A Platform for Electronic Patient Record Integration

WP2 Deliverable D2.2

Abstract

This document comprises one of the two main deliverables from *SynEx Work Package 2 - Norway SynEx Validation*. The objective of WP2 is to design and implementation a platform for software components that can be used to make clinical information available to health professionals using the paradigm of *shared, distributed electronic patient records*, based on the principles established by CEN/ENV 12265 and Synapses, and using the object-oriented paradigm and industry standard technology. As a proof of concept, WP2 includes a demonstration of how the platform and its support for shared, federated healthcare records can be used to support the continuity of cardiovascular care for patients that are examined at one of the verification hospitals (SiA) and undergoes cardiac surgery at the other hospital (RH).

This deliverable D2.2 presents the server-side components of the platform, while the other deliverable *"D2.1 Seamless Integration of Distributed Electronic Patient Records"* presents client-side components and how shared, distributed electronic patient records are achieved.

The work of WP2 is conducted by *Siemens Health Services (SHS), Sentralsykehuset i Akershus (SiA)* and *Rikshospitalet (RH)*.

Siemens Health Services, P.O.Box 10, Veitvet,
N-0518 Oslo, Norway
Phone +47 22 63 30 00, Fax +47 22 63 48 80

Author	:	**Egil.Paulin.Andersen@nr.no** (Norwegian Computing Center, http://www.nr.no)		
Distribution	:	**All Consortium members**		
Date	:	**16th March 2000**	Version :	**1.0**
Status	:	**Final**		
Filing code	:	**SHS-024[WP2]**	Classification :	**Public**

Contents

Author	:	**Egil.Paulin.Andersen@nr.no** (Norwegian Computing Center, http://www.nr.no)		
Distribution	:	**All Consortium members**		
Date	:	**16th March 2000**	Version :	**1.0**
Status	:	**Final**		
Filing code	:	**SHS-024[WP2]**	Classification :	**Public**

1. Motivation and Background

The management of electronic patient data was relatively easy as long as the data was collected, stored and viewed in a closed environment like a hospital or doctor's practice. Heterogeneous and compatible IT systems were provided by one company and externally or internally employed, centralised system administrators were responsible for the smooth use of all installed components. No connection to the "outside world" was established which eased the protection of patient data immensely. Data exchange was only possible by paper, mail, fax or telephone, which lead to extra work and unreliability: feedback and results had to be manually entered into a new system. Increased computerisation throughout the health sector has given rise to a proliferation of independent systems storing patient data. However, the growing trend towards shared care requires that these systems are able to share their data. This has led to the development of projects such as *Synapses* [3][4] and its follow-up *SynEx* [1][2] which aims to provide healthcare professionals with integrated access to patient records and related information, regardless of where this information resides.

The goal of SynEx Work Package 2 is to support the continuity of care through *shared federated healthcare records (FHCR)*. That is, a migration from FHCRs in the Synapses perspective, where they are primarily used to integrate legacy systems, to FHCRs in the SynEx perspective where records are shared across extranet, and where parts of a record can be regarded as part of another record. More specifically, the tasks of WP2 was to design, implement and demonstrate

- *a platform for software components that make clinical information available to health professionals using the paradigm of distributed electronic patient records, based on the principles established by CEN/ENV 12265 and Synapses, and using the object-oriented paradigm and industry standard technology.*
- *a federated health care record (FHCR) supporting the continuity of cardiovascular care for patients that are examined at one of the verification hospitals (SiA) and undergoes cardiac surgery at the other hospital (RH).*

The work in WP2 has resulted in a distributed, component-based information system where users can access information in FHCR servers that provide Synapses compliant record information in XML, on an extranet based on common internet technology.

2. Scenario for Demonstrating Shared, Federated Healthcare Records

The platform and software components developed in WP2 will be used to demonstrate, in a semi real life situation, clinical collaboration based on a federated healthcare record (FHCR) shared between two closely collaborating hospitals providing shared cardiovascular care. The FHCR can be used as an alternative for sharing information by being composed from two different records in two hospitals.

The Norway clinical sites for the installation and evaluation of SynEx products are:

- *The Department of Medicine at the Central hospital of Akershus (**SiA**–Sentralsykehuset i Akershus)*
- *The Department of Cardiac Surgery at the National Hospital (**RH** - Rikshospitalet)*

Both hospitals are reference sites for SHS prospects in the Norwegian market, and these two hospitals collaborate in the treatment of patients with angina pectoris that needs surgical treatment. Patients with Angina Pectoris in the County of Akershus are examined and considered for bypass operation at SiA. If candidate for operation, the patient is transferred to RH for the actual operation.

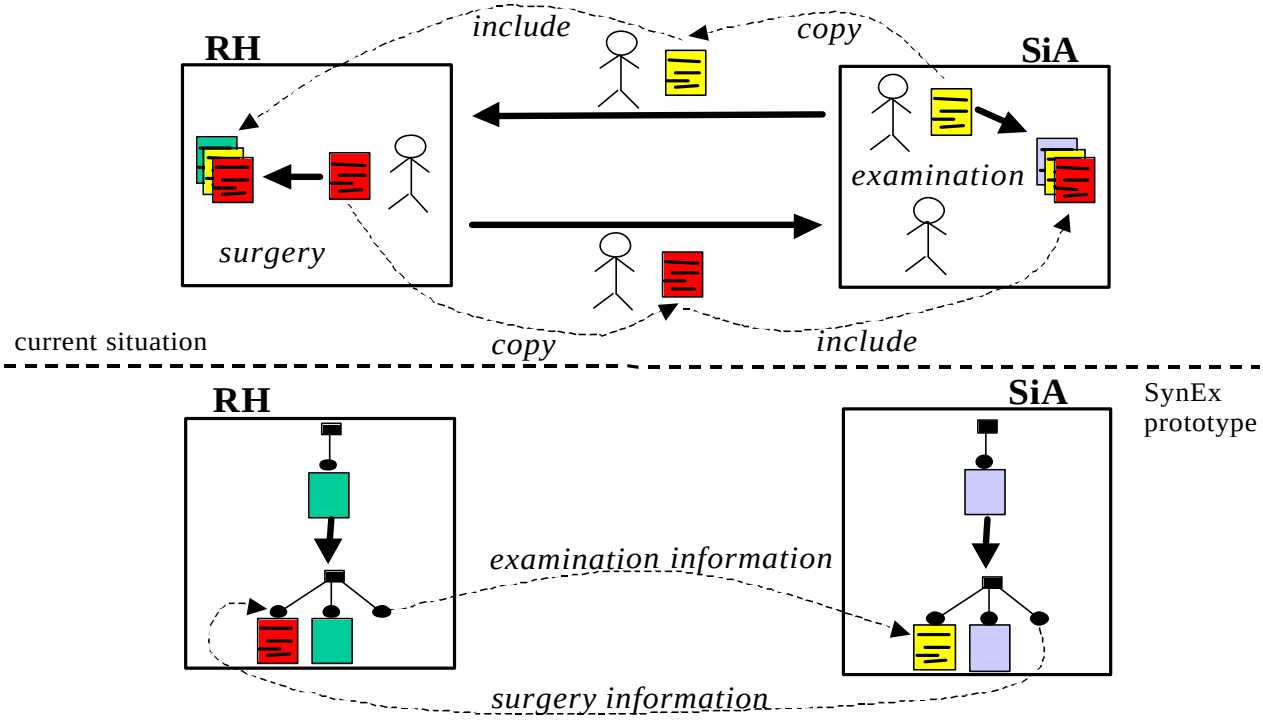

Figure 1. *Demonstration scenario.*

Current situation

Today, when a patient is admitted to SiA, a record, or a new section in an existing record, is created. Examinations are made and results recorded in the record. If surgery is required the relevant record information is copied, and a discharge letter written. This accompanies the patient to RH. At RH a new record, or a new section in an existing record, is created. The accompanying information is re-typed, and the required treatment recorded. When treatment is finalised the patient and the relevant parts of his record, together with a discharge letter, is transferred back to SiA.

Utilizing Shared Federated Healthcare Records

Figure 1 illustrates a scenario for how shared FHCR's can be used to simplify this procedure. Relevant parts of the record at SiA will be made available to the appropriate doctors at RH, while other parts are hidden. These parts will be considered part of the patients record at RH, and thus part of the federated record.

Correspondingly, the part of the RH record relevant for the continuity of care will be made available to the appropriate doctors at SiA, and become part of their record.

Validation

Validation of the demonstrator will be based on combining several real patient cases and real patient data, in an anonymous way. It will thus behave as if it were a real life case, but it will not be used on actual cases. The current legislation in Norway prevents us from transferring real patient data across extranet.

Benefits

The benefits of shared FHCR as demonstrated by the prototype are:

- it facilitates the continuity of care between organisations
- relevant information is available and thus reduces the risk of making wrong decisions due to lack of information
- it reduces administrative work
- the basis for decision making is more explicit and available
- it facilitates quality monitoring/control, research, etc.

3. The Platform Architecture

3.1 Exchanging Semantically Meaningful Information

XML for Exchanging Healthcare Record Information

XML [5] has the power to become the independent data exchange format of the future. The use of XML to exchange data between heterogenous systems provides support for hierarchically structured patient data, user defined tags and machine-understandable assertions for searching, reasoning and analysing healthcare information like federated healthcare record objects.

An XML DTD, called the *SynExML (SynEx Markup Language)*, has been defined within the SynEx project to be used for inter-site exchange of FHCR information. That is, SynExML is the basis for semantical interoperability between SynEx components that relate to FHCR information. The 2.1 beta 4 version of SynExML, which is likely to become the final version for the duration of the SynEx project, is included in appendix B below.

SynExML is based upon the generic FHCR structure defined within the Synapses project, and most of its elements and attributes correspond one-to-one with record component concepts and properties defined within the Synapses Server specification. The following section outlines the key principles of the Synapses Record Architecture.

Federated Healthcare Records according to Synapses

The Synapses Record Architecture consists of a single class hierarchy, and every Synapses patient record consists of a set of objects where each object is instantiated from a class in this hierarchy. Each class in the hierarchy belongs to one out of two main groups of classes. The structure of a record is made out of objects instantiated from the "structural classes", called *Record Item Complexes (RICs)*, while the data (information) within a record consists of objects instantiated from "data value classes", called *Record Items*.

The structure of a Synapses record corresponds to a tree structure of RIC objects, and each record can have unidirectional links to other such tree structures, i.e., to other records. The tree structure of each record is rooted in a single particular *RecordFolder* object, i.e., instantiated from class *RecordFolder*, which represents the overall record. Below this object there will be a structure of folders (*FolderRIC* objects) and documents (*ComRIC* objects). Each document will itself consist of a tree structure of objects, which can be *DataRIC* objects and/or *ViewRIC1* objects. The former contains information that is explicitly recorded in the record, while the latter are used to represent computed or derived information.

In addition there are objects that represent links to other records, called *ViewRIC2* objects. These are the key to the integration of Synapses records. They contain the unique identification of another RIC object, and they are used as follows. The root object of a record, or a folder within a record, may contain a single ViewRIC2 object that references another record or folder object, respectively. In addition, a document may contain one or more ViewRIC2 objects that each reference some subset of other documents. The RIC object referenced by a ViewRIC2 object is either local, intra- or inter-record, or remote, and if remote then the ViewRIC2 object contains an URL that identifies the server where the target RIC object resides.

Each RIC object instantiated from one of the "structural classes" will have a small set of static, predefined attributes, e.g. as required for their unique identification, or the target address of a ViewRIC2 object. However, most of the information content of a record, and all the medical information, exists in *RecordItem* objects instantiated from the "data value classes". That is, a set of RecordItem objects can be attached to a structural RIC object and thus function as its *dynamic attributes* with actual data values like e.g. a blood pressure measurement. The RecordItem objects that belong to a particular RIC object can also themselves be organized into a tree structure. Due to these RecordItem objects, the information content of a record can by dynamically extended over time, and with information of a kind that may not have been foreseen at the time the record itself was created.

The distinction between RIC classes versus RecordItem classes comprises a kind of "vertical" grouping of the overall Synapses class hierarchy. In addition the class hierarchy is split "horizontally" into a predefined set of base classes common to every Synapses server, called the *Synapses Object Model (SynOM)*, and an *extendable* set of classes that are derived from these SynOM classes, called the *Synapses Object Dictionary (SynOD)*. The above RIC classes RecordFolder, FolderRIC, ComRIC, etc, all belong to the SynOM, and they

define the core part of Synapses' generic record model. The SynOD classes on the other hand, which are site specific and thus may differ for each Synapses server, are the classes from which the actual patient record objects are instantiated. Thus while every record object has the above SynOM characteristics and properties, they can also be customised to the needs of each individual site.

3.2 Technical Architecture

Figure 2 illustrates the layered architecture that has been designed and implemented in WP2. There is a client presentation and interaction layer, a web server, an application layer, and a data layer. The web server is *IIS (Internet Information Server)* [11] with *ASP (Active Server Pages)* objects and scripts as its interface. The application layer consists of *COM (Component Object Model)* [9] components under the control of *MTS (Microsoft Transaction Server)* as the transaction server, and the data layer is an *SQL Server* database [8] containing healthcare record information. At the Oslo site the latter is the *Oslo Synapses Server (OSS)*.

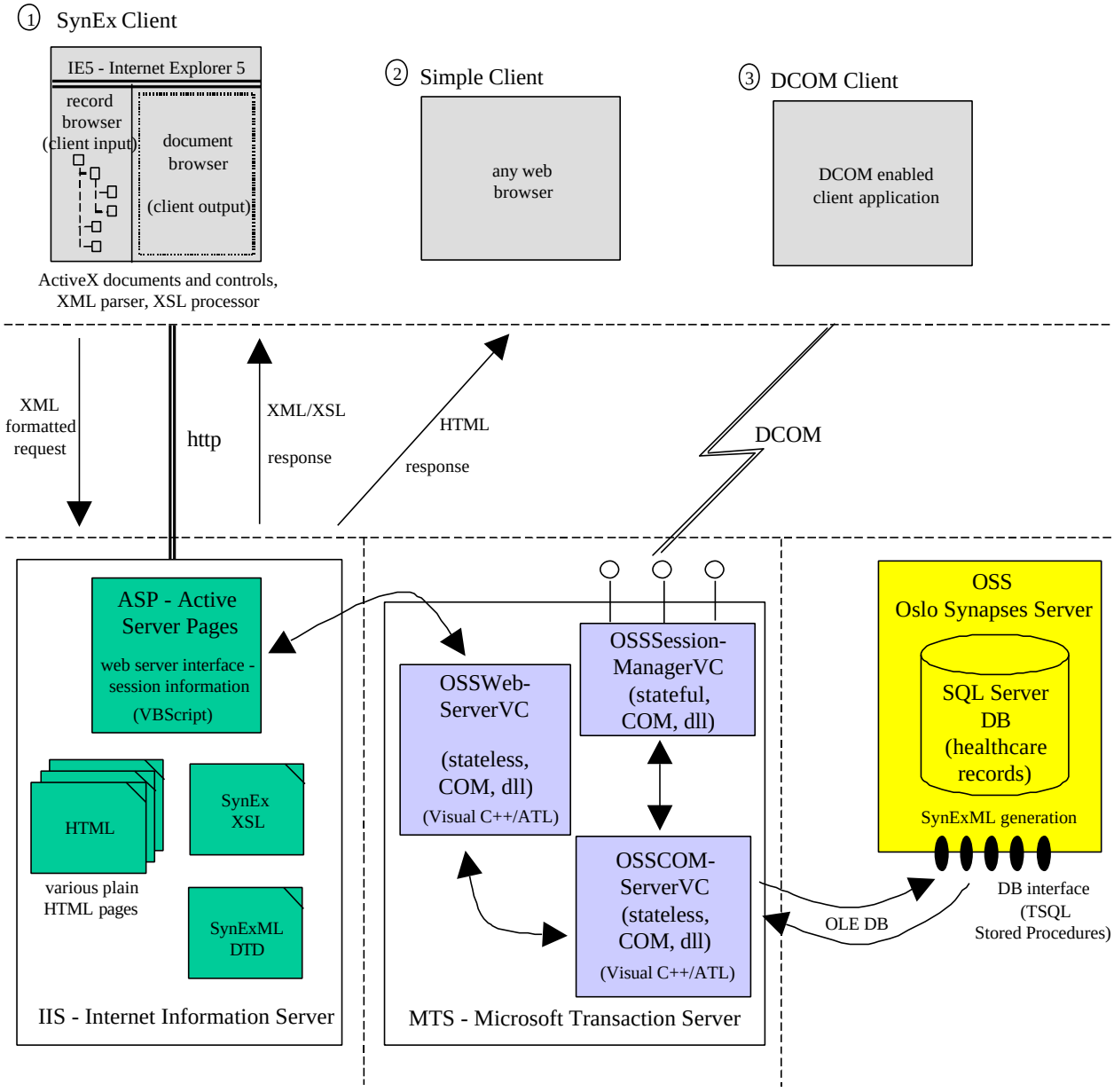

Figure 2. *The technical architecture for the WP2 platform (also known as "one happy Microsoft family")*

Client-side Components

There is a wide variety of different client types possible for a distributed information system. At one end there are highly specialized and domain dependent "thick" clients, including e.g. extensive caching, and flexible options for checking in and out information and then work on this information offline before updating the server with changes made. At the other end there are "thin" clients with common web browsers, or also ultra-light mobile devices with WAP/WML.

For reasons described below (XML+http=SOAP), clients can be made independent of the server-side technology. However, for clients based on Microsoft technology the following are some relevant examples, as illustrated in figure 2:

- *"Simple client"* - Any web browser will be able to log on to the server and then select, retrieve and update information.
- *"SynEx client"* - The client consists of a set of *ActiveX components* (documents and controls) that execute within e.g. *Internet Explorer* as their container.
- *"DCOM client"* - The client accesses the server via *DCOM (Distributed COM)*.

The "simple client" corresponds to what can be seen as a "true" web-client, while the "SynEx client" is more like a traditional client application using web protocols for server interaction. The "simple client" may be WAP based mobile devices with WML browsers. For reasons described below DCOM is not a particularly interesting option unless there is a very close relationship between the client and the server with respect to whom has developed them, and which servers the clients will connect to. The "SynEx client" alternative has been prioritised in WP2, and the other WP2 deliverable D2.1 describes this client in further detail.

For clients of the "SynEx client" type there are at least two different ways to combine the use of ActiveX components and e.g. Internet Explorer. One alternative is to implement a number of *ActiveX controls* that are included into an ordinary HTML page, e.g. with each control in a different frame. Another alternative is to use an *ActiveX document* with Internet Explorer as its container. In the latter case the ActiveX document can utilize features made available via Internet Explorer, e.g. its menues, its history lists, and more. For ActiveX documents that exist on the user's computer locally, the user can navigate to these within Internet Explorer the same way he can navigate to other web pages.

Network Protocol

To make record information available on an extranet using common internet technology, the internet protocol *http (HyperText Transfer Protocol)* is used for client-server interaction both ways.

XML for Request/Response Interaction and Information Transmission

Microsoft currently works on a specification called *SOAP (Simple Object Access Protocol)* [6][7] where the communication between a client and a server is formatted as XML over http both ways; i.e., as opposed to e.g. *DCOM (Distributed COM)* or *IIOP (Internet Inter-ORB Protocol* for *CORBA* [14]*)* also requests from a client to a server is formatted as XML (as functions with arguments according to a particular SOAP XML format [2]) which can then be parsed by the server and acted upon. There are several advantages by this despite its functional simplicity relative to DCOM and IIOP.

http is a simple protocol with good coverage and few demands on the client, and, not the least, most firewalls are readily configured for common security options dealing with well known internet protocols and ports. This as opposed to DCOM or IIOP for which firewalls can pose a problem. In practice, the ability for remote machines to interact via DCOM and IIOP is more limited. That is, DCOM and IIOP can be well-suited for computers within e.g. a limited area, but not between "any" remote client and server on the internet.

Since XML amounts to strings of text it is well-suited for transmission via http, and the great benefit of SOAP's combined use of XML and http is that it makes the underlying client- and server-side technology transparent to each other. Thus similar to how component technology like COM provides for programming language independence and technical interoperability locally, SOAP provides for platform independence and technical interoperability globally.

Furthermore, WML which is used for accessing information from e.g. WAP based mobile phones is itself XML, i.e., it is XML according to a particular XML schema definition. Thus information transmission with XML is well-suited for such mobile clients.

Since client requests received by a server is formatted as XML, instead of being received as e.g. low-level RPC's, a lack of complete client-server version compatibility can be handled more gracefully. For example, if an "old" client uses a "newer" server then the server may support a slightly different or extended set of requests than those requested by the "old" client. However, to the degree that the server is able to understand the "old" client's request, despite that it may not fully correspond to a request as expected by the "newer" server, the server may still be able to serve the "old" client. This is not possible with e.g. COM requests where e.g. a missing function argument leads to an immediate error.

Finally, several software development tools and techniques already has good support for XML; e.g.:

- Datastructures in *ADO (Active Data Objects* - a part of Microsoft's *Universal Data Access* (see below)*)* on the server-side can (from v.2.5) be automatically converted into XML, transferred to a client over http, and then used to automatically reconstruct an equivalent ADO datastructure on the client-side.
- *Microsoft Repository* supports export and import of XML for exchanging meta-information on UML models, relational database schemas, COM components, and more (see below).

Source Code

The source code for the current implementation of the WP2 platform comes as a zip file named:

*oss-server-wp2-v**n.m**.zip*

where the bold ***n.m*** states the version. The content of the file catalogs within this file will be further explained below.

4. Web Server Access

4.1 Client Requests

XML DTD for Client Requests

The following XML DTD defines the client request format that is expected by the WP2 platform:

```
<!-- XML DTD for requests accepted by the Oslo Synapses Server -->
<!ELEMENT OSSrequest ( Function+ ) >

<!ELEMENT Function ( Arg* ) >
<!ATTLIST Function  Name  CDATA  #REQUIRED>

<!ELEMENT Arg (#PCDATA)>
<!ATTLIST Arg  Name  CDATA  #REQUIRED>
```

Notice: The current implementation of the WP2 platform complies with SOAP technically by using XML over http, but it does **not** comply with the SOAP protocol with respect to the SOAP XML DTD for client requests.

No validation will be performed against this DTD when parsing it on the server side (if invalid then a more informative error message will be returned to the client), but the DTD will be available on WP2 web-servers in a file named *"oss-client-request.dtd"*.

The following is a list of functions currently supported. Notice that arguments in bold (**Arg**) indicates mandatory arguments, and in those cases where there is a predefined set of alternative argument values then the value in bold indicates the default value if this argument is not provided.

```
<Function Name="LogOn">
        <Arg Name="User">....</Arg>
        <Arg Name="Password">....</Arg>
        <Arg Name="ResponseType">..{html | xml | wml}..</Arg>
</Function>

<Function Name="LogOff">
        <Arg Name="User">....</Arg>
</Function>

<Function Name="RecordInfo">
        <Arg Name="User">....</Arg>
        <Arg Name="RecordID">....</Arg>
        <Arg Name="Retrieval">..{shape | all}..</Arg>
        <Arg Name="ResponseType">..{html | xml | wml}..</Arg>
</Function>

<Function Name="FolderInfo">
        <Arg Name="User">....</Arg>
        <Arg Name="RecordID">....</Arg>
        <Arg Name="RCID">....</Arg>
        <Arg Name="Retrieval">..{shape | all}..</Arg>
        <Arg Name="ResponseType">..{html | xml | wml}..</Arg>
</Function>

<Function Name="DocumentInfo">
        <Arg Name="User">....</Arg>
        <Arg Name="RecordID">....</Arg>
        <Arg Name="RCID">....</Arg>
        <Arg Name="Retrieval">..{shape | all}..</Arg>
        <Arg Name="ResponseType">..{html | xml | wml}..</Arg>
</Function>
```

The "ResponseType" argument is explained in section 4.2 below (the two defaults are due to this being client browser dependent).

The "Retrieval" argument can be either *"shape"* or *"all"*. "Shape" means that only the structure of a record, folder or document will be returned, not information within a document (no RIC's within a ComRIC except the ComRIC itself, in Synapses terms).

The reason why the user name must be provided with each request is that a particular *client* accessing the information can be logged on to the same server as several *users* during the same *session*. In a later version a default (the most recent logon) will be provided for this argument since in most cases a client will be logged on as a single user for the duration of a session.

According to the DTD, several functions can be combined into a single request, e.g. the following request:

```
<OSSrequest>
      <Function Name="LogOn">
            <Arg Name="User">onordmann</Arg>
            <Arg Name="Password">xyz</Arg>
            <Arg Name="ResponseType">xml</Arg>
      </Function>
      <Function Name="RecordInfo">
            <Arg Name="User">onordmann</Arg>
            <Arg Name="RecordID">93003449</Arg>
            <Arg Name="Retrieval">all</Arg>
      </Function>
</OSSrequest>
```

but notice that (in the current implementation) results will only be provided to the client for the last of the functions in the request. Thus there is no use in combining several record/folder/document requests.

Sending Client Requests

The current implementation of the WP2 platform supports three different alternatives for sending a request from a client to the web server. Two of them for "production" use, and one only for simple "demonstration" purposes.

1. *QueryString - http GET command*

   The http GET command, as a socalled QueryString, means that the XML formatted request from the client is added to the web address as follows:

   ```
   http://citroen.nr.no/synexdemo/oss.asp?<OSSrequest><Function Name="LogOn">
                        <Arg Name="User">admin</Arg><Arg Name="Password">x</Arg>
                        <Arg Name="ResponseType">xml</Arg></Function></OSSrequest>
   ```

   This can be useful for demonstration purposes to make things explicit, but there are also several disadvantages; e.g. there is a limit to the length of GET commands so parts of it may be truncated, and the requests will be visible to "anyone" (e.g. in logs).

2. *HTML Forms - http POST command*

   Using an HTML Form to send a POST command is a better solution than the above GET command. There are no (at least practically important) limitations to the length of the XML request within a POST command. The following is an example of making such a request:

   ```
   <FORM METHOD="POST" ACTION="http://citroen.nr.no/synexdemo/oss2.asp">
         <INPUT TYPE="hidden" NAME="XMLRequest"
                              VALUE='<OSSrequest><Function Name="LogOn">
                                    <Arg Name="User">emil</Arg>
                                    <Arg Name="Password">x</Arg>
                                    <Arg Name="ResponseType">xml</Arg>
                                    </Function></OSSrequest>'/>
         <INPUT TYPE="submit" VALUE="Log On"/>
   </FORM>
   ```

Notice: The name of the input/form field with the XML formatted request must be "XMLRequest" to be accepted/found by the current WP2 implementation.

3. XMLHttpRequest ActiveX control - http POST command

A http POST command can alternatively be sent by using an ActiveX control like *XMLHttpRequest*, which is available within the Micrsoft XML v.2.0 parser *"msxml.dll"*. The following is an example of how to use this control from Visual Basic:

```
Dim refHttp          As MSXML.XMLHTTPRequest
Dim refXMLDoc        As MSXML.DOMDocument
Dim strServerAddress As String
Dim strXMLrequest    As String

strServerAddress = "http://citroen.nr.no/synexdemo/oss.asp"
strXMLrequest = "<OSSrequest><Function Name=" & Chr(34) & "LogOn" & Chr(34) & _
                ><Arg Name=" & Chr(34) & "User" & Chr(34) & _
                >admin</Arg><Arg Name=" & Chr(34) & "Password" & Chr(34) & _
                >x</Arg><Arg Name=" & Chr(34) & "ResponseType" & Chr(34) & _
                >xml</Arg></Function></OSSrequest>"

Set refHttp = New MSXML.XMLHTTPRequest

Call refHttp.Open("POST", strServerAddress, False)

' NB! To distinguish this POST command from the Forms POST command
Call refHttp.SetRequestHeader("XMLRequest", "XMLHttpRequest")

Set refXMLDoc = New MSXML.DOMDocument
refXMLDoc.Async = False
refXMLDoc.ValidateOnParse = False
If (Not refXMLDoc.LoadXML(strXMLrequest)) Then
    ...error in XML request...
End If
Call refHttp.Send(refXMLDoc)

...result available in refHttp.responseXML
```

Notice: To be able to distinguish this POST command from the other Forms POST command, a request header variable named *"XMLRequest"* is defined with the value *"XMLHttpRequest"*. Without this variable defined and set, a WP2 server will not be able to get the XML request sent.

The SynEx client uses the *XMLHttpRequest* component in its implementation, and the other deliverable D2.1 describes this in further detail.

4.2 Server Response

SynExML

As explained in section 3.1, the server response to a valid request for record, folder or document information will be XML valid according to the *SynExML*, which again is based on the generic FHCR structure defined by the Synapses Server specification [3][4]. That is, such XML, and including a reference to a default XSL specification, will be returned to the client provided that *either* the client makes an explicit request for *"xml"* via the "ResponseType" argument, *or* no "ResponseType" argument is provided *and* the client browser is Internet Explorer v.5.0.

If the client specifies *"html"* for the "ResponseType" argument, *or* no "ResponseType" argument is provided *and* the client uses any other browser than Internet Explorer v.5.0, then the XML generated for the information requested will be transformed into HTML on the server-side via the use of an XSL specification.

Of course, requesting HTML instead of XML is only relevant in those cases where the client only wants to browse the information received, and the browser is unable to transform XML into e.g. HTML via XSL; either the default XSL provided from the server, or some other XSL that the client has access to.

WML formatted information is not available in the current version.

Other Server Responses

The server does not respond with XML valid according to SynExML for LogOn and LogOff requests, nor if an error occurs when processing a request (e.g. trying to retrieve information without being logged on, or information for which the user lacks authorization).

The following DTD defines alternative server responses for a successful LogOn, a successful LogOff, or an error situation:

```
<!-- XML DTD for non-SynExML responses returned by the Oslo Synapses Server -->

<!ELEMENT OSSresponse ( Success | Failure )* >

<!ELEMENT Success ( Function )>
<!ELEMENT Function (#PCDATA)>

<!ELEMENT Failure ( Error* ) >
<!ELEMENT Error ( Source, Number, Description )>
<!ELEMENT Source (#PCDATA)>
<!ELEMENT Number (#PCDATA)>
<!ELEMENT Description (#PCDATA)>
```

This DTD will be available on WP2 web-servers in a file named *"oss-server-response.dtd"*.

5. Web Server Implementation

Source Code

The source code for the web server implementation is indeed very "thin", namely a small *VBScript* in the following file, which is also the entry point when accessing the server:

../ServerSource/ServerWeb/oss.asp

The Web Server Interface is an ASP Script

ASP scripts constitute the interface of the IIS web server. In the current version the web server interface is a single ASP script; i.e. *"oss.asp"*. This script receives an XML request from a client, and it immediately, without any processing, hands the request over to a COM object instantiated from the *OSSWebServerVC* component described below.

In this case, and except for some error handling statements, the ASP script consists of just 2(!) lines of code, namely:

```
Set objServer = Server.CreateObject("OSSWEBSERVERVC.COSSWebServer.1")
objServer.HandleASPClientRequest()
```

i.e., one line to create a COM object from the *COSSWebServer* class within the *OSSWebServerVC* component, and one line to invoke its *HandleASPClientRequest()* function which then starts processing the client request, and which will eventually be responsible for returning results back to the client.

An ASP script in e.g. *VBScript* or *JScript* can access a database itself by using *ADO (Active Data Objects)*, but it is much better to let COM objects under the control of MTS handle database access. Performance is one reason, but also, script languages are generally unsuitable as application programming languages relative to e.g. *Visual C++* or *Visual Basic* that are supported by *Visual Studio* as a professional *IDE (Integrated Development Environment)*. Script languages are better used as thin "glue" between e.g., as in this case, the web server and COM components under MTS.

6. Application Layer Implementation

6.1 MTS - Microsoft Transaction Server

Transaction servers are important for scalability with respect to the number of concurrent users, and thus performance, and also for managing distributed transactions and resources like database connections. *MTS* is the transaction server from Microsoft, and it supports:

- Distributed transactions via *DTC (Distributed Transaction Coordinator)*
- Database connection pooling
- Object pooling (only available from MTS v.3.0)

This support for distributed transactions means that COM objects residing in the MTS of different computers can participate in the same atomic transaction. Similarly, if an MTS COM object works on several databases on different computers then these database operations can be combined into a single transaction.

Database connection pooling means that instead of assigning a dedicated database connection to each client, a pool of equal database connections are reused as and when required to serve client requests. The benefit is that database connections, which are a limited resource, can be utilized more efficiently such that the number of concurrent users can exceed the number of concurrent database connections, and the performance overhead of creating database connections can be reduced.

Object pooling means that instead of creating an object instantiated from a particular component from scratch each time it is needed, which can be time-consuming, objects that are currently "free", i.e., not participating in a particular transaction initiated by a client, can be reused *as if* they were newly created objects.

COM objects that should be used in object pooling must be made "stateless"; i.e., they may well have state when they participate in transactions, but after a transaction is ended, and the object returns to its pool of now ready objects, then no one else should depend on its state since as soon as it is assigned to a new transaction, its previous state will be overwritten as if it were a newly created object.

COM components at the MTS application layer should preferably be implemented in Visual C++ since from v.3.0 (in COM+) then MTS will support object pooling, but *only* for components implemented in Visual C++. *COM+* is the latest version of COM, just recently released, and here MTS is made an inherent part of COM.

ATL (Active Template Library) is a useful Visual C++ library that makes it easier to implement COM components.

6.2 OSSWebServerVC Component

Source Code

The OSSWebServerVC component is implemented as a dll in *Visual C++* with *ATL v.3.0 (Active Template Library)*. It executes under the control of MTS, it is *"stateless"* in the MTS sense, and it is registered as *not supporting transactions*. The latter because no transaction will span the invocation of several of its functions (i.e., functions supported via XML requests from a client), nor will it be created as part of a transaction.

The source code for the OSSWebServerVC component exists in the following Visual C++ project:

../ServerSource/OSSWebServerVC/OSSWebServerVC.dsw

Interface and Functionality

The OSSWebServerVC component is implemented by the class *COSSWebServer* and, as seen from the outside, it has a single COM interface with a single function *HandleASPClientRequest()* that takes no arguments. Its IDL specification is listed in appendix A. In addition the file *"/ServerSource/ServerUtility/-wstringutility.h"* contains some general "wide character string" utility functions used by all the COM components in the platform.

The ASP script "oss.asp", which first receives the client request, creates an OSSWebServerVC object and invokes this function (as shown in section 5 above). Notice that due to the "stateless" MTS programming model, a "new" OSSWebServerVC object is created for each client request ("new" because it may be retrieved

from a pool of previously instantiated objects that are now again ready to do some work - that is, such *object pooling* will be available from MTS v.3.0).

The OSSWebServerVC object uses the ASP *Request object*, further described below, to retrieve the XML string that constitutes the client request, as described in section 4.1 above. The *ParseClientXMLRequest()* function is used to parse this XML request, and currently the following 5 services are supported and implemented as follows in functions with the following names:

LogOnRequest():

- If the user is already logged on, return with a successful log on indication.
- Create a new OSSCOMServerVC object.
- Call the *LogOn()* function of this object.
- If the log on is successful, store the UserID received from the OSSCOMServerVC object in the ASP *Session* object (further described below).
- Store also the ResponseType, as either explicitly specified in the request or implicitly decided by the OSSCOMServerVC object based on the client's browser type, in the ASP *Session* object.
- Return an indication of successful log on; either as XML/XSL or HTML depending on the ResponseType.

LogOffRequest():

- Remove the specified user, and its UserID, from the *Session* object. If the current client has no more log ons then the Session object is terminated.
- Return an indication of successful log off; either as XML/XSL or HTML depending on the ResponseType.

RecordInfoRequest():

- Get the UserID for the user making the request from the ASP *Session* object, or return an error no such UserID exists; i.e., if the user has not (been able to) log on successfully.
- If the record request contains no preferred ResponseType, use the ResponseType from the initial log on request, which is also stored in the Session object.
- Create a new OSSCOMServerVC object.
- Call the *GetRecordInfo()* function of this object.
- Return the record information as an XML or HTML string received from the *GetRecordInfo()* function.

FolderInfoRequest():

Similar to *GetRecordInfo()* except invoking the *GetFolderInfo()* function of the OSSCOMServerVC object.

DocumentInfoRequest():

Similar to *GetRecordInfo()* except invoking the *GetDocumentInfo()* function of the OSSCOMServerVC object.

Finally, the OSSWebServerVC object returns the requested record, folder or document information, as XML with a default XSL, or an indication of a successful or failed log on or log off request, together with proper http headers, to the client via the ASP *Response object* (see below).

IIS/ASP COM Objects - Request, Response, Session, and more

IIS/ASP offers several COM objects that can be used by COM components under MTS, like e.g. OSSWebServerVC objects, to handle and service client requests received by an ASP script like "oss.asp".

The Request object is an object that contains information on the client request, e.g. whether it is an http GET or POST command issued, what is the XML content of the request, what kind of browser is the client

using, and so on. After the "oss.asp" script has created the OSSWebServerVC object, and invoked its function *HandleASPClientRequest()*, the OSSWebServerVC object will find all the information it needs on the client request in a corresponding Request object created due to the client request.

Similarly, accompanying a client request there is also a Response object that e.g. an OSSWebServerVC object can use to return information to the client that made the original request. The Response object has amongst others a *Write()* function that can be used to send text like XML or HTML to the client.

Furthermore, IIS/ASP offers a *Session* object that is useful to record the state of a user session over several http requests. A particular Session object is unique to a particular client, and it remains in existence as long as the client navigates within the same *IIS "web application"* (corresponding to a particular virtual catalog in IIS). A Session object is also able to store arbitrary (name,value) pairs, and can thus be used to record stateful information between client requests; e.g. log on information.

Notice that a *Session* object will terminate due to a time-out after a prespecified interval if no further requests are made (here set to 20 minutes), or if the client navigates to some other URL outside the virtual catalog in IIS.

There are also other IIS/ASP objects; e.g. an *Application* object which is shared by every client that accesses the same IIS "web application".

Authentication

The authentication mechanism used for this platform is as follows. Each read or write operation on the Oslo Synapses Server database, via a stored procedure, is accompanied by a UserID that identifies the user. These UserID's are not known to clients, however. They are only used internally on the server-side. The database contains a relationship between a UserID and a (user name, password) pair. When a client logs on he provides a user name and a password. If there exists a match between these and a UserID in the database, the UserID is forwarded to the OSSWebServerVC object, via the OSSCOMServerVC object, and stored in the Session object. The UserID is never sent to the client. Next time the client makes a request, the UserID will be retrieved from his Session object, and then used as an argument when executing stored procedures via an OSSCOMServerVC object.

A prerequisite for this scheme is of course that the relationship between a client and his Session object is secure. This relationships is handled by IIS/ASP, and the WP2 implementation relies on the default IIS/ASP security provided for this.

XSL Specifications and Client side Presentation

Different *XSL (eXtensible Stylesheet Language)* specifications can be used to transform the same XML into different presentation formats like e.g. different HTML. As explained in section 4.2 above, depending on which "ResponseType" argument the user provides, either "xml" or "html", the transformation of SynExML into HTML will either take place on the client side, or on the server-side, respectively.

In the current version of the implementation only three XSL specifications are provided, namely

../ServerSource/ServerWeb/oss-response.xsl

../ServerSource/ServerWeb/oss-record.xsl

../ServerSource/ServerWeb/oss-document.xsl

"oss-response.xsl" is used by default for server responses other than SynExML, *"oss-record.xsl"* is used for SynExML returned due to a request for a record or a folder, and *"oss-document.xsl"* is used for SynExML returned due to a request for a document.

XSL is very flexible for presentations, but it is also quite time-consuming to create XSL specifications. The quality of these XSL specifications accompanying the current version are very crude, but work is underway to produce more document specific XSL specifications.

6.3 OSSCOMServerVC Component

Source Code

The OSSCOMServerVC component is implemented as a dll in *Visual C++* with *ATL v.3.0*. It executes under the control of MTS, it is *"stateless"* in the MTS sense, and it is registered as *supporting transactions*. The latter is not strictly necessary in the current version, but in later versions, with a much more elaborate application layer, then transactions may in general span the invocation of several of its functions, and its objects may be created as part of transactions.

The source code for the OSSCOMServerVC component exists in the following Visual C++ project:

../ServerSource/OSSCOMServerVC/OSSCOMServerVC.dsw

Interface and Functionality

The WP2 application layer is very "thin" and simple, containing only the OSSCOMServerVC component which is responsible for accessing the healthcare record database. As the functionality of the platform is extended there may be a lot of processing going on in the application layer.

The IDL specification of OSSCOMServerVC is listed in appendix A. The component is responsible for 4 different services; log on, and retrieving a record, a folder or a document. Beside this component, implemented by the class *COSSCOMServer*, the application layer implementation consists of 5 other ordinary Visual C++ classes that are used to handle the OLE DB [10] database access. In addition the file *"/ServerSource/ServerUtility/wstringutility.h"* contains some general "wide character string" utility functions used by all the COM components in the platform.

Class *CSPConnector* offers functions to connect to a database (either via an *OLE DB provider for SQL Server*, which is strongly recommended for performance reasons, or an *OLE DB provider for ODBC*, which is used in the current implementation due to some technical problems with the current demonstration DBMS - do not use this if possible!).

Class *CSP_GetUserAccessor* is used to invoke the stored procedure *_GetSynExUser*, which is used during log on (see below).

The classes *CSP_GetSynXMLDocumentInfoAccessor*, *CSP_GetSynXMLFolderInfoAccessor* and *CSP_GetSynXMLRecordInfoAccessor* are used to invoke the stored procedures *_GetSynXMLDocumentInfo*, *_GetSynXMLFolderInfo* and *_GetSynXMLRecordInfo*, respectively, which are used to service requests for a document, a folder or a record, respectively.

Notice that, except for the general *CSPConnector* class, the other class all correspond to the invocation of a particular stored procedure. It is possible to create a more generic class to handle several stored procedure invocations, but this is not necessarily a benefit, e.g. concerning maintenance.

The following is a brief description of how each of the OSSCOMServerVC services are implemented.

LogOn():

- Connect to the database with *CSPConnector.ODBCConnect()* or (better) *CSPConnector.OLEDBConnect()*.
- Set parameter values for invocation of the stored procedure *_GetSynExUser* with *CSP_GetUserAccessor.SetParameterValues()*.
- Execute the stored procedure with *CSP_GetUserAccessor.ExecuteSP()*.
- Get the UserID for the user logging on with *CSP_GetUserAccessor.GetOutputValues()*, and test if this is a valid user with a valid password.
- Close the stored procedure accessor, and close the database connection (but this in a manner that assures database connection pooling).
- Return the UserID and a boolean indicating success or failure for the logon request.

GetRecordInfo():

- Connect to the database with *CSPConnector.ODBCConnect()* or (better) *CSPConnector.OLEDBConnect()*.

- Set parameter values for invocation of the stored procedure *_GetSynXMLRecordInfo* with *CSP_GetSynXMLRecordInfoAccessor.SetParameterValues()*.
- Execute the stored procedure with *CSP_GetSynXMLRecordInfoAccessor.ExecuteSP()*.
- Get the XML string for the record with *CSP_GetSynXMLRecordInfoAccessor.GetXMLString()*.
- Close the stored procedure accessor, and close the database connection (but this in a manner that assures database connection pooling).
- Wrap necessary XML around the record XML string.
- Transform the record XML into HTML, according to a default XSL, if HTML is requested.
- Return the final record XML or HTML string.

GetFolderInfo():

Similar to *GetRecordInfo()* except invoking the *_GetSynXMLFolderInfo* stored procedure.

GetDocumentInfo:

Similar to *GetRecordInfo()* except invoking the *_GetSynXMLDocumentInfo* stored procedure.

6.4 OSSSessionManagerVC Component

Source Code

The OSSSessionManagerVC component is implemented as a dll in *Visual C++* with *ATL v.3.0*. It executes under the control of MTS, it is ***not*** *"stateless"* in the MTS sense (see below), and it is registered as *not supporting transactions*. The latter because no transaction will span the invocation of several of its functions, nor will it be created as part of a transaction.

The source code for the OSSSessionManagerVC component exists in the following Visual C++ project:

../ServerSource/OSSSessionManagerVC/OSSSessionManagerVC.dsw

Interface and Functionality

The OSSSessionManagerVC component, i.e., objects instantiated from this component, are used to manage user sessions. An object is instantiated from this component when a user first logs on, and the object exists for the duration of the session, i.e., until the user logs off. All services requested from this object are forwarded to an OSSCOMServerVC object, except log on and log off. Each service request is also "intercepted" in the sense that the UserID for a particular user is stored in the OSSSessionManagerVC object for the duration of the session, and this UserID is handed over to the OSSCOMServerVC object instead of a user name and a password when making requests after a successful log on.

Notice that a particular client can be logged on as several different users during the same session. For each log on received from a particular client, the corresponding UserID is recorded in the session manager object. Thus a session manager object will contain a set of UserID's, and the Visual C++ classes *CUserCollection* and *CUserInfo* are used to manage this set of UserID's.

The IDL specification of OSSSessionManagerVC is listed in appendix A. The file *"/ServerSource/Server-Utility/wstringutility.h"* contains some general "wide character string" utility functions used by all the COM components in the platform.

The following is a brief description of how each of the OSSSessionManagerVC services are implemented.

LogOn():

- Create a new OSSCOMServerVC object.
- Call the LogOn() function of this object.

- If the log on is successful then add the returned UserID for this user to the *CUserInfo* object collection managed by a *CUserCollection* object local to the OSSSessionManagerVC object.

LogOff():

- Remove the specified user, and its UserID, from the *CUserInfo* object collection managed by a *CUserCollection* object.

GetRecordInfo():

- Get the UserID for the user making the request from the *CUserInfo* object collection, or return an error no such UserID exists; i.e., if the user has not (been able to) log on successfully.
- Create a new OSSCOMServerVC object (notice that due to the "stateless" MTS programming model, a "new" object is created for each service invocation - "new" because it may be retrieved from a pool of previously instantiated objects that are now again ready to do some work).
- Call the *GetRecordInfo()* function of this object.
- Return the record XML or HTML string produced by this function.

GetFolderInfo():

Similar to *GetRecordInfo()* except invoking the *GetFolderInfo()* function of the OSSCOMServerVC object.

GetDocumentInfo:

Similar to *GetRecordInfo()* except invoking the *GetDocumentInfo()* function of the OSSCOMServerVC object.

7. Database Access

Stored Procedure Database Interface

When designing and implementing a relational database schema in e.g. SQL Server, then there are several good reasons for always encapsulating the database tables behind an "interface" of *stored procedures*. First, constraints and business rules that are inherently linked to the database schema, independent of which application (i.e., application layer - several applications may share the same database) is using the database, should be enforced within the database, and stored procedures may be the only means for achieving this. Furthermore, the tables of a database schema are often subject to minor changes, e.g. for performance reasons. Such changes should be transparent to the application layer, however. For example, if a database implements a UML model in a "one-to-one" manner, then the stored procedure interface would contain procedures for creating, updating, deleting objects of each class, amongst others. However, due to the inherent "mismatch" between a relational schema with tables and an object-oriented UML model, it may well be necessary to implement different inheritance hierarchies in the UML model differently with respect to table layout, just depending on the individual characteristics of these inheritance hierarchies. Finally, executing a set of SQL statements within a stored procedure is often more efficient than executing SQL statements individually requested by application components.

SynExML Generation

The Oslo Synapses Server provides an external "interface" of stored procedures in SQL Server's *Transact-SQL*, and such stored procedures are used to generate SynExML based on requests received (in a single string - Text datatype).

No source code will be provided for the stored procedures that generate SynExML from healthcare information in an Oslo Synapses Server since, formally, the Oslo Synapses Server is a component outside the SynEx project. For more information on the Oslo Synapses Server, please contact Siemens Health Services.

OLE DB and Universal Data Access (UDA) [10]

OLE DB is a key part of Microsoft's strategy to enable seamless access from an application layer to several heterogenous information sources; see figure 3. OLE DB itself is a set of COM interfaces that are implemented by *data providers* which are objects that offer information from e.g. a database, a file, etc, and *data consumers* which are objects that make requests for certain information from one or more data providers. In this case a *OSSCOMServerVC* object is a data consumer that uses a ready-made OLE DB provider for SQL Server as its data provider.

ADO is a simplied version of OLE DB made to make its functionality available also to Visual Basic and scripting languages. As described in the other WP2 deliverable D2.1, the "SynEx client" uses ADO for caching record information.

In a sense the purpose of OLE DB is similar to the purpose of SOAP, except that OLE DB is made for seamless access to heterogenous data sources, while SOAP is made for seamless access to heterogenous web servers.

Server-side Legacy System Integration

The WP2 platform can be used to make information available from other data sources than the Oslo Synapses Server. The data source, e.g. a legacy system with healthcare record information, must satisfy the following two requirements:

- An OLE DB provider must be made that can be used by the COM Server Component as it is, or alternatively the COM Server Component is reimplemented (but supporting the same interface) such that it can access the legacy system by whatever means are offered by this system.
- The legacy system must produce XML that is valid according to the SynExML.

Of course, independent of such *server-side legacy system integration*, the seamless integration of distributed records described in the other WP2 deliverable D2.1 is *client-side integration* and thus independent of server-side technology.

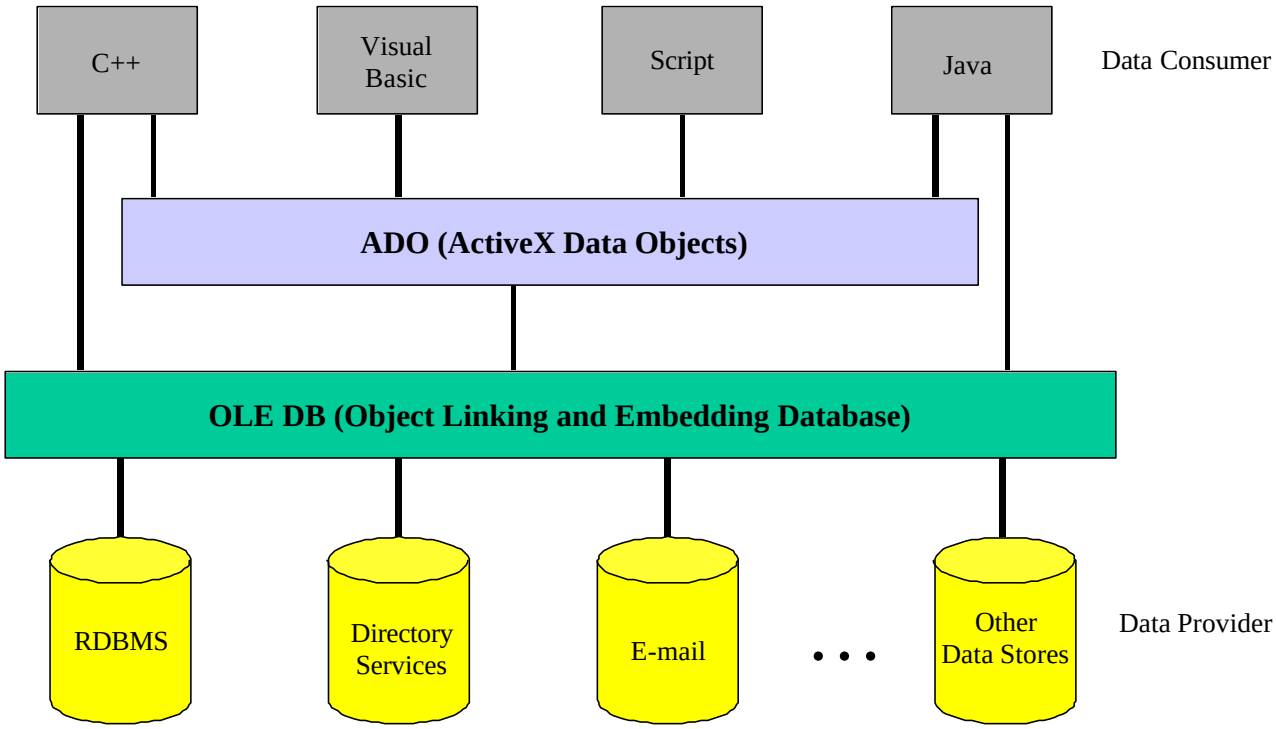

Figure 3. *Microsoft's UDA - Universal Data Access.*

8. Distribution

Shared Federated Healthcare Records in SynEx

The Oslo Synapses Server (OSS) and the Synapses Server specification does not rely on a central catalog service for retrieving information on where various parts of a particular patient record resides. Instead this information is distributed such that every record on a particular server has the information required to access other parts of it that exists in other servers. Thus the hyperlink capabilities of OSS and Synapses, together with XML and web technology as explained above, offer a good basis to realize shared federated healthcare records. A full description of how this has been achieved within WP2 is described in the other deliverable D2.1.

Client versus Application versus Database Distribution

We can distinguish between different kinds of distribution, namely (at least) database-, application- and client distribution.

Database distribution utilizes e.g. SQL Server features for this, while distribution at the application layer is handled by MTS as a collaboration of MTS objects that reside on different computers. Distributed transactions in both MTS and SQL Server are handled by the same *DTC (Distributed Transaction Controller)*.

By *"client distribution"* is here meant that distributed information is integrated at the client level, i.e., as the result of a client's request for related but distributed information (e.g. distributed information on a particular patient record). The application and data layers are not involved in the distribution except that they may contain information on how and where to get access to related information that resides elsewhere.

Database distribution should be transparent to the application layer, while distribution at the application layer should be transparent to its clients. Typically, the database and application layer distribution are "local" distribution in the sense that the databases or application components exist on computers in physical proximity, or at least the same development organization is in charge of configuring each participant in the distribution.

Client side distribution on the other hand may well be a global distribution where the organization offering the server side functionality may have no knowledge of who are the clients, and the client devices can be common web browsers, mobile phones, etc, making a connection to the server from anywhere in the world.

9. Security

Providing secure access to information is a key issue for healthcare information systems.

Authentication means that the server must be able to verify that the client is who he claims to be (e.g. via password like mechanisms, smartcards, etc), and also that no other person can take over a client's session on the server without the server (and the client) being aware of this. The authentication mechanisms for the WP2 platform has been described above.

Authorization means that a client can only access or update information for which he is authorized to do such operations. Authorization should preferably be considered an inherent part of the overall system- and information modelling. This to make it possible to assign domain specific read and write authorizations to various levels of granularity, e.g. read access to a particular document, but write access only to a particular field in this document, and also to be flexible with respect to dynamically changing authorizations. Often there will be a trade-off between flexible authorization versus performance. Thus authorization mechanisms should be taken into consideration right from the start of the analysis/design phase. The Oslo Synapses Server has already built-in authorization mechanisms that are very flexible and fine-grained. These are always used when objects in the application layer accesses the database, by providing the UserID as an argument to every stored procedure invocation.

Security relating to encryption of transferred information, client download of applets or ActiveX components, and more, are also important security topics, but there has not been an in-depth consideration of these issues within WP2 beyond what are common techniques for this.

10. Generic Platforms with UML and Meta-Information Management

The WP2 platform is made to offer two kinds of services, namely session handling with log on/log off requests, and services for record information retrieval. However, while parts of its implementation is certainly domain specific, parts of it is also generic in the sense that it can be reused for other domains. This section provides a brief outline of how UML models [12][13], or more specifically UML Class Diagrams, preferably together with a tool for meta-information management, can be used to generate some of the domain specific parts of the platform implementation thereby making it easier to customise the platform for new domains, or for extending its current domain.

Modelling with UML

UML (Unified Modelling Language) is becoming much of a standard for object-oriented and relational modelling, analysis and design. An important benefit of using UML is for communication purposes, both between software developers, but also between software developers and domain experts since it is relatively easy to enable domain experts to understand what is expressed by a UML model. However, while it is easy to read and understand a UML model, creating good UML models is something else. Due to its simplicity, mostly "boxes and lines between them", it can be difficult to distinguish sound models from nonsense.

UML can be a useful tool not only for designing the information system itself, but also for making it more generic and more adaptable to changes and extensions over time. In order for a client and a server to interact and communicate in a meaningful way they must have a common understanding regarding which requests can be made by the client, and what kind of information will be returned from the server. For example, the healthcare record integration in SynEx relies on an agreement on which requests can be made by clients, and the healthcare record information returned is structured according to SynExML, as agreed between the sites involved.

A more generic solution is to use UML models, or for the information part more specifically UML Class Diagrams, to describe which requests are available from clients, and what is the structure of the information returned. Client requests may then correspond to methods on various UML classes, and the information returned will be information on a particular set of objects instantiated from particular classes in the UML Class Diagram. The benefit of this is that regardless of which model changes and extensions are later made, the foundation for the common understanding necessary between a client and a server will remain the same, namely basic UML concepts. Of course, this is more of a syntactical foundation than a semantical foundation, but the domain specific semantics of concepts must be agreed regardless.

The modelling tool from *Rational Rose* currently offers the best support for UML. It does not cover all aspects of UML, but its Class Diagrams are useful for information modelling, and its Collaboration and Sequence Diagrams are useful for modelling object/component interaction. Rational Rose offers automation COM interfaces which makes it easy to create models or retrieve model information programmatically. It is also relatively easy to customise and extend its functionality.

Meta-Information Management with MS Repository and the Open Information model

Microsoft Repository is a tool for meta-information management. It is much more than just a database schema, more like an architecture for the management of meta-information in continuously evolving domains. It uses UML for modelling domain-dependent meta-data, relational databases for data storage, SQL for data retrieval, and XML for meta-information exchange via export/import. It consists of two major components; a set of interfaces for defining Open Information Models and to operate on the information they describe, and a Repository engine that is the underlying storage medium for these models and their data.

The *Open Information Model (OIM)* is a model that describes the objects that represent meta-information in the Repository, and their relationships. The OIM is defined in UML and it is organized into several domains (subject areas) as illustrated to the right in figure 4. That is, the OIM consists of an abstract core model that is extended with new domains; e.g. to store information on UML models, relational database schemas, COM components, datatypes, and more. There are mechanisms for extending the OIM to enable further evolution of the models, and the inclusion of new domains.

The *Repository engine* sits on top of a Repository database, which can be either *SQL Server* or a *Microsoft Jet* database (Access). It manages data in the Repository, and it provides basic functions to store and retrieve Repository objects and relationships between them. The repository engine exposes repository information as

COM objects with interfaces for manipulating data stored in a Repository, and thereby making the database schema and its storage format transparent to a user. It allows for both COM and Automation programming, thus making it relatively easy to integrate Repositories and different programming tools on the Microsoft platform.

In addition to basic object management, Microsoft Repository v.2.0 offers version and configuration management facilities. To support evolving meta-information shared by different development teams, the repository engine allows objects to be versioned, grouped into configurations, and managed using checkout-checkin.

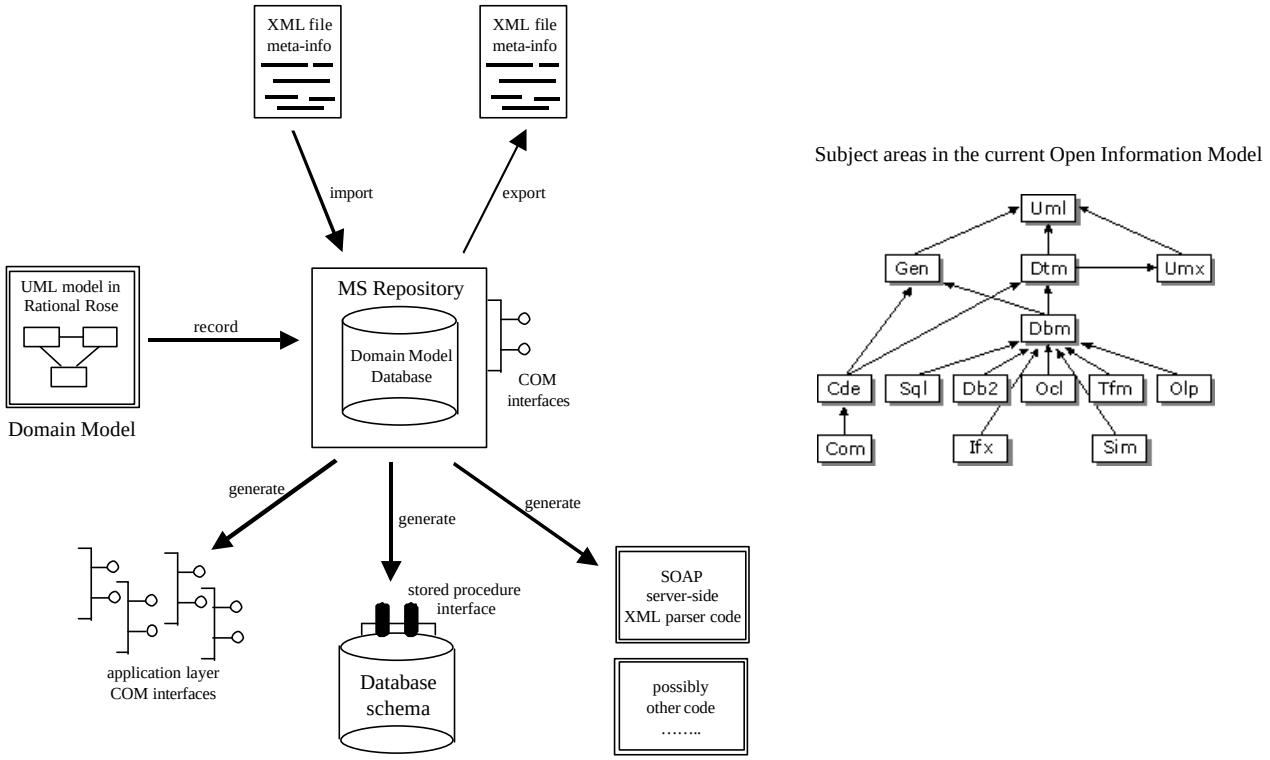

Figure 4. *UML with MS Repository for generic domain model management.*

Figure 4 illustrates how MS Repository can be used as a central repository for the various UML models that belong to a particular system, and different parts of the overall architecture can be generated on demand from this model information (meta-information). Examples of the latter can be database schemas with tables, stored procedures, indexes, etc, COM interfaces (specified in IDL) for application layer components, XML parser code for server-side parsing of client requests, and possibly more.

The primary advantage of the Repository and its OIM is that it is reasonably standardised by using UML, COM and XML, and it is likely that there will be many development tools supporting it. A current disadvantage is that while tools like Rational Rose supports it, it only does so "70%". Due to the extendability of Rational Rose you can provide the rest yourself, but there is a certain amount of coding involved. Furthermore, the Repository object model is huge and quite complex and it will take some time to get familiar with it.

Microsoft Visual Studio includes the tool *Visual Component Manager (VCM)* which closely resembles MS Repository (it is partially based on the same meta-information model). VCM can be used as a repository for organizing and storing information on components, models, projects, and more, to make them readily available to a development organization.

11. References

1. SynEx Homepage, http://www.gesi.it/synex/
2. B.Jung, E.P.Andersen, J.Grimson; *Using XML for Seamless Integration of Distributed Electronic Patient Records*; the XML 2000 Scandinavia conference; http://www.xml2000.org/program/index.html
3. Synapses Homepage, http://www.cs.tcd.ie/synapses/public/
4. P.Hurlen, K.Skifjeld, E.P.Andersen, *The Basic Principles of the Synapses Federated Healthcare Record Server*, International Journal of Medical Informatics, Vol. 52, Nr. 1-3, 1998
5. World Wide Web Consortium; *XML*; http://www.w3.org/XML
6. *SOAP specification*, http://msdn.microsoft.com/xml/general/SOAP_V09.asp
7. D.Box; *A Young Person's Guide to The Simple Object Access Protocol: SOAP Increases Interoperability Across Platforms and Languages*; http://msdn.microsoft.com/msdnmag/issues/0300/soap/soap.asp
8. Microsoft, *SQL Server*, http://www.microsoft.com/sql
9. Microsoft, *COM/DCOM*, http://www.microsoft.com/com
10. Microsoft, *UDA/OLE DB/ADO*, http://www.microsoft.com/data
11. Microsoft, *IIS/ASP*, http://www.microsoft.com/iis
12. Object Management Group (OMG), *UML*, http://www.omg.org/uml
13. Rational Rose, *UML Resource Center*, http://www.rational.com/uml
14. Object Management Group (OMG), *CORBA*, http://www.omg.org/corba

A. IDL (Interface Definition Language) Specifications

OSSCOMServerVC IDL

Source: */ServerSource/ServerIDL/OSSServer.idl*
/ServerSource/OSSCOMServerVC/OSSCOMServerVC.idl

```
// OSSServer.idl
//
// This IDL defines the interfaces for the Server component
// of the Oslo Synapses Server (OSS).

import "oaidl.idl";
import "ocidl.idl";

interface IOSSServerLogin;
interface IOSSServerXML;

	[
		object,
		uuid(7C184225-DE18-11d3-960D-0060979B4844),
		oleautomation,
		dual,
		helpstring("IOSSServerLogin interface"),
		pointer_default(unique)
	]
	interface IOSSServerLogin : IDispatch
	{
		[id(1), helpstring("Function LogOn")]
		HRESULT LogOn([in] BSTR user,
				[in] BSTR password,
				[out] long* userID,
				[out] VARIANT_BOOL* okLogOn,
				[out] BSTR* message);
	};

	[
		object,
		uuid(7C184226-DE18-11d3-960D-0060979B4844),
		oleautomation,
		dual,
		helpstring("IOSSServerXML interface"),
		pointer_default(unique)
	]
	interface IOSSServerXML : IDispatch
	{
		[id(1), helpstring("Function GetRecordInfo")]
		HRESULT GetRecordInfo([in] long userID,
					[in] long recordID,
					[in] BSTR retrievalMode,  // "SHAPE" or "ALL"
					[in] BSTR resultType,     // "XML" or "HTML"
					[out] BSTR* strResult);

		[id(2), helpstring("Function GetFolderInfo")]
		HRESULT GetFolderInfo([in] long userID,
					[in] long recordID,
					[in] long RCID,
					[in] BSTR retrievalMode,  // "SHAPE" or "ALL"
					[in] BSTR resultType,     // "XML" or "HTML"
					[out] BSTR* strResult);

		[id(3), helpstring("Function GetDocumentInfo")]
		HRESULT GetDocumentInfo([in] long userID,
					  [in] long recordID,
```

```
                                        [in] long RCID,
                                        [in] BSTR retrievalMode,  // "SHAPE" or "ALL"
                                        [in] BSTR resultType,     // "XML" or "HTML"
                                        [out] BSTR* strResult);
        };


// OSSCOMServerVC.idl : IDL source for OSSCOMServerVC.dll
//
// This file will be processed by the MIDL tool to
// produce the type library (OSSCOMServerVC.tlb) and marshalling code.

// Include MTS header to specify transaction mode for CoClass below
#include "mtxattr.h"

import "oaidl.idl";
import "ocidl.idl";

// Import OSS server specific IDL
import "OSSServer.idl";

[
        uuid(3D41E7C1-E05D-11D3-960E-0060979B4844),
        version(1.0),
        helpstring("OSSCOMServerVC 1.0 Type Library")
]
library OSSCOMSERVERVCLib
{
        importlib("stdole32.tlb");
        importlib("stdole2.tlb");

        [
                uuid(3D41E7CF-E05D-11D3-960E-0060979B4844),
                helpstring("COSSCOMServer Class"),
                TRANSACTION_SUPPORTED
        ]
        coclass COSSCOMServer
        {
                [default] interface IOSSServerLogin;
                          interface IOSSServerXML;
        };
};
```

OSSSessionManagerVC IDL

Source: */ServerSource/ServerIDL/OSSSessionManager.idl*
/ServerSource/OSSSessionManagerVC/OSSSessionManagerVC.idl

```
// OSSSessionManager.idl
//
// This IDL defines the interfaces for the SessionManager component
// of the Oslo Synapses Server (OSS).

import "oaidl.idl";
import "ocidl.idl";

interface IOSSSessionManagerLogin;
interface IOSSSessionManagerXML;

        [
                object,
                uuid(7C184222-DE18-11d3-960D-0060979B4844),
```

```
        oleautomation,
        dual,
        helpstring("IOSSSessionManagerLogin interface"),
        pointer_default(unique)
    ]
    interface IOSSSessionManagerLogin : IDispatch
    {
        [id(1), helpstring("Function LogOn")]
        HRESULT LogOn([in] BSTR user,
                      [in] BSTR password,
                      [out] VARIANT_BOOL* okLogOn,
                      [out] BSTR* message);

        [id(2), helpstring("Function LogOff")]
        HRESULT LogOff([in] BSTR user,
                      [out] VARIANT_BOOL* okLogOff,
                      [out] BSTR* message);
    };

    [
        object,
        uuid(7C184223-DE18-11d3-960D-0060979B4844),
        oleautomation,
        dual,
        helpstring("IOSSSessionManagerXML interface"),
        pointer_default(unique)
    ]
    interface IOSSSessionManagerXML : IDispatch
    {
        [id(1), helpstring("Function GetRecordInfo")]
        HRESULT GetRecordInfo([in] BSTR user,
                              [in] long recordID,
                              [in] BSTR retrievalMode,  // "SHAPE" or "ALL"
                              [in] BSTR resultType,     // "XML" or "HTML"
                              [out] BSTR* strResult);

        [id(2), helpstring("Function GetFolderInfo")]
        HRESULT GetFolderInfo([in] BSTR user,
                              [in] long recordID,
                              [in] long RCID,
                              [in] BSTR retrievalMode,  // "SHAPE" or "ALL"
                              [in] BSTR resultType,     // "XML" or "HTML"
                              [out] BSTR* strResult);

        [id(3), helpstring("Function GetDocumentInfo")]
        HRESULT GetDocumentInfo([in] BSTR user,
                                [in] long recordID,
                                [in] long RCID,
                                [in] BSTR retrievalMode,  // "SHAPE" or "ALL"
                                [in] BSTR resultType,     // "XML" or "HTML"
                                [out] BSTR* strResult);
    };


// OSSSessionManagerVC.idl : IDL source for OSSSessionManagerVC.dll
//
// This file will be processed by the MIDL tool to
// produce the type library (OSSSessionManagerVC.tlb) and marshalling code.

// Include MTS header to specify transaction mode for CoClass below
#include "mtxattr.h"

import "oaidl.idl";
import "ocidl.idl";
import "OSSSessionManager.idl";
```

```
[
	uuid(E376DFC0-E095-11D3-960E-0060979B4844),
	version(1.0),
	helpstring("OSSSessionManagerVC 1.0 Type Library")
]
library OSSSESSIONMANAGERVCLib
{
	importlib("stdole32.tlb");
	importlib("stdole2.tlb");

	[
		uuid(E376DFCD-E095-11D3-960E-0060979B4844),
		helpstring("COSSSessionManager Class"),
		TRANSACTION_NOT_SUPPORTED
	]
	coclass COSSSessionManager
	{
		[default] interface IOSSSessionManagerLogin;
			  interface IOSSSessionManagerXML;
	};
};
```

OSSWebServerVC IDL

Source: */ServerSource/OSSWebServerVC/OSSWebServerVC.idl*

```
// OSSWebServerVC.idl : IDL source for OSSWebServerVC.dll
//
// This file will be processed by the MIDL tool to
// produce the type library (OSSWebServerVC.tlb) and marshalling code.

// Include MTS header to specify transaction mode for CoClass below
#include "mtxattr.h"

import "oaidl.idl";
import "ocidl.idl";

	[
		object,
		uuid(1011ECFC-E094-11D3-960E-0060979B4844),
		dual,
		helpstring("ICOSSWebServer Interface"),
		pointer_default(unique)
	]
	interface IOSSWebServer : IDispatch
	{
		[id(1), helpstring("method HandleASPClientRequest")]
			HRESULT HandleASPClientRequest();
	};

[
	uuid(1011ECF0-E094-11D3-960E-0060979B4844),
	version(1.0),
	helpstring("OSSWebServerVC 1.0 Type Library")
]
library OSSWEBSERVERVCLib
{
	importlib("stdole32.tlb");
	importlib("stdole2.tlb");

	[
		uuid(1011ECFD-E094-11D3-960E-0060979B4844),
```

```
            helpstring("COSSWebServer Class"),
             TRANSACTION_NOT_SUPPORTED
      ]
      coclass COSSWebServer
      {
            [default] interface IOSSWebServer;
      };
};
```

B. UML Component Diagram of the WP2 Platform

Figure 5 is a Rational Rose/UML component diagram that illustrates each of the components that together constitute the WP2 platform. The rectangles with smaller interface rectangles to the left are *COM components*; except the *Oslo Synapses Server* which is an *SQL Server database*; *"oss.asp"* is an *ASP script*; *CSP_nnn* are ordinary *Visual C++ classes*, and *_Getnnn* are *Transact SQL stored procedures*.

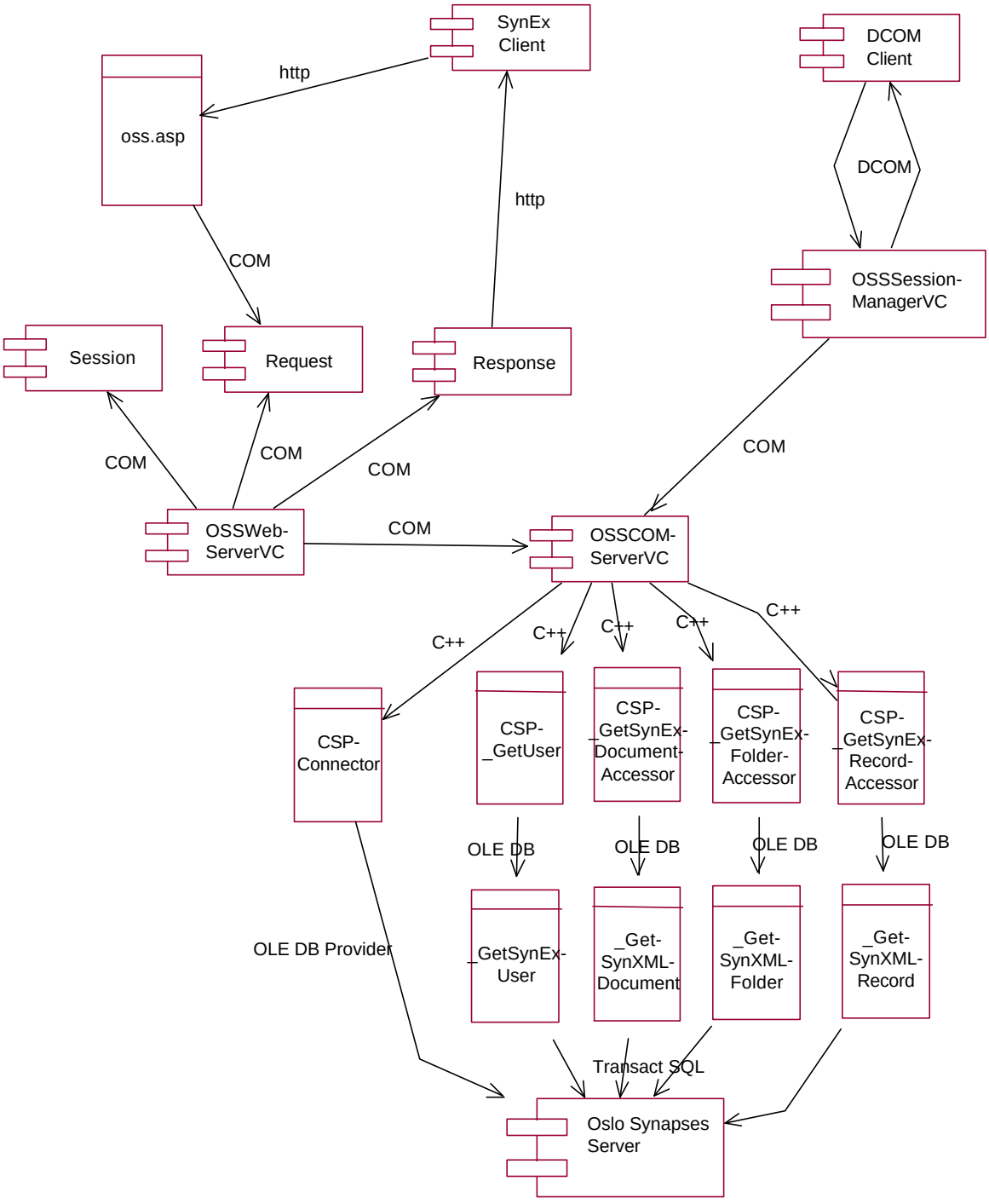

Figure 5. *A detailed outline of the WP2 platform components and their relationships.*

C. SynExML DTD (Document Type Definition)

```
<!-- ========================================================== -->
<!--    Name: SynExML                                           -->
<!--    Version: 2.1 beta 4                                     -->
<!--    Date: 04/02/2000                                        -->
<!--    Copyright: SynEx                                        -->
<!--                                                            -->
<!--    Editor:                                                 -->
<!--    Benjamin JUNG (TCD, <benjamin.jung@cs.tcd.ie>)          -->
<!--                                                            -->
<!--    Contributing editor:                                    -->
<!--    Tony AUSTIN (UCL, <t.austin@chime.ucl.ac.uk>)           -->
<!--                                                            -->
<!--    Contributing authors:                                   -->
<!--    Jose ANDANY (HUG, <jose.andany@dim.hcuge.ch>)           -->
<!--    Egil P. ANDERSEN (SHS, <egil.paulin.andersen@nr.no>)    -->
<!--    Stephane SPAHNI (HUG, <stephane.spahni@dim.hcuge.ch>)   -->
<!--    Yigang XU (Broussais, <xu@hbroussais.fr>)               -->
<!--    Vladimir YURPALOV (IBEX, <vdy@ibex.ch>)                 -->
<!--    Andrei EMELIANENKO (IBEX, <ave@ibex.ch>)                -->
<!--    Dipak KALRA (UCL, <d.kalra@chime.ucl.ac.uk>)            -->
<!-- ========================================================== -->
<!-- ========================================================== -->
<!--      GENERAL COMMENTS                                      -->
<!--      It is recommended to use the ISO Date/Time format to  -->
<!--      express Times and Dates in ELEMENT content and        -->
<!--      ATTRIBUTE values.                                     -->
<!-- ========================================================== -->

<!-- ========================================================== -->
<!--      % RICattributes                                       -->
<!--      Attributes common to every RIC in the Synapses Server -->
<!--      specification; i.e., attributes defined in class      -->
<!--      Record Component and class RIC in the Synapses Object -->
<!--      View.                                                 -->
<!--                                                            -->
<!--      Class RIC inherits class Record Component in the      -->
<!--      Synapses Server specification.                        -->
<!--                                                            -->
<!--      Record Component is the root class in the Synapses    -->
<!--      Object View. The Object View contains the classes     -->
<!--      from which objects constituting actual healthcare     -->
<!--      records are instantiated. The Synapses Class View     -->
<!--      contains classes from which objects constituting      -->
<!--      healthcare record classes are instantiated            -->
<!-- ========================================================== -->

<!ENTITY % RICattributes "ClassName          CDATA    #REQUIRED
                          RCID               ID       #REQUIRED
                          RecordID           CDATA    #IMPLIED
                          LogUserID          CDATA    #IMPLIED
                          LogTime            CDATA    #IMPLIED
                          InvalidationUserID CDATA    #IMPLIED
                          InvalidationTime   CDATA    #IMPLIED">

<!-- ========================================================== -->
<!--      % RIAttributes                                        -->
<!--      Attributes common to every RecordItem in the Synapses -->
<!--      Server specification; i.e., attributes defined in     -->
<!--      class Record Component and class RecordItem in the    -->
<!--      Synapses Object View.                                 -->
<!--                                                            -->
<!--      Class RecordItem inherits class Record Component in   -->
<!--      the Synapses Server specification.                    -->
```

```
<!-- ========================================================== -->

<!ENTITY % RIattributes "ClassName          CDATA    #REQUIRED
                         RCID               ID       #REQUIRED
                         RecordID           CDATA    #IMPLIED
                         LogUserID          CDATA    #IMPLIED
                         LogTime            CDATA    #IMPLIED
                         InvalidationUserID CDATA    #IMPLIED
                         InvalidationTime   CDATA    #IMPLIED
                         EventBeginTime     CDATA    #IMPLIED
                         EventEndTime       CDATA    #IMPLIED">

<!-- ========================================================== -->
<!--      % CommonRICAttributes                                 -->
<!--      CommonRICAttributes are attributes of RIC's that are  -->
<!--      not defined in the Synapses specification, but which  -->
<!--      all sites agree to add to this DTD.                   -->
<!--                                                            -->
<!--      The Language attribute is used to specify the language-->
<!--      used for terms within the element to which it belongs.-->
<!-- ========================================================== -->

<!ENTITY % CommonRICAttributes "Type      CDATA  #IMPLIED
                                Language  CDATA  #IMPLIED">

<!-- ========================================================== -->
<!--      % CommonRIAttributes                                  -->
<!--      CommonRIAttributes are attributes of RecordItem's     -->
<!--      that are not defined in the Synapses specification,   -->
<!--      but which all sites agree to add to this DTD.         -->
<!--                                                            -->
<!--      The Language attribute is used to specify the         -->
<!--      language used for terms within the element to which   -->
<!--      it belongs.                                           -->
<!--                                                            -->
<!--      The DataType attribute is used to specify type of     -->
<!--      data value carried by the RecordItem to which it      -->
<!--      belongs.                                              -->
<!-- ========================================================== -->

<!ENTITY % CommonRIAttributes "Type      CDATA  #IMPLIED
                               Language  CDATA  #IMPLIED
                               DataType  CDATA  #IMPLIED">

<!-- ========================================================== -->
<!--      SynExML (SynEx Markup Language)                       -->
<!--      A SynExML file can contain a set of RecordFolder's,   -->
<!--      FolderRIC's and ComRIC's in any sequence.             -->
<!--                                                            -->
<!--      Source specifies from where the XML is produced       -->
<!-- ========================================================== -->

<!ELEMENT SynExML (RecordFolder | FolderRIC | ComRIC)*>
<!ATTLIST SynExML Version CDATA  #REQUIRED
                  Source  CDATA  #REQUIRED>

<!-- ========================================================== -->
<!--      RCproperty                                            -->
<!--      RCproperties are (name,value) pairs.                  -->
<!--      They are not part of the Synapses Server              -->
<!--      specification, but they are included to support       -->
<!--      site-specific attributes. That is, conceptually they  -->
<!--      should be considered a site-specific addition to the  -->
<!--      ATTLIST for a particular element (e.g. the            -->
<!--      RecordFolder), and they are only included as nested   -->
<!--      elements within e.g. RecordFolder for DTD-technical   -->
```

```
<!--      reasons. For this reason they must always be the      -->
<!--      first elements within the element to which they       -->
<!--      belong (when parsing e.g. RecordFolder its attributes -->
<!--      should be known).                                     -->
<!-- ========================================================== -->

<!ELEMENT RCproperty (#PCDATA)>
<!ATTLIST RCproperty Name CDATA #REQUIRED>

<!-- ========================================================== -->
<!--      RecordFolder (a healthcare record)                    -->
<!--      In Synapses every healthcare record is rooted in a    -->
<!--      single RecordFolder object, and the structure of a    -->
<!--      HCR is seen as a tree-structure of RIC's with         -->
<!--      hyperlinks (ViewRIC2's) between them.                 -->
<!--      The elements that can be nested within a RecordFolder -->
<!--      element is as specified in the Synapses Server        -->
<!--      specification; i.e., either a single ViewRIC2, or a   -->
<!--      set of ComRIC's and/or FolderRIC's in any sequence.   -->
<!--      In Synapses RecordItem's are used to represent data   -->
<!--      values (as "dynamic attributes") attached to a        -->
<!--      particular RIC (a RIC as a structural element in a    -->
<!--      HCR). Thus beside its RIC children, a RecordFolder    -->
<!--      can also contain a set of RecordItem's.               -->
<!-- ========================================================== -->

<!ELEMENT RecordFolder
          (RCproperty*,
      ( (ComRIC | FolderRIC | RecordItem)* |
        (ViewRIC2, RecordItem*) ) )>
<!ATTLIST RecordFolder %CommonRICAttributes;
                       %RICattributes;>

<!-- ========================================================== -->
<!--      FolderRIC (a healthcare folder)                       -->
<!--      The elements that can be nested within a FolderRIC    -->
<!--      element is as specified in the Synapses Server        -->
<!--      specification (the same as for a RecordFolder -       -->
<!--      RecordFolder is a specialisation of FolderRIC in      -->
<!--      Synapses).                                            -->
<!-- ========================================================== -->

<!ELEMENT FolderRIC
          (RCproperty*,
      ( (ComRIC | FolderRIC | RecordItem)* |
        (ViewRIC2, RecordItem*) ) )>
<!ATTLIST FolderRIC %CommonRICAttributes;
                    %RICattributes;>

<!-- ========================================================== -->
<!--      ComRIC (a healthcare document)                        -->
<!--      The elements that can be nested within a ComRIC       -->
<!--      element is as specified in the Synapses Server        -->
<!--      specification; i.e., a set of DataRIC's, ViewRIC1's   -->
<!--      and/or ViewRIC2's in any sequence.                    -->
<!--      In addition it can contain a set of RecordItem's      -->
<!--      representing data values (as "dynamic attributes")    -->
<!--      attached to this ComRIC.                              -->
<!-- ========================================================== -->

<!ELEMENT ComRIC
          ( RCproperty*,
            (DataRIC | ViewRIC1 | ViewRIC2 | RecordItem)* )>
<!ATTLIST ComRIC %CommonRICAttributes;
                 %RICattributes;>
```

```
<!-- ========================================================== -->
<!--      DataRIC (a "field" within a healthcare document)      -->
<!--      The elements that can be nested within a DataRIC      -->
<!--      element is as specified in the Synapses Server        -->
<!--      specification; i.e., a set of more DataRIC's,         -->
<!--      ViewRIC1's and/or ViewRIC2's in any sequence.         -->
<!--      In addition it can contain a set of RecordItem's      -->
<!--      representing data values (as "dynamic attributes")    -->
<!--      attached to this DataRIC.                             -->
<!-- ========================================================== -->

<!ELEMENT DataRIC
          ( RCproperty*,
            (DataRIC | ViewRIC1 | ViewRIC2 | RecordItem)* )>
<!ATTLIST DataRIC %CommonRICAttributes;
                  %RICattributes;>

<!-- ========================================================== -->
<!--      ViewRIC1 (a "computed field" within a healthcare      -->
<!--                document)                                   -->
<!--      In Synapses a ViewRIC1 is similar to a DataRIC except -->
<!--      that its RecordItem's (its data values as "dynamic    -->
<!--      attributes") are computed on demand.                  -->
<!-- ========================================================== -->

<!ELEMENT ViewRIC1 (RCproperty*, RecordItem*)>
<!ATTLIST ViewRIC1 %CommonRICAttributes;
                   %RICattributes;>

<!-- ========================================================== -->
<!--      ViewRIC2 (a hyperlink between RIC's in two healthcare -->
<!--                records)                                    -->
<!--      A ViewRIC2 specifies a link either to another RIC     -->
<!--      within the same record, to a RIC within another record-->
<!--      at the same server, or to a RIC within another record -->
<!--      at another server. The Destination element specifies  -->
<!--      the link target.                                      -->
<!-- ========================================================== -->

<!ELEMENT ViewRIC2 (RCproperty*, Destination?, RecordItem*)>
<!ATTLIST ViewRIC2 %CommonRICAttributes;
                   %RICattributes;>

<!ELEMENT Destination EMPTY>
<!ATTLIST Destination ServerID CDATA #REQUIRED
                      RecordID CDATA #REQUIRED
                      RCID     CDATA #REQUIRED>

<!-- ========================================================== -->
<!--      RecordItem                                            -->
<!--      RecordItem is defined within the Synapses Server      -->
<!--      specification, but it is not defined with any content -->
<!--      (DataItem, as a specialisation of RecordItem, is just -->
<!--      included as an example (page 6 in the computational   -->
<!--      viewpoint)). An implementation of a Synapses Server   -->
<!--      is therefore free to define the content of            -->
<!--      RecordItem's as suits it best. However, their purpose -->
<!--      are as "dynamic attributes" to RIC's; i.e. RIC's      -->
<!--      define the structure of HCR while RI's contain the    -->
<!--      data values attached to them. Therefore, to make      -->
<!--      their "value" explicit, a Value element is added to   -->
<!--      their DTD definition.                                 -->
<!--      To allow for RecordItem's to define tree-structures   -->
<!--      of values, "RecordItem*" is added to the DTD          -->
<!--      specification.                                        -->
<!--      As for the RIC's defined above, RCproperty* is only   -->
```

```
<!--      meant to be used for extending the ATTLIST with       -->
<!--      site-specific attributes.                             -->
<!--      It is recommended to attach childelements of          -->
<!--      RecordItem in the following order: RCproperty,        -->
<!--      ElementItem, LinkItem, RecordItem, #PCDATA. It is     -->
<!--      also also recommended to keep the #PCDATA in a single -->
<!--      'data-island'.                                        -->
<!-- ========================================================== -->

<!ELEMENT RecordItem
          ( #PCDATA | RCproperty | ElementItem | LinkItem |
            RecordItem )*>

<!ATTLIST RecordItem %CommonRIAttributes;
                     %RIattributes;>
<!-- ==== END OF SynExML v.2.1 beta 3 ========================= -->
```