A Cloud Service for COTS component-based Architectures

Jesús Vallecillos[a], Javier Criado[a], Nicolás Padilla[a], Luis Iribarne[a]

[a] *Applied Computing Group, University of Almeria, Spain*

Abstract

The management of software architectures is an important subject, especially in component-based web user interfaces to enhance their accessibility, dynamism and management at run-time. The Cloud offers some favorable mechanisms for this kind of systems, since it allows us to manage the software remotely, guarantees high availability of the resources and enables us to perform mass-storage. This article presents an infrastructure solution, based on the use of web services and cloud computing, for managing COTS-based architectures.

Keywords:
Cloud Service; Software Architectures; Component-based Systems; COTS

1. Introduction

In general, the software available on the web is increasingly becoming an element that needs to be changed, updated and adapted to the users demands. In some cases, this software is built from components or component-based architectures are used to describe its structure. In both approaches, it is useful for those architectures to be accessible at any moment, dynamic, managed at run-time and adaptable to changes (Bradbury et al., 2004). For this purpose, the use of web services and cloud computing offers a favorable infrastructure, since it allows us to manage the software remotely, it guarantees high availability of resources and it allows mass storage. An example of such architectures are the component-based web interfaces, which do not get out of this necessity and also require to be dynamic and adaptive to the user. With this aim, new projects and proposals have come up in the last few years to build customized web User Interfaces (UI) through the configuration

of widgets that the user wants to visualize (Hoyer and Fischer, 2008). For these applications, the user has normally one Graphical User Interface (GUI) available that can be configured to create some kind of dashboard. This type of interface is built from graphical components of high or medium granularity (that is, they are not simple buttons or text fields) that group together some functionalities related to each other and give rise to mashup applications based on widgets (Yu et al., 2008), such as MyYahoo, Ducksboard or Netvibes projects (Sire et al., 2009).

Within this context, we became interested in the development of an infrastructure for managing component-based software architectures. In particular, our research work is focused on dynamic management of component-based UIs. With this aim, the three pillars on which our proposal is based are: CBSE (*Component-based Software Engineering*), MDE (*Model-Driven Engineering*) and *Cloud Computing*. **CBSE** (Crnkovic and Larsson, 2002) is a software engineering discipline that improves software development by reusing it, contributing reliability, and reducing the time required for creating such software. Contrary to traditional software development, CBSE is focused on integrating previously constructed software components in the construction of the system following a bottom-up development perspective instead of a traditional top-down one. Several component architectures industries have defined their own technologies, such as Sun *Enterprise JavaBeans* (EJB) or Microsoft COM. This concept of reuse and management of components is also present in standards such as IEC/PAS 62814 (Belli, 2013). Our proposal requires that the user interface be defined as a set of components, in which each component of the application represents an individual user interface component. This proposal follows a bottom-up perspective for the building (at run-time) of the structure of the user interface from those GUI components available in one or more third-party repositories. The UI components, in our proposal, are called COTSgets, a combination of the terms COTS (Commercial Off-The-Shelf) (Iribarne et al., 2004) and gadgets (understand as"*gadget*" any software that can work alone or as a piece of architecture).

The second pillar is **Model-Driven Engineering**. This engineering discipline is focused on constructing models on different levels of abstraction, facilitating the software specification, and providing several mechanisms to automate the development of the final product by means of the use of model transformation techniques. Some systems developed with these techniques attempt to provide software with adaptive capacities adapting the models at

run-time, so that their behavior can vary depending on the circumstances that surround their execution, for instance, changes in the user interaction, variation in available resources, different execution platforms, etc. In the particular domain of component-based software systems, the use of MDE techniques can facilitate the design and development of architectures, for example, for defining their structure, the behavior of their components and relationships, their interaction, or their functional and non-functional properties (Crnkovic et al., 2011). Furthermore, manipulation of architectural models at run-time makes it possible to generate different software systems based on the same abstract definition, for example adapting to the user preferences, the component status or the target platform (Bencomo and Blair, 2009). In Figure 1, we can see that our component-based architecture is structured on three levels of abstraction:

- Abstract architectural model, which corresponds to the *Platform Independent Model* (PIM) level in Model Driven Architecture (MDA) (Kleppe et al., 2003), and represents the architecture in terms of what type of components it contains and their relationships.

- Concrete architectural model, which corresponds to the *Platform Specific Model* (PSM) level and describes what concrete components comply with architectural abstract definition.

- Final software architecture, which represents the source code (our components) that will be executed or interpreted.

Thus, the adaptation of the architectures is done based on processes executed on the abstract and concrete architectural levels (Criado et al., 2010) (Iribarne et al., 2010). On the abstract level, *Model-to-Model* (M2M) transformation processes (Czarnecki and Helsen, 2003) are executed to change and adapt the abstract architectural models to the changes in context. However, the concrete architectural models are realized by a trading process (Iribarne et al., 2004), calculating the configurations of concrete components that best meet the abstract definitions. This provides the possibility of generating different software architectures based on the same abstract definition, for example to allow it to be executed on different platforms. The content of this paper focuses only on showing the technological infrastructure used on the concrete level and the final architecture, but not the adaptation performed on either the abstract level (PIM perspective) or the *trading* process that

obtains the concrete architectures (PSM). Similarly, this paper does not discuss synchronization issues between abstract models and final architectures, or how the changes in the models affect the executing architecture.

The third pillar is **Cloud Computing**. The strengths of cloud computing for users and organizations have been widely described in scientific literature, as in (Lee, 2010) or (Whaiduzzaman et al., 2014). The identified benefits include the use of *Software-as-a-Service* (SaaS) and specifically that of *Models-as-a-Service* (MaaS) as a software element of high level abstraction which is available for systems to use at any time on demand. The joint use of MaaS and MDE in turn brings many benefits (Brunelire et al., 2010), highlighting aspects such as the availability of such models, their run-time sharing, improved scalability and distribution, etc. In our proposal, instead of proposing a general use of this concept, our work focuses on the management of software architectures based on our COTSgets components. Therefore, inspired by the use of these components in the use of models as services and as a mechanism for access to these models through web services deployed in the cloud, we have created a cloud service called *COTSgets-as-a-Service*. To provide this service, a cloud infrastructure structured in three layers or levels has been created (Figure 1): the client layer, the platform-dependent server layer and the platform-independent server layer. The client layer is

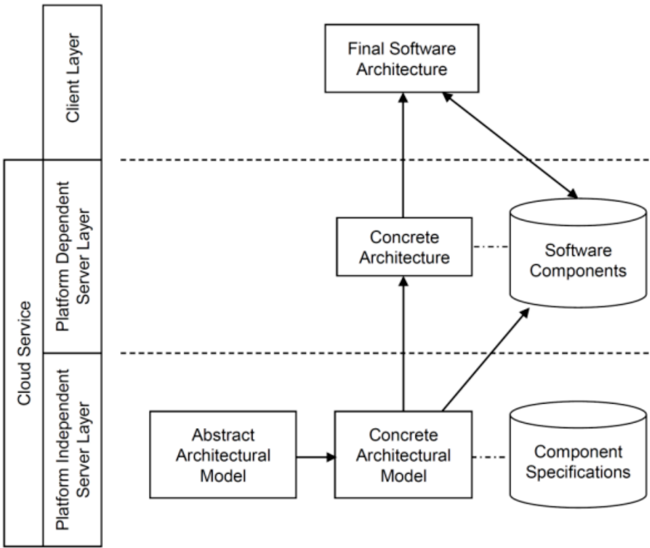

Figure 1: Abstraction levels and layers of our architecture

formed from the user built applications. Therefore, it is formed from the set of components that form the final software architecture shown in the figure. Currently all the developed applications use the web platform and have been implemented using widgets, following the recommendation of W3C, and deployed by means of a web browser. The platform dependent layer deals with providing the client with the required services and interacting with the independent layer, thus obtaining some services from it and providing it with others. Finally, the platform independent layer provides the system services that are valid for all platforms. For this, its functionalities are based only on the description of components and their relationships, regardless of the platform where they will be deployed.

This article is based on a previous research work (Vallecillos et al., 2014) framed in the field of distributed development of information system (Mishra et al., 2012). In this work the system architecture focusing on the three-level data model used in the different layers of our architecture is described. The work presented here in this manuscript focuses on the technological infrastructure, based on web services and cloud computing, which is used for the deployment of component-based architectures.

The rest of the paper is organized as follows: Section 2 shows an example scenario to explain many of the concepts we use in the rest of the article. Section 3 describes our architecture model (using a MDE perspective) based on COTSget components. Section 4 explains how to implement the *COTSgets-as-a-Service* cloud service. Section 5 illustrates the process used to validate and evaluate the Cloud Service and its performance in managing COTSgets-based architectures. Section 6 presents the related work, and finally, Section 7 summarizes the conclusions and discusses the future work.

2. An example scenario

This section will describe a web-technology based application, which will serve as an example scenario to explain many of the concepts used in the rest of the article. This application (Figure 2) has been dynamically constructed from COTSgets components, within a Project of Excellence funded by Junta de Andaluca (ENIA, 2010). The application deals with a query system of Geographic information, allowing us to load visual layers with this type of information. These layers offer data obtained from a set of *Open Geospatial Consortium* (OGC) services provided by the REDIAM (*Environmental Information Network of Andalusia*).

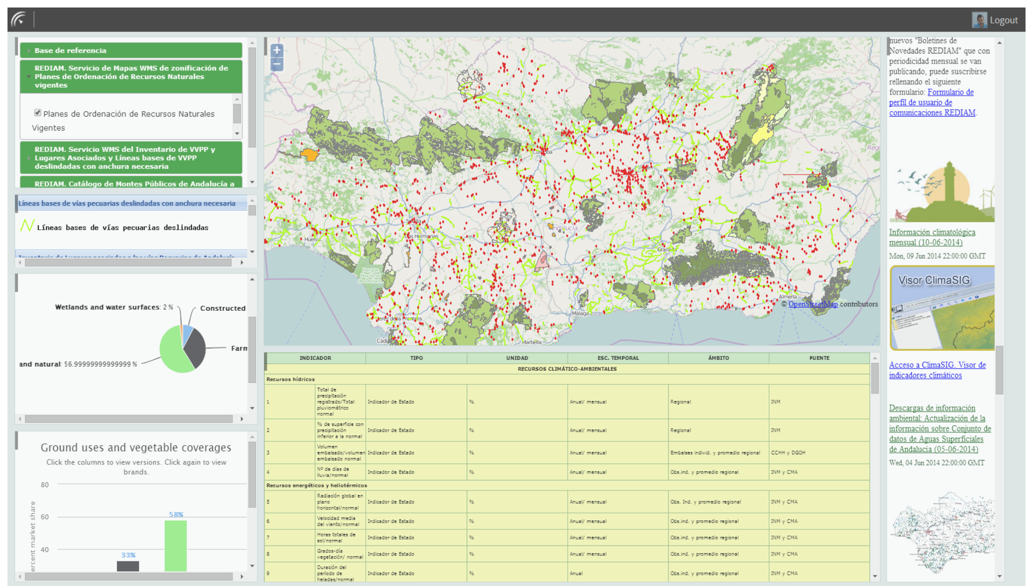

Figure 2: Example of a web application developed with COTSgets component technology

The components in this application are not assembled alone (independently of each other), but rather, as described below, they are related, helping to build complex interactive applications. This application will be examined with each component and the relationships between them described in more detail. Looking at the upper right portion of the image, we can see two components: *UserInfo* and *Logout*. The first is responsible for identifying the user who has connected and showing information about them such as the profile they are currently using in the system. Furthermore, the component *Logout* closes the session of that user in the system. Both components are not isolated from one another. In fact there is a component, called *Header*, that besides controlling the display in the top bar of the user interface, brings together both components. By this grouping, the *Header* component provides management of the access system. Below the menu bar we find components specific to the user profile and their role in the system. One is the *Map* component. This component is responsible for displaying the geographical information found in layers from OGC services. Nevertheless, this component does not have the implemented functionality to allow the user to indicate which layers they wish to be displayed. To do this, this component is related to the *LayerList* component (top left) which provides it with the information.

In addition, the web application displays certain information about the layers being viewed on the map in a certain moment. This activity is performed by the component *Legend* (left center) and as with the *Map* component, obtains the necessary information from the *LayerInfo* component. If we continue examining the figure, we can see that at the bottom we have a report in table format (in the medium) and two graphs (a pie chart and a bar chart) on the left side. Both the report (*ReportResults* component) and the graphs (*Piechart* and *Barchart* components) only handle display information. Another component exists (*TableParser*) which is responsible for generating the information and providing it to where it is needed. Finally on the right there is an external RSS service for REDIAM notices. This service has been encapsulated in a component (called RSS) and is an example of how we can integrate services or external components in our architecture transparently, although with limitations.

As can be seen from the example, it is possible to build web applications where the components can be used for different purposes, e.g. to interact with the user or simply display information, for activities in the background (with no user interaction), to group others and form more complex components, or to integrate external resources (developed by third parties). This diversity of components is analyzed in detail in Section 3.1. All the components that can be used in an application can be described by a set of abstract interface components (Silva, 2001). Figure 3 shows an abstract representation of the web application example scenario shown in Figure 2. In this figure we can see only the components that include user interface. Therefore, the *TableParser* component does not appear. This representation allows us to better understand our component architecture model allowing us to disengage from the visual aspect of the components and work solely with their features. Having described the example scenario, the structure of our component architecture and the relationships that exist between the various elements involved in this architecture will be examined.

3. COTSget-based Architecture Model

This section will describe our architecture model based on COTSget components. For the model we used a design inspired by MDE to build a *Domain-Specific Language* (DSL) of the architecture as can be seen in Figure 4. This representation (which has been described using a metamodel) will help in understanding the different parts that make up the architecture. Looking

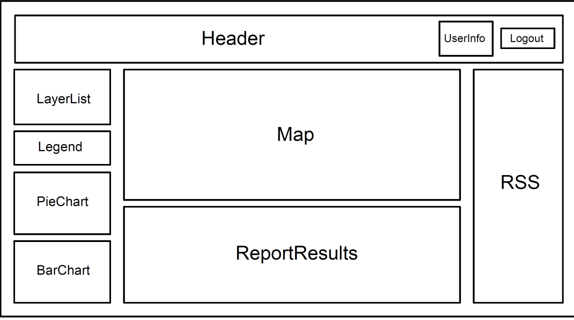

Figure 3: Abstract representation of example scenario

at the figure, it can be seen that our architecture is composed of two types of models: the abstract component architecture models (*AbstractArchitecturalModel*) and concrete component architecture models (*ConcreteArchitecturalModel*). As described in the introduction section, the former identify abstract components, which define the interface; that is, the types of components the interface must include to be considered correct. On the other hand, concrete component architecture models identify concrete components that have been selected as a solution to the types defined in the previous abstract component architecture models. Furthermore, both types of architectural model identify the relationships and links, which can be established between components, as described below. This article focuses on the concrete architecture models as discussed in the introduction. Therefore, hereafter, when we talk about component architecture, we are referring to the architecture of concrete components.

Looking at the figure, we can see that each concrete architecture model consists of a set of individual components (*ConcreteComponent*) and a set of relationships between said components (*Relationship*). Each component has a type, defined by *ComponentType*. The container component type identifies a component that is used to contain other components. This makes it possible to build more complex components from more basic ones. The functional component type is used to construct functional components, which do not include user interaction, therefore they can be built to execute background code (internal code of the component). The *userInteraction* component type is used to build components that include user interaction or simply to display information. Finally, the normal component type is the union of functional and *userInteraction* component types; components that include interaction

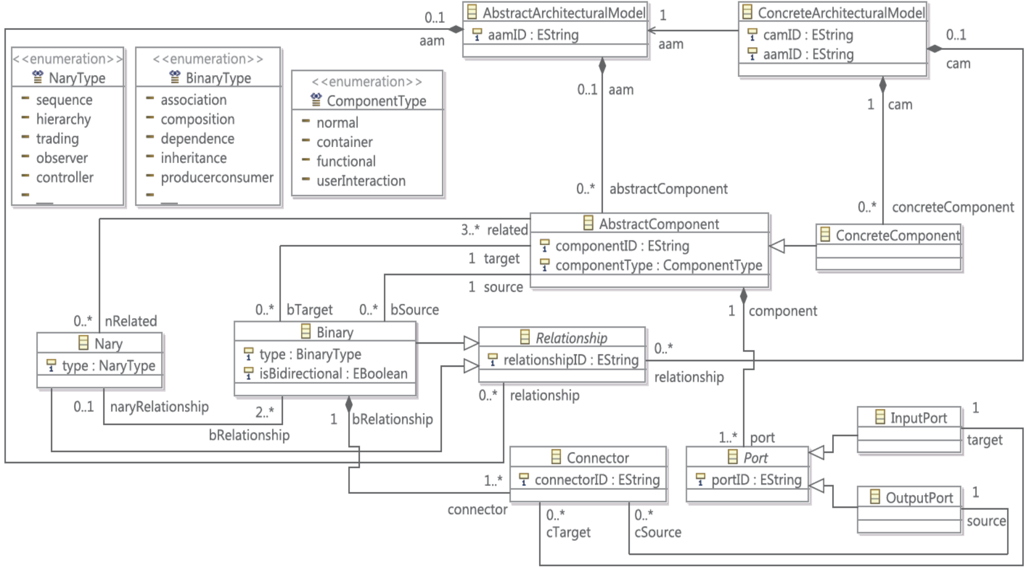

Figure 4: Metamodel of our component-based architecture

with the user and the internal functionality of the component. Through these types we define all the components present in a user interface. Each component must also have at least one port (*Port*) which is responsible for communication between components. In the following section, the types of components and ports associated with them will both be discussed in detail.

Furthermore, relationships (*Relationship*) are modeled between components. Each relationship connects two or more components simultaneously and may be formed by elements of *Connector* type. A *Connector* represents the link that takes information from the output port to the input port. Relationships can be of two types: *Binary* and *Nary*. Binary relationships are those relationships that exist between two different components (e.g. association, composition, etc.). The N-arys relationships are those that are composed of at least two binary relationships and are therefore related to at least three components of the architecture (e.g. hierarchy, sequence, etc.). These relationships will be described in more depth in Section 3.2.

Not all the constraints of our architectural model can be expressed in a metamodel. To do this, a set of OCL (*Object Constraint Language*) constraints has been defined, which help us to formally describe these restrictions and improve our model, giving coherence and reliability. Below, we describe some examples of these restrictions in connection with the relationships be-

tween components. Since our architecture allows only a single relationship between two components, i.e., a single binary relation whose origin is component A and target component B, our first constraint is expressed as follows:

```
context ConcreteArchitecturalModel
 inv:  not(relationship -> exists(r1 :  Binary, r2 :  Binary | r1 <> r2
  and r1.oclIsTypeOf(Binary) and r2.oclIsTypeOf(Binary) and r1.source =
  r1.source and r2.target = r2.target));
```

This constraint is basic in the architectural model and helps us to restrict the number of possibilities between components, making it more manageable and useful. The second constraint indicates that if the relationship between two components A and B is two-way, then there are at least two connectors: one whose origin is A and destination is B, and other connector whose origin is B and destination A. Therefore, the restriction in OCL would be:

```
context Binary
 inv:  isBidirectional = true implies (connector->exists(c1, c2 | c1 <> c2
  and c1.source.component = c2.target.component));
```

That is, at least one output port of the first component must be connected to an input port of the second, and *vice-versa*. Furthermore, it has been established that a component must have at least one port, however the restriction is stronger, since it must have at least one input port. Therefore, the corresponding OCL constraint would be the following:

```
context ConcreteComponent
 inv:  port->exists(p | p.oclIsTypeOf(InputPort));
```

This way we check that in the components collection of ports, there is at least one that is of the *InputPort* type. The following subsections will describe in detail the internal structure of the components and the relationships that can be established between these components, highlighting the structural aspects of these.

3.1. Components

As stated above, our model concrete architecture consists of a set of individual components. Each component has to be specified using a DSL. In Figure 5 we can see part of the metamodel, which describes the internal structure of said components. Each component is composed of four parts: *Functional*, *ExtraFunctional*, *Packaging* and *Marketing*. In the figure only the *Functional* y *ExtraFunctional* parts have been expanded because these are essential for

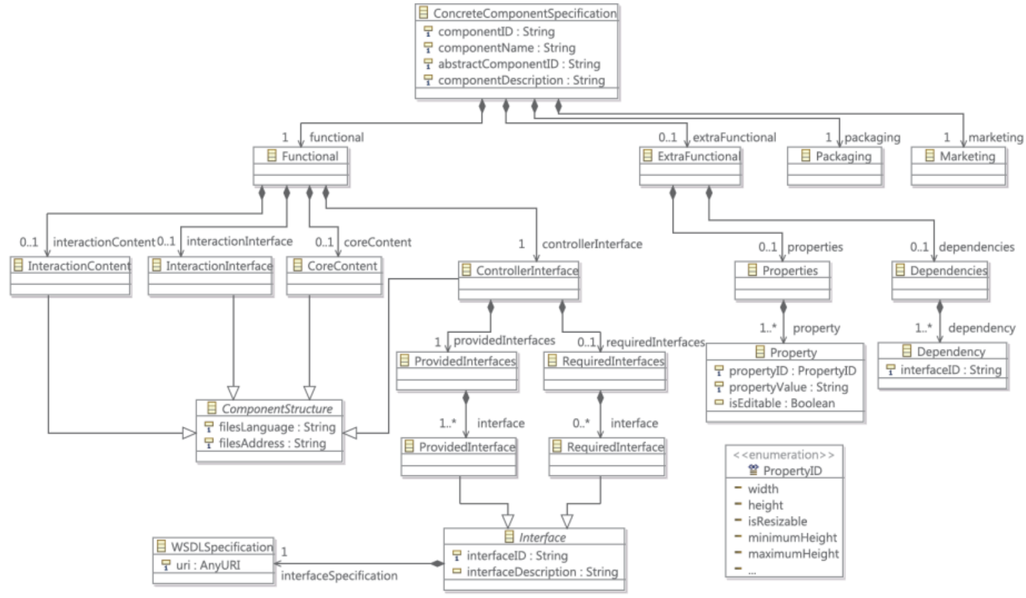

Figure 5: Metamodel of the internal structure of a component

understanding the internal structure of a component. When a component is constructed all parts must be specified except the *ExtraFunctional* part, which is optional. The *Marketing* part of a component identifies the information related to the entity that developed the component, such as the name of the organization, contact name, etc. The *Packaging* part provides information related to the packaging of the component, such as identifying the repository where the component is located, the programming language used, etc. The *ExtraFunctional* part identifies the set of extra-functional properties that a component can have. These properties may provide information on NFPs (*Non-Functional Properties*), properties related to quality of service (QoS), properties related to the appearance of the component such as width, height, etc., and the set of dependencies that a component can have with other components. Finally there is the Functional part which will be described in more detail since it is fundamental to the understanding of the structure of a COTSget component. Each component implements its own functionality, both related to the interaction with the user (*InteractionContent*) and the internal component itself (*CoreContent*). Both features are optional, which allows a developer to build a component to suit their needs. The different types of components are:

- *Functional Component*: defined by the functional component type (see list of types of components defined in *ComponentType* in Figure 4). This component only implements the internal functionality (*CoreContent*), without implementing functionality related to user interaction. It can be used to perform background tasks, such as accessing a database. In our example scenario the *TableParser* component is a functional component that is responsible for generating the information related to the reporting of results and table formatting. This component does not interact with the user or display information on the screen. Once this component generates and formats the data it is sent to the *ReportResults* component to display this information on screen.

- *User Interaction Component*: defined by the *userInteraction* component type. This component only includes the functionality associated with user interaction (*InteractionContent*) and is used to interact with the user. It can also be used to display reports or some graphical information. In our example scenario several components of this type appear. For examplethe *Map component* is used to display a map and interact with it and where layers with information that have been selected are shown in the *LayerList* component.

- *Container Component*: defined by the container component type. This component has neither internal functionality nor functionality related to user interaction. A container is a component that is composed of several components, to jointly develop a common task or purpose. An example of this type is the Header component in our example scenario. This component contains the *UserInfo* and Logout components, and is used to control access to the system. The *Header* component is a good example of how we can use a set of components to create a more complex common task.

In addition to the functionality that can be implemented in a component, the *Functional* part includes the specification of two interfaces: the interaction interface (*InteractionInterface*) and the controller interface (*ControllerInterface*). Through the interaction interface, the user interacts with the component entering data or receiving information. In this interface, user events supported by the component are managed depending on the device used for handling the component. The component developer should provide the setting in which that component is executed (e.g. mobile environment,

desktop, etc.), to develop its functionality according to the specific events of that setting. This feature means that the components are dependent on their setting. On the other hand we have the controller interface, which is used for communication between components, so it is inaccessible to the user. This interface allows the creation of a communications network between components, giving the architecture great versatility. The controller interface is comprised of a set of provided interfaces (*ProvidedInterfaces*) and required (*RequiredInterfaces*), with the stipulation that each component has at least one provided interface. The provided interfaces are those that define all the functionality that the component establishes as visible to the outside world, i.e., it describes methods that can be invoked in order to make the component perform some operation.

The required interfaces of a component describe those operations belonging to other components that are invoked by a component in order to operate completely and correctly. Each interface defined in the controller interface is specified by WSDL (*Web Service Description Language*), as can be seen in the concept WSDLSpecification. This specification uses the concept of *portType*, defined in WSDL 1.1 (`http://www.w3.org/TR/wsdl`), as a root element for describing each of the interfaces. In Figure 6 we can see the specification of a (*PortType*) interface.

Each interface has a name, so each interface within the same component can be referred to univocally, and a set of operations (*Operation*) with which

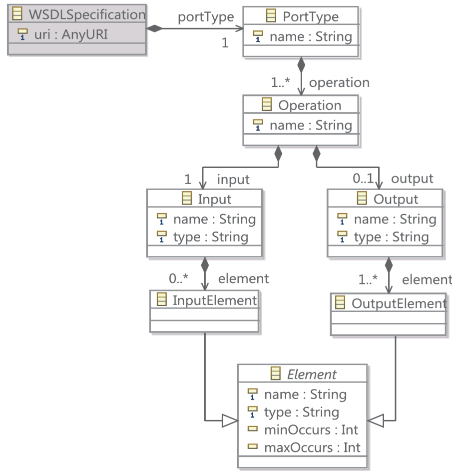

Figure 6: Specification of an interface (portType)

information can be sent (*Output*) or received (*Input*). An interface should always have an input operation and, optionally an output operation. The input operation may or may not be formed by a set of input elements (*InputElement*) while the output operation will consist of one or more output elements (*OutputElement*). Each Element defines its name, type and, optionally, the minimum (*minOccurs*) and maximum (*maxOccurs*) number of times that must be present.

In Figure 7 we can see a fragment of the specification of the interface provided *manageLegend* of the component Legend, from our sample scenario. This interface describes two operations: *loadLegend* and *removeLegend* (lines 44 y 48). The first is a "*request-response*" type, since it defines an input message and an output, while the second is a "*one-way*" type, since it only defines an input message (lines 45-46 y 49-50).

Moreover, the definitions of data that describe the structure of these messages can be seen in lines 9-32. For example, the *loadLegend* operation receives the URL as input of an OGC service followed by a list of the layers

```
1:  <?xml version='1.0' encoding='UTF-8'?>
2:  <wsdl:definitions name="RegisterImplService" targetNamespace=http://ws.cos.acg.ual.es/... >
8:  <wsdl:types>
9:  <xs:schema elementFormDefault="unqualified" targetNamespace=http://ws.cos.acg.ual.es/ version="1.0"
10:  xmlns:tns=http://ws.cos.acg.ual.es/ xmlns:xs="http://www.w3.org/2001/XMLSchema">
    ...
15:     <xs:complexType name="loadLegend">
16:       <xs:sequence>
17:             <xs:element minOccurs="1" maxOccurs="1" name="urlOGCService" type="xs:string"/>
18:             <xs:element minOccurs="1" name="layer" type="xs:string"/>
19:       </xs:sequence>
20:     </xs:complexType>
21:     <xs:complexType name="loadLegendResponse">
22:       <xs:sequence>
23:             <xs:element minOccurs="1" maxOccurs="1"  name="return" type="xs:string"/>
24:           </xs:sequence>
25:     </xs:complexType>
26:     <xs:complexType name="removeLegend">
27:       <xs:sequence>
28:         <xs:element minOccurs="1" maxOccurs="1" name="urlOGCService" type="xs:string"/>
29:             <xs:element minOccurs="1" name="layer" type="xs:string"/>
30:           </xs:sequence>
31:         </xs:complexType>
32: </xs:schema>
33: </wsdl:types>
    ...
43: <wsdl:portType name="manageLegend">
44:   <wsdl:operation name="loadLegend">
45:     <wsdl:input message="tns:loadLegend" name="loadLegend"> </wsdl:input>
46:     <wsdl:output message="tns:loadLegendResponse" name="loadLegendResponse"> </wsdl:output>
47:   </wsdl:operation>
48:   <wsdl:operation name="removeLegend">
49:     <wsdl:input message="tns:removeLegend" name="removeLegend"> </wsdl:input>
50:   </wsdl:operation>
51: </wsdl:portType>
52: </wsdl:definitions>
```

Figure 7: Fragment of a specification in WSDL

belonging to the service, which are to be loaded in the component caption (lines 17-18). As output, the operation returns the result of its execution in a text string (line 23), that is, whether it has been properly executed or if an error has occurred.

The functionality implemented in a component (specified by *Interaction-Content* and *CoreContent*), must be communicated to the relevant controller interface if it wants to send/receive information to/from other component(s) via ports. Since each component may be viewed as a black box, the only information that can be obtained from a component is the information it provides through its ports.

3.2. Relationships between Components

The previous section described the internal structure of a component by examining its parts and identifying the different types that exist. In this section we will describe the next aspect included in the architecture: the relationships between components. As discussed above, each component is related to other(s) through relationships, leading to different forms of communication between these components. Before listing the set of relationships between components, we must define the concept of a relationship. Every relationship connects a component with other(s) through the entry and exit ports of these components, i.e. if there is a relationship between two components, at least one set of output ports of a component must be connected to a set of input ports belonging to another component.

Each connection between two ports of two distinct components is called a connector (*Connector* concept in Figure 4). In Figure 8a, a graphical representation of a component can be seen, while in Figure 8b, an example with two components connected via their ports can be seen. As can be seen in Figure 8b, there may be ports in the components which are not connected by any connector and therefore do not participate in the architecture. These ports may be linked by other relationships or may not be used in a specific system. Therefore, depending on the application being implemented and the relationships defined in the architectural model, some ports or others will be used. Figure 8b also represents the connections, which exist within the relationship between the Legend and *LayerList* components in the example. It should be noted at this point that, when the architecture is built from components, the interfaces defined in the specification (Figure 5) are translated to the ports of each component defined in the architecture (Figure 4).

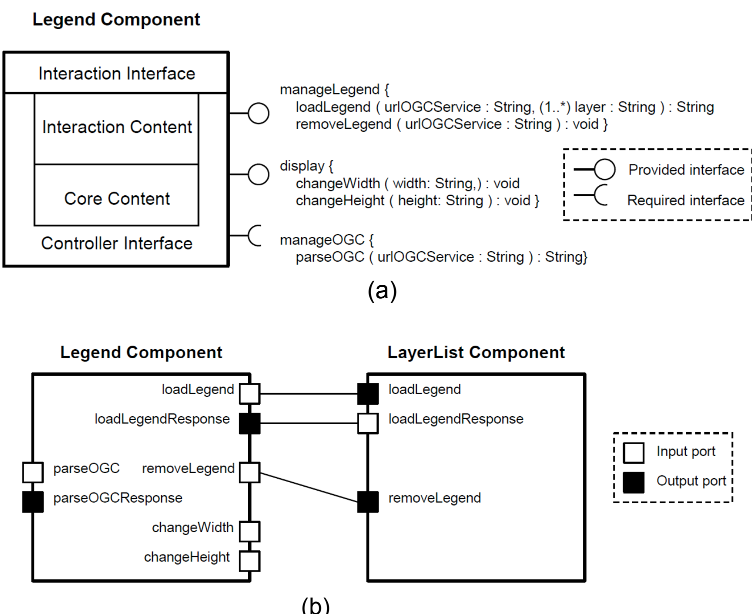

Figure 8: (a) Graphic representation of a component; (b) Example of two components connected by ports

Once the connectors have been defined, the concept of relationship can be defined. A relationship (concept Relationship in Figure 4) is used to connect two or more components at once, allowing communication between them. The relationships between components can be binary or N-ary. Binary relationships are those relationships that exist between two different components. These relationships have a Boolean property called *isBidirectional*, to indicate whether the involved components are connected so that the performance of one affects the other and vice versa. A connector connects two ports, one input and one output, from each of the components of a relationship. The binary relations also include the *BinaryType* concept to identify the existing types of binary relations and therefore different behaviors that can occur in such relationships. In Figure 9 we can see the main binary relations and symbols that represent them. These symbols are used to graphically model component-based architectures.

To help understand the significance of these relationships, the architecture components associated with our example scenario will be used (see Figure 10). The association relationship is the most common and occurs when a

Name	Symbol
Association	$\approx$
Composition	$\sqsupset$
Dependence	$\gg$
Inheritance	$\ni$
Producerconsumer	$\vdash$

Figure 9: Main binary relationships

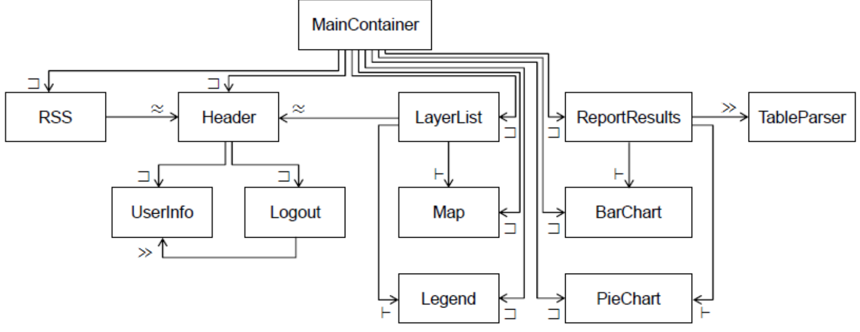

Figure 10: Relationships between components in the example scenario

generic relationship exists between two components A and B. This relationship is used to represent the regular exchange of information between two components, where such exchange cannot be done with any of the other relationships. An example of this relationship appears for instance between the *Header* and *LayerList* components; This relationship represents the request of user information to the *Header* component by the *LayerList* component. When the second component builds the list of available layers to show, it attempts to acquire information from the first by calling on one of its methods. However, if the component is not available or it is not in the architecture, *LayerList* will display a list of default layers. This behavior of weak dependence causes the relationship to be an association type.

The composition relationship is used when all of the interfaces of B are in A. Any component wishing to access B will need to do it through A. This relationship is used to create components from other components. In this way a component can contain one or more components that together perform a specific task or purpose. An example of composition relationship can be seen with the *Header*, *UserInfo* and *Logout* components. The Header component is a container type component containing the other two components. Thus all operations produced from the *UserInfo* or *Logout* components outward,

and all requests made on the methods of both components originating from the outside, must be made through the ports of the *Header* component.

The dependence relationship appears when component A cannot exist without component B. In our example scenario, this relationship appears between the *Logout* component and the *UserInfo* component. The first depends on the second given that, without the necessary information about the user and their session, it is impossible to execute the closing operations and, such as cannot function correctly. This behavior of strong dependence causes the relationship to be a dependence type. The inheritance relationship is given when component A includes all the ports that have been defined in B. This relationship does not appear in our example scenario. However, should we suppose that a component called *MapDB* existed which was connected to the *Map* component by this relationship. This component would include all the ports (and such as the entire functionality) that have been defined in the map. Furthermore it would be able to implement new functionalities such as, for example, saving the searches made in the map in a database. These new functionalities would be able to be offered optionally through additional ports of the *MapDB* component.

The final binary relationship that will be described is the producer-consumer relationship (*producerconsumer*). This relationship is given when a relationship between A and B exists in which A produces information which B consumes. This relationship is very useful in cases of access to databases by the producer component, which will supply information (modified or not) to the consumer component. In our example scenario there are various producer-consumer relationships. One case of this type is established between the *LayerList* and Map components. The first of these acts as the producer supplying the OGC services needed to be visualised on the map by the second component.

All these relationships between two components cover a wide range of possible scenarios in the construction of componentbased systems. Nevertheless, on occasion more complex relationships are necessary. For these cases N-ary relationships have been created. An N-ary relationship is defined as a set of relationships, which two-by-two (i.e. binary relationships) link the components existing within the N-ary relationship. Figure 11 shows the main N-ary relationships together with the graphic symbols.

The first relationship that appears in the table is the sequence relationship. This relationship provides a sequence order to a set of components for the realisation of a common task or purpose. In Figure 12a, a graphical

Name	Symbol
Sequence	$\dotplus\{,\ldots,\}$
Hierarchy	$\bar{\bar{\wedge}}\{,\ldots,\}$
Trading	$\top\{,\ldots,\}$
Observer	$\circledast\{,\ldots,\}$
Controller	⋇{,...,}
Sink	$\curlyvee\{,\ldots,\}$

Figure 11: Main N-aray relationships

representation of this relationship can be seen. An example of this N-ary relationship can be seen between the *LayerList*, *Map* and *Legend* components, and is formed by the two binary *producerconsumer* type relationships that link the *LayerList* and *Map*, and *LayerList* and *Legend* components, respectively. The N-ary relationship sequence will describe that firstly the operations associated with the first binary relationship are executed and secondly those associated with the second binary relationship.

Another N-ary relationship is the hierarchy which is given when all the components of the set have a inheritance relationship with their parent component (see Figure 12b). This relationship defines an inheritance tree between all the components, which form part of the relationship. The trading relationship is given when one of the components carries out a task of mediation with the others (see Figure 12c). As can be seen an association relationship exists between component A and component B. When component T carries out its job of trading it defines how components A and B must communicate. Finally, the N-ary observer, controller and sink relationships remain to be defined. These relationships have a similar structure, defined graphically in the shape of a star in which the relationships, which arrive or leave from the central component change. In Figure 13 graphic examples of these structures can be seen. Although both of these relationships have a similar graphical representation they are not used for the same purpose.

The observer relationship implies that the central component (component O in the example) carries out observation tasks on all the other components and acts in consequence. On the other hand the controller relationship implies that the central component (component C in the example) does not only observe but rather additionally sends control orders to the other components. Therefore, although the binary relationships linked are association type, this N-ary relationship is stronger than the observer relationship. Finally, the sink relationship describes a relationship in which the central component (component S in Figure 13), is the receiver of the information generated by the other

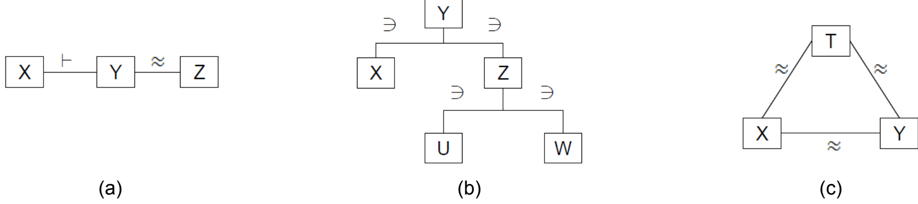

Figure 12: Graphic representation of relations (a) sequence, (b) hierarchy and (c) trading

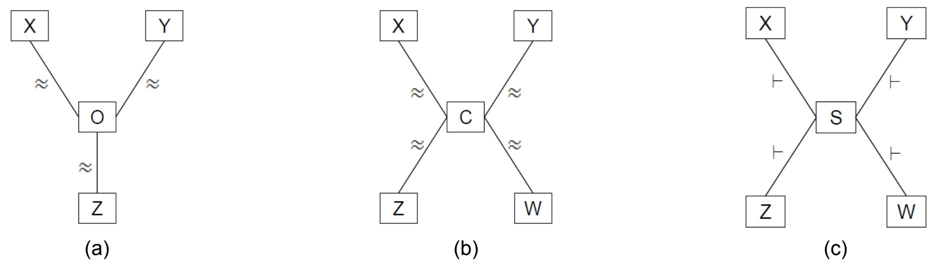

Figure 13: Graphic representation of relations (a) observer, (b) controller and (c) sink

components involved in the relationship. Therefore, all binary relationships of this relationship are *producerconsumer* type, in which the consumer is the central component.

Once both the internal structure of the COTSgets components and the principal relationships have been defined, the service of components in the cloud and how it is implemented will be seen.

4. COTSget-as-a-Service

As indicated in the introduction, the main objective of our research is to create a service of architectures based on COTSgets components (*COTSgets-as-a-Service*) to support interactive systems running on different platforms. Currently the service has been implemented on the web platform. To provide this service, a cloud-based, three layer infrastructure has been created: the client layer, the platform-dependent server layer and platform-independent server layer. The client layer consists of user applications. The components of these applications use the web platform (such as web-based technology) and have been implemented by widgets as recommended by W3C. These applications are deployed by means of a web browser. The platform dependent layer constitutes the intermediate layer of the system architecture. This layer deals

with providing the client with the required services and interacting with the independent layer, thus getting a number of services from it and providing it with others. Finally, the platform-independent layer provides the system services that are valid for all platforms. For this, its functionalities are based solely on the description of components and their relationships, regardless of the platform in which they will be deployed.

The following section describes the two layers of servers, fundamental to understanding the infrastructure. Subsequently, in Section 4, some implementation details related to the deployment of all layers will be examined.

4.1. Architecture Server Layers

Our system architecture consists of a set of servers located in the dependent and independent layers of the platform. To describe the operation of these servers, we will use the business process diagram of Figure 14. Firstly, the platform dependent layer will be described. This layer constitutes the middle layer of the architecture and is responsible for providing the necessary services to the applications that are running on the client layer as well as to communicate with the platform independent layer. The services offered to the client application layer are as follows:

- *Initial loading of the application*: when the execution of an application is to be initiated (Figure 14 task (a)), it is necessary to load the components that make the application. The platform-dependent layer will support this initial loading by means of a web application server (task (b)), given that, as already stated above, it is only currently possible to construct web-platform applications. As part of the initial loading, the layer collaborates with the browser, where deployment is done, to obtain and embed the widgets used to build the user interface on the website (task (c)). This service is managed by a *Node.js* server (`http://nodejs.org`).

- *Component repository support*: Given that applications are built with components, it is necessary to have a server that manages the repositories of said components. This management consists essentially of creating and obtaining instances of widgets (task (d)). *Different* repositories will exist depending on the platform on which they are run. For components implemented with widgets, an Apache Wookie server is used (`http://wookie.apache.org`).

- *Communication between components*: the dependent-layer indirectly manages the communication established between components (widgets), i.e. it is not responsible for calculating the communication paths, but is responsible for receiving and sending messages to and from the components. To manage this service a Node.js server is used. This communication may occur for various reasons, for instance by user interaction with the application (task (e)). When messages are sent, is the client the responsible for receiving and executing the corresponding operations arising from the communication (task (f)).

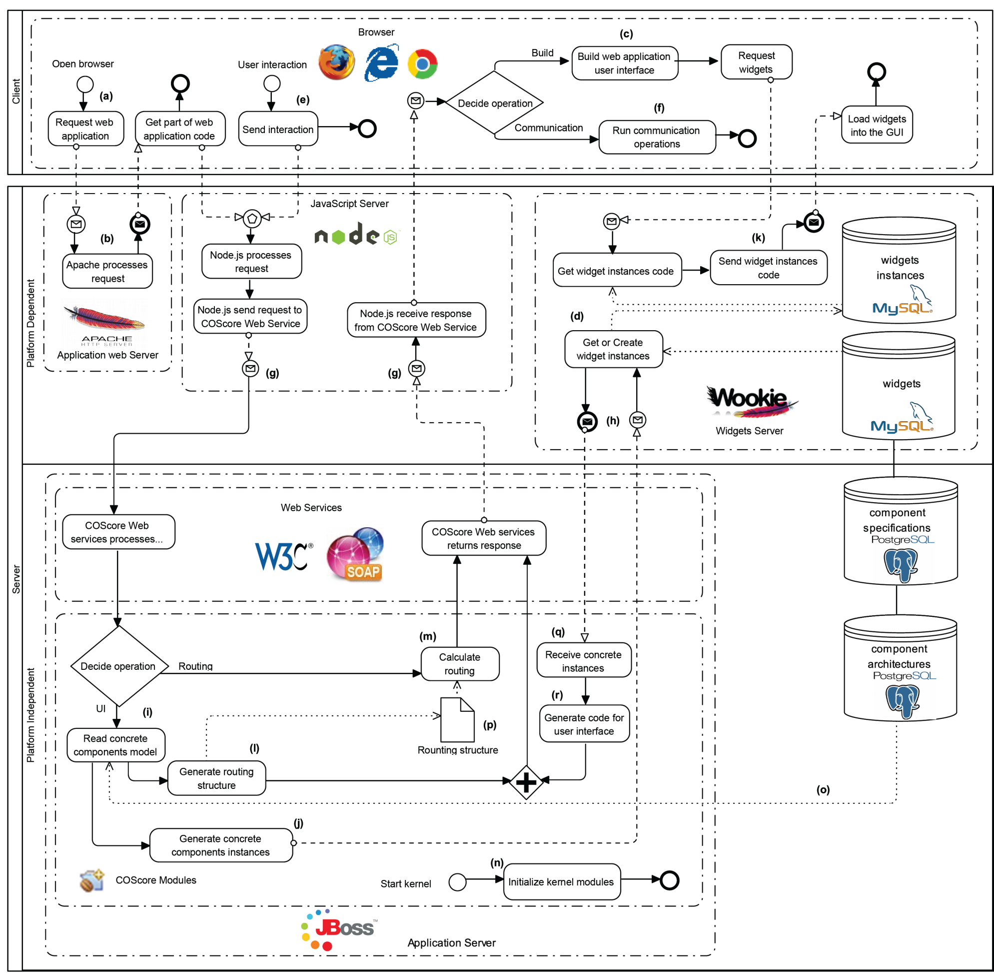

Figure 14: Business process diagram of the architecture behavior

By interacting with the independent-layer, the dependent-layer obtains a series of services (task (g)) and provides others to the independent-layer (task (h)). Regarding the services obtained, the dependent-layer obtains the necessary code to create the initial web application and also consults which route the information should follow to allow the components to communicate between themselves. This is achieved using a web service implemented in the independent-layer. Additionally, the dependent-layer offers information to the independent-layer about the components it manages such as the address of the instances of the widgets it uses to generate the application code.

The other layer is the platform independent layer. This layer has been built to support different platforms and offers a number of features related solely to the description of the components and the relationships between them, regardless of the specific platform in which they will be deployed. It is true that, in terms of deployment, there are certain particularities for each platform which must be taken into account, for example the initialization of a web user interface may be different from the initialization of component architecture in a Java implemented application or the invocation of methods between components as part of a communication task. However, there is one part in common, an abstract view of the behaviors that can be extracted and implemented independently of the platform.

Therefore, the services offered in this layer will be the same for all platforms, although depending on the case, invocations to internal services related to a specific behavior for each platform will be made. This mechanism hides the specifics of each platform to the outside and allows our approach to be modular and scalable, allowing the progressive addition of new features to new platforms that support the system in the future. The independent layer has been implemented with a single server. This server (called COScore, Cotsgets-based architecture Operating Support core) forms the core of the management of component-based architectures. Its basic functions are:

- *Initialize the architecture of concrete components*: when a new client initiates the connection to the system, the component architecture that is to be provided must be obtained (task (i)). Therefore, the corresponding component instances are created (task (j)) and it will return the appropriate data structure to be represented by the architecture. In the particular case of web user interfaces, this service is responsible for initializing the UI based widgets and re-turns the associated code (task (k)).

- *Obtain the communication relationships between components*: as mentioned above, this layer handles communication between components indirectly. This service of COScore takes responsibility for providing routing information for messages between components (task (l)) i.e., this service calculates which components should be sent a message using a message send request that comes from the dependent layer and the information held on the component architecture and the relations between these components, (task (m)).

These services are the only ones to be published externally, although there is a number of internal services which complement the business logic of the above. In brief these services include:

- *Initialize COScore services*: during the system boot all necessary data structures are constructed to allow all the offered services to be managed (task (n)).

- *Read the representative component architecture model*: with the client information provided by the application, the component architecture model, which corresponds to the application is obtained (task (o)).

- *Generate a data structure for routing*: from the architecture model, a data structure representing the relationships between the components of said model is generated (task (p)). Recall that these relations are used for communication between components. This data structure must be updated each time the architecture model alters.

- *Create instances of architectural components* (task (q)): from the architecture model, the component instances are also constructed and information on said instances is associated to the corresponding client.

- *Generate code of the architecture* (task (r)): This service aims to build the compiled or interpreted code for the platform on which the architecture will be deployed.

4.2. Architecture deployment

In order to understand the structure of the architecture proposed, in this section we are going to demonstrate a number of implementation details about the deployment from the client-side and from each of the servers. To

facilitate understanding, we present some code fragments in order to illustrate the behavior developed. In the three tables in this section the first column indicates the execution order (step) of each code fragment.

Starting from the client layer, the user interfaces are developed by sharing common web technologies such as HTML, CSS and JavaScript. With the goal of registering the user interface to the system, it is necessary to connect the web user interface with the platform dependent layer. Therefore, the interface needs to establish a socket connection with Node.js server and to request the GUI initialization (Figure 15-step#1).

step #1

```
var socket = io.connect('http://acg.ual.es:6969');
 socket.on('connect', function(){
   socket.emit('initGUI', { userID: userID }, function(confirmation){
     if(!confirmation) alert('Error in GUI initialization'); });
 });
```

step #7

```
socket.on('addComponent', function(data){ $('body').append(data); });
```

step #8

```
<html> ...
   <body> ...
     <div id="legend">
       <iframe id="ilegend" src="http://acg.ual.es/wookie/deploy/acg.ual.es/wookie/
         widgets/Legend/index.html?idkey=u0MfpWWU3eT057fkAeVkb4GKzwY.eq.&amp;proxy=
         http://acg.ual.es:80/wookie/proxy&amp;st=&amp;userID=user1&amp;nodejsURL=
         acg.ual.es:6969"></iframe>
     </div>
     <div id="layerList">
       <iframe id="iLayerList" src="http://acg.ual.es/wookie/deploy/acg.ual.es/wookie/
         widgets/LayerList/index.html?idkey=YRdICRqjAYRJ7z1tkWFV7Xl3HNg.eq.&amp;proxy=
         http://acg.ual.es:80/wookie/proxy&amp;st=&amp;userID=user1&amp;nodejsURL=
         acg.ual.es:6969"></iframe>
     </div>
     <div id="map">
       <iframe id="imap" src="http://acg.ual.es/wookie/deploy/acg.ual.es/wookie/
         widgets/Map/index.html?idkey=tuV6YYw9ztkjVRHkTC6NDz25D.sl.Y.eq.&amp;proxy=
         http://acg.ual.es:80/wookie/proxy&amp;st=&amp;userID=user1&amp;nodejsURL=
         acg.ual.es:6969"></iframe>
     </div>
     ...
   </body>
 </html>
```

step #9

```
function emitLoadLayer(layer){
  websocket.emit('send', getUrlVars()['userID'], 'layerList', 'loadLayer', layer);
}
```

step #12

```
var websocket = io.connect(getUrlVars()['nodejsURL']);
 websocket.on('connect', function(){ websocket.emit('adduser',
    getUrlVars()['userID'], getUrlVars()['componentID'], '');});

 websocket.on(getUrlVars()['componentID']+ '.loadLayer', function(data){
     var newData = new Array();
     for (var i = 1; i < data.length; i++) newData[i-1] = data[i];
     newLayer(data[0], newData);
});
```

Figure 15: Client side code fragments

Moreover, the application is waiting for a message with the HTML code of the widgets, which is used to add these widgets into the user interface (Figure 15-step#7). As a result, the corresponding HTML code of the widget-based user interface is deployed in the client layer (Figure 15-step#8). Furthermore, for each widget to communicate with Node.js server, a connection to said server and a declaration of the input and output ports must be established (this code resides within each widget). In Figure 15-step#9, it is shown the code executed when the interaction with the *LayerList* component requires to emit information (to the platform dependent layer) about the layers which have to be loaded. On the other hand, in Figure 15-step#12 we can see the connection of the Map component as well as the declaration of an input port named as *loadLayer*. This input port will receive (from the platform dependent layer) the information generated and caused by the interaction mentioned in Figure 15-step#9. This implementation of ports is a concrete case for web platforms and it corresponds to the interface definitions that the COScore server manages (see Section 3.1).

On the platform dependent layer, we have deployed two servers: an Apache Wookie server and a *Node.js* server. The first server allows us to deploy widgets based on the W3C specification (`http://www.w3.org/TR/widgets/`). The components that reside in the server are used for the web platform case, where graphical user interfaces are built from widgets. This server provides an API (`http://wookie.apache.org/docs/api.html`) based on REST services (Richardson and Ruby, 2008) for the management of widgets (insertion, elimination, modification, creation of instances, etc.). In our approach, the server is deployed at `http://acg.ual.es/wookie` and an example address of *Map* component for user1 would be:

```
. http://acg.ual.es/wookie/deploy/acg.ual.es/wookie/widgets/Map
    /index.html?idkey=tuV6YYw9ztkjVRHkTC6NDz25D.sl.Y.eq.&
    proxy=http://acg.ual.es:80/wookie/proxy&st=.
```

In this link, the address part `http://acg.ual.es/wookie/widgets/Map` is the component identifier, and "tuV6YYw9ztkjVRHkTC6NDz25D.sl.Y.eq." is the user's instance identifier.

The second server is a JavaScript server, called Node.js, which can be used as a link between the client and the platform independent server. In order to manage communications with Node.js, we use sockets. This requires installation of the socket.io module (`http://socket.io`). We can see an example

of this use in the code shown in Figure 16-step#2. In this code, the Node.js server establishes an input port *initGUI* in its initialization (connection). This input port is invoked from the client web application to initialize the user interface and, consequently, the web service of the COScore in charge of getting the components, will be called. The result is sent through the output port *addComponent* to the web client. Another example is shown in Figure 16-step#10. This code establishes an input port (send) which is invoked from the client layer to solve the communication process of the components. Again, we can see that a web service of the COScore is called (in this case, *calculatedConnectedPorts*) and then, the information is routed to the corresponding components through the calculated ports.

On the other hand, to invoke the SOAP-based web services, we need to install the node-soap module (`https://github.com/vpulim/node-soap`). In the code fragment shown in Figure 16-step#3, we make use of this module to create a SOAP client that invokes the corresponding method of the web

step #2

```
io.sockets.on('connection', function (socket) {
   socket.on('initGUI', function(data, fn) {
   ...
     callWS('http://acg.ual.es:8080/cos/COSWS?wsdl', 'initGUI',
     args_iniGUI, function(ws_response) {
       if(ws_response != 'Error') {
         ws_response.forEach(function(value, index) {
          io.sockets.in(users[data.userID]).emit('addComponent', value); });
         fn(true);
       } else fn(false);
     });
   });
 });
```

step #3

```
function callWS(wsurl, methodname, args, callback) {
   soap.createClient(wsurl, function(err, client) {
     if(client==null) { callback('Error'); }
     else client[methodname](args, function(err, result) {callback(result.return);});
   });
 }
```

step #10

```
socket.on('send', function(user, component, port, data){
  var args_calculateConnectedPorts = {componentID: component, portID: port};
  callWS('http://acg.ual.es:8080/cos/COSWS?wsdl', 'calculateConnectedPorts',
     args_calculateConnectedPorts, function(ws_response) {

    var pairs = ws_response.split('-');
    pairs.forEach(function(pair){
      pair_s = pair.split(',');
      var c = pair_s[0];
      var p = pair_s[1];
      io.sockets.in(users[user]).emit(c + '.' + p, data);
    });
  });
});
```

Figure 16: Node.js side code fragments

service and returns the result obtained. To describe the client layer deployment, we will use the implementation of the GUI initialization as an example. On the client-side, it is necessary to include the code shown in Figure 15-step#1 which is in charge of starting the connection between the web application and the *Node.js* server. Next, through the *initGUI* port, it sends a message with its user identifier to initiate the interface. Additionally, it establishes an input port named *addComponent* that will be called from the code described in Figure 15-step#7. This port adds the widgets into the web interface. As a result of the initialization, we can see a code fragment of the resulting widget-based web application in Figure 15-step#8.

Regarding the platform independent layer, a JBoss application server has been deployed (`http://www.jboss.org/jbossas`). This application server offers a set of web services developed with JAX-WS (Chinnici et al., 2006). These services are called from the layer that is dependent on the platform through SOAP messages (Graham et al., 2004) and offer the functionalities of the independent layer. All modules (capacities) of COScore are implemented via EJB (Johnson, 2005) and internally managed through the lookup mechanisms (lookup) of this server and through different types of session (stateful, stateless and singleton) of beans. Following the previous code fragments, there is a service in charge of resolving the initial widget list to be inserted into the web user interface (named as initGUI). This service is responsible for reading the data structure that represents the architecture of the user interface and for creating the corresponding code that will be deployed by the client (Figure 17-step#4). For this purpose, widget instances are created through REST services by using the Wookie API (Figure 17-step#5); and then, the HTML code is constructed using the information from the instances previously created (Figure 17-step#6). Another example of these services is shown in Figure 17-step#11. This service implements the behavior of calculating which components (and through which ports) are going to receive to information obtained from the client interaction.

5. Experimentation process

This section illustrates the process used to validate and evaluate the Cloud Service and its performance in managing COTSgets-based architectures. The previous section described the infrastructure of the architecture supporting our proposal, explained how it has been deployed, and provided implementation details. Nevertheless, a performance evaluation is also required to

<table>
<tr><td>step #4</td><td><pre>public List<String> initGUI(String userID){
  List<String> resultList = new ArrayList<String>();
  ManageWookie wookie = new ManageWookie();
  Iterator<Map.Entry<String,List<String>>> it=componentTable.entrySet().iterator();
  while(it.hasNext()){
    Map.Entry<String,List<String>> entry = it.next();
    //Create or get the instance for each widget
    String widgetID = entry.getKey();
    WidgetData widgetData = wookie.getOrCreateWidgetInstance(userID, widgetID);
    ...
    //Store the instance identifier into the existing table
    ...
    String divString = composeDiv(widgetTitle, instanceID, instanceURL, userID);
    resultList.add(divString);
  }
  //Return the code of the components that must be inserted
  return resultList;
}</pre></td></tr>
<tr><td>step #5</td><td><pre>public WidgetData getOrCreateWidgetInstance(String userID, String widgetID) {
  //Invoke REST service to create new widget instances
  ClientConfig config = new DefaultClientConfig();
  Client client = Client.create(config);
  WebResource webResource = client.resource(UriBuilder.fromUri("http://acg.ual.es/
    wookie/widgetinstances").build());
  // Form with widget data
  MultivaluedMap formData = new MultivaluedMapImpl();
  ...
  ClientResponse response = webResource.type(MediaType.APPLICATION_FORM_URLENCODED_
    TYPE).post(ClientResponse.class, formData);
  String result = response.getEntity(String.class);
  WidgetData widgetData = null;
  String xml = result;
  DOMParser parser = new DOMParser();
  try {
    // Get widget attributes ('url', 'identifier', 'title', etc.) from REST response ...
    widgetData = new WidgetData(url, identifier, title, height, width);
  } catch (SAXException e) { log.error(e.printStackTrace()); }
  catch (IOException e) { log.error(e.printStackTrace()); }
  return widgetData;
}</pre></td></tr>
<tr><td>step #6</td><td><pre>private String composeDiv(String widgetTitle, String instanceID, String
 instanceURL, String userID){
  String divString = "<div id = '"+ widgetTitle + "'>" +
    "<iframe id = 'i" + widgetTitle + "' src='" + instanceURL +
    "?userID=" + userID + "&" + "nodejsURL=" + nodejsURL + "'>" +
    "</iframe>" +   "</div>";
  return divString;
}</pre></td></tr>
<tr><td>step #11</td><td><pre>public String calculateConnectedPorts(String componentID, String portID)
{
  String result = "No results obtained";
  TMC tmc = null;
  try{
    Context initialContext = new InitialContext();
    tmc = (TMC)initialContext.lookup("java:module/TMC");
  }
  catch(NamingException e) { e.printStackTrace(); }

  try {
    result = (tmc.calculateConnectedPorts(componentID, portID)).get();
  } catch (Exception e) { ... }

  return result;
}</pre></td></tr>
</table>

Figure 17: COScore side code fragments

validate our approach. Performance is measured in terms of the execution and response times observed in the two main processes of the developed architecture. These two processes are: (1) initialization of the GUI, and (2) communication between the components present in the GUI.

To this end, we performed different tests to analyze the behavior of our setup taking into account three parameters that could affect the performance. These parameters are: (a) the size of the initial GUI which is loaded and shown to the user, (b) the coupling degree of the architecture, i.e., the amount of connections between components, and (c) the number of concurrent users accessing at the same time. We know that other input parameters exist which affect response times, such as the network latency or the browser used by the client, among others. However, to ensure the correct performance of the system, only experiments which validated features which can be managed or limited were performed. To execute these experiments and measure the performance times, we used a computer with an Intel(R) Core(TM) i5 CPU 660 @ 3.33 GHz, with 4 GB of physical memory and running the Windows 8.1 Professional 64 bits operating system. This machine included the platform, dependent and independent servers deployed. For testing purposes a web application has been developed as a client. Each result of the response times is calculated as the average time for 100 executions of the same test unit.

As the proposed architecture has three layers (see Figure 14) firstly, we evaluated the meas-ured times in (A), (B), and (C), as shows Figure 18.

The time obtained in (A) indicates the execution time of the functions implemented in the independent layer server (COScore). Subsequently, the time represented by (B) is derived from (A) but it incorporates the time elapsed to perform the behavior implemented in the platform dependent layer. Finally, (C) represents the total time elapsed between the call being performed by the client and the moment that a response is obtained and shown to the user. Figure 19 shows the response times for the GUI initialization when we vary

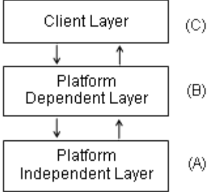

Figure 18: Possible measurement locations in the evaluation process

the number of components. The differences between times measured in (A), (B) and (C) are very small. For this reason, the following performance times are focused on the response time (C), because it is the largest response time (in fact, it is equal to the total time of the process) and corresponds to the real time that the user experiences as a response to the service usage.

The first group of tests that has been performed is intended to evaluate the performance of the initialization of a GUI. We evaluated the times obtained for different initial GUI sizes as they are shown to the user. Thus, measurements were taken with models with 1, 2, 3, 5, 10, and 20 components. These sizes were chosen since GUIs (developed within our COTS-based architectures) are usually composed of less than 20 components. For example, a typical user interface with only one component in the aforementioned domain is a GUI composed by a map with geospatial information. In addition to the map, the user interface may contain a legend component and/or a component with a list of the layers shown, thus forming typical examples of GUIs with two and three components. Due to visual limitations and typical user preferences, the normal scenario is to manage GUIs with a maximum of four, five or six components. For completeness, we included experiments with 10 and 20 components with the aim of testing the performance in less favorable testbeds. At the very least the obtained results allow us to infer the response times for scenarios with additional components. In addition, we evaluated the performance for different degrees of coupling: "low", "medium" and "high". If "n" is the number of components of the GUI, low coupling represents a number of communications between components lower than "n – 1". High coupling corresponds to a number of communications higher than "n * n (n – 1) / 2", and medium coupling represents the intermediate levels. Figure 20 shows the results of this test. It is noteworthy that the times shown in Figure 19 correspond to the different times shown in Figure 20 for a 'medium' coupling.

We can extract three conclusions from Figures 19 and 20: (1) the response times grow in proportion to the number of components; (2) the coupling degree does not affect (or has an insignificant influence on) the performance; and third, the performance results obtained are suitable for the GUI initialization process, since the worst response time is below 600 ms, resulting in a good user experience. Nevertheless, we performed another group of tests to evaluate the initialization process from concurrent users, with the aim of testing the performance in a real exploitation/production environment. The result of this experiment is shown in Figure 21, where rows Ui correspond to

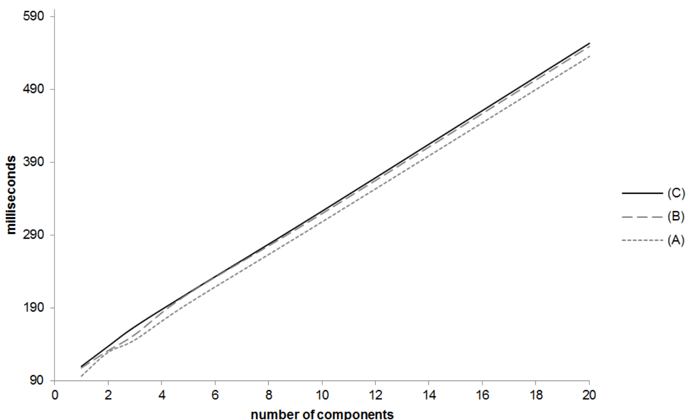

Figure 19: Difference between performance times in (A), (B) and (C)

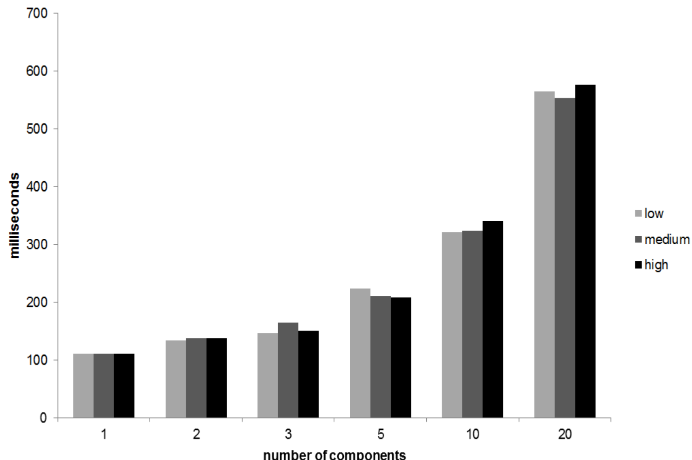

Figure 20: Performance results: Initialization of GUIs with different model sizes and different coupling values

the response times obtained when different numbers of concurrent users i accessed the user interface, and Ci represents the size of the loaded model. Low, medium and high coupling is represented by "l", "m" and "h", respectively. The performance results are measured in milliseconds.

When multiple concurrent users perform a service request for the GUI initialization simultaneously, the server side of our architecture is not able to respond to all users at the same time. In this sense, the column "min" of Figure 21 contains the time taken for the first user to receive the response

and complete the initialization process. The column “max” depicts the time taken for the last user to receive the response and complete the process. Figure 22 shows a representation of the obtained results (one for model).

From results depicted in Figure 21 and the different graphs in Figure 22, we are able to draw the following conclusions: (1) the “min” response time remains almost constant and is not affected by the number of concurrent accesses, (2) the “max” response time increases in proportion to the number of concurrent users, (3) the larger is the model of the GUI, the higher is the increase of the “max” value, and (4) the coupling degree of the architecture representing the GUI does not influence the performance even if we increase the number of concurrent users (as we previously stated for one user, see Figure 20). If we summarize the results of the Figure 22, focusing on the “min” and “max” results, we obtain the graphics shown in Figure 23 and Figure 24, which represent the lowest and the highest times obtained, respectively.

The performance results obtained from these experiments are relatively good in terms of response times for the initialization of the GUI with different numbers of concurrent users. Nevertheless, for models with 10 components the response time experimented by the last user (“max” value) begins to exceed acceptable levels (3.5 seconds) once there are over 40 concur-rent accesses. For models with 20 components, the highest response time is excessive (over 3.6 seconds) once there are over 25 concurrent initializations. The reason the number of concurrent users influences the response time is due to the application server (i.e., the platform-independent server) which has some common points (common EBJs) is accessed by all the users and a bottleneck can be created. An example of this common point is the EJB in charge of managing the user sessions. For this reason, we performed another experiment, which tried to reduce response time and to improve performance. In this test, we deployed two and three platform-independent servers and we balanced the workload by distributing the requests between the servers. The results are show in Figure 25.

The results of the experiment allow us to take the decision of deploying our cloud service architecture with two platform-independent servers instead of one or three. This decision is based on two factors: (a) the use of more than one server is useful to avoid a bottleneck in the system, and (b) using more than two servers takes longer to choose the target server and affects performance. In Figure AD it is possible to observe that using three servers gives the same improved results for “max” times, but worse results in the case of “min” times.

U_i \ C_i	1		2						3					
	-		l		m		h		l		m		h	
	min	max	min	max	min	max	min	max	min	max	min	max	min	max
1	110.00		132.50		137.50		137.67		147.00		164.50		150.33	
2	109.33	111.67	133.67	136.33	142.33	144.67	139.33	151.33	144.00	146.67	155.00	157.00	178.67	182.00
5	114.33	236.00	145.00	189.67	176.00	218.67	160.67	230.33	182.67	234.33	191.00	242.33	188.00	266.00
10	112.67	319.50	162.00	381.67	131.33	353.00	157.67	393.33	192.67	447.67	168.33	491.00	198.00	455.00
25	125.67	732.00	154.33	894.00	129.67	885.33	150.00	894.00	158.67	1061.67	173.33	1027.33	175.33	1036.00
50	88.00	1476.33	120.67	1784.33	133.33	1815.33	115.67	1766.67	188.33	2058.33	168.00	2049.67	139.67	2080.33
75	86.00	2157.67	131.67	2773.00	111.33	2813.33	119.33	2684.00	135.67	3186.33	142.00	3117.00	173.33	3133.67
100	118.33	2946.33	152.67	3709.67	116.00	3651.33	153.00	3682.67	130.67	4185.67	168.67	4392.33	142.67	4589.67

5						10						20					
l		m		h		l		m		h		l		m		h	
min	max	min	max	min	max	min	max	min	max	min	max	min	max	min	max	min	max
223.33		210.67		208.00		321.00		323.00		340.33		565.00		553.33		576.33	
196.00	201.67	219.33	222.33	229.67	233.00	344.00	360.00	319.67	324.33	336.67	342.67	548.33	555.00	551.33	568.67	567.00	578.33
261.33	315.00	281.33	332.33	265.33	321.00	451.67	528.67	406.67	464.67	401.00	477.33	699.00	800.67	724.00	808.33	683.67	782.00
264.00	565.00	225.67	574.67	244.00	561.33	414.33	937.67	440.67	915.67	412.00	923.00	653.00	1452.33	681.33	1593.67	684.67	1466.00
247.00	1392.67	214.00	1409.67	220.67	1349.00	394.00	2183.33	382.67	2203.67	408.00	2145.67	689.00	3707.00	699.00	3677.00	692.67	3863.00
202.67	2747.33	196.67	3017.33	216.67	2759.33	399.33	4311.33	476.00	4500.67	369.67	4307.33	670.67	7433.33	736.00	7502.67	715.67	7390.33
215.67	4082.33	244.33	4040.67	211.67	4096.33	383.67	6474.67	385.00	6561.00	393.67	6643.67	682.33	11155.00	666.33	11411.67	743.67	11181.67
251.67	5806.67	224.67	5730.33	225.67	5704.00	375.67	9083.33	464.33	9252.33	402.67	9152.00	677.00	14898.67	676.00	15043.00	695.67	16071.67

Figure 21: Performance: Initialization of GUIs varying the number of concurrent users

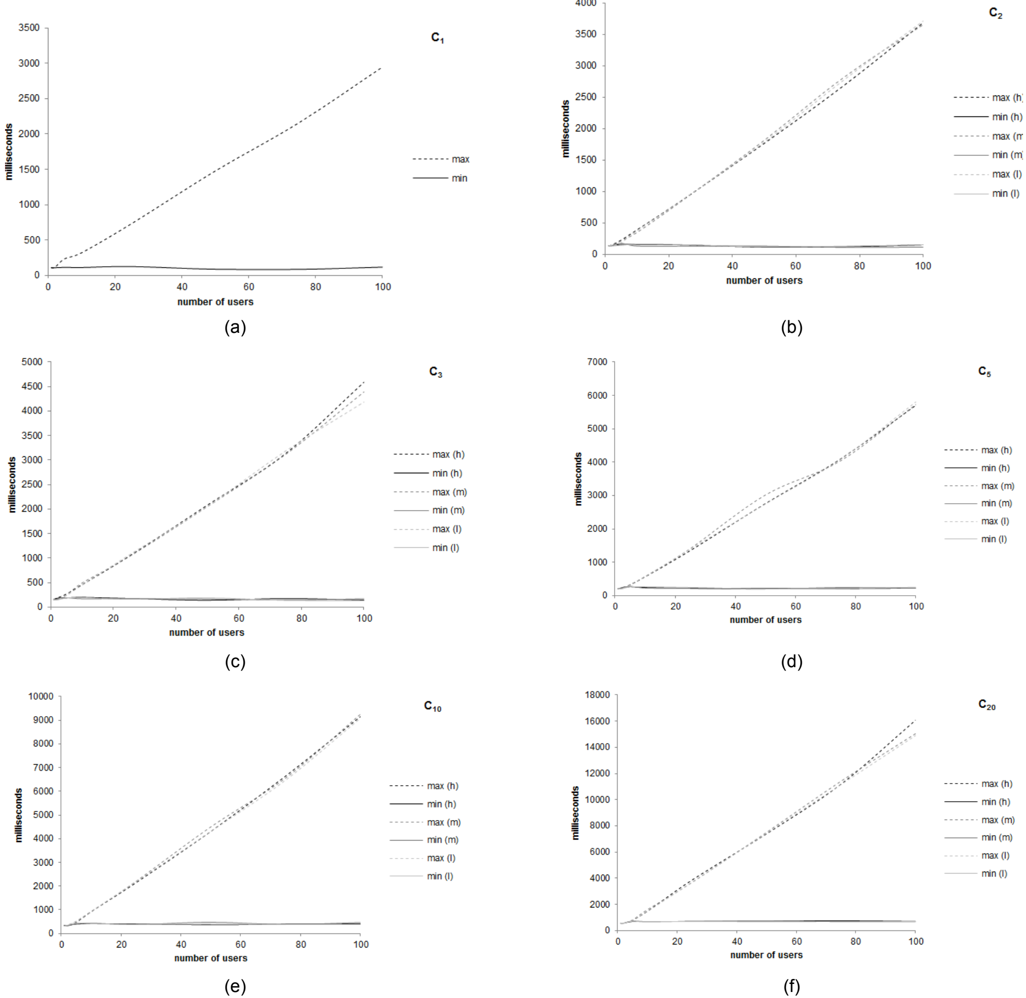

Figure 22: Performance results for the initialization of GUIs with different model sizes and different coupling values

Regarding the communication process, we measured the response times with different numbers of components and with different coupling values. For practical reasons, we did not perform the test with low coupling values of the architectures, because there are very few connections enabling the communication. In addition, we only executed the test for GUIs with a number of components between 3 and 20, as these scenarios provide sufficient connections to evaluate the process. In Figure 26 we can see the results of the

experiment. The response times for the communication remain under 100 ms for a "medium" coupling value and less than 140 ms in the case of "high" coupling. These times grow in proportion to the number of components, but with a very low gradient. For this reason, we can assert that the communication process is executed in a suitable time.

Furthermore, the following link `http://acg.ual.es/enia/COTSbasedArchitectureExample` offers a GUI example developed for the ENIA research project (ENIA, 2010). This GUI application serves as an example of a real application developed within our proposal. This way, the reader is able to test and (probably) better understand the appearance of a COTS-based architecture and the addressed goal.

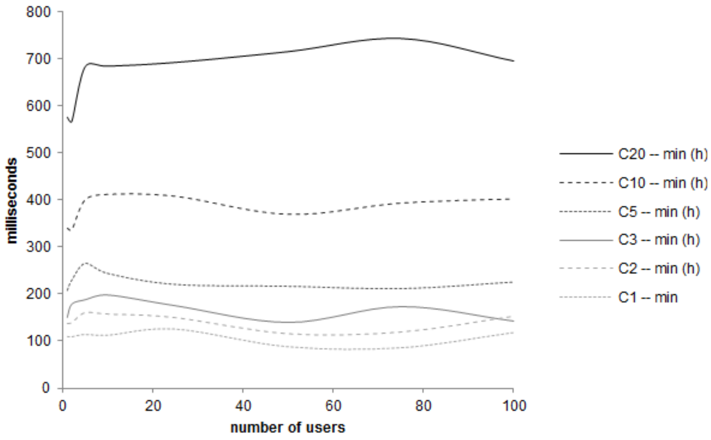

Figure 23: Lowest response times (obtained from the first user who is responded)

6. Related work

The use of cloud-based computing, as has already been presented in (Lee, 2010), offers a number of advantages for both users and organizations that want to make better use of the resources they manage. Among the benefits identified, this work names the use of SaaS and specifically the use of MaaS as a software element of a high level of abstraction available for systems to use at any time, for example, to build software "top-down" from an approximation. The concept of MaaS as an on-demand decision element that could be used by a software system was introduced in (Bhargava et al., 1997) and has had numerous applications since. For example, in (Kridel and Dolk, 2011),

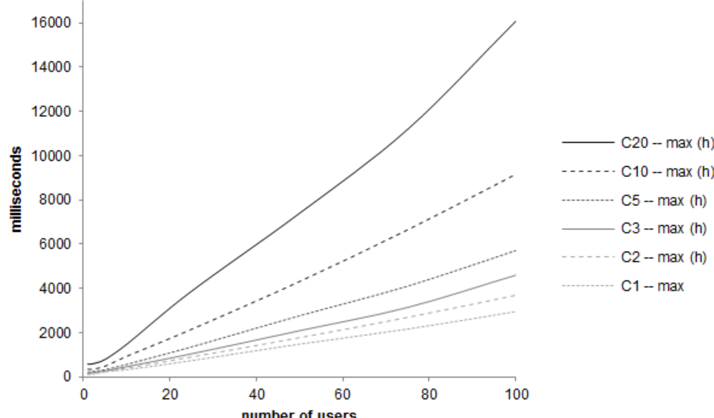

Figure 24: Highest response times (obtained from the last user who is responded)

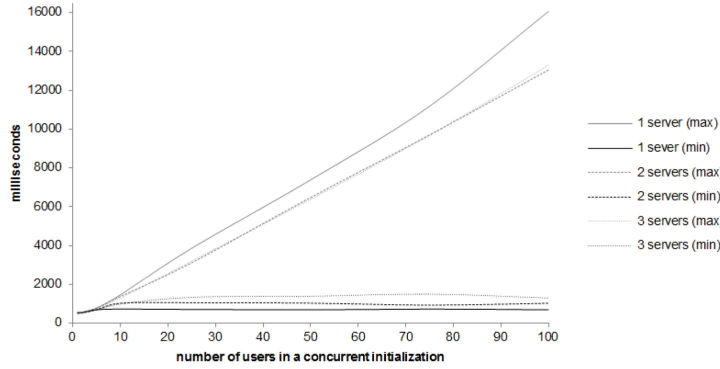

Figure 25: Using 1, 2 or 3 platform-independent servers

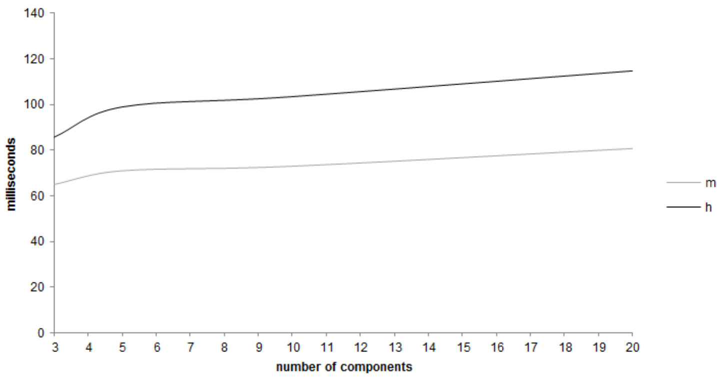

Figure 26: Evaluation of the communication process with different sizes of GUIs

the authors make use of this concept to provide data and decision analysis in model form from expert knowledge and an automatic modeling system. In (Brunelire et al., 2010) benefits that can be obtained from the MaaS concept and the use of models with cloud computing are also identified. In this work, the authors highlight aspects such as the availability of these models, their run-time sharing, improved scalability and distribution, possible implementation, adaptation and evolution of these models or even building mashups as a combination of MDE services offered by different "vendors".

Inspired by this concept and the use of the mechanism for accessing models through web services, we also make use of cloud computing and models as services in our proposal. Nevertheless, instead of proposing a general use of this concept, our work focuses on the management of software architectures based on specific named components such as COTSgets. Moreover, we show the use of the developed system to manage and communicate graphical user interfaces constructed from these components.

The MaaS concept was also used in (Geller and Turner, 2007) to identify ecological models represented and stored by a web system, for customers who would like to share this information for management tasks and decision making. Other authors (Roman et al., 2009), demonstrate their potential uses, the challenges to be addressed and provide a case study of their use within a task which analyses oil spill risks. Other works (Goodall et al., 2011) propose a definition of web service interface and a specification for data exchange in order to expose and offer "water resources" as web services.

On the other hand, there have also been other initiatives to enhance interoperability between models repre-senting geospatial information and its access through web services, additionally trying to make these services increasingly reused (Nativi et al., 2013). Our proposal also uses this mechanism of accessing models through services. However, although we also use geospatial information as a domain, with our research being linked to a regional project named ENvironmental Information Agent (ENIA, 2010), the models which we access through the cloud are models that represent the structure of widgets-based GUIs and it is precisely these medium or high granularity graphical components which are reused in our system.

Therefore, the domain could be any other application that uses a graphical interface in which components may be developed as widgets. Looking to the future, we also believe that our proposal and the architecture presented could be extended to other domains in which our COTSgets components do not represent GUI widgets, but rather are used to describe software archi-

tectures based on components from other areas, such as home automation (Chinnici et al., 2006), robotics (Edwards et al., 2009), communication network infrastructures (Garlan et al., 2004), etc.

Within the domain of our proposal (i.e. UI), there are examples of works that integrate the use of cloud computing with user interfaces. In (Grønli et al., 2011) the authors work with Android devices to achieve more flexibility in the user interfaces making use of information about the environment (for example, via the ambient light sensor, the battery status, user interaction, etc.). Our developed architecture is also designed to store information about the environment, including the ability to capture the user's interaction with the components thanks to the management of the communication of the Node.js server. In the literature there are other notable works that are not directly related to cloud computing, but with GUIs built from components or services. In (Wilson et al., 2012) the authors analyzed the features of building *mashup* GUIs by using widgets of the W3C and carried out a proposal to extend the widget model. The reason for aiming to extend this model is to give support to a variety of patterns of communication between components. In contrast, our proposal developed a communication mechanism that did not need a revision of the component model. In (Hsu, 2013), the authors carry out a proposal based on MDA modeling to develop Web 2.0 applications such as *mashup*. For UML constraints they make use of a profile developed specifically for this domain. In our case, in place of a UML profile, we restrict the modeling language using a domain-specific language (Gronback, 2009) to describe our architectures and COTSgets component types.

7. Conclusions and future work

This paper presents a developed infrastructure for managing component-based architectures by using web services. The proposal takes a general approach with the aim of being applied to any type of software architecture. Nevertheless, the architectures handled in our proposal must be defined using a DSL that has been developed. This language allows us to define which components are present in the architecture and how they relate to each other. For a description of these relationships, our language allows different types of binary and N-ary relationships between components to be defined, providing information about the behavior of the communication and interaction between them. In addition, this language describes how the architectural components are connected at port level.

When specifying components, our approach proposes the use of another DSL. Thus, the possible application domains are restricted to those software architectures built from components that meet our definition of component. Our type of component has been named COTSget, from the combination of COTS and "*gadget*" where gadget can be any software appliance that can work alone or as a piece of architecture. Such components are components of medium or high granularity that encapsulate some functionality and can interact with other elements of architecture through their interfaces. Therefore, inspired by the description of third party COTS components, the proposed DSL defines the interfaces, properties and dependencies, as well as providing information on implementation, packaging and marketing for each component.

As has been noted in the metamodels, which represent the syntax, both languages have been defined using MDE technologies. As a result, our proposal benefits from all the development and implementation mechanisms and tools that exist within this paradigm. In this sense, the article provides some examples of OCL constraints that are applied in the construction of our models, which allow for syntactic and semantic checks that cannot be expressed through the sole use of metamodels.

Building on established foundations, this paper demonstrates a three-layer implementation infrastructure based on web services and cloud management for component-based software architectures. The three layers consist of: a client-side layer and two layers belonging to the server side, one being dependent on the platform and the other platform independent. The platform independent layer includes all the management services of the architectures common to all possible platforms, such as management of communication between components. The platform dependent layer management provides the services that are particular to a specific platform, such as creating instances of widget-like components in the case of the web platform. The client layer is where the final software architecture opens and interacts with the platform dependent server layer to resolve all the required services.

As a specific application platform, a component-based prototype developed in the domain of user web interfaces is shown. Within this domain, a web interface that serves as a "running example" throughout the article is shown to explain the definitions of architectures and components and the three layer infrastructure. To better understand how the platform works, the deployment of technology, which solves the proposed infrastructure is shown, providing specific examples of functionalities and their implementation.

As future work, we intend to use the infrastructure, which has been developed and equip it with new capabilities and functionalities. One of our main objectives is to apply elements from the field of "*Business Intelligence*" (Moss and Atre, 2003) and its uses from cloud computing through the "*Big Data*" (Agrawal et al., 2011). Big Data involves a massive amount of data collected from different sources over time and aims to facilitate the task of analyzing this data via cloud services. Thus, in our architecture, user interaction with the components can be recorded for future decision making, for example, to tailor the user interface to their specific needs.

Furthermore, the languages developed for specifying components and architectures offer the possibility of being applied in different fields and platforms. Although this paper shows an example for application in web technologies, we intend to extend the use of our infrastructure to other scenarios such as home automation or robotics. These scenarios complement the validity of research and open new lines to improve our proposal.

Regarding the use of MDE technologies, there are different possibilities that can greatly complement this work. Processing operations or refactoring of models can be used to adapt or modify software architectures (at run-time); for example, from changes in context or due to user interaction (Rodríguez-Gracia et al., 2012). Furthermore, both the use of mediators and solutions based on "*trading*" can provide an interesting mechanism for the resolution of different platform dependent configurations from the same platform independent architecture (Criado et al., 2013).

Acknowledgments

This work was funded by the EU ERDF and the Spanish Ministry of Economy and Competitiveness (MINECO) under Project TIN2013-41576-R, and the Spanish Ministry of Education, Culture and Sport (MECD) under a FPU grant (AP2010-3259), and the Andalusian Regional Government (Spain) under Project P10-TIC-6114. This work was also supported by the CEiA3 and CEIMAR consortiums.

References

Agrawal, D., Das, S., El Abbadi, A., 2011. Big data and cloud computing: current state and future opportunities. In: Proceedings of the 14th International Conference on Extending Database Technology. ACM, pp. 530–533.

Belli, F., 2013. Dependability and software reuse coupling them by an industrial standard. In: IEEE 7th International Conference on Software Security and Reliability Companion (SERE-C). pp. 145–154.

Bencomo, N., Blair, G., 2009. Using architecture models to support the generation and operation of component-based adaptive systems. In: Software engineering for self-adaptive systems. Springer-LNCS, pp. 183–200.

Bhargava, H. K., Krishnan, R., Müller, R., 1997. Decision support on demand: Emerging electronic markets for decision technologies. Decision Support Systems 19 (3), 193–214.

Bradbury, J. S., Cordy, J. R., Dingel, J., Wermelinger, M., 2004. A survey of self-management in dynamic software architecture specifications. In: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems. ACM, pp. 28–33.

Brunelire, H., Cabot, J., Jouault, F., 2010. Combining model-driven engineering and cloud computing. In: Modeling, Design, and Analysis for the Service Cloud-MDA4ServiceCloud'10: Workshop's 4th edition (co-located with the 6th European Conference on Modelling Foundations and Applications-ECMFA 2010).

Chinnici, R., Hadley, M., Mordani, R., 2006. The Java API for XML-Based Web Services (JAX-WS) 2.0 Spec. JSR, 224.

Criado, J., Iribarne, L., Padilla, N., 2013. Resolving Platform Specific Models at runtime using an MDE-based Trading approach. In: On the Move to Meaningful Internet Systems: OTM 2013 Workshops. Springer-LNCS, pp. 274–283.

Criado, J., Vicente-Chicote, C., Iribarne, L., Padilla, N., 2010. A model-driven approach to graphical user interface runtime adaptation. Workshop on Models@run.time. International Conference on Model Driven Engineering Languages and Systems (5/13: 2010: Oslo, Norway). CEUR Workshop Proceedings 641, 49–59.

Crnkovic, I., Larsson, M., 2002. Challenges of component-based development. Journal of Systems and Software 61 (3), 201–212.

Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M., 2011. A classification framework for software component models. IEEE Transactions on Software Engineering 37 (5), 593–615.

Czarnecki, K., Helsen, S., 2003. Classification of model transformation approaches. In: Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture. Vol. 45. Citeseer, pp. 1–17.

Edwards, G., Garcia, J., Tajalli, H., Popescu, D., Medvidovic, N., Sukhatme, G., Petrus, B., 2009. Architecture-driven self-adaptation and self-management in robotics systems. In: Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on. IEEE, pp. 142–151.

ENIA, 2010. ENvironmental Information Agent: Developement of an intelligence Web agent of environmental information. Research Project of the Andalusian Regional Government (Spain), ref. P10-TIC-6114, http://acg.ual.es/enia.

Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., Steenkiste, P., 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37 (10), 46–54.

Geller, G. N., Turner, W., 2007. The model web: a concept for ecological forecasting. In: Geoscience and Remote Sensing Symposium, 2007. IGARSS 2007. IEEE International. IEEE, pp. 2469–2472.

Goodall, J. L., Robinson, B. F., Castronova, A. M., 2011. Modeling water resource systems using a service-oriented computing paradigm. Environmental Modelling & Software 26 (5), 573–582.

Graham, S., Davis, D., Simeonov, S., Boubez, T., Neyama, R., Nakamura, Y., 2004. Building Web Services with Java: Making Sense of XML. Soap, Wsdl, and Uddi, Sams, Indianapolis, IN.

Gronback, R. C., 2009. Eclipse Modeling Project: A domain-specific language (DSL) toolkit. Pearson Education.

Grønli, T. M., Hansen, J., Ghinea, G., 2011. Integrated context-aware and cloud-based adaptive home screens for android phones. In:

Human-Computer Interaction. Interaction Techniques and Environments. Springer, pp. 427–435.

Hoyer, V., Fischer, M., 2008. Market overview of enterprise mashup tools. In: Service-Oriented Computing–ICSOC 2008. Springer, pp. 708–721.

Hsu, I., 2013. Visual modeling for web 2.0 applications using model driven architecture approach. Simulation Modelling Practice and Theory 31, 63–76.

Iribarne, L., Padilla, N., Criado, J.and Asensio, J., Ayala, R., 2010. A model transformation approach for automatic composition of cots user interfaces in web-based information systems. Information Systems Management 27 (3), 207–216.

Iribarne, L., Troya, J., Vallecillo, A., 2004. A trading service for cots components. The Computer Journal 47 (3), 342–357.

Johnson, R., 2005. J2EE development frameworks. Computer 38 (1), 107–110.

Kleppe, A. G., Warmer, J. B., Bast, W., 2003. The model driven architecture: practice and promise. Addison-Wesley Professional.

Kridel, D., Dolk, D., 2011. Automated self-service modeling: predictive analytics as a service. Information Systems and e-Business Management 11 (1), 119–140.

Lee, C. A., 2010. A perspective on scientific cloud computing. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. ACM, pp. 451–459.

Mishra, A., Mnch, J., Mishra, D., 2012. Distributed Development of Information System. Journal of Universal Computer Science 18 (19), 2599–2601.

Moss, L. T., Atre, S., 2003. Business intelligence roadmap: The complete project lifecycle for decision-support applications. Addison-Wesley Professional.

Nativi, S., Mazzetti, P., Geller, G. N., 2013. Environmental model access and interoperability: The geo model web initiative. Environmental Modelling & Software 39, 214–228.

Richardson, L., Ruby, S., 2008. RESTful web services. O'Reilly Media, Inc.

Rodríguez-Gracia, D., Criado, J., Iribarne, L., Padilla, N., Vicente-Chicote, C., 2012. Runtime adaptation of architectural models: an approach for adapting user interfaces. In: Model and Data Engineering. Springer-LNCS, pp. 16–30.

Roman, D., Schade, S., Berre, A. J., Bodsberg, N. R., Langlois, J., 2009. Model as a Service (MaaS). In: AGILE Workshop: Grid Technologies for Geospatial Applications, Hannover, Germany.

Silva, P., 2001. User interface declarative models and development environments: A survey. In: Interactive Systems Design, Specification, and Verification. Springer, pp. 207–226.

Sire, S., Bogdanov, E., Palmr, M., Gillet, D., 2009. Towards collaborative portable web spaces. In: 4th European Conference on Technology Enhanced Learning (EC-TEL). Workshop on Mash-Up Personal Learning Environments (MUPPLE09). No. LA-CONF-2009-018.

Vallecillos, J., Criado, J., Iribarne, L., Padilla, N., 2014. Dynamic mashup interfaces for information systems using widgets-as-a-service. In: On the Move to Meaningful Internet Systems: OTM 2014 Workshops. Springer-LNCS, pp. 438–447.

Whaiduzzaman, M., Nazmul Haque, M., Rejaul Karim Chowdhury, M., Gani, A., 2014. A Study on Strategic Provisioning of Cloud Computing Services. Volume 2014 (2014), Article ID 894362, 16 pages. The Scientific World Journal.

Wilson, S., Daniel, F., Jugel, U., Soi, S., 2012. Orchestrated user interface mashups using W3C widgets. In: Current Trends in Web Engineering. Springer, pp. 49–61.

Yu, J., Benatallah, B., Casati, F., Daniel, F., 2008. Understanding mashup development. Internet Computing, IEEE 12 (5), 44–52.