IT 10 019
Examensarbete 30 hp
Maj 2010

JavaScript based web service access to a functional DBMS

Di Jin

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

JavaScript based web service access to a functional DBMS

Di Jin

A new way to access different kinds of services on the web is to develop web service interfaces and call the web service operations directly from a JavaScript client. This report describes a general such web service interface from JavaScript, called the Functional web Service Client (FSC). FSC loads the WSDL document into DOM object and parses its structure to build the web service request data. It provides a public API that can be called directly from a JavaScript application. The communication between the client and the web service operations uses the XML-based SOAP protocol. FSC simplifies Ajax applications by making it very simple to call web service operations as functions. It allows both synchronous and asynchronous function calls to the web service operations. To illustrate the functionality of FSC, an existing web application, the course manager, was re-implemented as a web service as an alternative to the previous implementation as a conventional server side TomCat application. The course manager uses the functional DBMS Amos II to represent information about courses, students taking courses, exercises, etc. Rather that providing the functionality of the course manager as a web based user interface, the new implementation provides a course manager web service along with a JavaScript application implementing a user interface that calls the course manager web service operations using FSC. The course manager web service is automatically generated by the WSMOS system that, for given functions implemented in Amos II, deploys web service operations and generates a WSDL document.

Handledare: Silvia Stefanova
Ämnesgranskare: Tore Risch
Examinator: Anders Jansson
IT 10 019
Tryckt av: Reprocentralen ITC

Index

1. Introduction

The normal way to access different kinds of services on the Internet is using a web portal where a web application is accessed from a web browser. The application runs on a server and the client provides an HTML-based user interface. Alternatively the user needs to download software from the Internet before using a web-based service.

A new way to enable different kinds of web application is to provide application programs as *web services*, which provide web based Application Programming Interfaces (API)s that implement a set of services. The web service operations can be called over the Internet from application programs. The API of each web service operation is described using the Web Services Description Language (WSDL) [22] and stored in a WSDL-file accessible from the Internet. One disadvantage with conventional web service applications is that they usually call the web service API from a conventional program, e.g. written in Java. This requires that the program has to be downloaded before the application can be run.

The communication between web service client applications and web service operations is usually based on the SOAP protocol [18]. SOAP is based on calling web service operations from clients by passing XML messages between the client and the server. Both synchronous and asynchronous operation calling is supported.

This thesis presents the *Functional web Service Client (FSC)*, which allows web service operations to be transparently called in a functional style directly from JavaScript programs running in a web browser. The programmer can call a web service operation as a function by simply giving an operation name along with the actual parameter values list, without having to know anything about SOAP or XML. FSC generates the SOAP request head and body automatically. As the response, the user receives the result from the FSC call as a JavaScript object. Since the application is written completely in JavaScript no software needs to be downloaded and installed when using FSC-based applications. To make the application development simple, FSC provides a functional web service API in JavaScript that dynamically generates web service requests. To be able to automatically form the messages, FSC needs to read the WSDL document of the called web service operation and store it in a JavaScript accessible variable. The WSDL document describes the interface of the web service operations to be called in the application. Based on reading the WSDL document, FSC generates the SOAP messages used for calling the web service operations. FSC supports both synchronous and asynchronous client server communication with web service operations.

To illustrate and test the functionality of FSP, an existing *course manager* system was re-implemented using FSP. The system manages course assignments for students. The previous implementation was based on Java Server Pages (JSP) [11] and implemented with TomCat [1]. The course manager uses the functional Database Management System (DBMS) Amos II [8] as the database server to store information about courses, students, assignments, etc. The server side code was implemented as JSP documents that call Amos II functions to search and update a course database. The Amos II system was thus embedded in TomCat and the Amos II functions called from JSP pages. The Java code in the JSP pages only managed the HTML-based user interface. All application logic was provided as Amos II functions and stored procedures. To run the course manager the administrator had to install and deploy the TomCat server, the Java runtime environment, and the Amos II based course database server.

The new course manager system is implemented in an alternative way by providing the functions in the Amos II course database as web service operations. It uses the techniques from the WSMOS system [7] to automatically deploy web service operations calling the Amos II functions managing the course database. The deployed web service operations are described in an automatically generated WSDL document. Thus rather than providing the course manager as a web user interface a *course manager web service* (CMS) is provided, and this course manager web service is automatically deployed by WSMOS.

The web user interface of the new course manager is implemented in JavaScript using FSC. The web page accessed by the client's browser contains only JavaScript and HTML code, so that no code needs to be downloaded by the users. The JavaScript code utilizes both synchronous and asynchronous calls to CMS. This illustrates how FSC can be used for implementing an application calling a web service and that WSMOS can automatically generate the necessary server side code.

2. Background

This chapter introduces the background knowledge about the web services and functional web service client. It covers the definition of web service and some important related technologies, such as XML schema, WSDL, SOAP, JavaScript, Ajax technologies, and the Amos II DBMS. The last section overviews the WSMOS system that automatically deploys the course manager web service.

.

2.1 Web Services

Nowadays, web services instead of classical client-server (CS) applications are becoming a preferred architecture. It does not provide the user with a Graphical User Interface (GUI) through a web browser. Instead it shares business logic, data, and processes through a programmatic interface across a network [20]. W3C defined a web service like this:

A web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards [21].

A web service is not depending on any platform or programming language. For example, the programmer can write the client in PHP, call a web service operation implemented in Java, or the client can run on a windows system calling a UNIX web service operation. It incorporates with several standards, such as XML, WSDL, SOAP and UDDI, to support interoperable machine-to-machine interaction over a network.

2.1.1 XML

XML (short for Extensible Markup Language) is the foundation of building web services. It has some important aspects which satisfy most of the requirements of building web services such as representing simple and complex data structures, open standard, platform independence, and extensibility.

XML can describe data and semantics of data by using one or more *elements* associating meaning with data. For example:

```
<student>
  <fullname>Di Jin<fullname/>
  <email>Di.Jin@student.uu.se<email/>
  <pnumber Format=”any ten digits”>8402151420<pnumber/>
  <group Format=”any natural number”>1<group/>
</student>
```

The XML code above not only represents data values but also defines the composing of the sub-elements to group data into a complex object. The element *student* is composed of four sub-elements: *fullname*, *email*, *pnumber*, and *group*. The element *fullname* is a descriptive element tag, and *Di Jin* is the data value contained within

the element. In the same way, the element *student* is the descriptive element tag, and four sub-elements are the data value contained in *student*. Associated with the element name, each element has one or more attributes which present as a name-value pair.

Because the data structure is flexible in XML, it allows user to define arbitrary elements. This may make it hard for different users and applications to understand and interpret the data. Web services therefore uses *XML schema* [23] to describe a required structure of elements. When exchanging XML data, the service provider and the service requestor can understand and interpret the elements by sharing the same XML schema. The benefit of using XML schema is that it describes meta-data such as the names, types, structures and semantic meanings of the elements. For example:

```
<xsd:element name="USERTYPE" type="xsd:string" />
```

The prefix *xsd* in the code above identifies the element as an XML schema element. The name of the element is *USERTYPE*, and *xsd:string* means the data type of the element is a schema simple type. The simple types can be any atomic data type, such as string, integer, Boolean, double, date, and time. Furthermore, it can be XML schema defined specific collection types, in our project, e.g. *VectorofOID*, *VectorofINTEGER* , *VectorofanyType,* and so on.

XML is the foundation of specifying the description of the web services and building the web service description language. It represents and formats the data and messages used by the web services and transmitted over the Internet.

2.1.2 WSDL

WSDL (short for Web Services Description Language) [22] is a specification language for creating, describing, and publishing web services. It specifies the data formats, the communication protocols, and the public interfaces of web services in a standard way which can be accessed by other underlying programs or software systems.

Typically, a WSDL document contains descriptions of data by using one or more XML schema definitions, enabling web services providers and requestors to understand the described data. The WSDL document also contains the details of the binding protocol, the transport and parameter details of the operations, so the requestor knows how to call the operations in a correct way. It also gives the result data types of each operation, so the requestor can understand what they got from the provider. Figure 1 shows the major elements in a WSDL document.

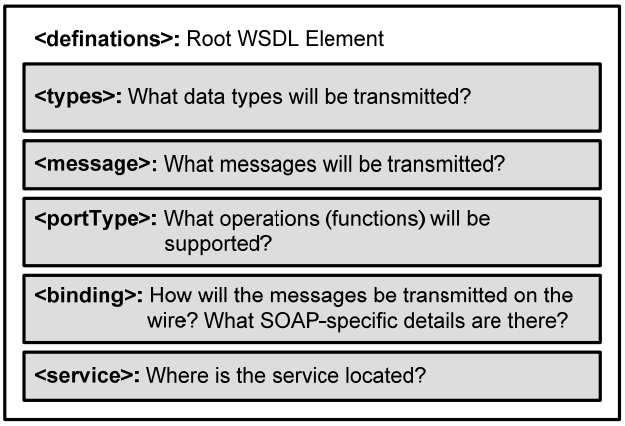

Figure 1: The WSDL specification in a nutshell.[4]

- **Definitions**: The *definitions* element is the root element of a WSDL document. It contains name, *targetNamespace* of a web service and other namespaces that are used throughout the document. These namespaces enable the WSDL document to reference external specifications, including WSDL specification, SOAP specification, and XML Schema specification [23].

- **Types**: The *types* element describes all data types in a web service. WSDL documents usually use W3C XML Schema specification as it default typing system. If the service only uses the XML Schema simple data type, such as *string, integer, Boolean, double, data* and *time*, the element *types* is not necessary. Otherwise, the user must declare own data types with the element *types*.

- **Message**: The *message* element describes the messages that are transmitted over the network. It defines the message name and zero or more message *parts*. The message part *request* specifies operation name and its parameters, and the message part *response* message specifies the result values.

- **PortType**: The *portType* element includes one or several *operation* elements as sub-elements. Each *operation* element combines the input and output elements referring to the *request* and *response* messages in the message element. It can describe not only synchronous round-trip operations but also one-way operations.

- **Binding**: The *binding* element describes the concrete specification of how a web service is communicates over the Internet and how external applications access the web service. SOAP is the most common transport for the web service, which is provided by specifying *SOAP* in the *binding* element. The sub-element *soap:binding* specifies the SOAP transport protocol and the request style: *RPC* or *document*.

- **Service**: The *service* element defines the address of the deployed web service. The user can invoke the web service operations by sending request to the specified URL specified by the *address* element.

WSDL provides the connection between the web services provider and the web services requestor. In this project, FSC reads the WSDL document and analyses its structure automatically so that the web services requestor does not need to know anything about WSDL.

2.1.3 SOAP

SOAP (short for Simple Object Access Protocol) provides communication capabilities for Web services to interact with applications over the Internet. The SOAP specification is based on a messaging framework for exchanging XML format data across the internet [5]. This messaging framework is independent of any operating system or programming language. It provides a simple and extensible communication approach.

The structure of a SOAP message is made up of several elements as shown in Figure 2:

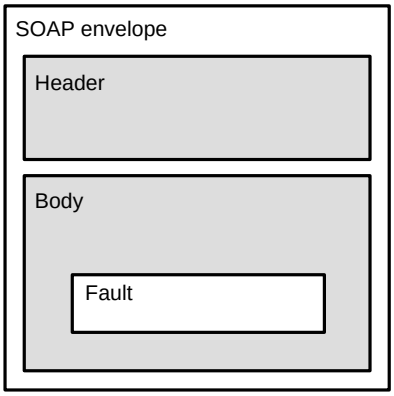

Figure 2: Main elements of the SOAP envelope

The *SOAP envelope* element is the outermost element of a SOAP message. The SOAP envelop encapsulates the header and body elements together to present the SOAP message.

The *header* is the first immediate child of the SOAP envelop. The SOAP header element provides optional features and functionalities of the SOAP message, such as security, transactions, and other facilities. Since the header is not mandatory in SOAP messages, most of the service requestor applications ignore it.

The *body* element is a mandatory element in all SOAP messages. It contains the application data of a call to a web service operation. The SOAP body is transported to the web services provider for processing. One example of a SOAP body is as follows:

```
<soap:Body>
  <LISTSTUDENTS xmlns="urn:WSAmos">
       <COL xsi:type='xsd:int'>3</COL>
       <ORDER xsi:type='xsd:string'>inc</ORDER>
       <CNAME xsi:type='xsd:string'>DBT-HT2007</CNAME>
  </LISTSTUDENTS>
</soap:Body>
```

In the code above, the immediate child of the SOAP body, the *LISTSTUDENTS* element is the operation's name. It has three sub-elements containing the name, type and value of the operation's parameters. The service provider can get the request body according to the *soap:Body* element.

If the SOAP message cannot be processed by the web service, an error exception is raised by the request. The SOAP Body then sends back a *fault* element including the *faultcode, faultstring, faultactor*, and *detail* as sub-elements [22]. The *faultcode* is a predefined code to identify a class of errors. The *faultstring* provides a human readable explanation of the fault. The *faultactor* is a text string to indicate who caused the fault. The *detail* element contains the application specific error information related to the *Body* element. It MUST be present if the contents of the Body element could not be successfully processed [6]. This SOAP error element is very useful if the SOAP message passed through several nodes and the user need to know which node caused the error.

In the project SOAP is used as the transport protocol connecting the web service and the Functional web Service Client (FSC) JavaScript code running in the client. Since SOAP is an independent and abstract communication protocol, it is easy to connect FSC to services developed in different languages or deployed on different platforms.

2.2 JavaScript and Ajax

JavaScript [13] is an object-oriented scripting language. It primarily uses in the client side of a web application and JavaScript programs run in a web browser. This enables to implement dynamic web sites enhancing the appearance of the user interface.

2.2.1 Ajax

Ajax, short for Asynchronous JavaScript and XML [6], is a group of technologies combined together to build interactive web applications. It is used to create dynamic web pages which retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing browser page [2]. The Ajax technology is incorporated with several Internet standards:

- *XMLHttpRequest* object is used to send a HTTP request to a web server. It can exchange application data asynchronously with a server.
- DOM (short for Document Object Model) is a platform and language independent data representation of XML documents.
- CSS (short for Cascading Style Sheets) is a style sheet language to describe the format and the style of a web page.
- XML is often used for formatting and transferring data.

In Ajax, instead of sending the HTTP request to the server directly, the user script first sends a message to the Ajax engine, which is an intermediary responsible for both communicating with the server and rendering the interface to the user. By introducing this intermediary, Ajax interacts between client and server asynchronous. Moreover, displaying the user interfaces is independent of the communication with servers making web applications dynamic.

Calling a web server from JavaScript is the fundamental of Ajax technology. FSC sends the web services request by using the *XMLHttpRequest* object of Ajax.

2.2.2 Synchronous call and Asynchronous call

There are two ways that Ajax accesses a server from a client. With a *synchronous* call the script running in the browser waits for response from the server before it continues to execute the script. With an *asynchronous* call the browser is not waiting for replies from the server but continues to process the script. The response is handled when it arrives to the client. None of them require the browser to reload the web page. The synchronous call must wait for the web content to be downloaded before continuing. If the amount of data transported between client and server is large, the reload time might be substantial. By contrast, the asynchronous call can make the web page run faster by downloading data in the background.

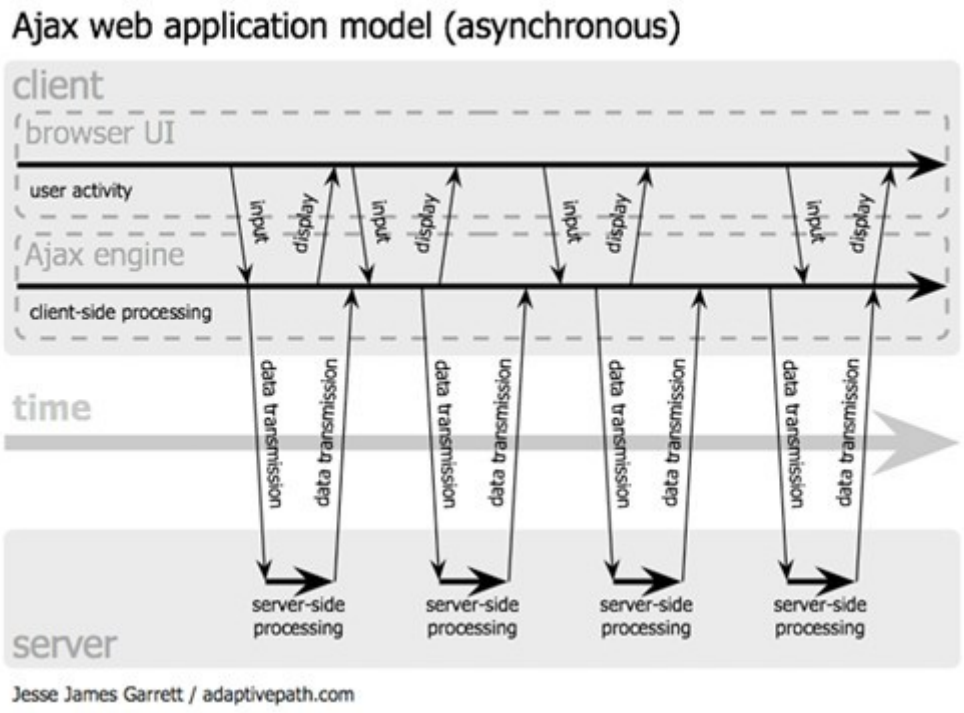

Figure 3: Asynchronous pattern of an Ajax application. [10]

In Figure 3, the Ajax engine acts as an intermediary between the client and server, it runs in a browser making the interaction more efficient. The use of Ajax does not need to install any plug-in application but requires your browser to support JavaScript.

Each user's action first generates an HTTP request object *XMLHttpRequest*, and sends the requests to the server by using the OPEN and SEND methods. If the user action does not require server manipulation, such as simple data validation, static web content display, page navigation and data edit in the browser memory, it can be handled by the Ajax engine. If the engine needs to get data from the server to satisfy the user's requirement, such as submitting data to the server for processing, loading additional interface code, or retrieving new data, the Ajax engine calls the server to complete those requests asynchronously.

Using the idle processing ability on the client-side to process some lightweight request can reduce the burden on the server-side and bandwidth. The Ajax asynchronous call does not effect on other scripts running on the page during the execution so that it is not stalling the user's waiting time [10].

2.3 Amos II

Amos II [17] is an extensible database system with a functional data model. It can store data in a main-memory and query the stored data through a functional query language, AmosQL. There are three primitive concepts in the data model of Amos II: *types, objects,* and *functions*.

Types in Amos II are used for classifying objects. They correspond to the *entity types* in an Entity-Relationship Diagram [15]. The `create type` statement creates a new user type in the database. The syntax is as follow:

```
create type <type-name>;
```

Objects in Amos II are instances some types. The syntax of creating new user objects is:

```
create <type-name> instances <variable>;
```

For example:

```
create Student instances :jd, :linda, :joh;
```

:jd, :linda, and *:joh* are environment variables which bound to three new instances of type *Student*. In the same transaction, other statement can refer to the object through these variables.

Functions in Amos II represent all properties of objects and the relationships between objects. There are four kinds of functions:

- *Stored Functions* explicitly store in the database the relationship between the argument and the result, analogous to a *table* in a relational database [8]. The "as stored" is the syntax to indicate the function type. In this project, the *course manager* system uses stored functions to represent the properties of users, courses, assignments, and other data. The syntax of defining a stored function is:

  ```
  create function <function-name(type-name)> -> <result-type> as stored;
  ```

- *Derived Functions* are defined in terms of other functions as a query [8]. A derived function uses the *select* statement of AmosQL to define the query. A derived function is side-effect free and makes no changes to the stored data. In this project, the CMS uses derived functions to represent relationships between the students, the assignments and the courses. The syntax of defining a derived functions is as follow:

  ```
  create function <function-name(type-name)> -> <result-type>
  as select [from-clause] [where-clause];
  ```

- *Foreign Functions* are Amos II functions implemented in some external language such as C or Java. This enables access from Amos II to external databases, storage

managers, or computational libraries. In this project, WSMOS uses foreign functions in Java to generate the WSDL document for the *course manager.*

- *Stored Procedures* are defined as a sequence of AmosQL statements. A stored procedure may have side-effects on the stored data, e.g. calling database update statements. Procedures may return a stream of results by using a special *result* statement. Each time *result* is called another result item is emitted from the procedure. In the project *course manager* logic is often implemented as stored procedures. These stored procedures are mapped to web service operations for the *course manager* web service. An example of a stored procedure is:

```
create function registerstudentgroup(charstring studName,
 charstring studPwd, charstring cname)
-> charstring registergroupResult
as begin
   if (studentexist(studName, studPwd, cname))
   then
   result registergroupvalidate(studName, studPwd)
   else
   result 'none';
end;
```

2.4 WSMOS

The WSMOS system [7] can dynamically deploy any Amos II function as a web service operation. Figure 4 show the architecture of WSMOS. It includes two parts. One is the WSMOS web server, the other one is the WSDL generator.

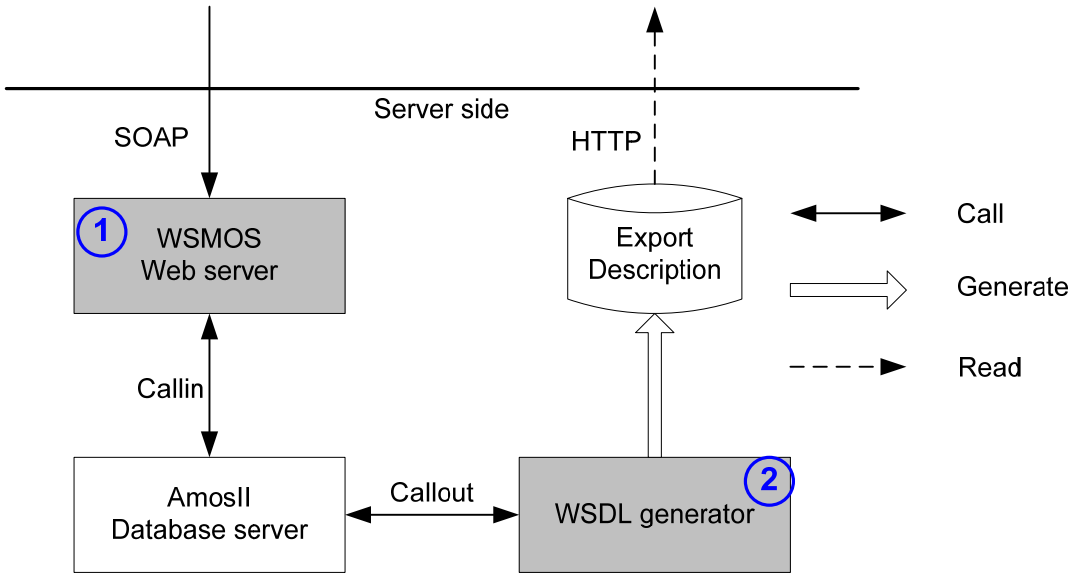

Figure 4: Architecture of WSMOS [7]

The WSMOS web server is a Java HTTP SOAP server receiving the SOAP requests and calling different Amos II functions based on the request. The WSDL generator is a foreign function of Amos II. WSMOS also contains a WSDL generator to automatically generate the WSDL document describing the web service interface for the Amos II functions provided as web service operations.

3. The Functional web Services Client (FSC)

This chapter starts with a high level view of FSC to indicate the role of FSC in the project. Then it gives an example of how to use FSC for making web service requests. After that, the chapter describes the implementation details top-down including the public APIs, the request module and the response module. The last two sections are about how FSC handles error and timeout error.

3.1 The FSC package

The Functional Web Services Client (FSC) enables web applications to call web service operations directly from a web browser. Some implementations of FSC are based on the *JavaScript SOAP Client* [14]. I made a lot of improvements so that it makes functional-styled requests to the web service operations through SOAP. It can also make request to any web service provider as long as a WSDL document is available. Figure 5 shows the high level view of FSC.

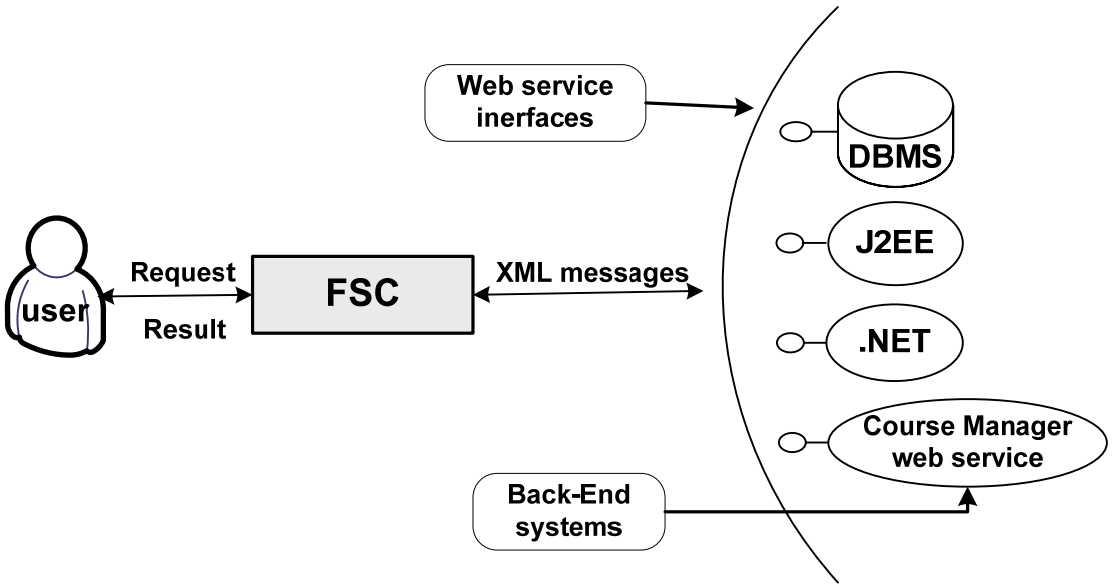

Figure 5: High level view of FSC

3.2 Example of using FSC

This is an example of using FSC to query the Amos II database to list all the students who registers to the course. An Amos II function, called *liststudents*, is defined as a derived function which is made available as a web service operation by WSMOS:

```
  create function listStudents(Integer col, Charstring order, Charstring
cname)
  ->Vector StudentInfo
  as select sortbagby((select studentInfoPublic(s)
                       from Student s
                       where coursename(s) = cname),
                       col,
                       order);
```

This function selects the public student information given the course name and gives the result as a vector of the student information called *StudentInfo*. There are three arguments in the function, the first is an integer which the column number to order the result by, the second is a string indicating the sort order ('inc' or 'desc'), and the third is the course name for which to list the students.

When deployed as a web services operation the WSMOS System generates a WSDL document and defines the function as the web service operation *LISTSTUDENTS*. The function parameters and the query result are the input and output messages. The JavaScript function *SOAPClient.invoke()* is the public API to call a web service operation, e.g. *LISTSTUDENTS*:

```
SOAPClient.invoke(“LISTSTUDENTS”, [3,”inc”,coursename],
                  true, listStudent_callBack, error_callBack);
```

The first argument is name of the web service operation to call, i.e, *LISTSTUDENTS*; the second argument is a JavaScript array holding the arguments in the call. In the example, *3* means that the result is ordered by the third column, *“inc”* means that the result is ordered increasingly, *coursename* is a string variable holding the course that students registered to. The next parameter *true* means that the request is called asynchronous. The fourth argument, *listStudent_callBack* is a JavaScript function called by FSC when the result from the request has arrived. If an error occurs, the JavaScript function *error_callBack* is called.

When the result is arriving, FCS calls the function:

```
function listStudent_callBack(o, soapResponse){
     var table = resultToTable(o,null);
   document.getElementById("listDiv").innerHTML = table;
}
```

O is the result object, a vector called *StudentInfo*. Each result in the *StudentInfo* Vector includes three *Charstring* type data, e.g. *<Charstring, Charstring, Charstring>*. According to the data type matching in FSC, vector is converted into array. The *<Charstring, Charstring, Charstring>* is also converted into an array. So *o* in this example is an array of an array, which is [[di jin, dijin@uu.se, none]]. *SoapResponse* is the response SOAP envelope helping user to analysis the result if necessary. The *resultToTable()* function is a utility function in the course manage web service, which builds the result object into a HTML table. The *listDiv* is identity of the DIV element that places the result table in the HTML page.

Table 1 shows how Amos II data types match corresponding Java class, XML data type, and JavaScript in FSC.

Amos II data type	**Java class**	**XML data type**	**JavaScript**
INTEGER	Integer	xsd:int	number
REAL	Double	xsd:double	number
BOOLEAN	Boolean	xsd:boolean	Boolean
CHARSTRING	String	xsd:string	string
Others types	Vector	tns:VectorofAnyType or other vector types	Array
Others types	Oid	xsd:string with special	string with

		syntax	special syntax

Table 1: Data type matching in FSC

In JavaScript, there is only one numeric type, *number*, so the integer and the double types are all mapped to the type *number*. The type *VectorofAnyType* or other vector types is converted to JavaScript type *Array*, since JavaScript has no vector type. The Java class *Oid* is a reference class for any Amos II surrogate data type. The WSMOS system [7] converts the *Oid* objects into a special proxy string: beginning with a prefix, "[OID", and ending by a postfix, "]". Between the prefix and the postfix it is the ID-number of the surrogate object, e.g., "[OID 1080]". [7] FSC keeps the special proxy string to represent the surrogate objects of the Amos II database.

3.3 Implementation details top-down

Figure 6 shows the implementation details in FSC. It consists of three modules, the public APIs, the request module, and the response module.

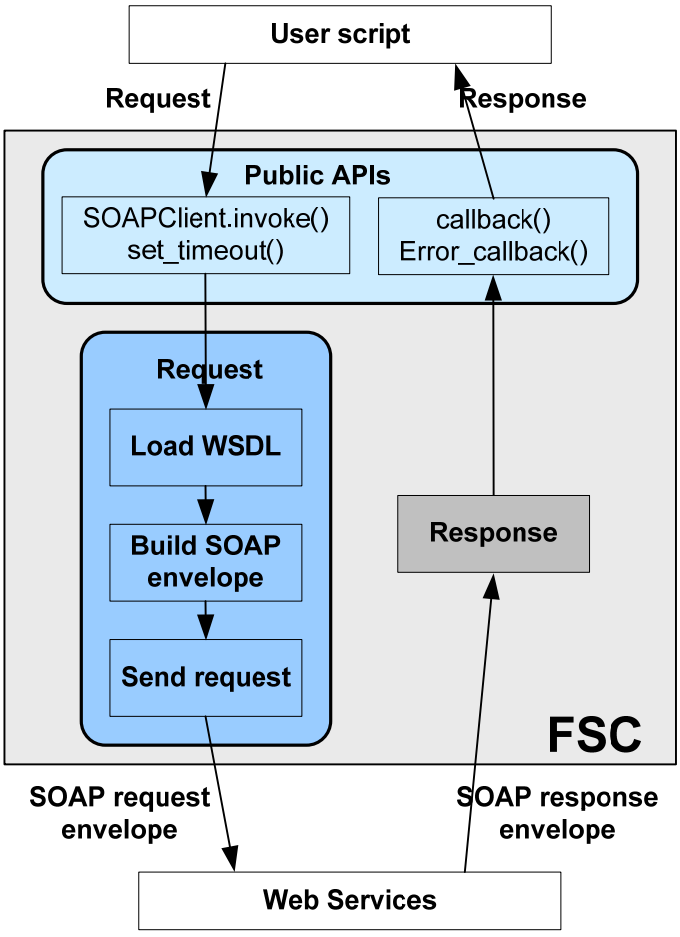

Figure 6: Architecture of FSC

The public APIs contains four public functions which can be called from user scripts. The left part is the request module consisting of three sub-modules. The *Load WSDL* sub-module reads the WSDL into a JavaScript accessible variable which is a global variable. To improve performance read WSDL objects are cached in a table in FSC so that a given WSDL document is read only once in a FSC session. The *Build SOAP envelope* sub-module parses the global variable and builds the SOAP request envelope together with the user input data. The *Send request* sub-module sends the SOAP message to the web service by using *XMLHttpRequest* object. The right part is the response module which analyzes the response according to WSDL description that saved in the global variable and extracts result objects from the SOAP envelope.

3.3.1 Application interface

In FSC, all of the code is running in a browser. There are two application interfaces in FSC for users to call web service operations:

- The JavaScript function *SOAPClient.invoke (method, parameters, asynchronous, callback, error_ callback)* is the main entry point of FSC to call a web service operation. There are five parameters:
  - *method* is the name as a string of the web service operation to call.
  - *parameters* is an array containing actual parameter values. The values must in the same order as in the WSDL document definition of the operation.
  - *asynchronous* is a Boolean variable indicating whether the call is asynchronous (true) or synchronous (false).
  - *callback* is a callback function called when the call is successful and returns a value
  - *error_ callback* is a callback function called when the call fails.
- The JavaScript function *SOAPClient.set_timeout(time)* lets the user set a timeout for each web service call. This function is not mandatory to call a web service, a user can set her own timeout time through this function or use system default timeout value.

The two call back functions have the following arguments:

- *callback (o, xmlDoc)* is called for successful results. *o* is the result from the web service operation call. *XmlDoc* is the response xml document.

- *error_ callback (e, errorCode)* is called when there is an error during the request. *e* is the error message from the web service operation *errorCode* is the error number.

3.3.2 The Request Module

The request module builds and sends SOAP requests. The user only needs to provide the operation name and parameter values since FSC loads and saves the WSDL document and parses its structure to get the necessary request data.
Figure 7 illustrates the work flow of one request:

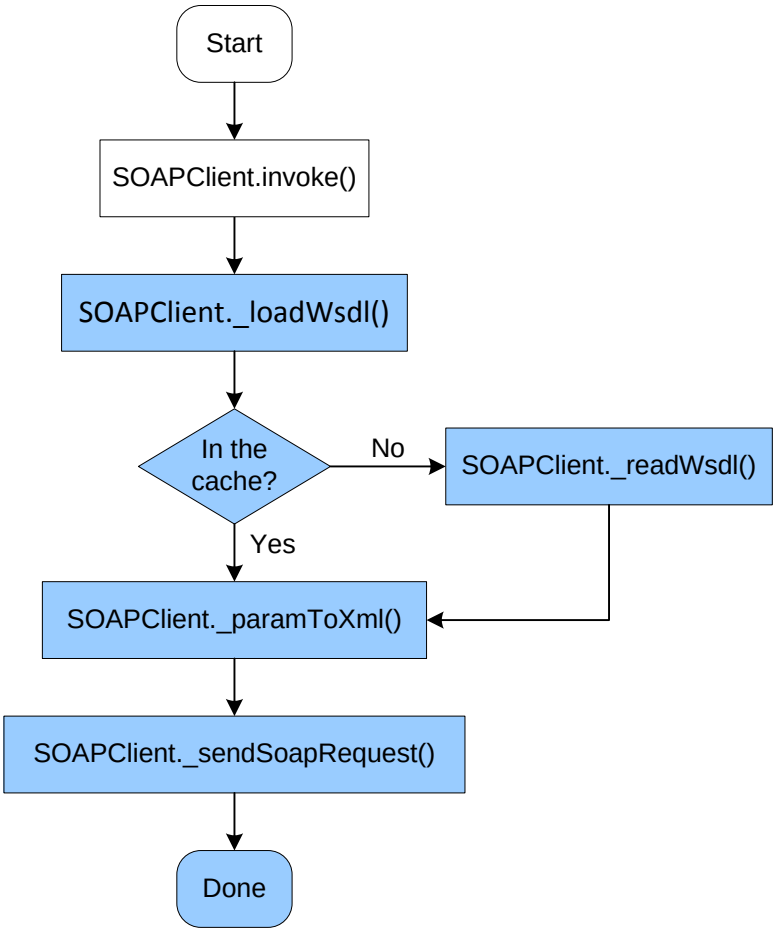

Figure 7: Send one request

The blue parts are all implemented by FSC, users do not need to know anything about these functions but simply calls the *SOAPClient.invoke()* function. There are four main internal functions in the request module.

SOAPClient._loadWsdl()

The *SOAPClient.invoke()* function first sends the request data to a *SOAPClient._loadWsdl()* function which is designed to read the WSDL only once and reuse it for all calls to the same web service in one page. Since the WSDL document is normally larger than the SOAP messages, it first checks whether the WSDL document with the user specific WSDL deployed URL has been saved in the browser cache, which is an associative array (wsdl_url and WSDL DOM object pair) called: *wsdl_cache[wsdl_url]*. If in the cache, the *SOAPClient._loadWsdl()* function sends the request to the *SOAPClient._paramToXml()* function to build the SOAP parameter body:

```
SOAPClient._wsdl = wsdl_cache[wsdl_url];
  if(SOAPClient._wsdl + ""!= "" && SOAPClient._wsdl + ""!= "undefined"){
  var ns = SOAPClient._wsdl.documentElement.attributes["targetNamespace"]…;
  var paramBody = SOAPClient._paramToXml(…);
  return SOAPClient._sendSoapRequest(…);
  …
  }
}
```

In the code above, the variable *ns* is the *targetNamespace* of the web service. The *targetNamespace* is a convention of XML Schema that enables the WSDL document to refer to itself [4]. This variable is sent to the *SOAPClient._sendSoapRequest()* to build the SOAP request.

If the WSDL document is not in the cache, the *SOAPClient._loadWsdl()* will call the *SOAPClient._getXmlHttp()* function to create an *XMLHttpRequest* object. Since the *XMLHttprequest* is supported slightly differently in IE, Safari and Mozilla-based browsers like Firefox, FSC considers this problem and uses code branching to support different browsers with right code.

```
var xmlHttp = SOAPClient._getXmlHttp(async,errorcallback);
SOAPClient._xmlhttp = xmlHttp;
xmlHttp.open("GET", wsdl_url, async);
xmlHttp.send(null);
```

The request will be created only if the WSDL document is not in the *wsdl_cache[wsdl_url]* array. In the code above, the first parameter of the *xmlHttp.open()* function initiates the request with the HTTP GET method. The second parameter specifies the WSDL document deployed URL, the third parameter means this is an asynchronous call. The *xmlHttp.send()* function is responsible for transmitting the request's content. Since no data need to be sent to the *wsdl_url*, the request content is set to *null*.

SOAPClient._readWsdl()

When the request is complete and successful, the *SOAPClient._readWsdl()* gets the *responseXML* and saves it into both *wsdl_cache[wsdl_url]* and a global JavaScript variable called *SOAPClient*.*_wsdl.* At last, the *SOAPClient._readWsdl()* function calls the *SOAPClient._paramToXml()* function to access nodes and build the request envelope.

```
  SOAPClient._readWsdl = function(url, method, parameters, async, callback,
errorcallback, req, wsdl_url){
  …
  if (httpstatus == 200 || httpstatus == 202) {
              SOAPClient._wsdl = req.responseXML;
              wsdl_cache[wsdl_url] = SOAPClient._wsdl;
        …
  var paramBody = SOAPClient._paramToXml(…);
  return SOAPClient._sendSoapRequest(…);
          }
      …
  }
```

SOAPClient._paramToXml()

The *SOAPClient._paramToXml(targetName, parameters, tagName, async, errorcallback)* function has five arguments, the *targetName* is the web service operation name input by the user, the *parameters* is the parameter value list input by the user, the *tagName* is the tag name of the specific element in a DOM object, the *async* is the flag of asynchronous or synchronous calls and the *errorcallback* is the callback function when an error occurs.

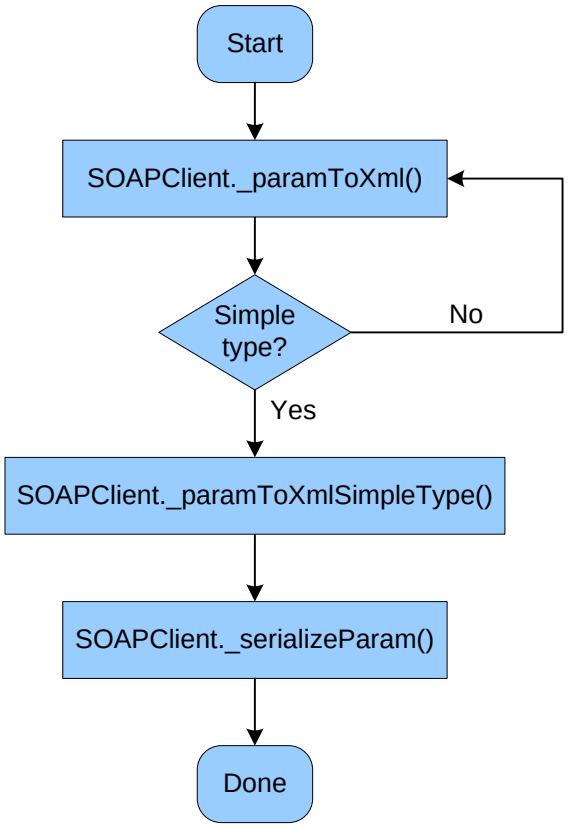

Figure 8: Build parameter SOAP body

As is shown in Figure 8, the *SOAPClient._paramToXml()* function combines a set of FSC functions together to retrieve and manipulate the content in the *SOAPClient._wsdl* variable. The entire process is handled by FSC.

It first calls a *SOAPClient._getElementByDiffTagName ()* function to save the elements in *SOAPClient._wsdl* with the same *tagName* into an array:

```
SOAPClient._getElementByDiffTagName = function(domObj, tagName){
   var ell = domObj.getElementsByTagName(tagName);
    …
   return [ell,useNamedItem];
}
```

The *ell* in the result array is an array of DOM objects. The *userNamedItem* is a Boolean argument acting as a flag to make the code support different browsers. After that, the *SOAPClient._paramToXml()* function searches the *ell* array to get the element with the given *targetName* name, and sets this element as current XMLDoc object, then calls *SOAPClient._getElementByDiffTagName ()* function again to get its *subElems* array with the same tagName. These *subElems* indicate the parameters of the operation:

```
for (var i = 0; i < elements.length; i++) {
  if (ell[i].attributes.getNamedItem("name").nodeValue == targetName;){
  var s = SOAPClient._getElementByDiffTagName(ell[i],tagName);
```

```
  var subElems = s[0];
  …
  }
}
```

If the type attribute of the *subElement[i]* is a simple XML Schema type, the *SOAPClient._paramToXml()* calls the *SOAPClient._paramToXmlSimpleType()* function to get the name and type attributes from the current element and build the parameter body of the SOAP request and adds it into a variable named *xml*:

```
var paramNames = subElems[i].attributes.getNamedItem("name").nodeValue;
var paramTypes = subElems[i].attributes.getNamedItem("type").nodeValue;
var xml = “”;
xml += "<" + paramNames + " xsi:type=\'xsd:" + paramTypes + "'>" +
SOAPClient._serializeParam (parameters[j]) + "</" + paramNames + ">";
```

If the type attribute of the *subElems[i]* is a complex type, the *SOAPClient._paramToXml()* function gets all of the elements with the complex tagName together into an array list and calls itself recursively until it gets down to a simple XML Schema type in the element. The user does not have to do this special for each web service operation call since the *Build SOAP envelope* module handles both simple and complex XML Schema types.

Finally, the *SOAPClient._paramToXmlSimpleType()* will call the *SOAPClient._serializeParam()* function to do the parameters serialization.

```
SOAPClient._serializeParam = function(o)
{
  var s = "";
  switch(typeof(o))
  {
      case "string":
        s += o.replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g,
     "&gt;");break;
      case "number":
      case "boolean":
        s += o.toString();break;
      case "object":
        if(o.constructor.toString().indexOf("function Date()") > -1){
            …
}
else if(o.constructor.toString().indexOf("function Array()") > -1){
   s = SOAPClient._serializeArray(o);
}
break;
return s;
}
}
```

From the code above we can see that FSC can handle most of the data types. When the input parameter is an array, the *SOAPClient._serializeParam()* function calls a *SOAPClient._serializeArray()* function to build the complex XML structure.

For example, Figure 9 is the DOM object representing a tree view of the XML document. It is the WSDL document of WSMOS system and saved in the variable *SOAPClient._wsdl*:

```
<wsdl:definitions ...>
  <wsdl:types>
    <xsd:schema ...>
      ...
      <xsd:complexType name="LISTSTUDENTSReturn0">
        <xsd:sequence>
          <xsd:element maxOccurs="unbounded" minOccurs="0" name="row">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="STUDENTS" type="tns:VectorofanyType" />
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
      ...
    </xsd:schema>
  </wsdl:types>

  <wsdl:message name="LISTSTUDENTSRequestMsg0">
    <wsdl:part name="COL" type="xsd:int" />
    <wsdl:part name="ORDER" type="xsd:string" />
    <wsdl:part name="CNAME" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="LISTSTUDENTSResponseMsg0">
    <wsdl:part name="results" type="tns:LISTSTUDENTSReturn0" />
  </wsdl:message>

  <wsdl:portType name="WebamosPortType">
    <wsdl:operation name="LISTSTUDENTS" parameterOrder="COL ORDER CNAME">
      <wsdl:input name="LISTSTUDENTSRequestMsg0"
                  message="tns:LISTSTUDENTSRequestMsg0" />
      <wsdl:output name="LISTSTUDENTSResponseMsg0"
                   message="tns:LISTSTUDENTSResponseMsg0" />
    </wsdl:operation>
  </wsdl:portType>
  ...
</wsdl:definitions>
```

Figure 9: SOAPClient._wsdl of WSMOS System

The root of the tree is the *wsdl:definitions* element. It has several elements *childNodes* that represent branches of the tree. If the tagName is *wsdl:message*, the *ell[0]* would look like this:

```
<wsdl:message name="LISTSTUDENTSRequestMsg0">
    <wsdl:part name="COL" type="xsd:int" />
    <wsdl:part name="ORDER" type="xsd:string" />
```

```
    <wsdl:part name="CNAME" type="xsd:string" />
</wsdl:message>
```

The targetName is the web service operation *name + "RequestMsg0"*, which is *LISTSTUDENTSRequestMsg0* in Figure 9. The *subElems[0]* is:

```
<wsdl:part name="COL" type="xsd:int" />
```

The parameter element of the SOAP body is:

```
<COL xsi:type='xsd:int'>3</COL>
<ORDER xsi:type='xsd:string'>inc</ORDER>
<CNAME xsi:type='xsd:string'>DBT-HT2007</CNAME>
```

SOAPClient._sendSoapRequest()

The *SOAPClient._sendSoapRequest(url, method, paramBody, ns, async, callback, errorcallback)* function contains seven parameters. The *url* is the web service deployed address, the *method* is the called operation name, the *paramBody* is the parameter body composed by the *SOAPClient._paramToXml()* function, the *ns* is the web service name space *targetNamespac,* the *async, callback* and *errorcallback* are the same as the corresponding parameters in the *SOAPClient.invoke()* function. It builds the SOAP request envelope as a XML sting by assembling all necessary components:

```
var sr =
    "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
    "<soap:Envelope " +
    "xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" " +
    "xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\" " +
    "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">" +
    "<soap:Body>" +
    "<" + method + " xmlns=\"" + ns + "\">" +
    paramBody +
    "</" + method + "></soap:Body></soap:Envelope>";
```

This SOAP request envelope is sent by an *XMLHttpRequest* object:

```
var xmlHttp = SOAPClient._getXmlHttp(async,errorcallback);
SOAPClient._xmlhttp = xmlHttp;
xmlHttp.open("POST", url, async);
```

It uses the HTTP POST method of the *XMLHttpRequest.open ()* function to initialize the request of sending a SOAP request message. Since the request can be either asynchronous or synchronous; the *XMLHttpRequest.send (sr)* function sends the request body in different ways.

```
if (async) {
   xmlHttp.onreadystatechange = function(){
```

```
      if(xmlHttp.readyState == 4){
         SOAPClient._getSoapResponse(…);
      }
    }
    xmlHttp.send(sr);
 } else if (!async){
    xmlHttp.send(sr);
    return SOAPClient._getSoapResponse(…);
 }
```

When the request is set as asynchronous, before calling *xmlHttp.send(),* FSC specifies an *xmlHttp.onreadystatechange()* function. In the code above the *readyState* is 4, which means that the request is complete. The function calls the *SOAPClient._getSoapResponse()* function immediately. When the request is set as synchronous, FSC will call the *SOAPClient._getSoapResponse()* function after sending the SOAP request.

The *onreadystatechange()* function acts as a time trigger, typically binding to a JavaScript function called whenever the state of the request is changing, or the *readyState* property value is changing. There are five *readyState* states describing different statuses of the request shown in Table 2.

State	Description
0	The request is not initialized
1	The request has been set up
2	The request has been sent
3	The request is in process
4	The request is complete

Table 2: Possible values for the readyState property [1]

Since FSC is a generic web service client package, it can communicate with different web services providers, not only the WSMOS web server but also other web services i.e., Terraservice, Yahoo web services.

3.3.3 The Response Module

Figure 10 illustrates the work flow of one response:

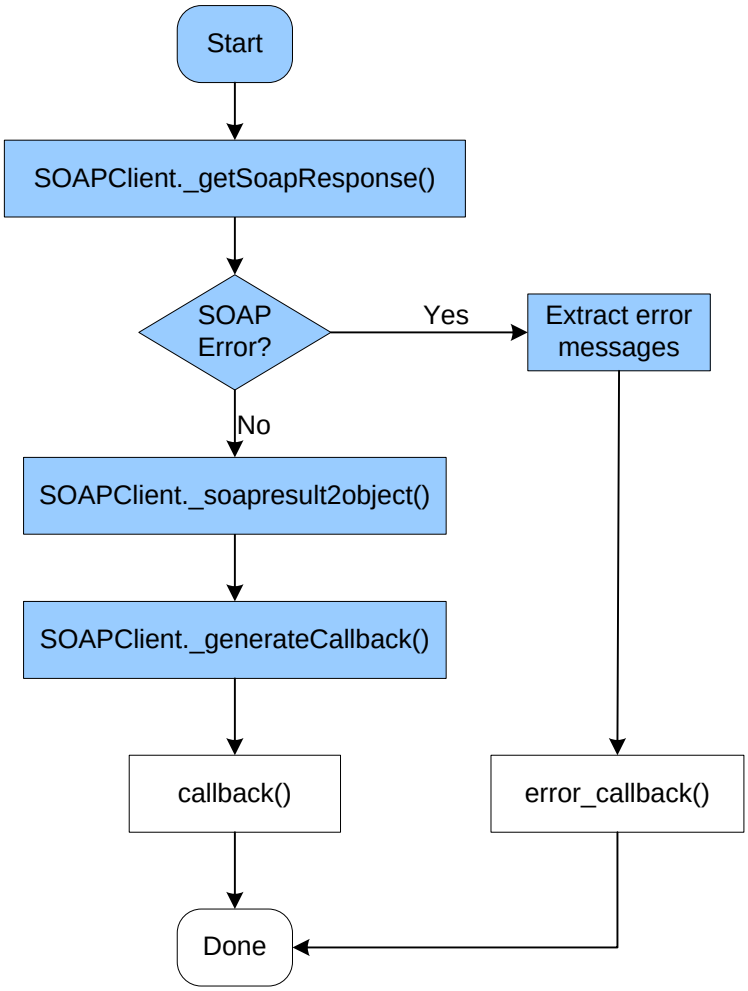

Figure 10: Get one response

When the request is complete and successful, the response is sent to the *SOAPClient._gerSoapResponse()* function. The function processes the result SOAP to see if there is an error. If there is an error, the function will extract the error detail and send it to the user through the *error_callBack()* function. If there is no error, the function will pass the result to *SOAPClient._soapresult2object()* function to extract the result object and sent it to the *callBack()* function. The blue blocks are all implemented by FSC, and the user can get the extracted result object or error messages directly.

SOAPClient._getSoapResponse()

A SOAP response structure is as following:

```
<SOAP-ENV:Envelope …">
  <SOAP-ENV:Body xmlns:tns="urn:WSAmos">
    <tns:EXISTCOURSEReturn>
      <tns:results>
        <tns:row>
```

```
          <COURSENAMES>
            <tns:member xsi:type="xsd:string">DBT-HT2006</tns:member>
            <tns:member xsi:type="xsd:string">DBT-ST2006</tns:member>
          </COURSENAMES>
        </tns:row>
      </tns:results>
    </tns:EXISTCOURSEReturn>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The *SOAPClient._getSoapResponse()* first gets the result element into the JavaScript variable *nd*:

```
var nd = SOAPClient._getElementsByTagName(req.responseXML, tagName);
```

The *req.responseXML* holds the XML format response data. The *tagName* is the result element tag name. It may be different depending on the response SOAP, however, FSC can handle different result elements using code branching. In the code above, the tagName is *tns:results*. If the length of *nd* is empty, an error has occurred during in the web service. I will discuss the error handling in chapter 3.4. If *nd* is not empty, the *SOAPClient._getSoapResponse()* calls the *SOAPClient._soapresult2object()* function to extract the result.

SOAPClient._soapresult2object()

The *SOAPClient._soapresult2object()* shown in Figure 10 is the entry point of extracting result objects. When getting the result element, it uses the *SOAPClient._getTypesFromWsdl()* function to go through the variable *SOAPClient._wsdl* again and save all result types in a variable called *wsdlType.* Then it calls *SOAPClient._node2object()* to simplify the structure of the result element:

```
SOAPClient._soapresult2object = function(node){
    var wsdlTypes = SOAPClient._getTypesFromWsdl();
    return SOAPClient._node2object(node, wsdlTypes);
}
```

After being processed by the *SOAPClient._soapresult2object()* function, the simplified element only contains the elements holding the result data. For example, in the above response SOAP, the simplified result element is:

```
<COURSENAMES>
     <tns:member xsi:type="xsd:string">DBT-HT2006</tns:member>
     <tns:member xsi:type="xsd:string">DBT-ST2006</tns:member>
</COURSENAMES>
```

The *SOAPClient._node2object()* function handles different type of result elements.

```
SOAPClient._node2object = function(node, wsdlTypes){
    if(node == null)
```

```
        return null;
    if(node.nodeType == 3 || node.nodeType == 4)
        return SOAPClient._extractValue(node, wsdlTypes);
    var isarray =
SOAPClient._getTypeFromWsdl(…).toLowerCase().indexOf("arrayof") != -1;
    if (isarray){
        return SOAPClient._handlArray(node, wsdlTypes);
    }
    if (node.hasChildNodes()){
        if (node.childNodes[0].hasChildNodes()){
            return SOAPClient._handleVector(node,wsdlTypes);
        }
        else{
            return SOAPClient._node2object(node.childNodes[0], wsdlTypes);
        }
    }
    else return null;
}
```

If the *node* is empty, the function will return a *null* to the result object. If the *nodeType* of the node is 3 or 4, which indicates that it is a text node, the function will call the *SOAPClient._extractValue()* to extract the node value. If the *SOAPClient._getTypeFromWsdl()* function returns an array, the function will call the *SOAPClient._handlArray()* function to handle the array element. If the result element includes multiple *childNodes*, the function will call the *SOAPClient._handleVector()* function to handle the vector element. Otherwise the function will call itself recursively to get the object.

Finally, the *SOAPClient._extractValue()* function extracts the result object according to the data type matching in Table 1 and give it back to the *SOAPClient._soapresult2object()* function.

```
SOAPClient._extractValue = function(node, wsdlTypes)
{
    var value = node.nodeValue;
    switch(SOAPClient._getTypeFromWsdl(…)
    {
        default:
        case "string":
            return (value != null) ? value + "" : "";
        case "boolean":
            return value + "" == "true";
        case "int":
        …
    }
}
```

So the result of above SOAP response is the array *[DBT-HT2006, DBT-ST2006]*.

SOAPClient._generateCallback()

The *SOAPClient._generateCallback()* function calls the *callBack(o, xmlDoc)* function in the user script and gives the results in both object and XML format to the user.

3.4 Error Handling

FSC acts as a black box for the user, there are only four public APIs connecting to the user. Since the interaction between the user script and the Web Service operation is complicated, it needs a complete, reliable and understandable mechanism to catch and throw exceptions. Fortunately, SOAP has such a comprehensive mechanism.

When an error occurs after sending the request, i.e, in the internal of the web services, SOAP returns a fault element in the SOAP body to the FSC:

```
<SOAP-ENV:Envelope …>
  <SOAP-ENV:Body>
         <SOAP-ENV:Fault>
            <faultcode>SOAP-ENV:Server</faultcode>
            <faultstring>callin.AmosException</faultstring>
               <detail>
                <tns:FaultDetail xmlns:tns=”urn:WSAmos”>
                  ERROR BAD_REQUEST.
                Function,CHARSTRING.existcourse does not exist.
                  </tns:FaultDetail>
             </detail>
         </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

FSC extracts the *faultcode*, *faultstring* and *detail* to the *error_callBack* function so that the user can get an intuitive error message from FSC. One extracted fault message looks like this:

```
faultcode: SOAP-ENV:Server
faultstring: callin.AmosException
detail: ERROR BAD_REQUEST. Function,.existcourse does not exist.
```

This fault occurred on the server side, the explanation of the fault is the string *callin.AmosException* and the fault detail is *ERROR BAD_REQUEST. Function,.existcourse does not exist*. If an error occurs before or during the HTTP request, FSC will call the *XMLHttpRequest.abort ()* function to stop current request and throw the exceptions by using try and catch statements.

```
Try {
var httpstatus = req.status;
var httpstatusText = req.statusText;
if (httpstatus == 200 || httpstatus == 202) {
          SOAPClient._wsdl = req.responseXML;
        }else{
```

```
            throw new Error(httpstatus + " , " + httpstatusText +  "\n
description of the error");
          }
  }catch (e) {
           return errorcallback(e.message,httpstatus);
  }
```

The code above shows how FSC catches an HTTP error. If the HTTP request status code equals to 200 (OK) or 202 (Accepted), the request is successful and the client gets the *responseXML*, otherwise the request has failed and the package returns an *httpstatus* code, an *httpstatusText* and a description of the error to the user. Like in Java programs, once the try statement throws an error, the catch statement catches it and calls the *errorcallback(e.message, httpstatus)* function. The *e.message* is the error message, *httpstatus* is the error code. If the error occurred during HTTP transmission, the error code is the standard HTTP status code [9]. If the error occurred in the FSC package, the error code is a self-defined FSC package error code shown in Table 3:

Status-Code	**Description**
600	Invoke argument error.
601	Invalid input parameters, the parameter is not correct.
602	Object could not be found.
603	Get Local Xml error.
604	Timeout error.
605	Browser does not support specified objects.

Table 3: Self define FSC package's error code

3.5 Timeout in FSC

Timeout time is the time interval between sending request to the server and getting response to the client. If the program does not limited the response time, the user might wait for the response for an unlimited time after sending the request. It reduces the efficiency of FSC. If there is no timer to count the request time and abort the request, it might also affect the web service performance. Therefore setting timeout in FSC is very important. When the request time exceeds the timeout time, FSC will return an error message to the client and abort the current request to release the occupied server resources. *Set_timeout()* is a public API in FSC, and users

can use it define their own timeout time or use the system default timeout time, *5000* milliseconds, in FSC. The time parameter in this function is milliseconds.

```
  SOAPClient._getTimeoutError = function (…){
      if (xmlHttp != null) {
          xmlHttp.abort();
      }
      window.clearTimeout(SOAPClient._timeout);
      SOAPClient.errorcallback("Timeout error,the server did not response in
" + (SOAPClient._delay/1000) + " second.", 604);
  }
```

The code above is the *SOAPClient._getTimeoutError()* function, it first aborts the current request then clears the timeout time eventually calls the *errorcallback* function to give timeout error to the user. The timeout error number is 604, the error message is "*Timeout error,the server did not response in (SOAPClient._delay/1000) second*".

4. The *course manager* web service (CMS)

This chapter describes the *course manager* web services (CMS) which is a proof-of-concept application of FSC. Then it compares the architectures of the old Tomcat based implementation with the FSC based one and discusses how FSC made the *course manager* implementation simpler. At the end, the chapter compares the code of calling a web server operation to show how FSC made the code simple.

4.1 Implementation of CMS

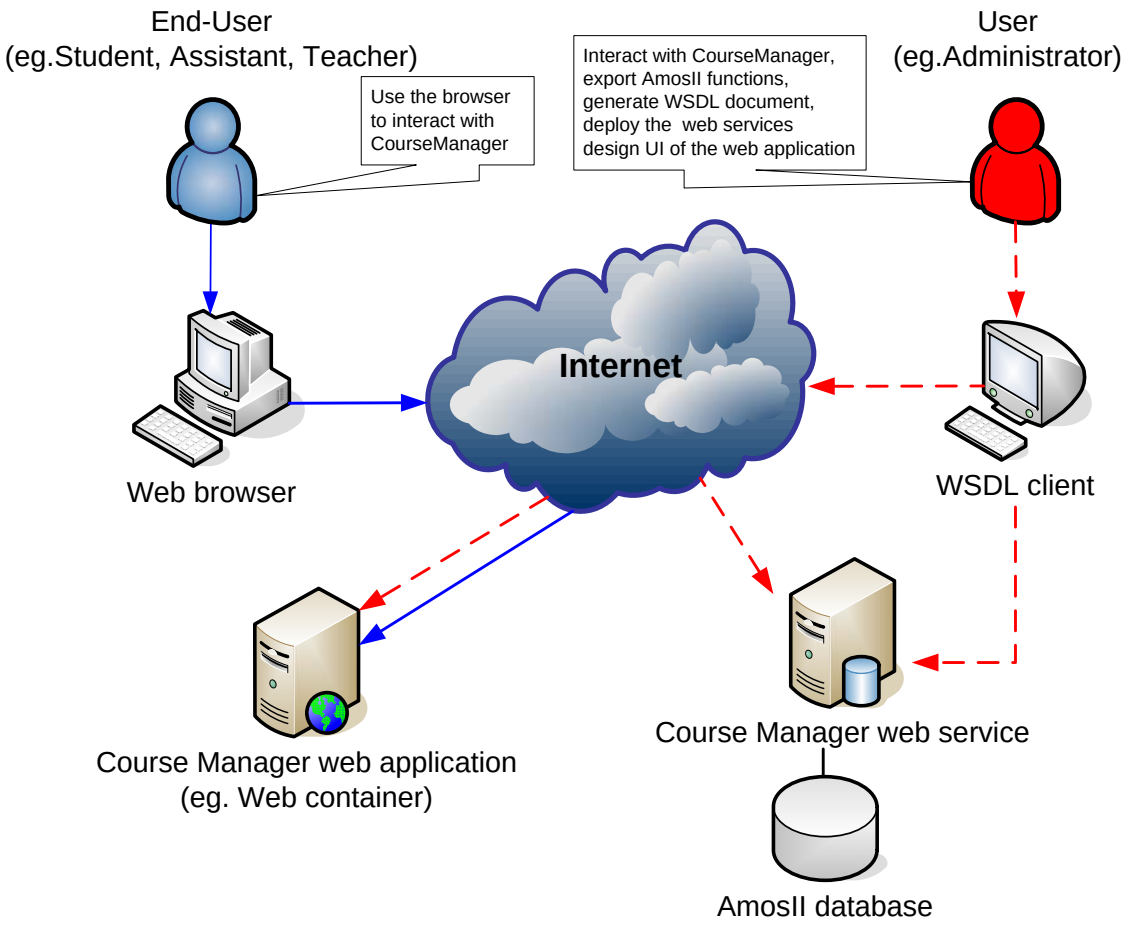

Figure 11: User communication with the CMS system

In Figure 11, CMS is provided as a web service, rather than a server side TomCat application [19]. The functions of CMS are provided as web service operations described by a WSDL document. There are two kinds of users in the system. One is the end-user, i.e, students, assistants, and teachers, who use CMS as a web application. The other is the administrator, who exports Amos II functions, generates WSDL documents, deploys the web service operations, designs user interfaces of the CMS web applications, and maintains the system.

The client side of CMS is a user script calling FSC. The user script presents the graphical user interface, and FSC builds the SOAP envelope and sends the web service request. The entire client side is defined as JavaScript and html code so that it can run on a standard browser and does not require the user to download any program. The server side of the system contains the WSMOS server and the Amos II database server, the WSMOS server helps the administrator to generate the operations and the Amos II database functions implements the CMS functionalities.

4.2 Comparing JSP based course manager with CMS

Comparing the architectures

The difference between the architectures of the JSP based *course manager* system and CMS are shown in Figure 12.

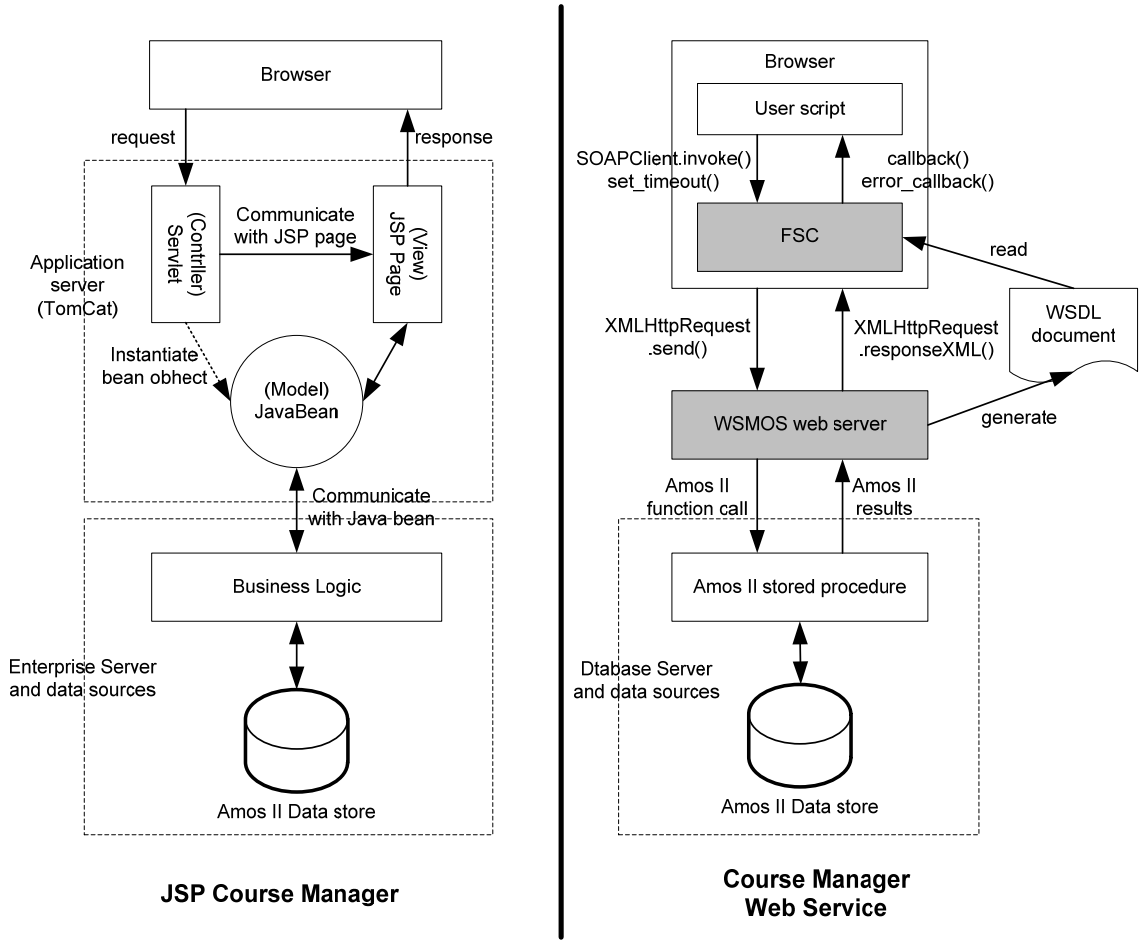

Figure 12: Comparison of JSP Course Manager and CMS

In the JSP Course Manager, when the user sends a request to the application server, the servlet first communicates with the JSP page and collects the requested data and instantiates Java Bean objects. Then the Java Bean object forwards the request to the business Logic. After querying the database, the enterprise server and data sources forms the result and composes the bean object that the JSP page needed and sends the response to the browser.

In CMS, there are four public APIs to help the user interact with the web service sending request and getting response. From the architectures of Figure 12, we can get that:

- CMS has no Java code in the client, it only contains JavaScript and HTML code so it runs in a browser.
- The FSC and WSMOS components are acting as two black boxes to the user. The user calls the web service operations by using the functional style public APIs. No internal technologies need to be known such as SOAP or XML. If the user wants to make changes in a function she only needs to re-write the Amos II stored procedures and re-deploy the function as a web service operation.
- CMS defines the Amos II stored procedures and exports them as Web Service operations by using the WSMOS system. The WSMOS generator generates the WSDL document automatically to help the user to get the information of the operations.

Comparing the codes

When invoking a web service call with FSC, the user only needs to give the operation name along with a parameter values list to the *SOAPClient.invoke()* function. The user can get the result object directly without knowing anything about XML, SOAP, WSDLE, etc. We take the code of *LISTSTUDENTS* function as an example to show how FSC make CMS's code simple and compact compared with a the JSP implementation on the server.

JSP Course Manager	CMS

`//Get Amos II connection`
 	
`Connection con = amos.getConnection();`
 	
`//Save arguments in Tuple`
 	
`Tuple arg = new Tuple(2);`
 	
`arg.setElem(0, 3);`
 	
`arg.setElem(1, "inc");`
 	
`//Call Amos II function`
 	
`Scan theScan = con.callFunction("INTEGER.CHARSTRING.LISTSTUDENTS->VECTOR", arg);`
	`//call web service`
	
	`Function listStudent(){`
	
	`SOAPClient.invoke("LISTSTUDENTS", [3,"inc",cname], true, listStudent_callBack,error_callBack);`
	
	`}`

//Set result table header
 	
String colHeader =
"<TR><TH>Name</TH><TH>e-
mail</TH><TH>Group No.</TH><TR>";
 	
//Format result into HTML table
 	
out.println(jspamos.Utilities.resultTo
Table(theScan, "",colHeader));
	function listStudent_callBack(o,
	xmlDoc){
	
	//Set result table header
	
	var tblHead = "<tr><th>Name</th><th>e-
	mail</th><th>Group No.</th></tr>";
	
	//Format result into HTML table
	
	var table =
	resultToTable(o,null,tblHead);
	
	}

//Catch error
 	
try{…}catch (Exception e) {
 	
out.println(e);
 	
e.printStackTrace();
 	
}
	//Catch error
	
	function error_callBack(e,errorCode){
	
	var error="Error message: " + error +
	", Error code: " + errorCode;
	
	}

Table 4: Comparison of the code in JSP Course Manager and CML

In Table 4, when making a request, the JSP Course Manager server side code first creates an Amos II *Connection*, *con*, sets creates an object of class *Tuple* holding the request arguments, and then uses the method *callFunction()*to call an Amos II function from Java code. The classes *Tuple* and *Connection* are defined in the *callin* interface. In contrast, the CMS user can make a call simply by calling *SOAPClient.invoke()* function from JavaScript code in the client. When processing the result, the JSP Course Manager returns a scan and uses a utility Java method, *resultToTable()*, to convert it into a HTML table. CMS returns an object and uses the utility JavaScript function, *resultToTable()*, to build HTML table. There are some utility functions in CMS that use DOM objects and methods to convert different type of result object into corresponding HTML components.

When catching errors, both implementations give the error messages. CMS puts the error into an *error_callBack()* function associating with an error code. Through the comparison in Table 4, we get that:

- FSC not only displays the User Interfaces but also builds the request to the web service operation directly so that it transfers some server burden to the web browser.
- FSC users can use the same *SOAPClient.invoke()* function along with the operation's name and parameter value list to call any Amos II function that is deployed by WSMOS. The result is handled through the callback function.

- CMS can make both synchronous and asynchronous call to the web services, which the JSP Course Manager can not. Asynchronous calls increases the interactivity of the user interface of CMS.

5. Conclusions

In conclusion, the project devises a different approach to query data through web services by introducing FSC. Using FSC, users can make a web service request by simply providing the web service operation's name and parameter values list as input data to a JavaScript function *SOAPClient.invoke()*. FSC builds the SOAP request envelope automatically and sends the request SOAP either asynchronously or synchronously. In the response, FSC extracts the result object from the SOAP response envelope. Since FSC is a generic JavaScript package, users can use it for calling standard web services through its public APIs. During this process users do not need to know anything about WSDL, SOAP or XML but get the result converted into JavaScript objects.

As a proof-of-concept application of FSC, the *cource manager* system (CMS) uses FSC as its web service client which makes the system more flexible and extendible than a corresponding implementation based on Java Server Pages. Asynchronous calls increase the efficiency of loading web pages. The CMS client can run in most of the standard web browsers because all code is written in JavaScript. Any Amos II function can be deployed as a web service operation by using the WSMOS system and called by FSC. I also evaluated FSC with some other web services. Most of the web service operations can be handled by FSC except those who need special SOAP header or tag name to identify the request.

References

[1]. AJAX - The XMLHttpRequest Object, http://www.w3schools.com/Ajax/ajax_xmlhttprequest_create.asp, last viewed 2010-02-18.

[2]. AJAX (Programming). http://en.wikipedia.org/wiki/AJAX, last viewed 2010-01-31.

[3]. Apache Tomcat, http://tomcat.apache.org/, last viewed 2010-04-01.

[4]. E.Cerami: Web Services Essentials Distributed Applications with XML-RPC, SOAP, UDDI & WSDL, First Edition February 2002, O'Reilly, ISBN 10: 0-596-00224-6

[5]. E.Newcomer: Understanding Web Services: XML, WSDL, SOAP, and UDDI, 1 edition May 23, 2002, Addison-Wesley Professional, ISBN-10: 0201750813

[6]. Extensible Markup Language (XML), http://www.w3.org/XML/, last viewed 2010-05-25.

[7]. F.Luan: Automatic web services Generator for Data Access, UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden, February 26, 2007, http://user.it.uu.se/~udbl/Theses/FengLuanMSc.pdf, last viewed 2009-02-18

[8]. G.Fahl and T.Risch: AMOS II Tutorial, Uppsala Database Laboratory, August 20, 2008, http://user.it.uu.se/~udbl/amos/doc/tut.pdf, last viewed 2009-02-18

[9]. Hypertext Transfer Protocol -- HTTP/1.1, http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html, last viewed 2010-02-20.

[10]. J. J.Garrett (2005-02-18). Ajax: A New Approach to Web Applications. AdaptivePath.com. http://www.adaptivepath.com/ideas/essays/archives/000385.php, last viewed 2010-01-31.

[11]. Java Server Pages Technology, http://java.sun.com/products/jsp/, last viewed 2010-03-27.

[12]. JavaScript: Use a Web Proxy for Cross-Domain XMLHttpRequest Calls, http://developer.yahoo.com/javascript/howto-proxy.html, last viewed 2010-02-21

[13]. JavaScript, http://en.wikipedia.org/wiki/JavaScript, last viewed 2010-05-25.

[14]. M.Casati: JavaScript SOAP Client, http://www.codeproject.com/kb/Ajax/JavaScriptSOAPClient.aspx, last viewed 2010-05-31.

[15]. M.Chapple: Entity-Relationship Diagram, http://databases.about.com/cs/specificproducts/g/er.htm, last viewed 2010-05-26.

[16]. R.Shannon, Ajax, http://www.yourhtmlsource.com/javascript/ajax.html, last viewed 2010-05-28.

[17]. S.Flodin, M.Hansson, V.Josifovski, T.Katchaounov, T.Risch, M.Sköld, and E.Zeitler: Amos II Release 12 User's Manual , Uppsala DataBase Laboratory, http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html, last viewed 2010-01-31.

[18]. Simple Object Access Protocol (SOAP) 1.1, W3C Note 08 May 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, last viewed 2010-02-20.

[19]. T.Risch: JavaScript based web services access to a functional DBMS, 2009-05-07

[20]. Web services, http://www.webopedia.com/TERM/W/Web_Services.html, last viewed 2010-02-18.

[21]. Web services Architecture, W3C Working Group Note 11 February 2004, http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, last viewed 2010-02-18.

[22]. Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001, http://www.w3.org/TR/wsdl, Retrieved 2010-02-18.

[23]. XML Schema, The XML Schema Working Group, http://www.w3.org/XML/Schema, last viewed 2010-5-26.

Appendix A: Source code of FSC application

Some of the important functions in FSC application:

```
SOAPClient.invoke = function(method, parameters, async, callback,
errorcallback)
{
    if(SOAPClient._delay == undefined){
        SOAPClient._delay = 50000;
    }
    try{
        if (arguments.length < 5 || arguments.length > 5)
        {
            throw new Error("Invalid input argument, SOAPClient.invoke
method requires 5 arguments, but " + arguments.length + (arguments.length ==
1 ? " was" : " were") + " specified.");
        }
    }
    catch(e){
        alert(e.name+" "+e.message);
        return;
    }
    SOAPClient.errorcallback = errorcallback;
    var xmlDoc = SOAPClient._getURL(async,errorcallback);
    try{
        var wsdl_url = xmlDoc.getElementsByTagName("wsdl-
url")[0].childNodes[0].nodeValue;
        var url = xmlDoc.getElementsByTagName("ws-
url")[0].firstChild.nodeValue;
    }
    catch(e){
        if(async)
            errorcallback("Line 362" + e.name + "url in the web.xml file not
specified.", 602);
        else
            return errorcallback("Line 364" + e.name + "url in the web.xml
file not specified.", 602);
    }
    if(wsdl_url!=null && url!=null){
        if(async)
            SOAPClient._loadWsdl(url, method, parameters, async, callback,
errorcallback, wsdl_url);
        else
            return SOAPClient._loadWsdl(url, method, parameters, async,
callback, errorcallback, wsdl_url);
    }
}

SOAPClient._loadWsdl = function(url, method, parameters, async, callback,
errorcallback, wsdl_url)
{
    SOAPClient._wsdl = wsdl_cache[wsdl_url];
    if(SOAPClient._wsdl + "" != "" && SOAPClient._wsdl + "" != "undefined"){
    var ns = (SOAPClient._wsdl.documentElement.attributes["targetNamespace"]
```

```
+ "" == "undefined") ?
SOAPClient._wsdl.documentElement.attributes.getNamedItem("targetNamespace").
nodeValue :
SOAPClient._wsdl.documentElement.attributes["targetNamespace"].value;
    var paramBody = SOAPClient._paramToXml(method, parameters, ns, async,
errorcallback);
    if (paramBody == null){
        return;
    }
        return SOAPClient._sendSoapRequest(url, method, paramBody, ns, async,
callback, errorcallback);
    }
    else{
    var xmlHttp = SOAPClient._getXmlHttp(async,errorcallback);
    SOAPClient._xmlhttp = xmlHttp;
    xmlHttp.open("GET", wsdl_url, async);
    if (SOAPClient._timeout == null) {
        SOAPClient._timeout = window.setTimeout(SOAPClient._getTimeoutError,
SOAPClient._delay,wsdl_url);
    }
    if (async) {
        xmlHttp.onreadystatechange = function()
        {
            if(xmlHttp.readyState == 4){
                SOAPClient._readWsdl(url, method, parameters, async,
callback, errorcallback, xmlHttp,wsdl_url);
            }
        }
        xmlHttp.send(null);
    } else if (!async){
        xmlHttp.send(null);
        return SOAPClient._readWsdl(url, method, parameters, async, callback,
errorcallback, xmlHttp);
    }
    }
}

SOAPClient._readWsdl = function(url, method, parameters, async, callback,
errorcallback, req,wsdl_url)
{
    try {
        var httpstatus = req.status;
        var httpstatusText = req.statusText;
        if (httpstatus != null){
            window.clearTimeout(SOAPClient._timeout);
            SOAPClient._timeout = null;
            SOAPClient._xmlhttp = null;
        }
        if (httpstatus == 200 || httpstatus == 202) {
            SOAPClient._wsdl = req.responseXML;
            wsdl_cache[wsdl_url] = SOAPClient._wsdl;
        var ns =
(SOAPClient._wsdl.documentElement.attributes["targetNamespace"] + "" ==
"undefined") ?
SOAPClient._wsdl.documentElement.attributes.getNamedItem("targetNamespace").
nodeValue :
SOAPClient._wsdl.documentElement.attributes["targetNamespace"].value;
```

```
        var paramBody = SOAPClient._paramToXml(method, parameters, ns, async,
errorcallback);
        if (paramBody == null){
           return;
        }
            return SOAPClient._sendSoapRequest(url, method, paramBody, ns,
async, callback, errorcallback);
        }else{
            throw new Error("Line 495: HTTP " + httpstatus + " , " +
httpstatusText +  "\n Server connection has failed.");
        }
    }catch (e) {
        return errorcallback(e.message,httpstatus);
    }
}

SOAPClient._sendSoapRequest = function(url, method, paramBody, ns, async,
callback, errorcallback)
{
    var sr =
    "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
    "<soap:Envelope " +
    "xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" " +
    "xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\" " +
    "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">" +
    "<soap:Body>" +
    "<" + method + " xmlns=\"" + ns + "\">" +
    paramBody +
    "</" + method + "></soap:Body></soap:Envelope>";
    var xmlHttp = SOAPClient._getXmlHttp(async,errorcallback);
    SOAPClient._xmlhttp = xmlHttp;
    xmlHttp.open("POST", url, async);
    if (SOAPClient._timeout == null) {
        SOAPClient._timeout = window.setTimeout(SOAPClient._getTimeoutError,
SOAPClient._delay);
    }
    var soapaction = ((ns.lastIndexOf("/") != ns.length - 1) ? ns + "/" : ns)
+ method;
    xmlHttp.setRequestHeader("SOAPAction", soapaction);
    xmlHttp.setRequestHeader("Content-Type", "text/xml; charset=utf-8");
    if (async) {
        xmlHttp.onreadystatechange = function()
        {
            if(xmlHttp.readyState == 4){
                SOAPClient._getSoapResponse(method, async, callback,
errorcallback, xmlHttp);
            }
        }
        xmlHttp.send(sr);
    } else if (!async){
        xmlHttp.send(sr);
        return SOAPClient._getSoapResponse(method, async, callback,
errorcallback, xmlHttp);
    }
}

SOAPClient._getSoapResponse = function(method, async, callback,
```

```
errorcallback, req)
{
    var httpstatus = req.status;
    var httpstatusText = req.statusText;
    try {
        if (httpstatus != null){
            window.clearTimeout(SOAPClient._timeout);
            SOAPClient._timeout = null;
            SOAPClient._xmlhttp = null;
        }
        if (httpstatus != 200 && httpstatus != 202 && httpstatus != 500) {
            throw new Error("Line 570: HTTP " + httpstatus + " , " +
httpstatusText +  "\n Server connection has failed.");
        }
    }catch (e) {
        if(async)
            errorcallback(e.message,httpstatus);
        else
            return errorcallback(e.message,httpstatus);
        return;
    }

    var o = null;
    var nd = SOAPClient._getElementsByTagName(req.responseXML, "tns:results",
async, errorcallback);
    if (nd.length == 0){
        nd = SOAPClient._getElementsByTagName(req.responseXML, method +
"Result", async, errorcallback);
    }
    if(nd.length == 0)
    {
        if(req.responseXML.getElementsByTagName("faultcode").length > 0){
            try{
                var faultCode =
req.responseXML.getElementsByTagName("faultcode")[0].childNodes[0].nodeValue;
                var faultString =
req.responseXML.getElementsByTagName("faultstring")[0].childNodes[0].nodeVal
ue;
                if(req.responseXML.getElementsByTagName("detail")[0] !=
null){
                    if
(req.responseXML.getElementsByTagName("detail")[0].hasChildNodes()){
                        var detail =
req.responseXML.getElementsByTagName("detail")[0].childNodes[0].childNodes[0
].nodeValue;
                    }
                }
                throw new Error(500, "Line 595: faultcode: " + faultCode +
"\n"+ "faultstring: " + faultString+ "\n"+ "detail: " + detail);
            }
            catch (e) {
                if(async)
                    errorcallback(e.message,500);
                else
                    return errorcallback(e.message,500);
            }
        }
```

```
    }else{
        o = SOAPClient._soapresult2object(nd[0]);
    }
    if(async)
        SOAPClient._generateCallback(o, req.responseXML, async, callback,
errorcallback);
    if(!async)
        return SOAPClient._generateCallback(o, req.responseXML, async,
callback, errorcallback);
}
```

Appendix B: Source code of CMS

There gives an example of the page which is listing all the students in the course in CMS.

The database functions are:

```
create function listStudents(integer col, charstring order, charstring cname)
-> vector students
as select sortbagby((select studentInfoPublic(s)
                     from student s
                     where coursename(s) = cname)
                    ,col,order);
```

The client html page is:

```
<html>
    <head>
        <title>List of students</title>
        <script type="text/javascript" src="scripts/utility.js"></script>
        <script type="text/javascript" src="scripts/SOAPClient.js"></script>
        <script type="text/javascript" language="javascript">
            var error;
            var oMyObject = window.dialogArguments;
            var cname = oMyObject.selectedCourse;

            function listStudent(){
                error = null;
                SOAPClient.invoke("LISTSTUDENTS",
                                  [3,"inc",cname],
                                  true,
                                  listStudent_callBack,
                                  error_callBack);
            }

            function listStudent_callBack(o, xmlDoc)
            {
                    Var tblHead =
                   "<tr style='background-
color:lightgrey'><th>Name</th><th>e-mail</th><th>Group No.</th></tr>";
                    var table = resultToTable(o,null,tblHead);
                    document.getElementById("listDiv").innerHTML = table;
            }

            function error_callBack(e, errorCode){
                if (e != null){
                    error = e;
                    document.getElementById("listDiv").style.display='none';
                    document.getElementById("errorDiv").innerHTML="Error
callBack:<br>" + error + "<br><br>Error code:<br>" + errorCode;
                }
            }

            function close_onclick(){
                window.close();
```

```
            }
        </script>
    </head>
    <body onload="return listStudent();">
        <h2>The students currently registered for the course are:</h2>
        <div id="listDiv" style="display:block;"></div>
        <div id="errorDiv"></div>
        <div align="center"><input type="button" value="close"
align="center" onclick="return close_onclick();"></div>
    </body>
</html>
```