A Platform-Centric Framework for the Web Exposure and Orchestration of Distributed Objects

Hervé Paulino
CITI / Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Portugal
Email: herve@di.fct.unl.pt

Paulo Cancela
Critical Software SA
Portugal

Tiago Franco
Critical Software SA
Portugal

Abstract—Many of the current service-oriented architectures are built on top of distributed object technologies. These technologies have, however, lost most of their initial appeal, mainly due to the inability to port their concepts to the World Wide Web, and to provide interoperability across many different platforms. The porting of these distributed object platforms to the current Web Service standard is, however, a costly process that requires high investments of both time and money.

This paper presents OHMS, a framework that provides an easy and not resource consuming way of exposing a platform to the Web, thus enabling Web access, business-to-business interaction and service composition, by the means of orchestration. We address the framework’s architecture, its implementation, and the support for orchestrating CORBA platforms. We evaluate the work regarding the initial requirements, the effort required to support a given distributed object technology, and performance issues. We also address the application of OHMS to a real-life scenario.

Index Terms—Distributed Objects, Web Services, Orchestration

I. INTRODUCTION

Many of the current Service-Oriented Architectures (SOA) are built on top of Distributed Object (DO) technologies, such as CORBA [11], DCOM [7] or Jini [20]. Although established in the market for more than a decade and therefore mature, these DO technologies have lost most of their initial appeal. This is mainly caused by their inability to overcome two crucial aspects of today’s businesses: to port the SOA concept to the World Wide Web and to provide interoperability across many different platforms, enabling business-to-business transactions. By tackling both these issues, the Web Service (WS) technology has become the community standard for the development of SOA infrastructures.

The porting of DO-based platforms to the WS technology is, however, a costly process that requires high investments of both time and money. The lack of WS support for some of the usual DO’s features, such as event-handling, may even force some architecture redesign, instead of just code rewriting. Furthermore, there are performance issues at stake, the overhead introduced by WS to support platform and language independence is not desired when it comes to the internals of a platform.

However, on the other hand, the exposure to the Web world opens a new range of prospects, essentially motivated by: 1) increased visibility - the business becomes accessible from the Web; 2) business-to-business interaction based on XML standards - a central aspect for today’s businesses, and; 3) service composition - the use of service composition to deploy new services by composing platform and other Web available services.

The last two items are closely related, since service composition plays a major role on the support of business-to-business and of enterprise application integration. The use of third-party services on the definition of a platform’s own business model is becoming a common solution.

The service composition concept is essentially driven by interoperability and thus, it is only natural the focus of the existing solutions [5], [9] on the WS technology. This state-of-the-art reduces the application of composition on a DO-based platform to the previous exposure of the original services as Web services, process we will refer to as bridging.

The bridging of DO services is already provided by some platforms [1], [3], [6], [8], [18], [19]. These, however, are bound to a specific technology, focus on the bridging of a single service, and require its posterior publishing in a WS repository, hence not providing a systematic and transparent procedure.

This paper presents OHMS, a framework to support the Web exposure and composition, concretely the orchestration, of services originating from distinct DO technologies. OHMS is platform-centric, meaning that it does not bridge a service, but rather a platform. The latter expose their composing services (partially or completely) by registering their naming-service in the OHMS directory. No alterations on the original code of the platform is required. Transparency is one of our main premises. The actual service orchestration is performed on a platform-centered extension of the Eclipse BPEL plug-in [4].

The work was developed in the context of a collaboration between academia (Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa) and the industry (Critical Software, SA). The motivation was the will to specialize COMCOP [17] (a generic Command and Control Platform implemented in CORBA) with mission-specific services by
orchestrating existing ones, such as the platforms’ own core services and Web accessible services.

The remainder of this paper is structured as follows: Section II overviews the current state-of-the-art on DO and WS interoperability; Section III introduces the OHMS platform, its architectural design and implementation; Section IV describes how OHMS can support CORBA-based platforms; Section V evaluates OHMS from different perspectives, which includes the application to the COMCOP platform; and finally Section VI presents our conclusions.

II. RELATED WORK

The existing service composition techniques, orchestration [5] and choreography [9], only support Web services. The lack of support for direct DO service composition, forces the exposure (bridging) of these services as Web services and the use of the available Web service composition tools. In the remainder of this section, we will further explain the concept of bridging and the currently available solutions.

A. On the Bridging of Distributed Objects

To the best of our knowledge, no solution has been proposed to address the bridging and consequent composition of DO platforms. There are solutions to bridge CORBA, DCOM, and Jini services, but these are bound to a single technology and require the posterior publishing of the Web service, thus not providing a systematic and transparent procedure.

The main idea of bridging in this context is to allow a Web service client to invoke a DO service in a transparent way, i.e., as if it was invoking a Web service. As is portrayed in Figure 1, a Web service poses as a two-way proxy between the client and the DO service, relaying the communication messages in both directions. The client discovers and interacts with the service using the usual Web service standards. The responsibility of converting the SOAP (the WS communication protocol) messages into the respective DO communication protocol and vice versa (depending on the direction of the communication flow) is left to the proxy.

1) Bridging CORBA Objects: In 2001, the Object Management Group (OMG) published the WSDL/SOAP to CORBA [14] and the CORBA to WSDL/SOAP [15] interworking specifications that provide a framework for the exposing of CORBA services as Web Services.

Concrete bridging implementations are provided by IONA Orbix [8], Borland Visibroker [3], Pontifex [1], and Apache Axis2 [19]. Since the first two are commercial platforms, hence lacking on design and implementation information, our discussion will focus only on the remainder (Pontifex and Axis2).

Pontifex is a generic bridge generator. Although the authors describe an architectural design, a CORBA implementation and a performance analysis, no implementation was found. The performance analysis is however interesting, since it compares the CORBA solution using Pontifex against a pure Web service. Although the case-study is a simple application, the analysis demonstrates that the overhead introduced by Pontifex, and bridging in general, is acceptable and that should not degrade performance.

Axis2 is an open-source Web service engine developed by the Apache Software Foundation. It currently features a module that allows a Web service client to invoke CORBA services in a completely transparent way. An incoming service invocation aiming for a CORBA service is intercepted by an handler and delivered to the CORBA Message Receiver that converts it to the CORBA communication protocol, following the usual bridging mechanism.

2) Bridging Jini Objects: The General brOkering Architecture Layer (GOAL) [6] is a service architecture that was initially thought for the development and management of highly-flexible, scalable and self-configurable distributed, and service-oriented applications over the Internet. GOAL runs on top of Jini, relying on it for dynamic registration, service lookup, notification of remote events, distributed object access and platform-independence.

Service clients and providers must be implemented in Java, since the communication is based on the exchange of Java objects. To overcome this restriction two meta-services were added, promoting the interoperability between GOAL services and Web services. The first meta-service, GOAL2WS, generates and deploys a bridge that makes a Jini service accessible as a Web service. The second, WS2GOAL, generates and deploys a bridge that performs the inverse operation, thus enabling have full platform interoperability.

3) Bridging DCOM Objects: Softing commercializes a solution to bridge communication gaps between applications. This includes a DCOM bridge [18] but no architectural information is available.

B. Service Component Architectures

The Service Component Architectures (SCA) model [16] is a step further in service interoperability, allowing the assembly and composition of heterogeneous services, i.e., services developed in distinct languages, such as Java, C#, and so forth. Legacy services can also be assembled by being wrapped into Web services, where wrapping is just a different terminology for the previously introduced bridging concept.

From a DO platform’s point of view, to bridge a subset of its composing services requires the wrapping of each of these, and the exposure of their interface in one of the supported interface description languages, of which WSDL or SCDL (an internal format) are examples. This wrapping is a technology dependent operation, since it includes the invocation of the legacy service and the relaying of the operation’s result.

As will be clear throughout the paper, our approach is conceptually different. The bridging unit is the platform rather
that the service. The focus is mainly on inter-platform interoperability and on the deployment of new platform services by the means of orchestration, while SCA focuses essentially on intra-platform service interoperability.

III. THE OHMS FRAMEWORK

The scope of this research is to provide support for the orchestration of DO middleware services in the context of their platforms. Our main requirements are: 1) to have a platform-centric approach, i.e., to focus on service-oriented DO platforms, rather than individual services; 2) to avoid, at all cost, the need to alter the platform’s implementation in order to be suitable for bridging and orchestration; and 3) to have a general solution that is not bound to any particular technology.

To meet such requirements we designed an architecture composed of two independent modules: naming-service directory module (or simply directory) and the orchestration module.

The directory module bridges a DO platform, partially or completely, by storing: a) information on its naming-service, such as the IP address and which services are to be bridged, and b) the logic required to expose these services. The orchestration module provides the means for the actual orchestration of the exposed services. It is an platform-centered extension of the Eclipse BPEL plug-in [4] that provides a simple way to access and orchestrate services of previously bridged platforms.

Figure 2 portrays the two modules and how these interact. As will be described in detail in Subsection 3.1, exposed DO services are registered on a UDDI registry embedded in the directory. This registry can be seen as the glue that binds the two components, since the orchestration module will be able to access and orchestrate the services it stores. All external requests are handled by the directory through a dedicated peer (the registration peer). A more detailed description of both the directory and the orchestration modules follows.

A. The Naming-service Directory Module

The directory is the core of OHMS. It is a general solution that is not bound to any particular DO technology, thus providing full interoperability. From the directory’s point of view, a DO technology is a set of Java classes that encapsulate all the logics necessary to bridge a service of the given technology (CORBA, DCOM, Jini, and so on). By logic we mean the computational steps necessary to perform the following three actions: 1) Extract the services to bridge, which requires holding the knowledge of which services are published in the naming-service of the target platform. This requires either the interception of the naming-service related communication, or the inspection of the naming-service’s registry. 2) Generate the bridge for each of these services. 3) Register the resulting bridges on the UDDI registry, making them visible to the orchestration module.

The overall architecture of the module, presented in Figure 3, embeds a Web server and the UDDI registry previously mentioned. The former serves as an access point to the Web services that relay the incoming invocations to the target platform services. The latter holds the registry of these Web services, publishing them to the network and serving as glue to bind both modules of the framework. Registered technologies are kept in map that associates the technology’s identifier to the bridging logic.

The remainder of this subsection will address the registry of technologies and platforms.

1) Technology Registry: The directory provides a Web service (registration peer in Figure 3) with operations to register, update and unregister technologies, supporting only the bridging of platforms from the technologies currently registered. This approach provides complete technology, and even version, independence. A directory may hold the registries of distinct technologies, such as CORBA or DCOM, or different registries for the same technology, e.g. CORBA 2.1 and 3.0.

Listing 1 presents a Java-like specification of the technology management interface.

```
Listing 1. The technology registry management interface

OpStatus registerTech(String techId, Logic logic);
OpStatus updateTech(String techId, Logic logic);
OpStatus removeTech(String techId);
```

Data-type Logic encapsulates the Java classes required to implement the bridging logic, as well as all their dependencies. Among theses classes one, the main class, must be compliant with a specific interface required by OHMS to enable and
interact with the technology.

Data-type OpStatus stores the status of the invoked operation: whether it concluded successfully or not, and, in the latter case, which was the resulting error.

The registerTech operation receives a new technology identifier and a bridging logic. Upon reception, the logic implementation classes are locally stored on the directory’s file-system and a new entry is added to the registered technologies’ map.

OHMS is platform-centric, hence, as is illustrated in the Figure 3, the directory does not keep services but rather information on the naming-services of the platforms to bridge. These naming-services are kept in their original location, completely unaware of the bridging process. Typically a technology will require their whereabouts and information on which services to bridge. Note, however, that the handling of naming-service registrations is delegated on the target technology, and, thus, many approaches are possible. This subject will be further discussed throughout this subsection.

Each technology bridging logic is executed by a dedicated execution flow. However, for the sake of resource usage optimization, this flow is only enabled when the technology has registered platforms. For this purpose the directory accounts the number of platforms registered for each technology, and ensures that CPU consumption is restricted to technologies with actual work.

The updateTech operation replaces the classes associated to a given technology (causing the latter to restart), while operation unregisterTech eliminates a technology from the system.

2) Platform Registry: The registry of platforms is also performed through the directory’s registration peer. The set of operations provided (Listing 2) is equivalent to the one for technologies.

Listing 2. The platform registry management interface

```java
OpStatus registerPlatform(String platformId, String techId,
                          Bytecode properties)
OpStatus updatePlatform(String platformId, String techId,
                         Bytecode properties)
OpStatus removePlatform(String platformId, String techId)
```

The registry of a platform requires: a technology-wide unique identifier; the identifier of the (previously registered) DO technology; and a properties file holding all the platform’s specific information. In order to keep the bridging process completely transparent and orthogonal to the platform’s implementation, the properties file must contain all the information required by the technology, such as the naming-service’s IP address and which are the services to bridge. Note that the set of services to bridge result from the interception of the services listed in the properties file and the ones actually registered in the naming-service.

The actual registry operation is not performed directly by the directory module. Given the technology’s identifier, the directory retrieves the associated logic from the registered technologies map and delegates the procedure. It is up to the bridging logic to: 1) select which services to bridge; 2) bridge the selected services, and; 3) register them in the embedded UDDI registry. Note that the bridging overhead is technology and implementation dependent, which may result in performance discrepancies.

The updatePlatform operation is also particularly important, since it allows for the modification of the platform’s properties on-the-fly, e.g. alter the location of its naming-service or the set of services to bridge without having to stop the directory.

3) Unbridging a service or a platform: Two factors can lead to the unbridging of a service: 1) its removal from the set of services to bridge, by updating the naming-service’s properties file (updatePlatform operation), and; 2) the detection, by the technology bridging logic, that the service has been removed from the platform’s naming-service. Both scenarios cause the bridge to collapse, thus making the serve inaccessible from the Web and unavailable to the orchestration module.

B. The Orchestration Module

The orchestration module is a platform-centric extension of the Eclipse BPEL plug-in. It allows the user to connect and import services from a DO platform, using the directory’s UDDI registry as intermediate. Once imported, these services can be treated as common partner links in the construction of a BPEL process.

Our extension to the plug-in addresses mostly the introduction of the platform concept. Partner links can now be added by browsing and selecting the services available on a given DO platform. These platform defining configurations can be stored for posterior retrieval. Figure 4 illustrates the use of the plug-in to perform an orchestration featuring services originating from a DO platform - Login and Domain - and a Web service - Weather Forecast.

For the sake of implementation independence, the plug-in does not resort to any particular UDDI implementation. Thus, it can be used out of the OHMS context to browse and import the contents of any UDDI registry.
C. Handling Heterogeneity

Service composition in OHMS is not restricted to the internals of a platform. As is illustrated in Figure 5, several platforms can use an OHMS layer (comprised of one or more OHMS platforms) to combine services from distinct platforms and possibly incorporate other Web available services. This provides full interoperability support at a platform-level and centric way, a concept different from SCA.

There are however limitations to this approach, due to the inability of the WS model to cope with events and DO references. The former can be overcome with the use of WS-Notification [10], a topic for future research. The latter, relates to the fact that the WS model is more loosely-coupled that, for instance, CORBA. WS references are WSDL documents not internal language representations. To allow for reference passing, the WS bridges must be able to translate these references in both ways. To the best of our knowledge, none of the existing bridging technologies perform such translations.

IV. BRIDGING A CORBA-BASED PLATFORM

This section addresses the use of OHMS to expose CORBA-based platforms. We have chosen CORBA to illustrate the development of a bridging logic because it is still the most used technology in the development of DO-based systems, and thus of interest to a wider audience.

As discussed in Section III, to bridge a DO platform is to register its naming-service in the OHMS directory, supplying the identifier of the correspondent technology and the server’s properties file. In turn, to support a DO technology is essentially to define and register its bridging logic. Next we discuss these issues in the context of the CORBA technology.

A. A CORBA Bridging Logic

The implementation of a bridging logic is a three step process (Subsection III-A) that drives the structure of this subsection.

1) Select the Services to Bridge: The selection of which are the services to bridge is a two phase process: Phase 1 is performed as soon as the platform is registered and consists on checking which of the services listed on the platform’s properties file are actually on the platform’s service registry. Phase 2 consists on detecting registry alterations, in order to ensure that only valid bridges are kept and that these are up-to-date.

Regarding CORBA, our initial approach was to accomplish phase 1 by polling the registered platform’s Common Object Services (COS) naming service [13] (or simply COSnaming). From that point, new registrations and de-registrations (phase 2) would be intercepted by resorting to the CORBA Portable Interceptors mechanism [12]. Unfortunately, this mechanism is not supported by many of the most used implementation of COSnaming, such as omniNames [2]. This conflicted with our premise of providing portable solutions, and therefore, for the sake of portability, we chose to resort uniquely to polling.

To deal with the overhead usually introduced by these strategies, we pushed to the properties file the definition of the polling time interval and the ability to enable and disable the mechanism. This allows for a better, and platform-tuned, management of the polling strategy. For instance, stable platforms that do not envision the registry of new services can simply disable polling, eliminating the overhead.

2) Generate a Bridge for Each Service to Expose: We resort to the Axis2 CORBA module to bridge individual CORBA services. Given the IDL of the service to bridge and a configuration file, the module generates the equivalent WSDL interface and the proxy required to relay the invocations. The configuration file contains service metadata, such as an informal description of the service, the name and path of the IDL file, the location of the service, its abstract name, its interface name, and so forth.

Both the IDL and the metadata information cannot, in general, be obtained directly from COSnaming. Usually, this level of abstraction is no longer present after the service’s registry. Some COSnaming implementations feature the Interface Repository ORB, that provides the means to obtain information about a service’s interface. But, once again, this feature is not available in all implementations.

We have thus chosen to follow another approach, close to the one found in Axis2. Both files have to be deposited in the directory’s folder tree. Note that a bridge in Axis2 is represented by the pair (IDL file, configuration file). This means that different instances of the same IDL will have distinct configuration files.

The actual bridging process includes the creation of inner control data-structures to keep track of which are the services currently bridged, and the copy of the correspondent IDL and configuration files to a pre-determined Axis2 folder.

3) Register the Bridges in OHMS’s UDDI: Once the bridging process is concluded, the resulting WSDL file is available at a specific URL from which the service is Web accessible. This location must be stored on the UDDI registry embedded in OHMS. Since this task is not bound to any particular DO-technology, it is factorized in a class featured in OHM’s’s API.

B. Bridging and Unbridging CORBA services

The CORBA service registration process is not altered, nor a platform needs to be halted in order to be bridged. An CORBA enabled OHMS shall detect any COSnaming operation and
trigger the bridging process, making the service available as a Web service and suitable for orchestration.

The unbridging, caused by either of the factors enumerated in Subsection III-A3, results in the deletion of the service configuration file from the Axis2 folder tree (not from the directory) which automatically causes Axis2 to disable the bridge. Since a IDL file may be affected to several service instances, i.e., to several configuration files, it is only deleted when no associated configuration files are left. With the bridge out, registration in the UDDI repository is also removed.

Regarding an entire platform, the de-registering operation causes the process described above to be triggered for each service of the platform that was bridged.

V. EVALUATION

The OHMS framework was evaluated regarding the functional requirements, operationality, the effort required to implement a bridging logic, and performance.

A. Functional Evaluation

This evaluation verified that the framework respects the initial requirements, namely: 1) to be platform-centric in opposition to service-centric; 2) to avoid the need to alter the platform’s implementation in order to be suitable for bridging and orchestration; and 3) to be a general solution that is not bound to any particular technology.

The first requirement was met by featuring a name-service directory that stores the name-server of the platform to be bridge. The directory module does not feature operations to manipulate individual services but rather entire platforms.

The second and third requirements lead to the definition of the bridging logic concept, to encapsulate all the logic required to bridge the services of a platform implemented on top of a given DO technology. This design provides a bridging functionality that is completely transparent to the hosting platform, which includes the absence of alterations on the platform’s original code. This is very important because it widens OHMS’s scope of application to platforms whose source code is not available. Moreover, the segregation of the logic required to bridge a DO technology from the core of the framework enable us to easily met the third requirement and propose a general solution. The support for new technologies can be added by resorting to the technology registry management interface.

B. Operational Evaluation - Application to COMCOP

As stated in section I, the driving-force for this work was the possibility of adding new services to the COMCOP platform by orchestrating the existing ones, along with other Web services. This section provides a general overview of COMCOP and of how OHMS was used to meet the initial expectations.

C & C platforms provide the facility of monitoring and controlling assets (either mobile or stationary). They have evolved from a tight coupling with military operations to a broader range of usage. COMCOP has an application range that crosses the aerospace, defense and civil markets. It is a service-oriented architecture infrastructure implemented completely in CORBA. Its internal architecture is composed of a core layer that includes general (solution independent) services, and a solution specific layer that includes the services required to specify the platform for a given purpose.

Usually the services exposed by a platform to the outside world comprise operations that follow a call-by-value semantics. This is the design that matches with the developed CORBA bridging technology, since, for now, reference exchanging is not supported. However, COMCOP goes a step further, even the internal operations are called by value, which permitted the use of OHMS in the building of new internal services.

Thus, the use of OHMS in this context enabled the population of this solution specific layer with services developed by composing existing platform services, extended with extra functionalities, and by composing these with publicly available Web services.

An example of a new implemented service is one that takes weather forecast into account when choosing the domain of assets for a given mission, e.g., when performing a sea rescue mission the sea conditions may determine the type of assets to use: maritime, such as a high-speed boat; or aerial; such as an helicopter.

The service was developed by orchestrating an available weather forecast Web service with the COMCOP services that manage domains of assets. By being also registered in the OHMS UDDI, the new services can, in turn, be partners in new orchestrations. For instance, a mission planning service may plan the missions according to the assets selected by the asset selection service.

This application of OHMS in a real-life scenario provided an ideal test-bed to evaluate operationally both the directory and the orchestration modules, as well as the implemented CORBA support. OHMS is currently able to bridge and orchestrate services originating from CORBA-based platforms, with the requirement that references must not exchanged between services.

C. Bridging Logic Development Effort

The development of a bridging logic follows the three-step process presented in subsection III-A, which in practice requires the implementation of two Java methods: the first to inspect the name-server of the target DO platform, and the second to construct the bridges. Remember that the registry in the UDDI directory is provided by the API.

The CORBA support presented in section IV was implemented in less than 500 lines of Java code. This includes polling the COSnaming server in order to inspect the target platform’s name-server and the interaction with Axis2 in order to create the bridges.

D. Performance

The purpose of OHMS is to construct bridges between Web services and DO services by resorting to external tools, e.g.
Axis2 CORBA module. The impact of this construction is, usually, restricted to a setting-up stage. We have no knowledge of scenarios that require a steady construction of bridges.

Once the bridges are up, the overhead of OHMS is limited to the handling of naming-service registrations, either by eavesdropping the requests or by periodically inspecting the service’s registry. The impact of these actions is negligible and should even be switched off when the set of services to bridge is static.

In conclusion, the performance of the overall system is bound to the performance of the bridging technology, e.g. the performance of the CORBA support presented in section IV is bound to the performance of the Axis2 CORBA module. The Apache Foundation does not provide any data on this subject, but the analysis presented in [1] concludes that overhead introduced by a CORBA bridge is acceptable and should not degrade performance.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we presented OHMS, a platform-centric framework for the Web exposure and orchestration of DO services. OHMS provides an easy and not resource consuming way of porting DO platforms to the WS world, enabling Web access, business-to-business interaction and service composition.

By registering their naming-service in the OHMS directory, DO platforms automatically expose their set of services to the Web. The process is completely transparent to the platform, and thus no alterations on its implementation are required. The definition of which services to bridge is coded on a properties file supplied to the directory during the platform’s registry.

OHMS’ directory is a compact engine that resorts to technology-dependent components, the bridging logics, to manage platform exposure. All of the work goes, thus, into developing the logic required by a given technology, something that must be done only once, and may be shared by the community.

Orchestration in OHMS is also platform-centered, the orchestration module connects to a platform, allowing for the browsing and retrieval of the bridged services. This module is, in fact, a platform-centric extension of the Eclipse BPEL plug-in. Thus, platform services can be orchestrated as any other Web service in a BPEL process, as well as be orchestrated with other services available in the Web.

As a proof-of-concept, in this paper we presented the implementation of a logic to bridge CORBA services. The whole three-step process (COSNaming polling, Axis2 interaction to create the bridges, and UDDI registry) was implemented in less than 500 lines of code.

A performance evaluation was also conducted and the conclusions indicate that, once the bridges have been built, OHMS impact is negligible. The weight of the overhead from Web exposure is bound to the technology used to perform the actual bridging, e.g. Axis CORBA module.

Concerning OHMS applicability, the COMCOP case-study illustrates how C & C and DO platforms in general can greatly benefit from the framework. OHMS enhanced COMCOP’s functionality and flexibility by allowing the creation of new services through the combination of existing ones. The scope of the framework’s applicability to C & C can be further extended to provide platform interoperability by allowing the orchestration of the services originating from the different platforms, and thus defining a higher abstraction layer. This interoperation is very important in scenarios where different C & C platforms have to work in conjunction to accomplish the same goal (e.g. disaster scenarios). This capability is, however, still restricted by the number of supported technologies, a limitation that we intend to correct soon.

Currently we are working on supporting the exchange of CORBA references and on the implementation of other technologies, such as DCOM. To the best of our knowledge, there are no non-commercial WS bridges that address the DCOM technology, which will add some extra complexity.

REFERENCES


