# A Web Service-Oriented Architecture for Implementing Web Information Systems

Flavius FRASINCAR [a,b], Geert-Jan HOUBEN [a,c], and Peter BARNA [a]

[a] *Eindhoven University of Technology*
*Den Dolech 2, 5612 AZ Eindhoven, the Netherlands*
[b] *Erasmus University Rotterdam*
*Burgemeester Oudlaan 50, 3062 PA Rotterdam, the Netherlands*
[c] *Vrije Universiteit Brussel*
*Pleinlaan 2, B-1050 Brussels, Belgium*

**Abstract.** Web services are one of the most popular ways for building distributed Web Information Systems (WIS). In this paper we propose a distributed implementation of the Hera methodology, a methodology based on models that specify the different aspects of a WIS. The Web services used in the implementation are responsible for capturing the required data transformations built around specific Hera models. A service orchestrator coordinates the different Web services so that the required WIS presentation is built. Based on the degree of support of the user interaction with the system two architectures are identified: one for the construction of static applications, and another one for the building of dynamic applications.

**Keywords.** Web Information Systems, Web services, RDF(S)

## Introduction

The Web The Web has more than three billion pages and around half a billion users. Its success goes beyond national frontiers and imposes it as the first truly global information medium. While the nineteenth century was dominated by the “industrial revolution", the beginning of this new century is marked by an “information revolution" with the Web as its main “engine".

A Web Information System (WIS) [1] is an information system that uses the Web paradigm to present data to its users. In order to meet the complex requirements of a Web Information System several methodologies have been proposed for WIS design. In the plethora of proposed methodologies we distinguish the model-driven ones that use models to specify the different aspects involved in the WIS design. The advantages of such model-based approaches are countless: better understanding of the system by the different stakeholders, support for reuse of previously defined models, checking validity/consistency between different design artifacts, (semi-)automatic model-driven generation of the presentation, better maintainability, etc.

Based on the principle of separation of concerns, model-driven methodologies like OOHDM [2], WebML [3], UWE [4], SiteLang [5], and Hera [6] distinguish: a conceptual model that describes the schema of the (multimedia) data to be presented, a navigation model that specifies how the user will be able to navigate through the data, and a presentation model that gives the layout and the low-level characteristics (e.g., font size, color, etc.) of the hypermedia presentation to be generated. For the model representations, these methodologies make use of different technologies: OOHDM is based on the OO paradigm, WebML offers a set of visual units (serialized in XML) to be graphically composed, UWE is based on the UML notation extended with a Web UML profile, SiteLang suggests a mathematical language (story algebra), and Hera uses Semantic Web technology to make explicit the semantics captured in a certain model and to subsequently exploit this semantics.

Today there is an increasing need for making WIS components interoperable. An isolated WIS is not able to provide all the information/computational power required by an organization. Web services (WS) appear to be the most popular mechanism to support application interoperability. Their success is based on the fact that they are cross-platform and cross-language distributed applications. The fact that the provider and the requester of a WS are loosely coupled ensures application robustness. A disadvantage of such an approach is the XML messaging involved, which affects application speed. Nevertheless using appropriate optimization techniques and/or fast computers/networks this disadvantage can be overcome. We distinguish between two classes of WS: WS that provide data-on-demand, and WS that offer computation-on-demand. As an example for the first type of WS we mention WS offered by Amazon or Google, and for the second type of WS we refer to the services developed in the Grid project [7].

Different science communities saw the opportunity of using WS to enable data sharing between their systems. For example, the geoscience community came up with a service-oriented solution for the interoperability of geographical information systems aiming at providing better forecasts. The proposed WS solution [8] is compared with a puzzle (of WS) as depicted in Figure 1. In this puzzle, shapes represent interfaces, and colors (shades of gray for a black and white printing) stand for data models/encodings. Interfaces are the operations that a WS provides, and data models/encodings are the input/output parameter types of the WS operations.

Most model-driven methodologies for WIS design have modular implementations made from components driven by different models. Based on the Hera methodology, this paper proposes the use of WS in realizing the different components and aggregating them in a WIS. In this way the software plug-and-play vision can be realized in the context of WIS.

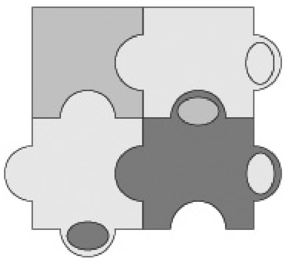

**Figure 1.** Web services puzzle.

## 1. Hera Methodology

Hera [9] is a model-driven methodology that uses Semantic Web technologies for building WIS. There are two main phases in Hera: the data collection phase responsible for integrating the data coming from different sources, and the presentation generation phase responsible for constructing a Web presentation of the integrated data. The focus of the paper is on the second phase of Hera, nevertheless some of the ideas presented here can be applied also for the data collection phase.

As a model-driven methodology, Hera distinguishes design models (specifications) that capture the different aspects of the presentation generation phase in a WIS. The conceptual model (CM) describes the schema of the data that needs to be presented. The application model (AM) expresses the navigational structure to be used while browsing through the data. The presentation model (PM) specifies the look-and-feel aspects (layout and style) of the presentation. The Hera methodology also proposes an implementation which builds a WIS based on the previously specified models.

One of the characteristics of the Hera methodology is its specific focus on personalizing the WIS based on the user characteristics and user browsing platform. The user profile (UP) is responsible to store the attribute-value pairs that characterize the user preferences and platform properties. These properties are used to customize the presentation generation before the user starts the browsing session. The adaptation technique utilized for implementing WIS personalization is based on adding visibility conditions to elements appearing in the different Hera models. These conditions make use of the data stored in the UP in order to define the presence/absence of a model element.

Another feature that Hera supports for a WIS is the ability to consider the user interaction with the system (by user interaction we mean here the interaction by forms in addition to merely link following) before the next Web page is generated. For this purpose the User Session (US) was defined to store all the data created at run-time by the user. The forms and the queries responsible for building new data are described in the AM.

All Hera models are represented in RDF(S), the Semantic Web foundation language. In this way we were able to cope with the semi-structured aspects of data on the Web (by using the loose RDFS schema definitions), support application interoperability (by making available the data semantics as RDFS descriptions), reuse existing models (by utilizing the refinement mechanisms supported by RDFS), etc. RDF(S) also allows for compact representations of the models as intensional data can be provided at run-time by an inference engine. For the user profile we did reuse an existing RDFS vocabulary, User Agent Profile (UAProf) [10], an industry standard for modeling device capabilities and user preferences.

As a query language we used SPARQL [11] one of the state-of-the-art query languages for RDF models. Differently than other RDF query languages (e.g., RQL) SPARQL allows for the construction of new models in addition to extraction of relevant information from existing models. This feature was useful for generating new data used for populating the US.

The implementation of the Hera methodology is based on several data transformations operating on the Hera models. There are three types of transformations: application-dependent, data-dependent, and session-dependent. The application-dependent transformations operate on CM, AM, and PM, and do not consider the instances of these models (e.g., model adaptation). The data-dependent transformations convert one model instance into another model instance (e.g., CM instance into AM instance). The session-dependent transformations make use of the data provided by the user (e.g., populating the US with form-related data).

In order to support the presentation of the Hera models we use a running example, a WIS that depicts Vincent van Gogh paintings, and allows the users to order posters of the displayed paintings. Figure 2 shows a snapshot of the hypermedia presentation for the considered WIS. It presents a Self Portrait painting of Vincent van Gogh.

### *1.1. User Profile*

The user profile (UP) specifies the user preferences and device capabilities. It is a CC/PP vocabulary [12] that defines properties for three components: `HardwarePlatform`, `SoftwarePlatform`, and `UserPreferences`. Figure 3 illustrates an excerpt from a UP. For the hardware characteristics one such property is `imageCapable`. This property is set to `Yes` for devices that are able to display images. The software characteristics have, among others, the property `emailCapable`. This property is set to `Yes` for browsers that are able to provide email functionality. For user preference characteristics one such property is `isExpert`. This property is set to `Yes` if the user is an expert of the application domain.

### *1.2. Conceptual Model*

The conceptual model (CM) represents the schema of the data that needs to be presented. It is composed of concepts, concept relationships, and media types. There are two kinds of concept relationships: attributes which relate a concept to a particular media type, and inter-concept relationships to express relations between concepts. A CM

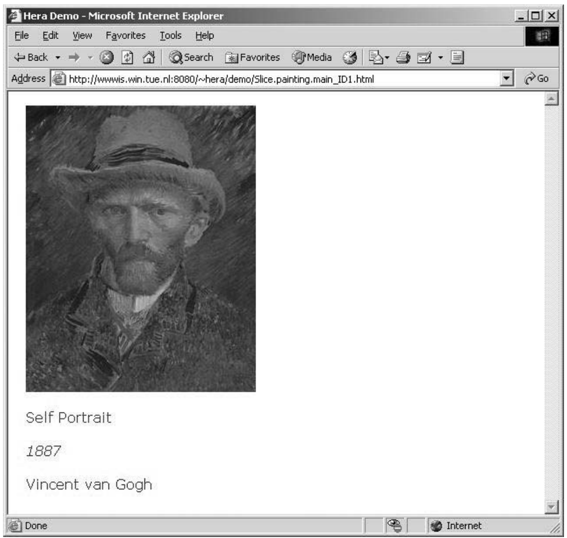


**Figure 2.** Presentation in Internet Explorer.

```
<ccpp:component rdf:ID="comp_ID1">
   <HardwarePlatform>
      <imageCapable>Yes</prf:imageCapable>
      ...
   </HardwarePlatform>
</ccpp:component>
<ccpp:component rdf:ID="comp_ID2">
   <SoftwarePlatform>
      <emailCapable>Yes</prf:emailCapable>
      ...
   </SoftwarePlatform>
</ccpp:component>
<ccpp:component rdf:ID="comp_ID3">
   <UserPreferences>
      <isExpert>Yes</isExpert>
      <levelOfVision>Normal</levelOfVision>
      ...
   </UserPreferences>
</ccpp:component>
```

**Figure 3.** Excerpt from UP serialization in RDF/XML.

instance (CMI) gives the data that needs to be presented. This data might come from an integration/retrieval service that is out of the scope of this paper. Figure 4 illustrates an excerpt from a CMI. It describes a `Self Portrait` painting of Vincent van Gogh. Such a painting exemplifies the Impressionism painting technique identified by `Technique_ID2`. In the description one can find the year in which the painting was made, i.e., `1887`, and the URL of the image depicting it.

The adaptation condition used in this case specifies that the image of the painting is shown only if the device is image capable.

```
<Painting rdf:ID="Painting_ID6">
   <name>
      <type:String>
         <type:data>Self Portrait</type:data>
      </type:String>
   </name>
   <year>
      <type:Integer>
         <type:data>1887</type:data>
      </type:Integer>
   </year>
   <picture>
      <type:Image rdf:about="http://.../SK-A-3262.ORG.jpg"
                  a:condition="up:imageCapable = 'Yes'"/>
   </picture>
   <exemplifies rdf:resource="#Technique_ID2"/>
   <painted_by rdf:resource="#Painter_ID2"/>
</Painting>
```

**Figure 4.** Excerpt from CMI serialization in RDF/XML.

### *1.3. Application Model*

The application model (AM) gives an abstract view of the hypermedia presentation that needs to be generated over the CM data. It is composed of slice and slice relationships. A slice is a meaningful presentation unit that groups media items coming from possibly different CM concepts. The AM is in this way built on top of the CM. A slice that corresponds to a page from the presentation is called a top-level slice. There are two kinds of slice relationships: aggregation which enables the embedding of a slice in another slice, and navigation which specifies how a user can navigate between slices. An AM instance (AMI) populates an AM with data from a CMI. Figure 5 depicts an excerpt from an AMI. It presents the slice `Slice.painting.main_ID6`, a top-level slice associated to the painting presented in the above subsection. This slice refers using slice aggregation relationships to four other slices. Each of the four slices contains a media item related to different painting/painter attributes.

The adaptation condition used in this example specifies that the year in which a painting was made is presented only for expert users. Other users, like for example the novice users, will not be able to view this information.

Let's consider now that the application is dynamic in the sense that the user is allowed to buy posters depicting paintings. In this case the user can place orders based on the currently displayed painting and a specified quantity (by means of a form) of this poster. The US stores the data that is created by the user at run-time. In our example the US contains the order to be added to the shopping trolley. Figure 6 shows an example of a query used in the AM to create an order (variable `x`) that stores the desired painting (variable `id`, the currently displayed painting), and the required quantity (variable `v`, the number input by the user in a form).

```
<Slice.painting.main rdf:ID="Slice.painting.main_ID6">
   <slice-ref rdf:resource="#Slice.painting.name_ID6"/>
   <slice-ref rdf:resource="#Slice.painting.year_ID6"/>
   <slice-ref rdf:resource="#Slice.painting.picture_ID6"/>
   <slice-ref rdf:resource="#Slice.painter.name_ID2"/>
</Slice.painting.main>

<Slice.painting.name rdf:ID="Slice.painting.name_ID6">
   <media>
      <type:String>
         <type:data>Self Portrait</type:data>
      </type:String>
   </media>
</Slice.painting.name>
...

<Slice.painting.year rdf:ID="Slice.painting.year_ID6"
                    a:condition="up:isExpert = 'Yes'">
   <media>
      <type:Integer>
         <type:data>1887</type:data>
      </type:Integer>
   </media>
</Slice.painter.name>
```

**Figure 5.** Excerpt from AMI serialization in RDF/XML.

```
CONSTRUCT { ?x <rdf:type> us:Order .
            ?x <rdf:contains> ?y .
            ?y <rdf:ID> ?id .
            ?x <us:quantity> ?v }
WHERE     { BuyForm <bf:quantity> ?v .
            Session <var:paintingid> ?id }
```

**Figure 6.** A SPARQL query.

The SPARQL language was extended with new constructs like: URI generator (e.g., the order identifier), aggregation functions (e.g., counting the number of orders that are in the trolley), and delete statement (e.g., the user should be able to delete orders from the trolley).

### *1.4. Presentation Model*

The presentation model (PM) defines the look-and-feel of the application. It is composed of regions and region relationships. A region is an abstraction for a rectangular part of the user display area. Each region corresponds to a slice that will provide the content to be displayed inside the region. A top-level region is a region that is associated to a top-level slice. There are two types of region relationships: aggregation relationships, which allow the embedding of a region into another region, and navigation relationships, which allow the navigation from one region to another region.

A region has a particular layout manager [13] and style associated with it. There are four layout managers: BoxLayout, TableLayout, FlowLayout, and TimeLayout. These layout managers describe the spatial/temporal arrangement of the regions in their container region. They allow us to abstract from the client-dependent features of the browser's display. The style specifies the fonts, colors, backgrounds, etc., to be used by a certain region. The definition of the style properties allows a uniform representation of style information irrespective of the CSS compatibility of the client. Regions that do not a have a style defined inherit the style of the container region. The top-level regions always need to have a style defined.

A PM instance (PMI) populates a PM with data from an AMI. Figure 7 depicts an excerpt from a PMI. It presents the region `Region.painting.main_ID6`, a top-level region associated to the considered Self Portrait painting. This region refers using region aggregation relationships to four other regions. Each of the four regions contains a media item related to different painting/painter attributes. The top-level region has a `BoxLayout` associated with a `y` orientation and the style uses `times` fonts.

The adaptation condition presented in this example states that a default style with normal fonts should be used for users with a normal level of vision. For users with a poor level of vision a default style with large fonts is prescribed.

## 2. Hera Web Service-Oriented Architecture

The previous implementation of Hera was based on one centralized application. This application is made of several components each responsible for performing a specific set of data transformations. In this paper we will show how one can distribute such an implementation by mapping components to WS. One of the advantages of using such an im-

```
<Region.painting.main rdf:ID="Region.painting.main_ID6"
                      layout="#BoxLayout_ID1"
                      style="#DefaultStyle_ID1"
                      style="#DefaultStyle_ID2">
   <region-ref rdf:resource="#Region.painting.name_ID6"/>
   <region-ref rdf:resource="#Region.painting.year_ID6"/>
   <region-ref rdf:resource="#Region.painting.picture_ID6"/>
   <region-ref rdf:resource="#Region.painter.name_ID2"/>
</Slice.painting.main>

<BoxLayout rdf:ID="BoxLayout_ID1"
           axis="y"
           width="100%"
           ...
</BoxLayout>

<Style rdf:ID="DefaultStyle_ID1"
       a:condition="up:levelOfVision = 'Normal'"
       font-family="times"
       font-size="normal"
       ...
</Style>

<Style rdf:ID="DefaultStyle_ID2"
       a:condition="up:levelOfVision = 'Poor'"
       font-family="times"
       font-size="large"
       ...
</Style>
```

**Figure 7.** Excerpt from PMI serialization in RDF/XML.

plementation is the loose coupling of Hera WS which can help realize the plug-an-play software vision in the context of WIS. For example a WIS can be constructing by composing a WS responsible for delivering the data, a WS that builds a navigations structure over this data, a WS that adds the layout and style details, and a WS that generates the presentation code.

The WS solution used for realizing the distributed Hera architecture has clear advantages compared to their predecessors CORBA, J2EE, and DCOM [14]. First of all WS are based on the XML document paradigm, a human readable language that abstracts from the implementation details. WS interfaces are specified in the universally accepted Web Service Definition Language (WSDL) [15], an XML-based language. Last but not least, WS use the popular HTTP protocol as the carrier of exchanged messages.

### *2.1. Hera Web Service-Oriented Architecture for Static Applications*

Figure 8 presents a Web service-oriented architecture (WSOA) for the Hera implementation based on six WS: the Data Service, the Application Service, the Presentation Service, the Code Generator Service, the User Profile Service, and the Adaptation Service. Note that this architecture is used to build static applications, i.e., applications for which the user is unable to change the presentation to be generated. After the description of this architecture we present another architecture that is used to generate dynamic applications, i.e., applications that take in consideration the user input data (from forms) before generating the next page.

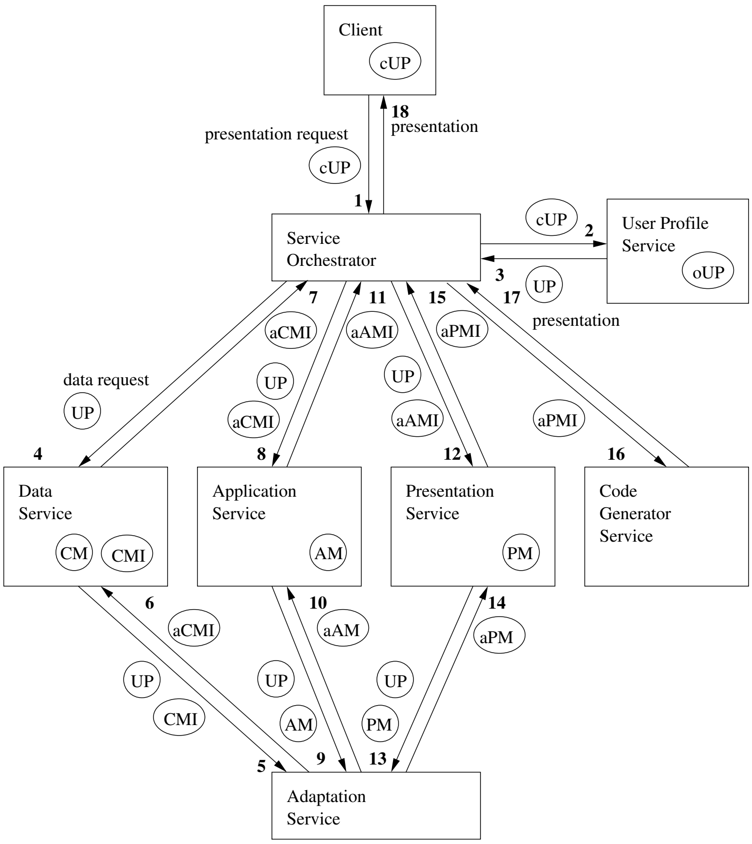


**Figure 8.** Web service-oriented architecture for static applications.

The User Profile Service is responsible for providing the UP to be used by the application. The Adaptation Service is able to execute the adaptation specifications for the CM, AM, and PM. The core services are: the Data Service, which provides the data, the Application Service, which builds the navigation structure over the data, the Presentation Service which sets the layout and style of the presentation, and the Code Generator, which constructs the actual presentation. We call these services core services because they can form a WIS even when the other services (i.e., the User Profile and the Adaptation Service) are absent.

In addition to these WS the application uses two other components: the Client which is the initiator of the presentation request, and the Service Orchestrator responsible for coordinating the collaboration among WS. The Service Orchestrator is the hub of the whole system as it is the interface with the client and the message dispatcher to the core WS. The communication between Client and services is done at SOAP [16] level which resides on top of HTTP while the communication between the Client (Web Server) and the Web Browser is done in plain HTTP.

Figure 8 also depicts the flow of information in our distributed system. Each message has a number associated that indicates the order in which these messages are exchanged.

The architecture components are depicted in rectangles, models are represented as ovals, and messages are shown as arrows. On the arrows one can read the models that are being passed, 'presentation request/data request' (as events), and 'presentation' which is the returned hypermedia presentation. All components are WS with the exception of the Client and Service Orchestrator. The models that are own by the application components are placed inside the corresponding rectangles.

First the Client sends a presentation request to the Service Orchestrator (step 1). It also passes the client UP (cUP), the UP residing on the Client (possibly given by the Web Browser). The Service Orchestrator sends the cUP to the User Profile Service (step 2). The User Profile Service computes an updated version of the UP, by integrating the cUP with the old User Profile (oUP), the user profile stored in this service. The profile attributes not defined in the cUP are taken from oUP, and the resulted UP replaces the oUP. The User Profile Service can be viewed as a shared memory service for user profiles. After the integration process the UP is passed back to the Service Orchestrator (step 3).

The Service Orchestrator sends a data request to the Data Service. It also passes the UP, the integrated user profile from the previous phase (step 4). The CM instance (CMI) and the UP are passed further to the Adaptation Service (step 5). The Adaptation Service has one task to fulfill: evaluate the conditions in the models (CMI, AM, PM) and prune the models from the elements that have an associated condition that does not hold. For the CMI these elements are concepts or media items. In our application the Adaptation Service works on an instance, i.e., the input data of the system (CMI), and two schema models (AM, PM), as the instances of the two schema models will be populated at run-time. The adapted CMI (aCMI) is passed back to the Data Service (step 6) which forwards it to the Service Orchestrator (step 7).

The Service Orchestrator sends the UP and the aCMI to the Application Service (step 8). The AM and the UP are forwarded to the Adaptation Service (step 9). The Adaptation Service removes the AM slices that have the associated condition not valid. The resulted adapted AM (aAM) is passed back to the Application Service (step 10). The Application Service transforms the aCMI into the aAMI, based on the specifications from the aAM. The resulted aAMI is forwarded to the Service Orchestrator (step 11).

The Service Orchestrator sends the UP and the aAMI to the Presentation Service (step 12). The PM and the UP are forwarded to the Adaptation Service (step 13). The Adaptation Service selects the appropriate layouts and styles for the PM by removing the PM elements that have the associated condition not valid. The resulted adapted PM (aPM) is passed back to the Application Service (step 14). The Presentation Service transforms the aAMI into the aPMI, based on the specifications from the aPM. The resulted aPMI is forwarded to the Service Orchestrator (step 15).

The Service Orchestrator sends the aPMI to the Code Generator Service (step 16). The Code Generator Service transforms the aPMI into code interpretable by the user's browser. At the moment the intelligence of this transformation is encapsulated in the service logic itself, but in the future we would like to have an explicit model that maps PM elements to the code specifics. We did experiment with several code generators: HTML, cHTML, HTML+TIME, WML, and SMIL. The resulted presentation is passed back to the Service Orchestrator (step 17), which forwards it to the Client (step 18).

Note that this is not the only way in which the information can flow through the system. For example the adaptations can be performed first, i.e., before transforming one model instance into another model instance.

### *2.2. Hera Web Service-Oriented Architecture for Dynamic Applications*

Figure 9 presents a WSOA implementation of Hera for dynamic Web applications. Differently than before where a full presentation was generated now we build one page-at-a-time as the application needs to take in account the user's input after each page before generating the next page. For this purpose we have added an extra service, the User Session Service responsible for collecting the user session data. We also have modified the logic of the Application Service so that the session data is considered in the data transformation that it performs.

Instead of presenting the full flow of information for this distributed architecture, we emphasize the major differences with respect to the distributed architecture for static applications.

At the beginning the Client sends besides the page request and the cUP, also the session data, which contains the data that emerges from the interaction of the user with the system (e.g., the data resulted from a form filling by the user). The presentation request (a presentation being a set of linked Web pages) used in the previous architecture is now replaced by a page request. The communication between the Service Orchestrator and the User Profile Service goes the same as before (steps 2 and 3).

The Service Orchestrator sends the page request, session data, and UP to the Application Service (step 4). In the previous architecture the Service Orchestrator could directly communicate with the Data Service as the full presentation was built at once. Here it is important to know the context of the presentation (what is the page that needs to be generated next), and this information is in the Application Service. For this reason the Service Orchestrator communicates first with the Application Service and not the Data Service.

The communication between the Application Service and Data Service (steps 5 and 8), and between the Data Service and Adaptation Service (steps 6 and 7) is similar as the communication between the Service Orchestrator and Data Service, and between the Data Service and the Adaptation Service from the previous architecture. The only difference is that the aCMI now corresponds only to the data necessary for computing the next Web page (instead of the whole presentation).

The Application Service sends the session data to the Session Service (step 9). The Session Service transforms the session data into a US instance (USI). After that the Session Service sends the USI back to the Application Service (step 10). The communication between the Application Service and the Adaptation service is the same as before (steps 11 and 12). Based on the aAM, the Application Service transforms the aCMI and USI into aAMI. As the aCMI, the aAMI contains only the data that is relevant for the current page. The computed aAMI is passed back to the Service Orchestrator (step 13).

The communication between the Service Orchestrator and the Presentation Service (steps 14 and 17) and between the Presentation Service and the the Adaptation Service (steps 15 and 16) is similar as before. Also the communication between the Service Orchestrator and the Code Generator (steps 18 and 19), and between the Service Orchestrator and the Client (step 20) is done in the similar way as in the previous architecture.

The only difference is that the aPMI corresponds only to one page and the presentation is being replaced by one page.

Note that this is not the only way in which the information can flow through the system. For example the communication with the User Profile and the Adaptation Service can be performed only at the beginning (before the user starts browsing the presentation or only during the first run through the system). This is because both the user profile and the adaptation specifications are static, i.e., they are not influenced by the user interaction with the system.

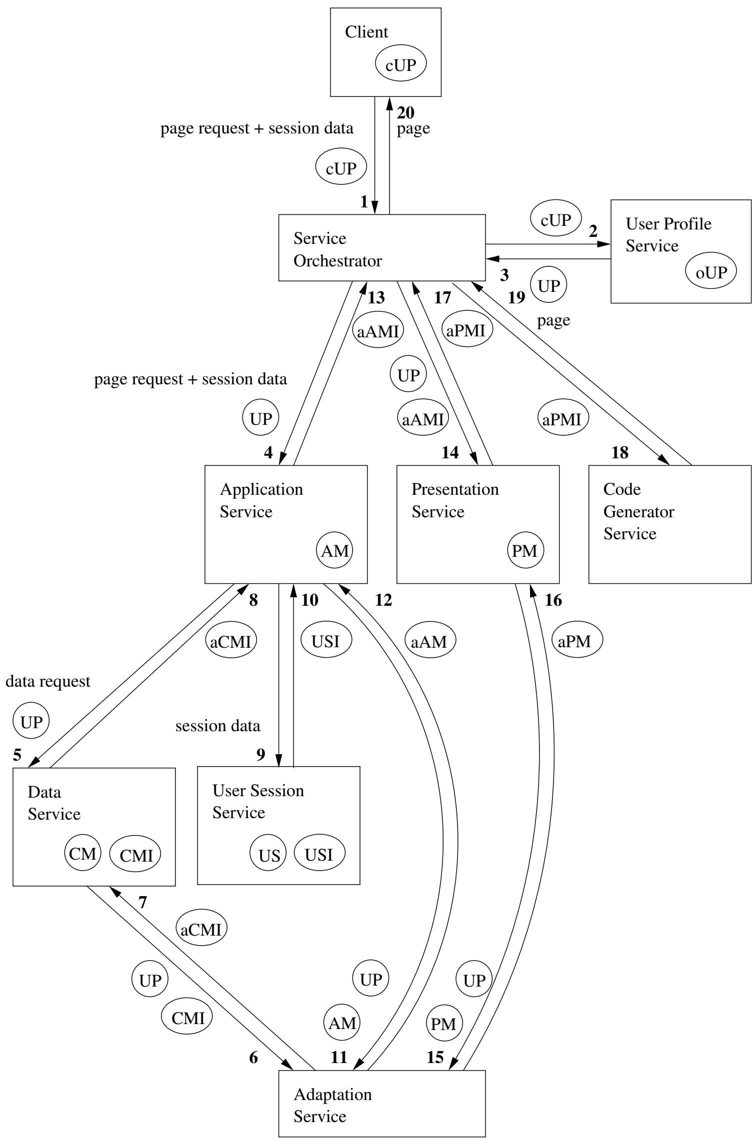


**Figure 9.** Web service-oriented architecture for dynamic applications.

### 2.3. Tools

In order to experiment with the proposed architecture a Java-based Hera tool was developed. Tomcat 4.1 [17] was used as the Web server that supports servlets. On this Web server we installed Axis 1.1 (Apache eXtensible Interaction System) [18], a SOAP 1.1 engine. By SOAP engine we mean a tool that supports both a SOAP server and SOAP clients. We did deploy on the SOAP server all WS. For their deployment we used appropriate Axis Web Service Deployment Descriptors. The SOAP Client that communicates with these services was installed on the Web server, outside the SOAP server. The WSDL specifications were generated by the Java2WSDL emitter. Tomcat and Axis are Java-based and freely available from the Apache Software Foundation. The services and the client were written in Java. All software is running on the Java 1.4 platform.

It is important to notice that when developing WS with Axis, the programmer doesn't need to bother about making WSDL interfaces or the actual encoding of the SOAP messages. All these will be automatically done by the system. Making all WS details transparent to the programmer enables him to focus only on the application logic implementation in Java and makes thus the system less error-prone.

The data transformations are implemented in Java using Jena [19]. These transformations replace the previous XSLT [20] data transformations implemented using Saxon [21], as Java transformations can exploit more of the RDFS semantics given by Hera models than the ones based on XSLT. For querying and constructing RDF(S) models we used ARQ, an implementation of SPARQL [11]. ARQ is now delivered as part of the Jena distribution kit.

## 3. Conclusion

In this paper we have have described a distributed implementation for the Hera methodology. The implementation is based on WS due to the popularity and easy-to implement features of WS. Because of the loose coupling of WS one can easily extend/replace a WS without affecting the other application services. Also these WS can be made available for other applications, for example the User Profile Service can be used by a Web querying application that wants to improve the accuracy of the returned results by making use of the user context. The Axis distribution kit proved to be a very flexible set of tools to support WS development. Users familiar with the Java programming language can seamlessly deploy Java methods as WS.

As future work we would like to extend the WSOA with new services like a data integration service responsible for integrating data from different, possibly heterogeneous data sources, or a data update service that allows for changes in the original data sources. In addition we would like to add a broker (based on UDDI) that will further decouple the communication between WS. Services will register themselves at the broker which will be responsible to forwarding requests and responses between WS. In this way the application doesn't need to know about WS location but only of the functionality these WS offer.

## References


[1] Tomas Isakowitz, Michael Bieber, and Fabio Vitali. Web information systems. *Communications of the ACM*, 41(1):78–80, July 1998.

[2] Daniel Schwabe and Gustavo Rossi. An object oriented approach to web-based application design. *Theory and Practice of Object Systems*, 4(4):207–225, 1998.

[3] Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, and Maristella Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, 2003.

[4] Nora Koch, Andreas Kraus, and Rolf Hennicker. The authoring process of the uml-based web engineering approach. In *First International Workshop on Web-Oriented Software Technology*, 2001. Available at: `http://www.dsic.upv.es/~west/iwwost01/files/contributions/NoraKoch/Uwe.pdf`.

[5] Klaus-Dieter Schewe and Bernhard Thalheim. Reasoning about web information systems using story algebras. In *Advances in Databases and Information Systems (ADBIS 2004)*, volume 3255 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2004.

[6] Flavius Frasincar, Geert-Jan Houben, and Richard Vdovjak. Specification framework for engineering adaptive web applications. In *The Eleventh International World Wide Web Conference, Web Engineering Track*, 2002. Available at: `http://www2002.org/CDROM/alternate/682/`.

[7] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tueke. The physiology of the grid: An open grid services architecture for distributed systems integration. Global Grid Forum, 2002.

[8] Stefano Nativi. Service-oriented technology to support geosciences. In *Expanding Horizons: Using Environmental Data for Education, Research, and Decision Making Workshop*, 2003. Available at: `http://my.unidata.ucar.edu/content/Presentations/2003_expanding_horizons/nativioverview.pdf`.

[9] Richard Vdovjak, Flavius Frasincar, Geert-Jan Houben, and Peter Barna. Engineering semantic web information systems in hera. *Journal of Web Engineering*, 2(1-2):3–26, 2003.

[10] Wireless Application Protocol Forum, Ltd. Wireless application group: User agent profile. 20 October 2001, 2001.

[11] Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf. W3C Working Draft 20 February 2006, 2006.

[12] Graham Klyne, Franklin Reynolds, Chris Woodrow, Ohto Hidetaka, Johan Hjelm, Mark H. Butler, and Luu Tran. Composite capability/preference profiles (cc/pp): Structure and vocabularies 1.0. W3C Recommendation 15 January 2004, 2004.

[13] Zoltan Fiala, Flavius Frasincar, Michael Hinz, Geert-Jan Houben, Peter Barna, and Klaus Meissner. Engineering the presentation layer of adaptable web information systems. In *Web Engineering - 4th International Conference (ICWE 2004)*, volume 3140 of *Lecture Notes in Computer Science*, pages 459–472. Springer, 2004.

[14] Annrai O'Toole. Web service-oriented architecture: The best solution to business integration. Cape Clear Software, 2005. Available at: `http://www.capeclear.com/technology/clearthinking/websoa.shtml`.

[15] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. W3C Note 15 March 2001.

[16] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1. W3C Note 08 May 2000.

[17] Apache Software Foundation. Apache tomcat, 2006. Available at: `http://jakarta.apache.org/tomcat/`.

[18] Apache Software Foundation. Webservices - axis, 2006. Available at: `http://ws.apache.org/axis/java/user-guide.html`.

[19] Hewlett-Packard Development Company, LP. Jena - a semantic web framework for java, 2006. Available at: `http://jena.sourceforge.net/`.

[20] Michael Kay. Xsl transformations (xslt) version 2.0. W3C Candidate Recommendation 8 June 2006, 2006.

[21] Michael Kay. Saxon (the xslt and xquery processor), 2006. Available at: `http://saxon.sourceforge.net`.