *(generationDomain),* and that it outputs data in *text/xml* format (*syntaxDomain*).

6.1.2 Service invocation

Since our platform wants to cope with potentially any kind of service, we need a way to transparently manage service invocation, no matter the technology in use.

To overcome this, we propose a *proxied service invocation*: once a workflow needs to request service activities, service invocation is mediated by an *Invoker* component. The Invoker component concretely determines the correct piece of service logic to invoke, and performs invocation by taking into account current location of the service (local or remote) and service implementation details such as realization technology (e.g., Web Services, EJB components and so on).

The *Invoker* interface is reported in Figure 15. Intuitively enough, the *configure* method is in charge of calling the *configuration method* of the corresponding service, and similarly for the *execute* and *deconfigure* methods.

```
public interface Invoker {

    public Object configure(Object[] arg);

    public Object execute(Object[] arg);

    public Object deconfigure(Object[] arg);

}
```

Figure 15. Invoker interface

Invoker proxies obviously heavily depend on both the service logic and its concrete implementation.

However, some cases exist that can greatly help in realizing such invocation proxies. Specifically, if the concrete service implementation comes in the form of a Plain Old Java Object (POJO) or of an EJB component that explicitly extend their operational interface, invoker proxy realization is straightforward, since methods to be called can be autonomously and directly inferred by inspecting operational interface annotations. As an example, in such a case, the invoker *execute* method will always call the service implementation method (either by direct method call or by remote method invocation) that implements the one marked as *@Execute* in the service operational interface.

A unique and fixed implementation based on code reflection[4] techniques [81]

4 *Code reflection refers to the practice of software that is able to reason on and inspect itself at runtime.*

could inspect (each time a service gets invoked) service implementation classes to dynamically determine the concrete method to invoke. However, reflection-based techniques have proven to be extremely heavyweight and resource consuming, and hence are not a practically viable solution. In our implementation, on the contrary, we adopt bytecode generation techniques[5] and, though still initially exploiting reflection techniques to determine the needed concrete methods, we generate invocation proxies (Java classes) with hard-coded logic to invoke suitable methods. This kind of invocation proxy can be automatically generated by the platform either *dynamically* when in need (e.g., at first service invocation) or *proactively* (for performance reasons) at service deploy time. We also provide similar dynamically-generated proxies that can cope with Web Services-based service implementations.

Finally, to cope with any kind of service, we allow service providers to supply also a suitable *custom invoker* (a simple Java class that implements the aforementioned interface) along with a service registration.

6.1.3 Service provider standpoint

According to the previous sections, a service provider willing to register services to our platform needs to provide the service catalogue with a package of information as follows (see Figure 16).

The *operational interface* and the *semantic metadata interface* are required, since they provide necessary information about composability and enactment of the service. The invoker proxy, on the contrary is required only for services whose logic is implemented in technologies other than the ones currently supported (so far, as stated before, POJOs, EJBs and Web Services).

This layered approach allows to clearly distinguish and shape different level of

---

5 *Java Bytecode generation techniques allow to on-the-fly and programmatically generate and manipulate Java bytecode from within a Java program. A typical example tool is the ObjectWeb's ASM code manipulation framework [82, 83].*

required service provider skill; the *semantic metadata interface* and *operational interface* provide high- and mid- abstraction level information about service composability (e.g., the semantic metadata of our composition model or operational details of the service); in our preliminary experiments, average service providers are usually able to easily express semantics and operational details on the produced services by following platform conventions (as can be seen in previous code excerpts). Realizing a custom service invoker, on the contrary, is a more challenging task, since it requires to have some knowledge also of the component-oriented model (EJB) our platform exploits to realize and manage invokers.

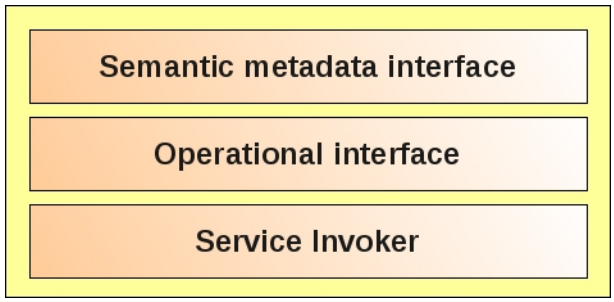


Figure 16. Service description layers

6.2 Kernel layer

Kernel layer components are realized by exploiting J2EE support features; specifically, component logic is generally implemented as EJB 3.0 components. Other relevant kernel components exploit more specific support features.

The Message Broker component exploits the JMS implementation provided by the JBoss application server; JMS provides native support to both point-to-point and to publish-subscribe asynchronous messaging, respectively by exploiting the notion of *Message Queues* and *Message Topics*. Message queues realize an end-to-end communication where a message sender asynchronously sends messages to an

endpoint of a *queue* (a JMS infrastructural object) and the receiving endpoint extracts messages from the other endpoint. *Message topics* allow for a publish-subscribe interaction between a message producer and one or more message consumers. Each *topic* (a JMS infrastructural object) should be used to send/receive specific classes of messages, e.g., to logically group messages whose content is similar. Figure 17 reports an exemplification of JMS queues and topics.

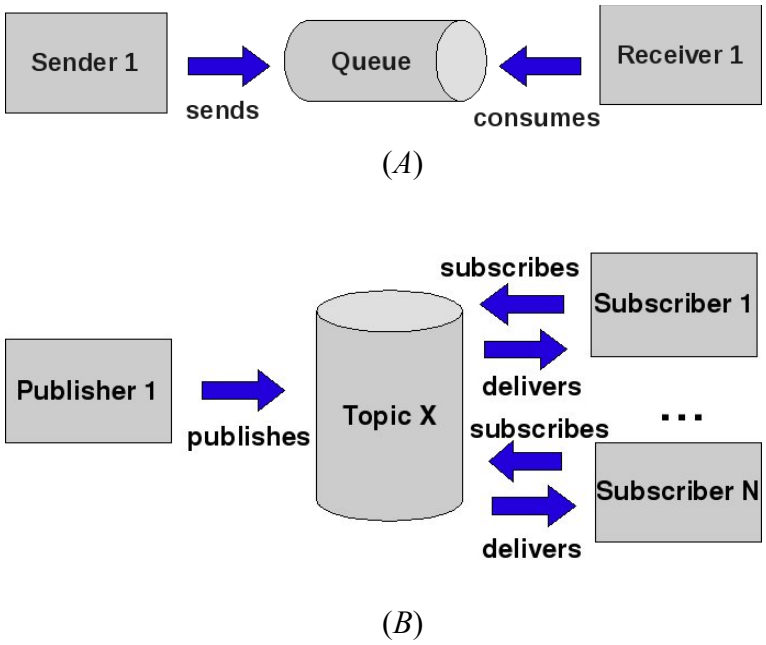


Figure 17. JMS Queue (A) and Topic (B)

The JMS *topic* notion naturally fits the Message Broker topic concept, hence, we realized the Message Broker as an EJB component that manages a set of JMS topics and provides features to instantiate them, register/unregister subscribers to topics and then send messages to a specific topic.

Workflow execution and management relies on the JBoss JBPM [84] tool. This tool provides a widely adopted and acknowledged open source workflow

description and execution engine; the execution engine is a centralized component that can execute and track services starting from a workflow description (typically expressed in the standard BPEL format or in the custom JPDL JBPM language). By exploiting the *Invoker* components, our JBPM engine is capable of executing any kind of service of our platform, either remotely or locally.

Finally, persistence layer exploits JBoss JPA implementation, namely the Hibernate Object-Relational Mapping (ORM) tool [85] and naming system relies on the JNDI Java naming service.

7 Case studies

The platform we have developed covers the discussed core architectural components and can be extended not only in terms of available service implementations, but also with different sets of semantic metadata and with the capability of performing different kinds of compositions (e.g., by adding novel templates). After providing our platform with the knowledge of an initial set of metadata and quite a numerous set of deployable services, we have developed several different use cases, representing the most usual ubiquity scenarios. Examples reported in the following relate to a given set of templates, semantic metadata (attributes, values and service properties) and concrete services we plugged into our platform to realize the following and other analogous scenarios; it is important here to notice that, even if, in our experience, this basic setup has proven to provide a sound basis to realize complex Ubiquitous scenarios, we are able to extend platform capabilities by adding novel service metadata, novel services or novel templates.

In a typical example, one user can access the Internet by means of her personal

smartphone, either by exploiting a slow GPRS connection or a faster WiFi one, and wants to read pages from the RSS campus portal. Furthermore, the college provides a news service (via RSS feed) she is particularly interested in, a shared student calendar with indication of important campus events and a blog service where students can post their considerations about aspects of campus life, music, politics, and so on.

User accesses a convenient service configuration Web interface to express her preferences; specifically, she chooses:

- to receive campus news by SMS messages on her phone as soon as news get published;
- to browse the campus portal by means of her smartphone, hence receiving content adapted to smartphone screen size (e.g., resized HTML pages) and network connectivity (no images on GPRS connection, or full content in WiFi connection);
- to request content from a generic RSS channel by means of an SMS message potentially from any mobile phone and to receive content via both a phone call (with content read by a synthesized voice) on the mobile device she is currently exploiting, and as a mail to her mailbox.

Notice here that, for the sake of simplicity, these scenarios access Web-related contents (specifically, RSS ones); actually, this is not a limitation since our platform is able to retrieve potentially any kind of content via suitable ad-hoc service logic.

The following sections describe each sub-scenario from both a user standpoint (to show the ease of configuration and requirements definition) and the infrastructural one, by showing how concretely the platform reacts to and fulfills user requirements.

7.1 Push-based interaction: *news-on-SMS*

In the following, we describe how user configures her template, by easily specifying service coordination, service choice and user interaction features. These requirements translate into a concrete workflow, whose enactment and runtime behavior is described at the end of this section.

Service coordination logic. User accesses a web interface by means of which she can choose among a catalogue of different templates (templates are described both verbally and graphically). Since she is not interested in complex coordinations of services, she chooses a simple two-stage sequence template whose first node is already marked with role *generator* (see Figure 18). The template provides a link consistency rule so as to enforce consistency between the two nodes.

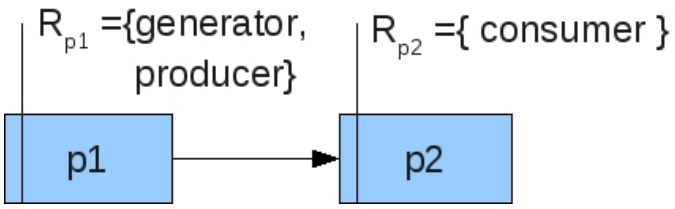


Figure 18. Two-stage sequence template

The template also comes with a rule that binds *typology* attribute (semantic domain *behavior*) to the "*generation"* value and the interaction style to a push-style:

$$generator.behavior.typology = generation$$

$$generator.behavior.interactionStyle = push$$

At a glance, this template easily communicates the user that its main goal is to autonomously (*push-style*) retrieve content from a generic source and to somehow deliver it.

Service choice logic. In order to specify services or contents she is interested in, user selects the *generator* role to have attribute *contentType* (of semantic domain *generationDomain*) to be "*RSS*". This translates to the following service consistency rule:

$$generator.generationDomain.contentType = RSS$$

User interaction logic. In order to specify how to interact with this service aggregation, user marks the first node with the built-in *eventInput* role and the last one as *userOutput* role in order to tell the platform she wants to be notified of the content and to receive it via a given output channel. The web frontend now proposes some choices about semantic features of the input and output nodes; specifically, the *behavior* semantic domain allows to specify high level information about the nature of a service. Hence, user specifies that the *typology* attribute for role *eventInput* must have value *novelContentEvent* in order to tell the system that she wants to be notified when content becomes available (other possible values are, for instance, *timerEvent* to bind to a specific time event or *localizationEvent* in case the user gets localized into a specific area). This choice translates into the following service consistency rule for the template:

$$eventInput.behavior.typology = novelContentEvent$$

Finally, to specify she wants the output to be via SMS messages, she selects the attribute *userOutput.behavior.typology* to be of type *delivery* and the attribute *userOutput.interactionDomain.outputChannel* to be of type *SMS*.

$$userOutput.behavior.typology = delivery$$

$$userOutput.interactionDomain.outputChannel = SMS$$

From now on, it is up to the composition engine to inspect available services and to translate (if possible) user requirements into a concrete workflow.

Template reification. An *RSSPoller* service provides metadata compatible to play both the role of *eventInput* and *generator;* in practice, the *RSSPoller* service is able to inspect a given RSS channel and to generate a suitable event when novel

content is available; content of the generated event is the novel RSS content. Similarly, an *SMSSender* service is compatible with rules on *userOutput* role. However, these services violate link consistency constraint (XML output and plain text input collide) but the composition engine is able to recursively remodel this template by adding an adaptation template, with an *XML-to-TXT* service in between the incompatible nodes.

The final result is a concrete *news-on-SMS* workflow as reported in Figure 19.

Figure 19. News to SMS scenario

The *news-on-SMS* workflow now is registered to the platform and immediately enters the workflow lifecycle; hence, workflow manager starts service configuration. The *RSSPoller* requires the RSS URL to be configured (see Figure 9); it depends on the service instance itself and needs to be configured by asking user for the preferred value. Therefore, the web interface asks the user to enter a suitable URL. The *SMSSender* and *XML-to-TXT* services, on the contrary, have no required configuration.

Runtime behavior. The *RSSPoller* service inherently features a *push-based* behavior, by notifying contents when available, e.g., by means of the Message Broker. Our platform is able to easily and consistently deal with this situation in a twofold way:

- the composition engine has registered the resulting workflow to the Message Broker, so as to trigger workflow execution each time the concrete RSSPoller logic sends messages to the broker;
- the RSSPoller has no concrete *execution* method, hence, when the workflow executes, no concrete logic is associated to the execution of the RSSPoller stage; as usual, the workflow engine is in charge of passing data

(the payload received with the broker message, e.g., novel RSS content) to subsequent stages.

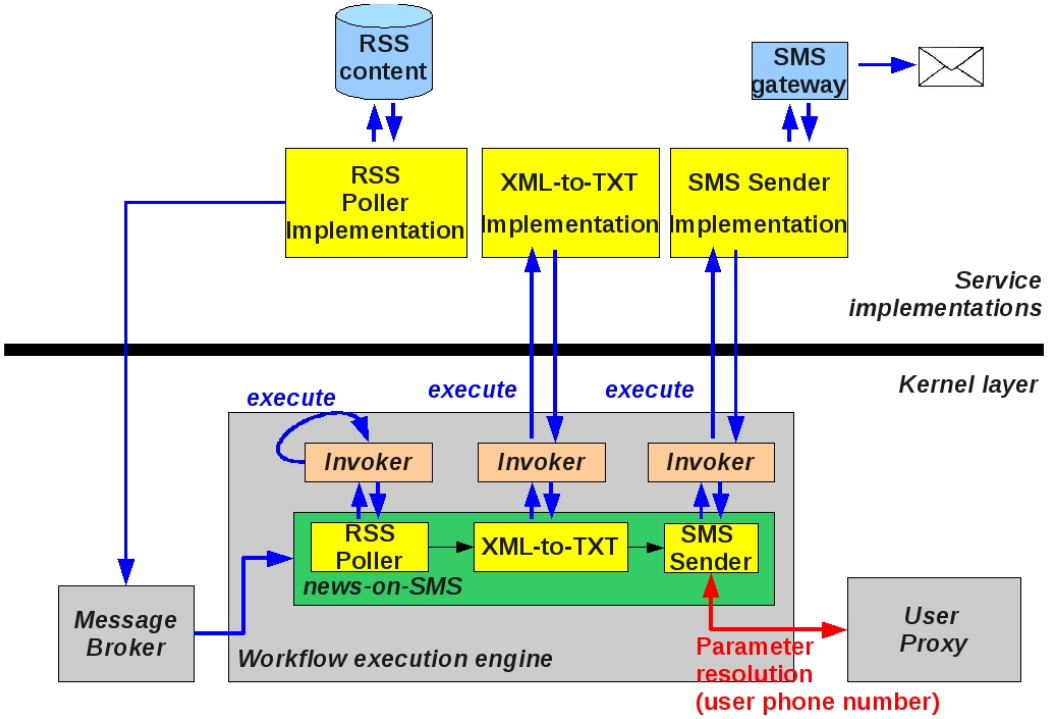


Figure 20. *News-on-SMS* runtime execution

Runtime (see Figure 20), when the RSS poller logic detects a novel content, sends a message to the Message Broker, which notifies all the interested subscribers (in this case the *news-on-SMS* workflow). The workflow execution engine invokes the execute methods of each service after one another (each service takes as input the output of the previous one). Notice here that the RSSPoller, as stated before, has no concrete execution logic since it behaves asynchronously and in a push-style interaction. Finally, the *SMSSender* needs to know current user phone number; since this piece of information is a typical

context information, it is managed by the user proxy. The corresponding parameter in the execution method of the operational interface is therefore mapped to the user proxy (see Figure 21 for an excerpt code).

```
@Execute( { "data", "destinationNumber" } )
public void sendMessage(
    ...
    @Declaration(
            name = "destinationNumber",
            description = "the message fallback destination",
            responsibility = ResponsibilityLevels.UserProfile,
    )
    @Mapping(
            mappedTo=PROFILE_USERDATA_PHONE,
            choice = ChoicePolicies.PickFirst
    )
    String userPhoneNumber,
    ...
```

Figure 21. SMSSender destination number mapping

Hence the parameter resolution process queries the user proxy for this piece of information; the *invoker* is now able to execute the concrete SMS sender logic, and hence to send a message to the correct phone number with the required content.

7.2 Pull-based interaction: *adaptedHTML*

Service coordination logic. User selects an *adaptation* template with four nodes. The second node is marked with a *generator* role and the third one with an *adapter* one. The template also comes bundled with the following rules:

generator.behavior.typology = generation

generator.behavior.interactionStyle = pull

adapter.behavior.typology = adaptation

At a glance, this template easily communicates the user that its main goal is to retrieve content on demand (*pull* interaction style) from a generic source and to adapt it. See Figure 22 for an exemplification.

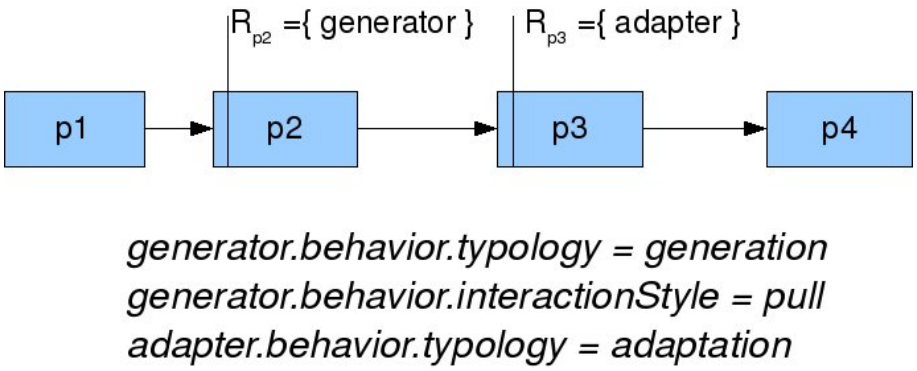


Figure 22. *Adapter* template

Service choice logic. User requires the *generator* to deal with RSS content, and the adaptation to be of type HTML. These requirements translate to the following rules:

generator.generationDomain.contentType = RSS

adapter.adaptationDomain.contentType = HTML

User interaction logic. User marks the first node with the built-in *userInput* role and the last one as *userOutput* role in order to tell the platform she wants to explicitly request the required content in a typical pull-style interaction. Moreover, she wants the interaction to be *symmetric* (e.g., request/response), and the input and output to be on an HTTP channel:

userInput.interactionDomain.interactionType = symmetric

$$userOutput.interactionDomain.outputChannel = HTTP$$

$$userInput.interactionDomain.inputChannel = HTTP$$

Template reification. An *RSSPuller* provides metadata suitable to play the *generator* role whereas an *HTMLAdapter* can play the role of the adaptor.

Finally, an *HTTPInterceptor* service provides metadata compatible to play both the *userInput* and the *userOutput* role. The service composition engine then translates these requirements into the *adaptedHTML* workflow described in Figure 23. Similarly to the previous case study, the RSSPuller requires the user to explicitly configure the RSS URL.

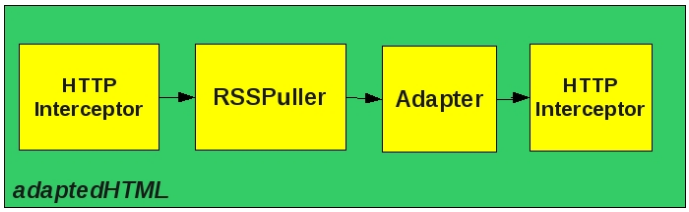


Figure 23. HTML adaptation template

Runtime behavior. The *HTTPInterceptor* service captures user requests on the HTTP channel (from a given device), passes them to the correct interaction manager and waits for the interaction manager to send back workflow result. Upon receiving result of the workflow, the *HTTPInterceptor* can arrange the response to send back to the client device. Similarly to the previous scenario, the adapter service needs to interact with the user proxy to determine the kind of connection the device is currently exploiting: in case of a GPRS connection, the *adapter* service removes images from the HTML content and resizes the page, whereas in case of a WiFi one, the adapter performs only a page resize. Figure 24 exemplifies runtime behavior.

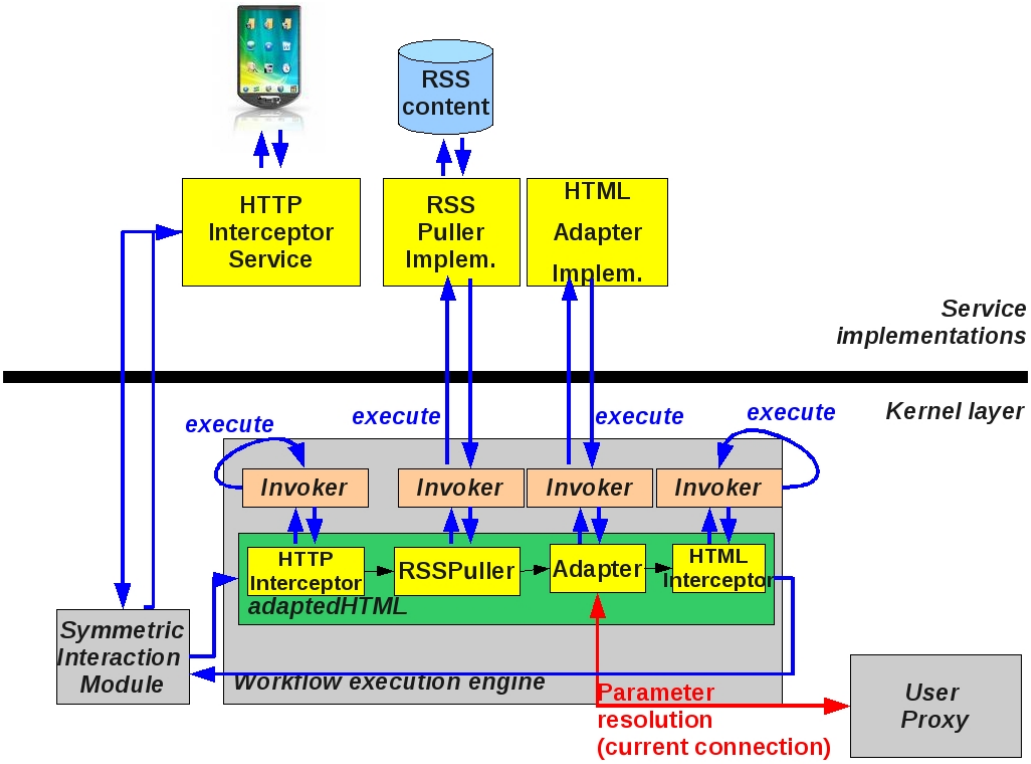


Figure 24. *AdaptedHTML* runtime execution

7.3 Multi-output interaction

In the previous examples, user just needed to configure some already existing templates; in this section we show how a more skilled user is able to configure more complex templates, hence more complex service arrangements.

Service coordination logic. User selects an adaptation template, similar to the one of the previous section. However, she is interested in personalizing it, by adding some novel features. A convenient section of the web interface allows user to reshape this workflow. Specifically, she arranges a novel workflow by inserting a *fork* control block that splits execution in two branches that can execute in

parallel (see Figure 25).

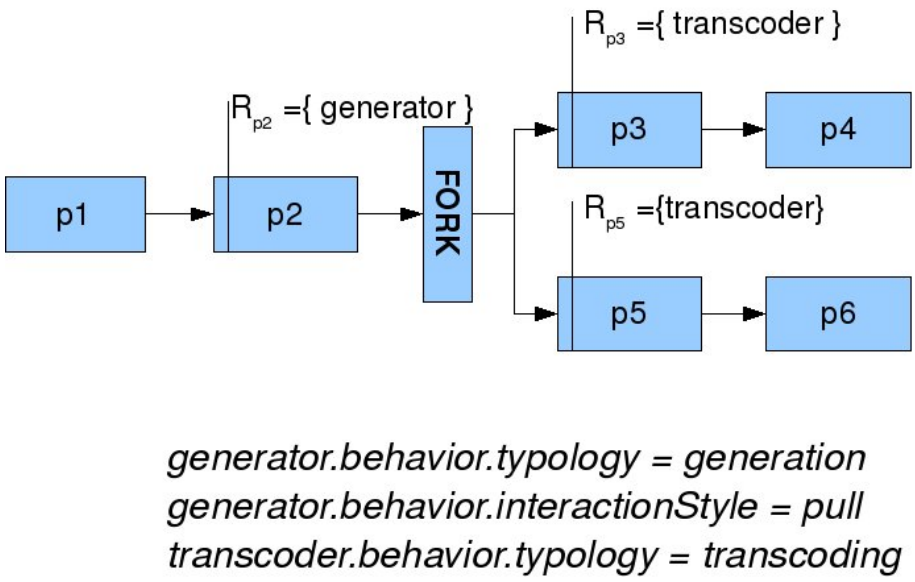


Figure 25. Custom template

User marks placeholders *p3* and *p5* with the *transcoder* role and specify a service rule that binds the transcoder *typology* to be *"transcoding"*, so as to tell the system she wants both nodes to transcode the content coming out of the fork node. Notice here the adoption of roles *(transcoder)* allowed to easily share the rule among different nodes.

Service choice logic. Similarly to the previous case, user requires the content to be of type RSS, hence imposing the following rule

$$generator.generationDomain.contentType = RSS$$

User interaction logic. To determine the required interaction, user marks the first node as having the *userInput* role and nodes *p4* and *p6* with *userOutput* role. These choices enable the usual rules on the typology of services (see previous examples). Similarly, the user selects the input to be of type SMS:

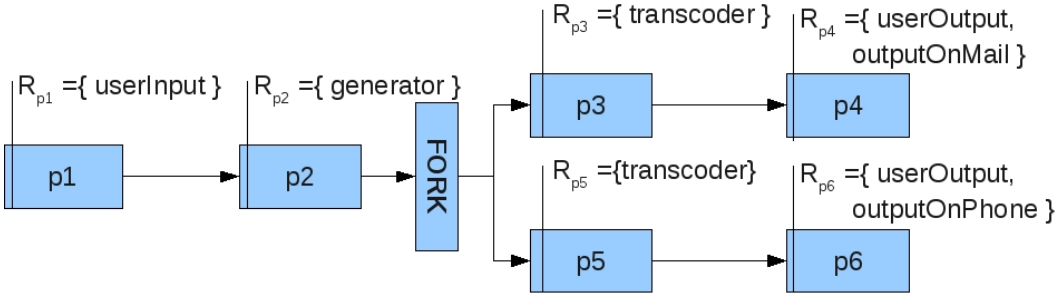
$$userInput.interactionDomain.inputChannel = SMS$$

However, to express semantics on the output channels, user needs to introduce a couple more roles, since specifying a rule over the common *userOutput* role would bind both nodes to the same output type. User then introduces two novel roles (e.g., *outputOnMail* and *outputOnPhone*) and adds the following rules:

$$outputOnMail.interactionDomain.outputChannel = mail$$

$$outputOnPhone.interactionDomain.outputChannel = phoneCall$$

The resulting template is shown in Figure 26.



*outputOnMail.interactionDomain.outputChannel = **mail***
*outputOnPhone.interactionDomain.outputChannel = **phoneCall***
*userInput.interactionDomain.inputChannel = **SMS***
...
generator.geneartionDomain.contentType = RSS
generator.behavior.typology = generation
generator.behavior.interactionStyle = pull
transcoder.behavior.typology = transcoding

Figure 26. Final template

Template reification. The template reification process easily determines services able to fill in placeholders *p1, p2, p4,* and *p6*; namely an *SMSInterceptor* service is able to intercept SMS messages from users, the previously mentioned *RSSPuller* is suitable to extract on-demand content from the RSS channel; an *EmailSender* service is able to send e-mail messages to users and a *PSTNGateway* service places phone calls via a PSTN phone network. Finally, an *RSS-to-Mail* service transcodes the RSS content into suitable HTML content to send via e-mail.

Requirements on placeholder *p5* are satisfied by two different *voice synthesizer* services; the *FreeVoiceSynthesizer* translates plain text (or web content) into an MP3 file with a low *bitrate* but at no fee; the *ProprietaryVoiceSynthesizer* service employs third-party routines that produce better MP3 files (higher bitrate) but requires a fee (Figure 27 reports the semantic metadata interfaces of both services)

```
@Composability(domains = "...transcodingQoSDomain;..." ;)
public interface FreeVoiceSynthesizer {
    ...
    @ComposabilityAttributes( domainName = "transcodingQoSDomain" )
    public String[] qos_attributes = {
           "bitrate=32kbps",
           "fee=0Eur",
    };
    ...
}
@Composability(domains = "...transcodingQoSDomain;..." ;)
public interface ProprietaryVoiceSynthesizer {
    ...
    @ComposabilityAttributes( domainName = "transcodingQoSDomain" )
    public String[] qos_attributes = {
           "bitrate=192kbps",
           "fee=15Eur",
    };
    ...
}
```

Figure 27. Semantic metadata interfaces for voice synthesis services

By means of some service scoring rules associated to the template, the composition engine is able to prompt user with both possibilities; the user selects the concrete workflow that features the *FreeVoiceSynthesizer* service.

The final workflow is depicted in Figure 28.

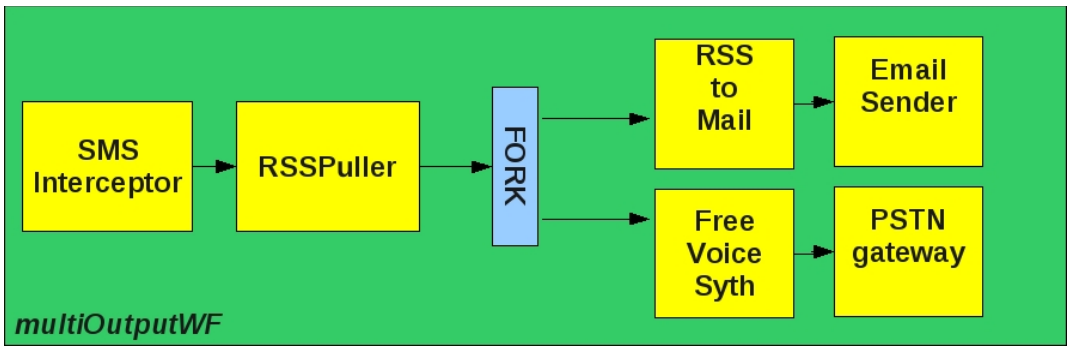


Figure 28. Multiple output custom workflow

Runtime behavior. Runtime, the *SMSInterceptor* service logic captures user requests via SMS messages and forwards them to an *Asymmetric Interaction Module* (this intrinsically is an asymmetric interaction, since the request arrives from an SMS channel and possible responses will be delivered asynchronously via other communication channels). The *Asymmetric Interaction Module* interprets user requests and determines the actions to perform (i.e., the workflow to execute), possible parameters (e.g., the RSS URL) and then enacts the concrete workflow. The *workflow execution engine* executes all the stages, and finally, the *EmailSender* and *PSTNGateway* respectively send a mail to the user and initiate a phone call to the user mobile phone to read the synthesized content. It is important here to notice that the user mail address can be taken via the user proxy (as in previous examples); contrarily, the phone number from which the initial request came is not available as a user profile element (the user may be using the phone number of a friend and attach a personal code to the message in order for the system to identify her). This can be easily realized by means of the Session Manager: the *SMSInterceptor* stores the phone number into the session with *Workflow execution session* granularity (we want the phone number to be valid only for the current execution, since subsequent executions could be activated by SMS messages coming from different phone numbers). Figure 29 exemplifies the described interactions.

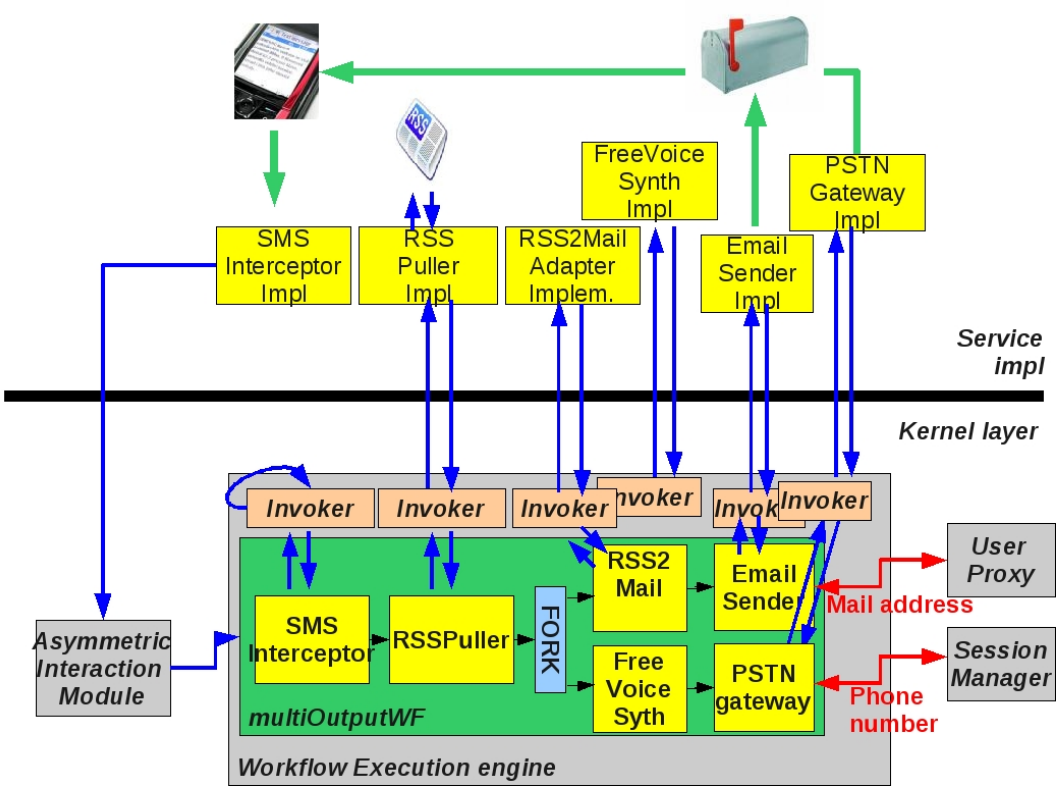


Figure 29. Custom multiple output workflow runtime execution

8 Designing Middleware reconfigurability

Heterogeneity is one of the most relevant characteristics of the Ubiquitous computing landscape: novel services or contents keep becoming available as well as novel communication channels, media formats and portable devices allow to exploit them in different and increasingly personalized ways. Separation of concerns has proven to be a powerful abstraction to tame the complexity of such a scenario; by following this principle, we modeled an extremely versatile middleware platform that is able to support provisioning of contents and services in a wide range of Ubiquitous scenarios. Moreover, since these scenarios are not predetermined and immutable, but rather are likely to rapidly change and grow in number and complexity, our platform is able to extend itself by plugging in novel applicative logic to realize novel scenarios.

However, the intrinsic heterogeneity and the dynamic nature of Ubiquitous scenarios typically have a great impact also on structural and non-functional aspects of the supporting middleware platform itself.

Facing reconfiguration of structural and non-functional aspects of the

middleware is becoming a compelling issue in realizing an efficient support for Ubiquitous computing scenarios. We believe that separation of concerns is, again, the key in designing a Ubiquitous support middleware that is able to reconfigure and adapt both the applicative logic support and its structural non-functional features. In the following, we describe how the adoption of separation of concerns allowed us to extend our proposal and has led to a dynamic and fully reconfigurable architectural solution.

8.1 Related work

This section provides some background concerning reconfiguration of software systems, with special focus on middleware platforms.

8.1.1 Reconfigurable systems

System reaction and adaptation to changes is becoming an acknowledged and challenging task, especially for extremely heterogeneous scenarios such as ubiquity-enabled ones.

Reflective middleware approaches have historically been the forefront of platform solutions to system reconfiguration. They usually rely on a causally connected self representation model that describes characteristics of the reconfigurable application, thus can be used as a basis to decide how to react and reconfigure. The decision on whether to reconfigure is up to a reflection layer which is able to dynamically inspect current status of the application (not surprisingly, by means of language reflection techniques) [86].

In the last years, the autonomic computing initiative [87] (the term autonomic was first coined by IBM as a metaphor to describe systems that behave as autonomously as human autonomic nervous system) has tackled this issue from a broader point of view, by pinpointing four main reconfiguration properties applications need to face. Self-configuration and self-optimization relate to the

capability of reacting to changes in order to reconfigure systems that became invalid or no longer optimized. Self-healing and self-protecting relate to the ability to reactively or proactively take actions to preserve system integrity against changes. Some reference models have been proposed to fully or partially address these issues; one of the most appreciated and adopted models is the IBM MAPE-K loop. This model basically identifies five main tasks of autonomic systems: Monitor and Analyze tasks aim at tracking current component status and at extracting information on whether system reconfiguration is necessary; Plan and Execute stages entail the organization and concrete enactment of reconfiguration tasks. Finally, Knowledge task aims at building and runtime updating a consistent model of both current system features and their evolution, in order to provide a sound basis for reconfiguration analysis and planning stages.

8.1.2 Reconfiguring Ubiquitous middleware

A substantial body of work exists in the domain of middleware platforms for ubiquitous pervasive support scenarios and some recent proposals try to cope with the non trivial task of system reconfiguration by borrowing ideas from the autonomic computing initiative. However, they typically tend to be extremely vertical, by supporting reconfiguration of specific applicative ubiquitous scenarios. As an example, some proposals [88, 89] describe context-aware middleware solutions able to cope with reconfiguration driven by context changes but lack to adapt to changes in user requirements and can only reconfigure the applicative layer. Other proposals [90] extend reconfiguration support to cope with changes in user requirements, but, again, can only reconfigure the application logic and do not tackle the non-functional layer reconfiguration.

From a different perspective, interesting work exists that tries to propose generic purpose (so, not particularly bound to ubiquitous pervasive scenarios) fully reconfigurable (so, both at application and at non-functional layer) middleware models by adopting component-based approaches. The OpenCOM

[91] generic middleware relies on a generic component model with a strong and clear separation of concerns among different layers; however, there is no evidence of concrete deployment and tailoring of such generic model to ubiquitous and pervasive scenarios. Another work [92] proposes a generic component metamodel that tries to support non-functional layer reconfiguration, though it specifically targets mobile environments.

8.2 Design principles

Reconfiguration of both the applicative and the non-functional aspects of ubiquity support platforms is a challenging task and current state of the art solutions only partially tackle the problem. The main issue in dealing with such problems is the inherent heterogeneity of scenarios as well as the diversity of environmental conditions and of user requirements.

We strongly believe that the same architectural principles that, inspired by a separation of concerns approach, guided the design of our architecture, are the basis to refine our model in order to provide a platform that is able to re-adapt from both an applicative and a non-functional standpoint.

8.2.1 Layered architecture

Our proposal strongly promotes a clear separation of concerns and therefore we refine the already proposed layered architecture to explicitly model the non-functional layer; hence, the resulting architecture stack basically features an *applicative layer*, a *non-functional layer* and a very minimal *kernel layer*.

Consistently to the previous architectural model, applicative layer groups all of the ubiquity-related logic, hence it provides content generation and retrieval facilities, as well as service and content adaptation and delivery or user interaction facilities.

Non-functional layer provides basic support facilities the applicative layer needs to exploit; typical examples include workflow execution logic, persistence and naming facilities, or user/device mobility management or communication facilities.

Finally, kernel layer offers basic low level features to enable both applicative and non-functional layer reconfiguration.

8.2.2 Delegating reconfiguration responsibility

To tame the complexity of reconfiguring both applicative and non-functional layer we propose to delegate reconfiguration responsibility: applicative layer reconfiguration is essentially a non-functional feature and as a consequence should reside at the non-functional layer, as well as reconfiguration of the non-functional layer is the lowest level facility our platform provides and therefore resides at the kernel layer.

By following MAPE-K model fundamental ideas, we identify the following main elements of the reconfiguration process.

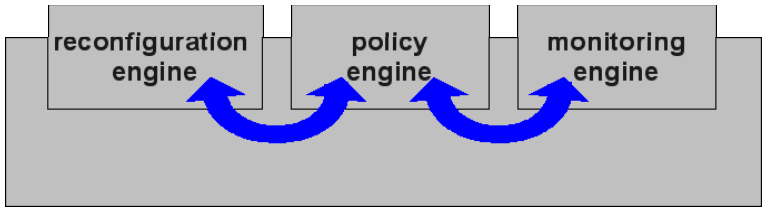


Figure 30. Reconfiguration tasks

As can be seen in Figure 30, basically each reconfiguration layer features a monitoring engine whose aim is to keep track of current status of elements of the (monitored) layer above. The policy engine is responsible to determine both whether reconfiguration needs to take place (by basing on current monitoring results) and how to carry reconfigurations out (e.g., which pieces of logic need to

be substituted or reconfigured). Finally, the reconfiguration enactment engine concretely manages to execute the reconfiguration actions determined by policies.

8.2.3 Decoupling non-functional logic

Ubiquitous pervasive scenarios are inherently extremely dynamic, heterogeneous and ever-growing. To manage the increasing demand of novel features, both the applicative and the non-functional support layer must obey two major requirements. First, they need to promote strong decoupling of business logic into small, manageable and well-defined pieces; second, they need to be dynamically extensible by either plugging in novel features (pieces of business logic) and/or replacing/reconfiguring existing ones. These principles however, need to cope with the inherently different nature of application logic pieces and non-functional logic ones.

Application logic that relates to ubiquitous pervasive scenarios typically presents well-marked isolation and loose coupling characteristics; this is why in previous sections we modeled the applicative layer in a service oriented fashion and let the underlying non-functional support layer provide all of the necessary basic service catalogueing, composition and coordination support features.

On the contrary, non-functional support features are typically much more tightly bound to each other and need to interact in a more autonomous way, without intervention of external coordination entities. As an example, a messaging support layer that needs to make dispatched messages persistent, could directly invoke functionalities of the persistence support layer. Component-oriented approaches [93, 94] have proven to naturally fit this scenario and several proposals have emerged to build generic purpose self-reconfigurable middleware platforms [91]. We therefore model the non-functional support layer as a set of generic software components able to interact with each other in an autonomous manner.

8.3 Architecture

By following the principles sketched in section 4, we propose the architecture represented in Figure 31.

The applicative layer concerns services that model typical ubiquity- and pervasivity-related application logic; common examples are content retrieving services (e.g., news and RSS feed readers, HTML scrapers,...), content adaptation services (e.g., audio/video transcoding modules, vocal synthesizers, ...) or content delivery services (e.g., media streaming servers, SMS gateways, DVB-T carousel servers, HTTP servers, ...). Novel services can be plugged in by need at any time, in a dynamic fashion, to realize novel ubiquity scenarios.

In order to build complex scenarios on top of such basic building blocks, services still need to be aggregated, executed, and managed; since these are inherently non-functional features, we model them as full-fledged non-functional layer components. Reconfiguration of the applicative layer is therefore completely treated and targeted by the non-functional layer and, since components of the non-functional layer themselves can be substituted and/or reconfigured as well, our system is able to easily change, substitute or implement different applicative reconfiguration strategies if in need.

The non-functional layer features the support facilities described in previous sections. In addition, in order to react to environmental or user requirements changes, a *service monitoring engine* observes both services and service compositions, in order to detect anomalies. The *service policy engine* is then in charge of determining whether reconfigurations need to take place by analyzing policies that were provided by the composition engine at composition build time.

Finally, the kernel layer provides coordination facilities to help reconfigure the non-functional layer. The component monitoring engine monitors current state of non-functional components (e.g., by monitoring QoS parameters such as responsiveness, average load and so on) whereas the component policy engine

determines when and which components need to be reconfigured in case environmental or user conditions/requirements change. Finally, the component reconfiguration engine is in charge of concretely enacting component (re)configuration.

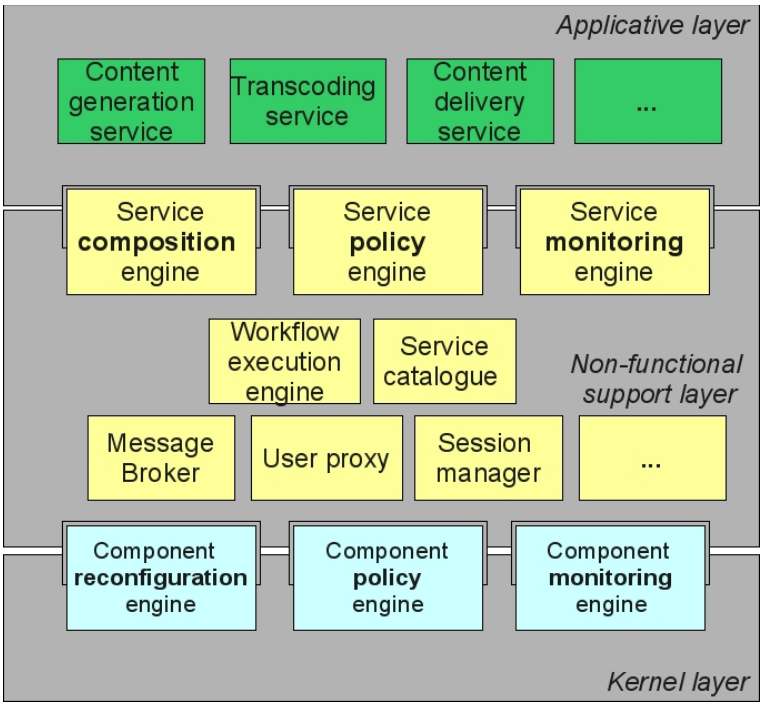


Figure 31. Reconfigurable architecture

8.4 Reconfiguration details

The intrinsically diverse nature of entities at the applicative and non-functional layers (services and components) requires to handle reconfiguration issues in different ways.

8.4.1 Applicative layer reconfiguration

Since workflow management system concretely handles service invocation and management, it can easily be used also to monitor and track service status; typical

examples of monitored properties involve both single-service and overall workflow characteristics such as average execution time or execution counters.

As already explained, the service composition engine is responsible for the crucial task of arranging services into workflows according to user needs, available services and environmental conditions. It relies on a set of *composition rules* that determine whether it is possible and, in case, how to translate user requirements into concrete workflows of currently available services. Such *composition rules* can constrain both semantic and syntactical features of services; for instance, to enforce correct sequences of services, each one operating on the result of the previous one, a rule may constrain the output of a service to be the same format as the input of the following service. Similar composition rules can be used to trigger service or workflow reconfigurations when environmental conditions or user requirements change. Hence, as an example, the aforementioned user proxy middleware components can monitor specific pieces of user context and, in case of changes, can trigger the re-execution of service composition routines.

8.4.2 Non-functional layer reconfiguration

Component-oriented models inherently promote autonomous and spontaneous cooperation and interoperability among components. To realize this, components willing to cooperate need to bind to each other (essentially, to know how to communicate) in more or less decoupled ways (by direct reference, by referencing component interfaces, and so on). This can however become a burden when reconfiguration needs to take place: suppose a component needs to be substituted by another one, in this case references to old component become invalid and need to be substituted for each collaborating component.

The Inversion of Control principle (and its most widespread implementation, the Dependency Injection technique) [95] is a novel approach that delegates component binding and resolution to the execution environment where

components live. Components willing to interact need only to declare interfaces they depend on and it is up to the component container to decide and transparently "inject" (into the declaring component) the component that currently best implements the required interface. Thus, our component reconfiguration engine heavily relies on dependency injection primitives to easily reconfigure component references.

Furthermore, since non-functional components execute autonomously with no external management or coordination (contrarily to the service-oriented approach where a centralized workflow management system is responsible of invoking services), monitoring task becomes a really compelling issue to implement. Typical naïve approaches could in fact require that each component implements its own monitoring logic, with quite obvious limitations to portability and modularity. Monitoring can be seen as a typical concern that cross-cuts several different components, as well as other low-lever features such as security or transaction management. More recent approaches to component-oriented computing solve the issue of modeling and reusing cross-cutting concerns by means of Aspect-oriented Programming (AoP) techniques [96]. Aspects are pieces of business logic that implement a certain cross-cutting concern and are defined outside of any specific component. The aspect management layer allows to programmatically and declaratively "decorate" component activities with as many aspects as needed, and it is in charge of concretely executing them when needed, typically before and/or after component activities themselves. This approach again fosters clear separation of concerns and decoupling principles. Our component monitoring engine thus heavily relies on AoP techniques to dynamically add or remove monitoring logic to managed components; typical examples of monitored features involve for instance method execution average time, persistence layer access statistics and so on. Aspects can be easily shared and reused across non-functional support components and can be automatically re-registered in case of component substitution or reconfiguration.

8.5 Implementation

The Spring framework [97] is becoming more and more largely adopted as a full-fledged component model that natively supports Aspect oriented Programming and Dependency Injection techniques. As a consequence we chose to implement the component reconfiguration engine and other kernel components by exploiting this framework facilities.

However, current implementation of the Spring dependency injection container does not natively support dynamic component addition or removal; as a consequence, we had to adopt the Spring Dynamic Modules extension, that targets this issue by integrating Spring with OSGI framework [98] features for dynamic service/component load/unload.

8.6 Case Study

This section depicts a successful deployment of our platform in a typical ubiquitous scenario: mobile users require notification of traffic news related to a certain urban area each time news get published on a specific traffic portal. Some users prefer getting notified by a phone call with a synthesized voice reading news contents whereas others prefer an SMS message be sent to their mobile phone. As depicted in Figure 32, our platform arranges three different kinds of service workflows, each one realizing a specific portion of the overall scenario.

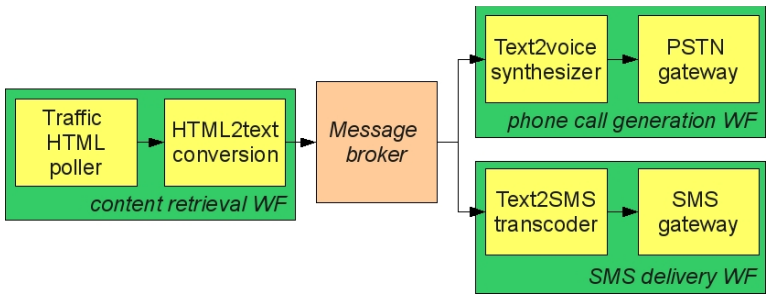


Figure 32. Reconfiguration case study

The first workflow is made up of a couple of services: the first one monitors the specific traffic portal to detect news publishing, whereas the second one extracts plain-text news content from the HTML page. Each time news get published, their plain-text version is made available to other workflows by means of the asynchronous messaging support component. Each user interested in receiving a phone call owns an instance of the phone call generation workflow running within our platform whereas users interested in an SMS message have an associated SMS delivery workflow instance. The phone call generation workflow translates plain-text news into an MP3 content by means of a voice synthesizer service, and then initiates the phone call by means of a PSTN gateway service (e.g., an Asterisk server). Similarly, the SMS delivery service splits plain-text content in trunks of 160 characters and then sends them as SMS messages to the user by means of an SMS gateway service.

As users keep getting registered, hence adding novel workflows, the platform experiences serious performance loss. Specifically, the component monitoring engine layer detects the Message Broker component is becoming a bottleneck and message delivery times are increasing. As a consequence, the component reconfiguration engine substitutes the current costless JMS-based component implementation with a costly but outperforming RTI DDS implementation that can better guarantee near real-time asynchronous messaging delivery. Similarly, the service monitoring engine detects the voice synthesizer is producing MP3 files at an increasingly lower pace. Therefore, the service composition facility reconfigures phone call generation workflow by substituting the synthesizer service with a lower quality one that produces MP3 contents at a consistently lower bitrate but in a shorter time.

9 Conclusions

The Ubiquitous computing scenarios are fostering users to access contents and services in most personalized ways, by exploiting them while moving, via different communication channels and formats, and with any device at hand. Service and content provisioning in the Ubiquitous computing scenarios requires an integration platform support that consistently manages this intrinsic heterogeneity and remains extremely easily accessible by final users.

This research work has investigated the state of the art of this area and has distilled some design guidelines that help in modeling and taming heterogeneity of the target scenarios. Separation of concerns has proven to be the key architectural approach to realize flexible and extensible solutions able to provide the correct abstraction level to final users, hence hiding unnecessary complexity. By following this approach, we designed a middleware architecture to support service and content provisioning in heterogeneous Ubiquitous computing scenarios. Our middleware pushes ubiquity-support applicative logic (e.g., content retrieval, adaptation and delivery, or user interaction management) outside the middleware core, and, by modeling it with the abstraction of services, allows to plug in novel

ubiquity-support features by need. The middleware thus retains only the core crucial tasks of coordinating and composing pieces of applicative logic into more complex aggregates that can easily fulfill user needs.

To assess the viability of our approach we extensively tested and deployed our middleware in a number of different usage scenarios, the most notable of which are reported and described carefully. The encouraging results obtained pushed our research work toward further investigation and extension of our platform with self-managing capabilities to help reconfiguring both the applicative layer and the middleware core functionalities according to Ubiquitous requirements variations. The principles that initially drove our work have proven to be extremely helpful and suitable also in designing strategies and architecture extensions to cope with middleware dynamic self-reconfiguration.

In our vision, in future years different and heterogeneous kinds of middleware support platforms for novel IT integration scenarios will keep permeating and 'disappearing' into everyday user life, just as Ubiquitous devices and wired and wireless connectivities are more and more moving to right now. Users will more and more access a landscape of coordinated, integrated and cooperating services, where the integration middleware becomes just a commodity that fades in the background.

Our work has demonstrated that truly extensible and heterogeneous support middleware platforms also struggling for being usable for end users, should adopt design guidelines intimately inspired by the principle of separation of concerns. In our opinion, these design guidelines will emerge in the next years and will be widely adopted in yet-to-come middleware platforms.

Our future work will certainly explore the adoption of our middleware into other applicative scenarios different from the ones addressed in this work, such as the automotive and Wireless Sensors Networks-based. Content and data fusion and provisioning can probably highly benefit from the adoption of our platform (or similar ones). In addition, we are also eager to challenge our platform with rather

different domains and research areas that stress heterogeneity as a key requirement, such as distributed network monitoring or load balancing and fault tolerance management for distributed applications.

Publications

A. Corradi, P. Bellavista, S. Monti, "Integrating Web Services and Mobile Agent Systems", First International Workshop on Services and Infrastructure for the Ubiquitous and Mobile Internet (SIUMI) (ICDCSW'05), Columbus, Ohio, USA, June 2005.

M. Boari, E. Lodolo, S. Monti, S. Pasini, "Middleware for Automatic Dynamic Reconfiguration of Context-Driven Services", 11th Symposium on Computers and Communications (ISCC'06), IEEE , Pula, Italy, June 2006.

M. Boari, E. Lodolo, S. Monti, S. Pasini , "Progetto SWIMM (Servizi Web Interattivi e Multimodali per la Mobilità). Risultati ed applicazioni ", AICA national conference, Milano-Mantova , Italy, September-October 2007.

A. Corradi, E. Lodolo, S. Monti, S. Pasini, "User-Centric Emergency Management: a Disappearing Middleware Approach", Wireless Rural and Emergency Communications Conference (WRECOM'07), Rome, Italy, October 2007.

M. Boari, E. Lodolo, S. Monti, S. Pasini, "Middleware for Automatic Dynamic Reconfiguration of Context-Driven Services", Microprocessors And Microsystems Journal, Issue 32, pages 145-148, Elsevier, November 2007.

M. Boari, A. Corradi, E. Lodolo, S. Monti, S. Pasini, "Coordination for the Internet of Services: a user-centric approach", 3rd International Conference on Communication System Software and Middleware (COMSWARE '08), Bangalore, India, January 2008.

A. Corradi, E. Lodolo, S. Monti, S. Pasini, "A user-centric composition model for the Internet of Services", 13th Symposium on Computers and Communications (ISCC'08), IEEE, Marrakesh, Morocco, July 2008.

A. Corradi, A. Landini, E. Lodolo, S. Monti, S. Pasini , "Integrating Service Composition with Mobile Code Technologies ", 2nd International Workshop on Distributed Agent-based Retrieval Tools (DART'08), Cagliari, Italy ,

September 2008.

S. Monti, S. Pasini, A. Corradi, E. Lodolo, M. Boari, “An eXtensible middleware for Multichannel, Multimodal, and Multipattern services (X3M)”, 5th International Workshop on Next Generation Networking Middleware (NGNM'08), Samos Island, Greece, September 2008.

A. Corradi, F. Di Marco, S. Monti, S. Pasini, “Facing Crosscutting Concerns in a Middleware for Pervasive Service Composition”, accepted for publication to the 14th Symposium on Computers and Communications (ISCC'09), IEEE, Port El Kantaoui, Tunisia, 2009.

A. Corradi, E. Lodolo, S. Monti, S. Pasini, “Dynamic reconfiguration of middleware for Ubiquitous Computing”, submitted to the 3rd Workshop on Adaptive and DependAble Mobile Ubiquitous Systems (ADAMUS'09), London, England, 2009.

References

[1] P. A.Bernstein. Middleware: a model for distributed system services. Commun. ACM 39, 2 (Feb. 1996), 86-98. DOI= http://doi.acm.org/10.1145/230798.230809, 1996.

[2] A. D. Birrell, B.J. Nelson. Implementing remote procedure calls. ACM Trans. Comput. Syst. 2, 1 (Feb. 1984), 39-59. DOI= http://doi.acm.org/10.1145/2080.357392, 1984.

[3] P. A. Bernstein. Transaction processing monitors. Commun. ACM 33, 11 (Nov. 1990), 75-86. DOI= http://doi.acm.org/10.1145/92755.92767, 1990.

[4] Object Management Group Common Object Services Specification: Atandt/Ncr, Bnr Europe Limited, Digital Equipment Corporation. John Wiley & Sons, Inc., 1994.

[5] K. Boucher, F. Katz. Essential Guide to Object Monitors. John Wiley & Sons, Inc., 1999.

[6] S. Davies, P. Broadhurst. WebSphere MQ V6 Fundamentals. IBM Redbooks publication. ISBN 073849299X, November 2005

[7] Microsoft Corporation. Message Queuing in Windows XP. Microsoft White Paper, 2001.

[8] Sun Microsystems Inc. Java Message Service Specification – version 1.1, 2002.

[9] Reference Model for Service Oriented Architecture 1.0, Committee Specification 1, http://www.oasis-open.org/committees/ download.php/19679/soa-rm-cs.pdf.

[10] W3 Consortium. Web Services Architecture. W3C Working Group Note, http://www.w3.org/TR/wsarch/.

[11] W3 Consortium. SOAP Version 1.2 Part 1. W3C Recommendation, http://www.w3.org/TR/soap12-part1/.

[12] W3 Consortium. Web Services Description Language (WSDL) 1.1. W3C Note, http://www.w3.org/TR/wsdl.

[13] OASIS. UDDI Version 3.0.2. UDDI Committee Specification, http://uddi.org/pubs/uddi_v3.htm.

[14] P.M. Papazoglou, D. Georgakopoulos. Service oriented Computing. Introduction to Commun. ACM 46, 10, pp. 24-28.

DOI= http://doi.acm.org/10.1145/944217.944233, 2003.

[15] M. Weiser. The computer for the 21st century. SIGMOBILE Mob. Comput. Commun. Rev. 3, 3 (Jul. 1999), 3-11. DOI=http://doi.acm.org/10.1145/329124.329126 , 1999.

[16] P. Bellavista, A. Corradi, C. Stefanelli. A mobile agent infrastructure for terminal, user, and resource mobility. Network Operations and Management Symposium, 2000. NOMS 2000. 2000 IEEE/IFIP , vol., no., pp.877-890, 2000

[17] N. M. Karnik, A. R. Tripathi. Design Issues in Mobile Agent Programming Systems. IEEE Concurrency, Vol. 6, No. 3, pp. 52-61, July-Sep. 1998.

[18] P. Bellavista, A. Corradi, C. Stefanelli. Mobile Agent Middleware for Mobile Computing. IEEE Computer, Vol. 34, No. 3, pp. 73-81, Mar. 2001.

[19] P. Bellavista, A. Corradi, L. Foschini, MUM: a middleware for the provisioning of continuous services to mobile users, in: Proceedings of the 9th International Symposium on Computers and Communications, vol. 1, pp. 498–505, June 2004.

[20] R. Cissee, A. Rieger, J. Wohltorf. BerlinTainment – an agent-based serviceware framework for context-aware services. IEEE Communications 43 (6)

(2004) 102–109.

[21] G. D. Abowd, A.K. Dey, P.J. Brown, N. Davies, M. Smith, P. Steggles. Towards a Better Understanding of Context and Context-Awareness. In Proceedings of the 1st international Symposium on Handheld and Ubiquitous Computing (Karlsruhe, Germany, September 27 - 29, 1999). H. Gellersen, Ed. Lecture Notes In Computer Science, vol. 1707. Springer-Verlag, London, 304-307, 1999

[22] B. Schilit, N. Adams, R. Want. Context-Aware Computing Applications. In Proc. of the Workshop on Mobile Computing Systems and Applications (Santa Cruz, CA, Dec. 1994), pp. 85-90, 1994

[23] R. Oppermann, M. Specht. Adaptive support for a mobile museum guide. Proceedings of the Conference on Interactive Applications of Mobile Computing, Rostock, Germany, November 1998.

[24] P. Bellavista, A. Corradi, C. Giannelli. Coupling transparency and visibility: a translucent middleware approach for positioning system integration and management (PoSIM), in: 3rd International Symposium of Wireless Communication Systems, Valencia, Spain, September 2005.

[25] G.D. Abowd, A.K. Dey, D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction Journal 16, 2001.

[26] Commission of the European Communities, eEurope 2005: An information society for all, Available from: <http://ec.europa.eu/information_society/eeurope/2002/news_library/documents/eeurope2005/eeurope2005_en.pdf>, Bruxelles, Belgium, May 28th 2002.

[27] T.K. Shih, T. Wang, C. Chang, T. Kao, D. Hamilton. Ubiquitous eLearning With Multimodal Multimedia Devices. IEEE Transactions on Multimedia, Vol. 9, No. 3, April 2007.

[28] M. Akay, I. Marsic, A. Medl, G. Bu, A System for Medical Consultation and Education Using Multimodal Human/Machine Communication. IEEE Transactions on Information Technology in Biomedicine, Vol. 2, No. 4, December 1998.

[29] R. Sharma, M. Yeasin, N. Krahnstoever, I. Rauschert, G. Cai, I. Brewer, A.M. Maceachren, K. Sengupta. Speech–Gesture Driven Multimodal Interfaces for Crisis Management. Proceedings of the IEEE, Vol. 91, No. 9, September 2003.

[30] W3 Consortium. W3C Multimodal Interaction Framework. W3C Note, http://www.w3.org/TR/mmiframework/, May 2003.

[31] T. V. Raman, G. McCobb, R. A. Hosn. Versatile Multimodal Solutions. The Anatomy of User Interaction. XML Journal, Vol. 4, No. 2, Apr. 2003.

[32] VoiceXML Forum. XHTML + Voice Profile 1.2. VoiceXML 2.0 Recommendation, http://www.voicexml.org/specs/multimodal/x+v/12/spec.html, Mar. 2004.

[33] Opera Software ASA. Opera Multimodal Browser. http://www.opera.com/products/verticals/multimodal/index.dml, 2001.

[34] IBM Corporation. Why IBM? – Leadership in Multimodal. http://www306.ibm.com/software/pervasive/multimodal/, 2006.

[35] IONA Technologies. Using Artix and Service-Oriented Architecture for Multi-Channel Access. http://www.iona.com/devcenter/artix/articles/0304soa.pdf, February 2008.

[36] C. Jefferies, P. Brereton, M. Turner. A Systematic Literature Review of Approaches to Reengineering for Multi-Channel Access. 12th European Conference on Software Maintenance and Reengineering, pp. 258-262, April 2008.

[37] O. Zimmermann, V. Doubrovski, J. Grundler, K. Hogg. Service-oriented architecture and business process choreography in an order management scenario:

rationale, concepts, lessons learned. Companion To the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 301-312, ACM Press, October 2005.

[38] M. Comerio, F. De Paoli, S. Grega, C. Batini, C. Di Francesco, A. Di Pasquale. A service re-design methodology for multi-channel adaptation. Proceedings of the 2nd International Conference on Service Oriented Computing, ICSOC 2004, November 2004.

[39] S. Monti, S. Pasini, A. Corradi, E. Lodolo, M. Boari. An eXtensible middleware for Multichannel, Multimodal, and Multipattern services (X3M). 5th International Workshop on Next Generation Networking Middleware (NGNM'08), Samos Island, Greece, September 2008.

[40] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. König, F. Leymann, R. Müller, G. Pfau, K. Plösser, R. Rangaswamy, A. Rickayzen, M. Rowley, P. Schmidt, I. Trickovic, A. Yiu, and M Zeller, WS-BPEL Extension for People (BPEL4People). version 1.0, 2007.

[41] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. König, F. Leymann, R. Müller, G. Pfau, K. Plösser, R. Rangaswamy, A. Rickayzen, M. Rowley, P. Schmidt, I. Trickovic, A. Yiu, and M Zeller. Web Services Human Task (WSHumanTask). version 1.0, 2007.

[42] N. Russell, W.M.P. van der Aalst. Evaluation of the BPEL4People and WS-HumanTask Extensions to WS-BPEL 2.0 using the Workflow Resource Patterns. Technical report, Queensland University of Technology, Brisbane, 2008.

[43] T. Holmes, H. Tran, U. Zdun, S. Dustdar. Modeling Human Aspects of Business Processes - A View-Based, Model-Driven Approach. European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA), 2008.

[44] G. Alonso, F. Casati, H. Kuno, V. Machiraju. Web Services Concepts, Architectures and Applications. Springer Verlag, ISBN 3-540-44008-9

[45] E. W. Dijkstra. Selected Writings on Computing: A Personal Perspective, Springer-Verlag, ISBN 0–387–90652–5, 1982.

[46] P. Tarr, H. Ossher, W. Harrison, S.M. Sutton. N degrees of separation: multi-dimensional separation of concerns. Software Engineering, 1999. Proceedings of the 1999 International Conference on , vol., no., pp.107-119, May 1999.

[47] D. L. Parnas. On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 12 (Dec. 1972), 1053-1058. DOI= http://doi.acm.org/10.1145/361598.361623, 1972.

[48] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, ISBN-10**:** 0201633612, 2004.

[49] Y. Zhang, B. Xu. A survey of semantic description frameworks for programming languages. SIGPLAN Not. 39, 3, 14-30. DOI= http://doi.acm.org/10.1145/981009.981013, March 2004.

[50] O. Lassila, J. Hendler. Embracing "Web 3.0". IEEE Internet Computing 11, 3 (May. 2007), 90-93. DOI= http://dx.doi.org/10.1109/MIC.2007.52, 2007.

[51] W3 Consortium. Semantic Web Activity. http://www.w3.org/2001/sw/

[52] F. Buschmann, K. Henney, D. C. Schmidt. Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing. John Wiley and Sons, 2007, ISBN 0470059028, 9780470059029.

[53] D. Braga, S. Ceri, F. Daniel, D. Martinenghi. Mashing Up Search Services. Internet Computing, IEEE, vol.12, no.5, pp.16-23, Sept.-Oct. 2008.

[54] Yahoo! Inc. Yahoo Pipes. http://pipes.yahoo.com/pipes/

[55] Intel Corporation. Intel Mask Maker. http://mashmaker.intel.com/web/

[56] Google Inc. Google Mashup Editor. http://editor.googlemashups.com/

[57] J. Yu; B. Benatallah, F. Casati, F. Daniel. Understanding Mashup Development. Internet Computing, IEEE , vol.12, no.5, pp.44-52, Sept.-Oct. 2008.

[58] B. Medjahed, A. Bouguettaya. A Multilevel Composability Model for Semantic Web Services. IEEE Trans. on Knowl. and Data Eng. 17, 7, 954-968. DOI= http://dx.doi.org/10.1109/TKDE.2005.101, July 2005.

[59] J. Robinson, I. Wakeman, T. Owen. Scooby: middleware for service composition in pervasive computing Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing, pp 161-166, 2004.

[60] F. Lécué, E. Silva, L. F. Pires. A Framework for Dynamic Web Services Composition. Whitestein Series in Software Agent Technologies and Autonomic Computing, Emerging Web Services Technology, Volume II, pp 59-75, DOI: 10.1007/978-3-7643-8864-5_5, October 28, 2008.

[61] IBM Corporation. Business Process Execution Language for Web Services version 1.1. http://www.ibm.com/developerworks/library/specification/ws-bpel/

[62] W3C Consortium. Web Service Semantics – WSDL-S – Version 1.0. http://www.w3.org/Submission/WSDL-S/

[63] W3C Consortium. OWL-S: Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S/

[64] C.A. Petri. Kommunikation mit Automaten. PhD thesis, Rheinisch-Westfäisches Institut fur Instrumentelle Mathematik an der Universität Bonn, 1962. In German.

[65] R. Milner. Communication and Concurrency. Prentice Hall, London, 1989.

[66] C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, London, 1985

[67] S. Narayanan, S.A. McIlraith. Simulation, verification and automated composition of web services. In Proceedings of the 11th international Conference on World Wide Web. WWW '02. ACM, New York, NY, 77-88. DOI= http://doi.acm.org/10.1145/511446.511457, 2002

[68] R. Reiter. Knowledge in Action—Logical Foundations for Specifying and

Implementing Dynamical Systems. MIT Press, Massachusetts, MA, 2001.

[69] P. Barthelmess, J. Wainer, Workflow Modeling, Proceedings of 1st CYTED-RITOS International Workshop on Groupware, pp. 1–13. September 1995.

[70] E.Tsang. A Glimpse of Constraint Satisfaction. Artif. Intell. Rev. 13, 3, pp. 215-227. DOI= http://dx.doi.org/10.1023/A:1006558104682, 1999.

[71] P. T. Eugster, P.A. Felber, R. Guerraoui, A. Kermarrec. The many faces of publish/subscribe. ACM Comput. Surv. 35, 2, pp. 114-131. DOI= http://doi.acm.org/10.1145/857076.857078, June 2003.

[72] Sun Microsystems Inc. Java Enterprise Edition http://java.sun.com/javaee/

[73] JBoss Community. JBoss Application Server. http://www.jboss.org/jbossas

[74] Sun Microsystems Inc. The Java Database Connectivity http://java.sun.com/javase/technologies/database/

[75] Sun Microsystems Inc. J2EE Connector Architecture http://java.sun.com/j2ee/connector/

[76] Sun Microsystems Inc. Enterprise JavaBeans – version 3.0 http://java.sun.com/products/ejb/docs.html

[77] Sun Microsystems Inc. Java Transaction API. http://java.sun.com/javaee/technologies/jta/

[78] Sun Microsystems Inc.Java Naming and Directory Interface. http://java.sun.com/products/jndi/

[79] Sun Microsystems Inc. JavaServer Pages Thechnology. http://java.sun.com/products/jsp/

[80] Sun Microsystems Inc. JavaServer Faces Technology. http://java.sun.com/javaee/javaserverfaces/

[81] S. Vinoski. A Time for Reflection. IEEE Internet Computing 9, 1 pp. 86-89. DOI= http://dx.doi.org/10.1109/MIC.2005.3, January 2005.

[82] E. Bruneton, R. Lenglet and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems, November 2002.

[83] E. Kuleshov. Using ASM framework to implement common bytecode transformation patterns. Available at http://aosd.net/2007/program/ industry/index.php

[84] JBoss Community. JBPM. http://www.jboss.org/jbossjbpm

[85] Hibernate framework.http://www.hibernate.org/

[86] F. Kon, F. Costa, G. Blair, R. H. Campbell. The case for reflective middleware. Commun. ACM 45, 6 pp. 33-38. DOI= http://doi.acm.org/10.1145/508448.508470, June 2002.

[87] M. C. Huebscher, J.A. McCann. A survey of autonomic computing—degrees, models, and applications. ACM Comput. Surv. 40, 3, pp. 1-28. DOI= http://doi.acm.org/10.1145/1380584.1380585, August 2008.

[88] D. Preuveneers, Y. Berbers. Adaptive Context Management Using a Component-Based Approach. Lecture Notes in Computer Science, 3543/2005, pp. 14-26. DOI=10.1007/11498094_2, May 2005.

[89] S. S. Yau, F. Karim, Y. Wang, B. Wang, S.K. Gupta. Reconfigurable Context-Sensitive Middleware for Pervasive Computing. IEEE Pervasive Computing 1, 3, pp. 33-40. DOI= http://dx.doi.org/10.1109/MPRV.2002.1037720, July 2002.

[90] J.P. Sousa, V. Poladian, D. Garlan, B. Schmerl, M. Shaw. Task-based adaptation for ubiquitous computing. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on , vol.36, no.3, pp.328-340, May 2006.

[91] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, T. Sivaharan. A generic component model for building systems software. ACM Trans. Comput. Syst. 26, 1, pp. 1-42. DOI= http://doi.acm.org/10.1145/

1328671.1328672, February 2008.

[92] S. Zachariadis, C. Mascolo. The SATIN Component System-A Metamodel for Engineering Adaptable Mobile Systems. IEEE Trans. Softw. Eng. 32, 11, pp. 910-927. DOI= http://dx.doi.org/10.1109/TSE.2006.115, November 2006.

[93] A. Sharma, R. Kumar, P.S. Grover. A Critical Survey of Reusability Aspects for Component-Based Systems. In Proceedings of World Academy of Science, Engineering and Technology, Volume 21, January 2007.

[94] C. Szyperski. Component Software: beyond Object Oriented Programming. New York: ACM Press/ Addison Wesley 1998.

[95] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern. Available at http://martinfowler.com/articles/injection.html.

[96] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, JM Loingtier, J. Irwin, "Aspect Oriented Programming", Proc. 11th European Conf., ObjectOriented Programming, ECOOP'97, pp. 220– 242, Lecture Notes in Computer Science, vol. 1,241, SpringerVerlag, 1997.

[97] Spring framework. http://www.springsource.org/

[98] OSGI Alliance. OSGI Framework http://www.osgi.org/About/Technology

Acknowledgments

First and foremost, I would like to thank my advisors, Maurelio Boari and Antonio Corradi, and my project manager at the Swimm research project, Enrico Lodolo, for their precious guide during these years. They have fostered my work and my growth as a researcher and as a professional with constant attention, providing me with continuous encouragement, advice and support.

I would also like to thank the many friends and colleagues at the Swimm research project, for being extraordinary people and great professionals.

Many thanks are also due to all the people of the Advanced Computer Science Laboratory (LIA) of the University of Bologna, which gave me support and advices.