ECOLE POLYTECHNIQUE
PROMOTION X2000
TAROPA Emanuel

RAPPORT DE STAGE D"OPTION SCIENTIFIQUE

Services Continus en Active XML

NON CONFIDENTIEL

Option: Département INFORMATIQUE
Champ de l'option: Majeure d'Informatique
Directeur de l'option: Monsieur Jean-Marc STEYAERT
Directeurs de stage: Monsieur Serge Abiteboul
Adresse de l'organisme : INRIA Futurs
91400 Parc Club Scientifique ORSAY, France

Abstract

During my internship at INRIA I have worked on continuous web services in the context of Active XML, a language used for peer-to-peer data integration using web services. I have defined a declarative language that permits to specify continuous services and I gave an implementation that allows us to use them in Active XML.

This report is formed by 4 chapters: the first one is an introduction presenting the context of information exchange and subscription services. The second chapter describes the technologies that form the Active XML's framework. My work on continuous services in Active XML and their related topics is present in the third chapter. The last chapter is a conclusion.

Pendant mon stage d'option scientifique à l'INRIA j'ai travaillé sur les services web continus dans le contexte d'Active XML, un language utilisé pour l'intégration pair-á-pair des données au moyen de services web. J'ai défini un language déclaratif permettant de spécifier des services continus et j'ai réalisé une implementation permettant de les utiliser dans Active XML.

Ce rapport comporte 4 chapitres : le premier est une introduction qui présente le contexte d'échange d'informations consideré et les services de souscriptions. Le deuxième décrit le technologies utilisées et le language Active XML. Ma contribution sur les services continus en Active XML et sur des sujets liés à eux est présentée dans le troisième chapitre. Le dernier chapitre contient mes conclusions.

Contents

Chapter 1

Introduction

1.1 Context

We are living in a world which is in continuous motion. Keeping this simple truth in mind prevents us from being amazed by the rapidity of changes occurring in our environment and helps us keep an open perspective on all possible evolutions. The keys for synchronism between us and our surrounding universe are the adaptability, expressed by the ability of identifying and understanding new emerging configurations, and the communication which allows us to add our results to the general knowledge base and gives us the possibility to probe the actual limits of our world. As the general tendency is to move from disparate to integrated, from isolated to interconnected, our need for effective communication and reliable information management grows.

1.2 Peer-to-Peer vs. Client-Server

In a centralized system endpoints are divided into servers and clients. This architecture is good for closed environments with simple relations between endpoints but it is unfit to express the complex interactions that we have in today's web applications. The main limitation comes from having only one role per endpoint (either client or server), thus the entire system being not extensible enough to model new relations starting from the existent ones.

In order to be able to proper model these interactions we need a new scheme, more flexible, where each endpoint has the ability to provide and consume services in an autonomous manner. This is best modeled by a peer to peer architecture.

We maintain the concept of service provider and requester but we change slightly the demarcation between them. By allowing multiple services on the same endpoint we can better model interactions between the different parts of a distributed application. Shifting the perspective from the centralized universe to a distributed one we enhance the overall functionality of the entire system while reducing its load.

1.3 Subscription Services

Although necessary, the above change of perspective does not offer us the entire power of expression we need, because we still find ourselves limited by the nature of the communication. Frequently a web service is a synonym with the retrieval of dynamic information. The usual scenario is that a client connects to a server, makes a request and obtains a response. This is a scenario of information ”pulling”. Although a pull only architecture is quite powerful, it turns out to be completely un-practical in some cases, as for example, in managing information subscriptions (i.e. a client wants to be updated when data changes).

The *pull paradigm is based on synchronicity*, relying heavily on request-response protocols and thus being unable to express asynchronous events. The usage of web services with unpredictable execution time is possible only by changing the communication between services.

Asynchronous behavior can be modeled by defining a new class of web services, called subscription services. These services define a *push architecture*, this time the server being the one that initiates the data transfer between itself and its subscribers.

In the following chapters I will describe the continuous services which basically are subscription services that periodically send results to their subscribers.

Chapter 2

Technical Context

In the followings we will define the basic notions that will be extensively used in the next chapters, establishing the framework used for developing continuous services.

2.1 XML

XML is the acronym for Extensible Markup Language. It is based on the International Standard for structured information (SGML). Another known language based on SGML is HTML (Hypertext Markup Language). All the implementations of SGML are text based languages that use tags to specify/define their structure. This makes them human readable and easy understandable. XML could be seen as HTML's brother in the respect that they are both applications of SGML but some essential differences make them complementary. The main difference is in their goal : HTML is designed to describe how data should look while XML stands for data structure. While HTML has a fixed set of tags, XML allows you to define your own description of your data, in such a way that it makes sense to your application. This gives us the essential difference between the two languages: HTML is display oriented while XML is a human readable representation of data, thus better suited for machine processing.

2.1.1 A Simple Example

A quick view of an XML document is:

```
<address>
  <street>Rue de la Pacaterie</street>
  <batiment>499</batiment>
  <chambre>473</chambre>
  <city>Orsay</city>
  <region>Ile-de-France</region>
```

```
    <country>France</country>
    <code>91400</code>
  </address>
```

The above document contains structured information about an address. We can see the elements clearly individualized by their tags which resume their content.

2.1.2 Main Characteristics

Structure

Before XML, data exchange on the web was done through HTML. The distant server usually had an HTML page through which it proposed it's services. The client's browser displayed this page and the client was free to examine the data it contained. For any dialog with the server, the request was submitted back to the server which did the processing and then send the results back to the client. A major speed-up to the entire process would have been possible if the client had the ability to perform simple actions on the data that it received instead of sending other requests to the service provider. This couldn't be done because HTML is inflexible and thus the client had no possibility of understanding the data's structure. All what he was able to do was to display it, conforming to directives encoded in the server's HTML response.

For instance, lets consider the following situation: we have a service provider that offers the telephone numbers of people living in Paris. Lets take now a client, named Bob, that wishes to call his friend Jean. The first thing he will do might be typing in Jean for a surname and wait for results. As expected, the list of phone numbers that are assigned to all Jeans in Paris will be quite big. When Bob will get this list he will want to refine his search criteria. So he will fill in his friend's family name: Valois and send the request back to the service provider. And so on, until Bob would have reduced the results list to a reasonable size that he will be able to scan himself. What is wrong with this scenario is that each refining of the search requires the service provider to perform an action and causes redundant data to be sent over the network.

If the client would be able to refine himself the initial list, the exchange of data between the two machines will be reduced only to the initial one. The gain is obvious: the server does not have to perform the same operation over and over again and the network overhead is drastically decreased by sending only once the results. We will also have a decrease in the server's usage and therefore it will be able to service more clients in the same time interval. This is what XML gives us: data structure. The client receives XML information that he is able to interpret for himself, with the aid of a simple XML parser. If Bob does not own a program for filtering the results(ie: a program that allows him to specify the family name, the address, etc.) the server could provide one for him and attached it to the initial message. This program can be a Java applet that once loaded will help Bob locate his friend's telephone number.

Interoperability

The diversity of machines and programming languages that we use is no longer a problem for data communication. Inter platform and inter language data exchanges are possible by the use of XML which is a standard [12] for information representation. Being text and using unicode as encoding, we are sure that it can be parsed and interpreted on any kind of machine while using any alphabet we want. The only thing that communicating applications need to know is what data structure to expect, which can be easily described through two mechanisms : DTD (Document Type Definition) and most recently, XML Schema. They allow to determine if an XML document is valid or not and most of the XML parsers support them. This type system is very useful because it gives us the means of specifying our own document format.

Transformability

Write once and display in infinite ways could be a short characterization of XML's transformability. The idea is that you can display the same content in multiple ways. This is accomplished through the use of XSL (Extensible Stylesheet Language), a language that controls how elements from our document are displayed. The main gain is that everyone can choose a different way of displaying the same content only through a simple re-writing of the XSL file.

Easy Querying

Having a hierarchical structure, XML Documents can easily be represented as trees. There are multiple ways of traversing these trees and they all rely on algorithms with strong theoretical background. All the operations that will be performed on the tree representation will be reflected in the XML document that describes it.

2.2 Web Services(WS)

The official definition of WSs' given in [13] states that: *a web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the WS in a* *manner prescribed by its definition, using XML based messages conveyed by internet protocols.*

A WS is an abstract notion denoting a specific functionality. A WS can have multiple implementations in different programming languages, thus being deployed on different platforms. In the remainders the agent that offers a concrete implementation of a web service is called *service provider* and the service's requester is called *client*.

2.2.1 Composability

Seen as concrete objects, web services are build following the component-based model's principles. This model expresses the composition of existing objects into new ones with complex functionality. By reusing available functionalities we can better concentrate on the essence of the new object and not on its implementation details.

2.2.2 Technology Stack

Web services are not implemented in a monolithic way but they rather represent a collection of related technologies.

Remote Procedure Call

At the simplest level we can see a WS as an interface that allows two applications on different endpoints interact. We model this interaction by performing remote procedure calls (RPCs). A remote procedure call means invoking one application's methods from distance. The data is exchanged in XML format and the transport protocol is HTTP. The main advantages are

- we do not need to design our own protocol of communication because it is already provided by HTTP
- exchanging data in XML format over an HTTP connection means little overhead on both participating endpoints

Simple Object Access Protocol (SOAP)

SOAP is a simple protocol for data transport. We give the definition of SOAP from [14]:*SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses.*

Our framework uses SOAP for data exchange between WSs. All data is converted into SOAP messages before being sent over the HTTP connection. In case we had errors when sending/receiving the data, we will find them in the SOAP response message, encoded in special error headers. Thus we have a reliable communication system with error detection using a simple HTTP connection.

Web Services Definition Language (WSDL)

WSDL has an XML representation, allowing us to describe a web service, specifying its methods' signatures. Its complete definition is given in [15]:*WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate.*

2.3 Active XML

Active XML is a language used for transparent data integration.

2.3.1 Context

With XML and Web services, seamless access to heterogeneous distributed information on the Web becomes feasible. This is crucial for many applications e.g. in the context of B2B, B2C or B2G, in particular those who perform data integration using Web information. Let us therefore consider data integration in more detail, and highlight some of the key benefits of the AXML approach for it.

2.3.2 Data Integration

Typically, the goal is to integrate information provided by a number autonomous, heterogeneous sources and to be able to query it uniformly. One usually distinguishes between two main alternatives in a data integration scenario: warehousing vs. mediation. The former consists in replication the data of interest from the external sources, and working on this data locally, whereas the latter relies on query rewriting mechanisms to fetch just the necessary data from the sources at query time.

AXML allows to introduce flexibility in various ways. First, it assumes a peer-to-peer architecture, so in particular, the integrator needs not be one particular server, i.e. many peers may participate to this task. Second, it allows to follow a hybrid path between mediation and warehousing. More precisely, it allows to warehouse only part of the information. Finally, it considers Web services as first class components of the system, hence any new Web resource that uses Web services standards may be discovered and integrated. Thus, in some sense, AXML also allows "service integration". It should however be observed that AXML is not a technique for data integration (such as e.g., Information Manifold), but a language and a system to facilitate data integration.

2.3.3 Distributed Data

Beyond data integration, the ambition of AXML is to facilitate the deployment of distributed applications based on distributed data sharing at large. It is thus a relevant technology for a wide range of applications such as comparative shopping or cooperative editing.

The essence of AXML is very simple. An *AXML document* is an XML document with embedded Web service calls. More precisely, it is a valid XML document (so it may benefit from all existing software tools for XML) where some particular elements (the ones labeled *sc* are interpreted as service calls. The presence of these elements make the document intensional, since some of the data of is given explicitly, whereas for some of it, a definition (i.e. the means to acquire it when needed) is given. AXML documents may also be seen as dynamic. The same service called twice may give a different answer (e.g., because the external data source changed), so the same document at different time will have a different semantics, possibly reflecting world changes.

We name a system responsible for storing and managing *AXML documents* an *AXML peer*.

2.3.4 Behavior

An *AXML peer* has a double role, acting as a:

- client because activations of the calls inside documents use Web services provided by other systems.

- server in the sense that it provides querying facilities on its repository of documents. These are exposed as Web services.

When a service call is activated, the data it returns is inserted in the document. Therefore, documents evolve in time as a result of service calls. This process of *materializing* some calls plays a crucial role in our approach.

The above description of Active XML is just an overview of the language. For better understanding we refer the reader to: [4], [5], [6], [2].

2.4 Technical Environment

In the followings we will describe only the relevant parts for continuous services of the project's technical environment, avoiding to present all the products used by Active XML's framework.

2.4.1 Programming Language

As a programming language we used Sun Microsystem's Java language. This is a high-level language that is: simple, architecture neutral, object oriented, portable, distributed, performant, interpreted, multi-threaded, robust, dynamic and secure. We choose to use Java because it modeled very well our intentions of building a portable, web service oriented application in a cost-free environment. Besides its high functionality what we used most in developing continuous services were the packages that offered an implementation of RPC.

2.4.2 Web Server

As web server we decided to use Apache's Tomcat Server. As defined in [17] Tomcat is *the servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies. The Java Servlet and JavaServer Pages specifications are developed by Sun under the Java Community Process.*

A servlet container is a runtime shell that manages and invokes servlets on behalf of users. Being one of the most performant web servers available and being developed in Java, Tomcat was a natural choice as a web server in our framework.

2.4.3 SOAP Engine

As SOAP provider we decided to use Apache's Axis. As defined in [16] Axis is : *essentially a SOAP engine (a framework for constructing SOAP processors such as clients, servers, gateways, etc).*

The current version of Axis being written in Java made it the perfect choice for our programming environment. Axis' main features are:

- speed - Axis uses SAX (event-based) parsing to achieve significantly greater speed than earlier versions of Apache SOAP.
- flexibility - The Axis architecture gives the developer complete freedom to insert extensions into the engine for custom header processing, system management, or anything else you can imagine.
- stability - Axis defines a set of published interfaces which change relatively slowly compared to the rest of Axis.

- component-oriented deployment - Reusable networks of Handlers permit to implement common patterns of processing for user's applications, or to distribute to user's partners.
- transport framework - Axis has a clean and simple abstraction for designing transports (i.e. senders and listeners for SOAP over various protocols such as SMTP, FTP, message-oriented middle-ware, etc), and the core of the engine is completely transport-independent.
- WSDL support - Axis supports the Web Service Description Language, version 1.1, which allows an easy building of stubs to access remote services, and also to automatically export machine-readable descriptions of deployed services from Axis.

Chapter 3

Continuous Services in Active XML

3.1 General View of the Architecture

A subscription is a special service call that is fired only once and that provokes periodical answers from the invoked continuous service. Keeping a simplified view of the subject, we can graphically model the interactions that occur between a subscriber and a continuous service provider as in Figure 3.1

Continuous services allow servers to push information to their clients that will integrate it into their applications. By adding support for continuous services in Active XML we added a push feature its architecture. This means that at a given moment, the server can open a connection with a client and send it the expected data. Having a distributed peer-to-peer architecture in Active XML made modeling this asynchronous behavior easy, in spite of the synchronous communication protocol between the different peers that we use. By defining multiple services on the same peer we make the clients act as servers, thus having multiple roles on the same endpoint. The same functionality would have been much harder to obtain in a traditional client-server architecture where the roles played by the endpoints are fixed and the extensibility of the system is reduced.

The general behavior of continuous services in Active XML is shown in Figure 3.1 where we have a service running on the client's peer, called Callback, that integrates the data received from the server in the client's application. Callback defines the aforementioned change in the roles played by the two peers: the service provider becomes a client of its client's Callback service.

The behavior depicted in Figure 3.1 can be expressed by the following steps:

1. The client sends a subscription request to the service provider which

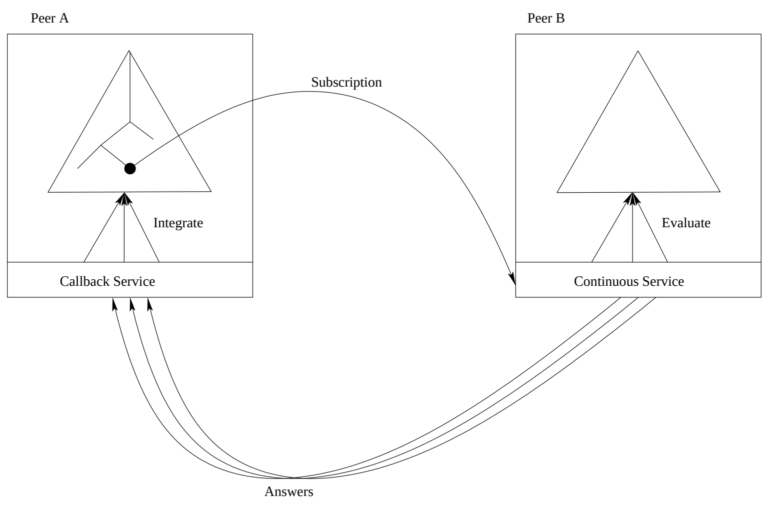

Figure 3.1: General Architecture View

registers it.

2. Periodically, the service provider will do the following actions:
   - Evaluate the subscription.
   - Call a possible post-processing service(if it is specified to do so in the subscription).
   - Call back the client sending it the results (perform the callback).
3. The client, in its turn will periodically integrate the results he receives form the service provider to which he subscribed.

The advantages of this model are obvious if we try to obtain the same behavior with a pull only architecture. The client would have to "poll" the server by firing the same service call over and over again, as often as we wish to receive our results. This means sending the same information over and over on the network, generating an overhead at network level. It also means that we have a good chance of asking a result from the server when the data is not available for the clients, thus introducing an extra overhead at peer level by forcing the peer to evaluate calls that it cannot respond to.

An even more important limitation that we overcome using a push architecture is the one given by the timeout of our synchronous connections. This means that if we are not using an open connection for a period of time greater or equal to the maximum allowed inactivity time interval, the connection will be automatically closed. It is obvious that this reduces our capacity of using(or integrating) in our application services that have a non-determined (or extremely large) execution time or services that give partial results. While a pull system relies on open connections to retrieve data from the service provider, a push one needs an open connection only for sending the subscription and every time when the service's result is available for the client. The rentability of a push architecture is evident if we consider the following simple example: lets say that we are stock broker and that we need monthly forecasts on stocks' prices. We're using a service offered by New York Stock Exchange (NYSE) that takes 20 minutes to calculate the forecast for a given month. Lets suppose that we're connecting to it using our mobile phone. In a pull only architecture this means that we must keep the connection open for 20 minutes, which is quite costly, while in a push system we can connect only as long as it takes to send the subscription and then we can disconnect and the service will send us the result when it will be available.

Our main goal was to make the entire subscription process transparent for the client and the definition of a continuous service as intuitive as possible for the service provider, while maintaining a high configurability.

Further on we will go into detail and analyze how things happen on server and client side. We will describe their behavior and give concrete examples.

3.2 Server Side Behavior

One of the major features we introduced in Active XML is the possibility to take any existing service and make it continuous. We can see the process of turning a normal service into a continuous one as a "wrapping" of that service in a continuous envelope.

The basic behavior on server side can be expressed by the following steps, which can be repeated several times:

- Call the wrapped service
- Retrieve its result
- Send the result back to the client's Callback service

This describes the basic process of dealing with a subscription.

3.2.1 Server Side Post-processing

We will denote by the general term of post-processing any computation performed on by a dedicated service.

Above we have presented the general route that data takes on the server's side. Having a modulable architecture allows us to plug-in services that perform post-processing on data before we send it to the client's Callback service. An example of post-processing that we can perform on server side is a service that returns the differences between a document and its prior version. We will name from now on this service "Diff", calling the function that gives us the differences between two versions of the same document doDiff. A scenario that shows its use could be the following one: lets say that we are NYSE and we offer a continuous service that sends every 10 minutes stock quotes. If we could send our clients only the quotes that changed since the last transmission we would reduce the overhead at network and peer level by transmitting a significantly smaller amount of data through the system. To do this, we use the aforementioned doDiff method on the set of actions, comparing it with the previous one. Then we will do the callback with this data. The client will then integrate the data as we will explain later.

3.2.2 Restrictions

Once we offer continuous services on our peer we must be able to control the load that they will bring to our system.

Response Frequency

The load that a continuous service introduces on the peer where it is defined is provoked mainly by the frequency of calls towards the wrapped service. It is therefore a necessity to be able to impose a maximum allowed value for this frequency, thus making sure that we will not exceed our peer's resources and that we will have a reasonable tradeoff between the number of clients that we will accept and the peer's performance. Through this maximum frequency we can express constraints that are linked with the wrapped service's functioning (e.g. there is no point in doing multiple calls to the wrapped service before it has the chance to execute once). Besides a maximum frequency we should also be able to define a default one, that we will use in modeling a default behavior of our continuous service at callback level.

Subscription Control

In some applications we need to perform a finer control of incoming subscriptions. Therefore, we need an optional set of parameters that allow us to impose subscription related restrictions.

In order to better control the load that a continuous service brings on the current peer we need to limit the number of subscriptions that we will accept. Imposing a maximum number gives us an idea about the maximum load that the respective service will bring to the peer.

Another part of the load introduced by a continuous service on its peer is given by the saving of missed results for registered subscriptions. In order to avoid saving results for a client that is almost never present we could limit the time interval for saving consecutive results.

Even if a client is always connected and the callback is always successful, we might want to limit its subscription validity. The importance of this feature is well shown by the following example: NYSE offers free access to the previously mentioned stock quoting service for a period of one week and after that the clients that would like to continue using the service must pay for their subscription.

Ongoing Work

Ongoing work is directed towards a dynamic load analysis on the local peer. The system that we are designing will be able to dynamically configure the peer (e.g. it will dynamically establish the response frequency for a continuous service, decrease or increase the number of its subscribers a.s.o.) using in an efficient manner available resources.

3.2.3 Error Handling

The subscription schema described in Section 3.1 does not consider the errors that might appear in the system's run. Because of various problems that we might encounter in different situations we have to define a fallback sequence that will make our system reliable. The main challenge that arises is to define a behavior for the case when the client's Callback service is not reachable (ie network error, client is not on line, a.s.o.). This behavior will depend on the parameters that are specified by the client when it makes the subscription and on the service's specific parameters and constraints.

Adding error recovery to the schema presented in Figure 3.1 we can now express server's behavior by two distinct actions: callback and fallback, their sequencing being:

1. Try to perform callback
2. If this fails, do fallback

The callback is performed as it was previously described and as a fallback strategy we propose the following one:

1. Perform forwarding - The client has the possibility of providing a list of endpoints where he wishes to have his results sent in case he is not reachable. The first step in the fallback sequence is to forward the information to the specified forwarding locations (i.e. we will make a call for each given endpoint until we have successfully contacted one of the peers or we have finished the list).
2. If forwarding was successful, then fallback is over.
3. If not, then we save the result locally and we will send it when the client is reachable again. We have a dedicated service that does the

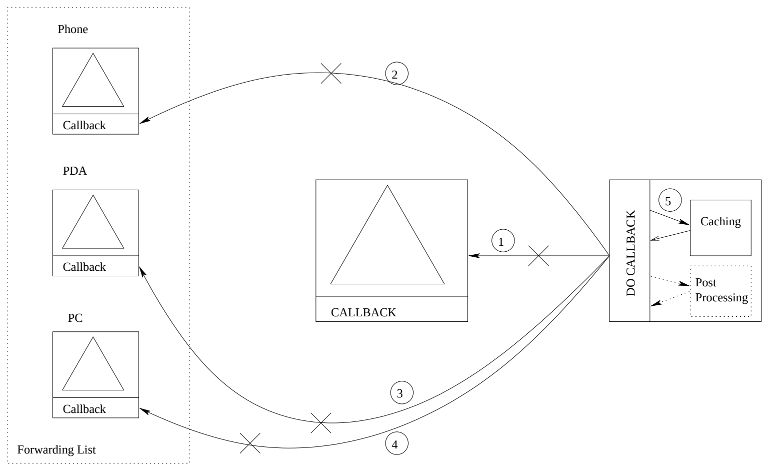

Figure 3.2: Server Side Behavior

saving of missed results. We will call it the "caching" service and we will discuss it more extensively in Section 3.5. The client is able to specify what results he wishes to receive from the cached ones when he will be reachable again.

This strategy is a tradeoff between the overhead on service provider's peer and reliability, other strategies being imaginable under different system requirements. A graphical example of the fallback sequence is offered by Figure 3.2 The sample behavior illustrated in Figure 3.2 passes through the following states:

1. First we try to call back the client. This call fails because of an error that occurred when sending information to the client (e.g. a network error)

2. We try then to call the endpoints provided by the client in the forwarding list. We call the Phone Peer and we have no success

3. We call the PDA peer without success

4. We call the PC peer without success

5. The forwarding failed because we didn't manage to successfully call any of the provided locations. We move to the next step and we do the caching of missed result

Our particular fallback strategy can be represented by the automaton shown in Figure 3.3 Below we give the evolution that models the fallback example provided in 3.2.

We identify the initial state of the system with the moment of receiving a subscription. Then we call the wrapped service. If the call to the wrapped

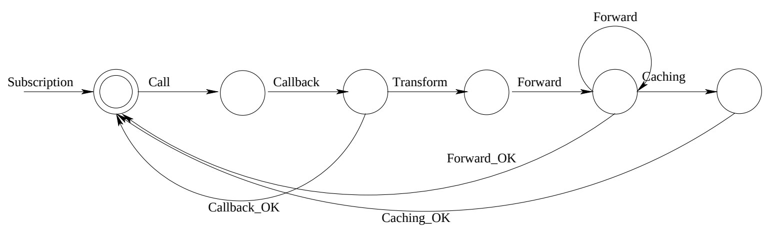

Figure 3.3: Server Behavior

service goes well we proceed forward and call the post processing service(if so specified in the subscription) and do the callback. If all goes well, we try to do the callback. If the callback goes well, we'll repeat the sequence again after the specified time interval has elapsed. If the callback does not go well, we proceed to fallback. We first try to forward the result to the specified list of endpoints until we succeed or we have no more peers to call. If forwarding fails, we do caching (if so specified) of the result and we start all over again.

3.3 Client Side Behavior

After subscribing to the continuous service, the client will start receiving periodical results from the respective service. The entry point on the client's peer for the data sent by the server is the Callback service. It is this service that will deal with the integration of received data in the client's application. We have defined the integration process according to Active XML's principle of merging data obtained from a service call next to the node that fired the respective call. We obtain the respective node using the service call's id, received as parameter by the Callback service. The result of the service will be inserted as service call's node next sibling and the document will be then updated by the rules provided in the service call's definition.

The client side effects of a subscription to a continuous service can be illustrated by the following example: lets consider that we do a subscription call to a continuous service called "ContinuousInLineGetStockPrice". This continuous service returns an element representing quoting information about a certain company, identified by its NYSE index. The result returned by the "ContinuousInLineGetStockPrice" has the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<company>
  <openingprice>34.49USD</openingprice>
  <closingprice>34.86USD</closingprice>
  <index>MSFT</index>
</company>
```

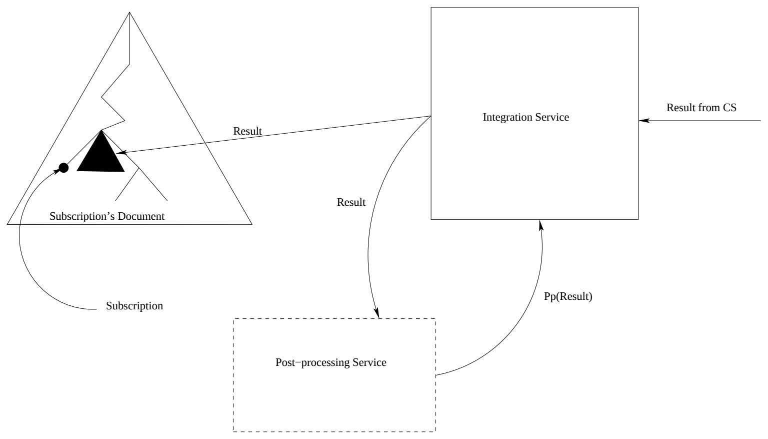

Figure 3.4: Client Side Bahavior

After receiving a couple of results from the invoked service, the document that fired the subscription call looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<test_compil_inline_cont...>
<!- Subscription call -->
<company>
  <openingprice>34.49USD</openingprice>
  <closingprice>34.86USD</closingprice>
  <index>MSFT</index>
  <date>Mon 29 Jun 2003 05:01:02 PM CEST</date>
</company>
<company>
  <openingprice>37.13USD</openingprice>
  <closingprice>37.23USD</closingprice>
  <index>MSFT</index>
  <date>Mon 30 Jun 2003 05:01:12 PM CEST</date>
</company>
</test_compil_inline_cont>
```

The received results are integrated as siblings of the node that contains the subscription.

3.3.1 Client Side Post-processing

The steps described above present the general route that information follows on the client side, from Callback service to the subscription's document. Having a modular architecture, we are able to plug-in between the Callback service and the moment of insertion in the document one or more dedicated services that perform post-processing on the received data.

To better grasp what client side post-processing is we could consider the following example: lets say that we use the NYSE stock quoting service defined in the previous section. This service sends us only the companies that have changed stock price since its last transmission. A client displaying the entire list of companies and their quotings subscribes to this service. If this client were to integrate directly the received results in his application, it would display information only about a few companies that changed quotes. It is obvious that we need some kind of received information's processing on client side in order to retrieve the entire list of companies, updated by the last minute changes, needed by the client. In order to perform the desired processing of data we use a dedicated service (the Diff) that is able to reconstruct a document starting from a given set of differences between the document and one of its prior versions and the prior version itself. We will call the method that does the aforementioned reconstruction applyDelta. This method will be used by the Callback service once new data is available and will return to the Callback service the new version of the document that will be further integrated in the subscription's document.

3.3.2 Server Directives

A number of directives can be passed by the client to the server in order to control the server side subscription behavior.

Desired Time Interval

The first thing we saw fit to define was a desired time interval(DTI) for receiving the results from the continuous service. The use of this parameter is evident if we consider the following example: lets say we are a financial site that displays stock quoting information three times a day. We are using a stock quoting service provided by NYSE which has the default frequency of response of 10 minutes. It is obvious that we do not need updates on stock quotes every 10 minutes so we would like to be able to specify to the service provider that we would like our results sent only three times a day, thus avoiding an overhead at network level.

Reducing the frequency of receiving results from a service might be the most straightforward application of the DTI but it is not the only one. A more subtle utility of this parameter could be to increase a service's response frequency. Lets take the reverse of the previous example and assume this time that we need results every ten minutes and NYSE's service usually sends results only three times a day. Then, by specifying a DTI of 10 minutes we inform NYSE's service that we have a subscription that needs special treatment. Depending on server provider's configuration, it can send us results at the required time interval or it can tell us that it is not able to do with such high frequency of pushing data. In the first case we will receive results with our desired frequency and in the second we can start looking for another service provider that is able to offer us the same data at our desired rate.

Server Side Post-processing of the Results

Another usefull option that we made available to the client is the possibility of defining post-processing of the received results. The client could instruct the service provider to use a desired processing service (e.g. Diff) and to do the callback with the processed data (e.g. to send only the differences). For the moment we are using a default post-processing service, the Diff service but ongoing work is to dynamically use any post-processing service specified by the client.

Caching the Missed Results

As the default server side behavior is not to cache the missed results it might not be convenient for some applications. A simple example is the following one: lets say that we are a financial web site that does statistics on stock quotes evolution. For retrieving the stock quotes we use the previously mentioned NYSE stock quoting service. Lets assume now that for a reason or other we find ourselves in the impossibility of communicating for a while with the continuous service. As we need the evolution of stocks' prices for our statistics, we need to know what happened during the time we were not able to communicate with the NYSE's service. Thus, we need to instruct the service to save the results for us while we are not reachable and send them when callback resumes. For the moment we are using a default Caching service that is on the server's peer but ongoing work is to define and implement a generic one.

Forwarding List

As the default server side behavior is not to perform the fallback sequence when the callback cannot be completed this might not be convenient for some applications. In some cases, the forwarding part of the fallback sequence might be crucial for the application. A typical example would be the one of a business man that needs to take decisions rapidly according to stock price. Lets assume that at some point the network connection between his laptop and the NYSE's stock quoting service is broken but his mobile phone is working. It would be then very usefull to have the information sent to his phone, as plain text messages. It is now obvious that we need to offer to the client the possibility of specifying a list of endpoints where he would like his results forwarded in case of callback error.

3.4 Syntax Definition

When building web services we have to take a dual perspective, regarding the system on its two facets: the service provider's one and its client's one. Although it complicates the overall treatment of the problem, doing this way allows us to design a flexible architecture where the roles are well defined and the interactions between the endpoints are carefully modeled. Every time that we add functionality to one side we must also take into account the impact that this will have on the other side. The process of building a syntax, of defining continuous services is an iterative one, every newly discovered aspect being added to the system in order to render it as complete and reliable as possible.

3.4.1 Server Side Syntax

The syntax we defined for the server side integrates the main parameters that we have presented in Section 3.2.

As a matter of technical detail, we allow the user to define directly in the continuous service's definition the wrapped service that will be made continuous. This is very useful because we spare the client from the sordid details of writing a separate definition for the wrapped service and of publishing it.

To see how the syntax looks like we can give a straightforward example of a continuous service definition that wraps an inline defined service:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<serviceDefinition name="ContinuousInLineGetStockPrice"
  type="continuous">
  <coreServiceDefinition type="query">
    <parameters>
      <param name="NYSE_Symbol"/>
    </parameters>
    <query><![CDATA[
       select <company>
         c/openingprice,
         c/closingprice,
         c/index</>
       from c in NYSE//quotations/company
       where c/symbol/ = NYSE_Symbol;]]>
    </query>
  </coreServiceDefinition>
  <!-- Continuous Service Parameters -->
  <parameters>
    <param name="NYSE_Symbol"/>
  </parameters>
  <!-- Post Processing Service SOAP Params -->
  <transformService
    URL="http://localhost:8080/axml/servlet/AxisServlet"
    methodName="doDiff"
    nameSpace="SimpleDiffService"/>
  <!-- Caching Service SOAP Params -->
  <cachingService
    URL="http://localhost:8080/axm/servlet/AxisServlet"
    methodName="doCaching"
    nameSpace="SimpleCachingService"/>
  <!-- Frequency Settings -->
  <frequency default="60000" max="10000"/>
  <!-- Subscription Settings -->
  <subscription maxLifeTime="8640000"
    maxSuspensionTime="86400"
    maxClientsNo="15000"/>
</serviceDefinition>
```

The correspondence between the tags present in the above definition and the parameters defined in Server Behavior Section is given by the followings:

- coreServiceDefinition - definition of an inline service that is just a usual declarative service definition in Active XML
- parameters (exterior to coreServiceDefinition) - parameters received by the continuous service.
- transformService - SOAP parameters of the Post-processing service
- cachingService - SOAP parameters of the Caching service
- frequency - maximum and default frequency values
- subscription restrictions:
  - maxLifeTime - maximum subscription validity time
  - maxSuspensionTime - maximum subscription interruption time
  - maxClientsNo - maximum number of subscribers

Without changing too much the syntax for defining a normal service we are now able to declaratively define continuous services, adding expressivity to Active XML.

3.4.2 Client Side Syntax

On the client side we had to define a way to call a continuous service. We have established that we will use a subscription paradigm. The scenario is that the client subscribes to the desired continuous service which will send back periodical answers. It is then obvious that we have to define a set of parameters that allow us to call a service, i.e. to properly invoke it. We will call them SOAP parameters. The SOAP parameters of a service are:

1. Endpoint URL - the address of the peer where the respective service is deployed

2. Name Space - the name of the service

3. Method Name - which one of the service's methods to use

Every time that we make a service call we need these parameters' values. For continuous services, following the behaviors depicted in the above sections, we will use SOAP parameters for calling the:

- Continuous Service

- Post-Processing Service (both client and server side)

- Callback Service

- Caching Service

As the architecture we presented for continuous services is a basic one it is obvious that in real life applications we will customize it by adding services required by our application's logic. These services will also be called using their SOAP parameters, just as we did for the four services enumerated above.

After establishing a general frame for continuous services on client side we had to define a syntax that will allow us to integrate this frame in Active XML. Our main goal was to make the syntax transparent to the user and to realize all the functionality we needed while making a minimum number of changes to Active XML. We tried to offer both a default and an advanced way of making a subscription. In the default way subscribing requires minimum intervention from the client. A bit more complicated, the advanced way lets the client specify a desired server side behavior for his subscription.

The default behavior for a subscription on server side is to perform the callback using neither post-processing of the results nor fallback procedure. This means that if the callback does not succeed, the missed information will not be kept and the client will start receiving again results from the service when it will be reachable again. The data that is sent when callback is performed is directly the answer from the wrapped service without any processing added on server side.

When using the default way of subscribing to a service the only thing that needs to be done is to identify the service call as a subscription. The system will then add the SOAP parameters of the Callback service as well as parameters needed by the Callback service to perform data integration. An example of making a simple subscription could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<test_compil_inline_cont
  xmlns:axml="http://www-rocq.inria.fr/verso/AXML">
  <axml:sc callable="true" frequency="once"
    methodName="doSubscribe" mode="replace"
    serviceNameSpace="ContinuousInLineGetStockPrice"
    serviceURL="http://xhost:8080/axml/servlet/AxisServlet"
    subscription="true">
    <axml:params>
      <axml:param name="NYSE_Symbol">MSFT</axml:param>
    </axml:params>
  </axml:sc>
</test_compil_inline_cont>
```

As you can see, the only thing that differs in the subscription given above and a standard Active XML service call is the attribute *subscription* set to *true*. The rest of the subscription's attributes are standard in an Active XML service call. Their meaning is:

- SOAP parameters of the continuous service we wish to subscribe to
- service call's frequency set to once (a subscription being a service call that is fired only once
- way of inserting the result in the document containing the subscription

Although the default behavior for a subscription is used in most of the applications there are some cases when it might prove totally un-adapted to application's needs. We must then define a set of optional parameters that express the server directives given in the Client Behavior Section.

The following example shows how these parameters integrate into a service call:

```
<?xml version="1.0" encoding="UTF-8"?>
<test_compil_inline_cont
  xmlns:axml="http://www-rocq.inria.fr/verso/AXML">
  <axml:sc
    <!-Same parameters as for the default subscribing -->
    desiredFrequency="15000" useCaching="true" useDiff="true"
    forwardingList="URL1 URL2...URLn">
    <!-- Same continuous service parameters as above -->
  </axml:sc>
</test_compil_inline_cont>
```

Because we use a generic Caching service and a generic post-processing service on the server, we just need to specify whether we want to use them or not by setting the values for *useCaching* and *useDiff* to *true*. Ongoing work is done to restructure the service call element, moving most of its attributes as interior elements (the same way we are grouping Active XML parameters of the called service). We are also working on allowing the client to choose a Caching service and a post-processing one that are defined on the server. The interest in having multiple post-processing services and multiple Caching services on the service provider's peer is that we render the entire system more flexible, more adapted to client's needs. Taking into account all the modifications that are in progress, a future view of the same service call could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<test_compil_inline_cont
  xmlns:axml="http://www-rocq.inria.fr/verso/AXML">
  <axml:sc
    <!-Same parameters as for the default subscribing --> >
    <forwardingList>
      <peer>URL1</peer>
      <peer>URL2</peer>
      .................
  <peer>URLn</peer>
    <forwardingList>
    <serverDirectives>
      <defaultFrequency>15000</defaultFrequency>
      <useCaching>true</useCaching>
      <useDiff>true</useDiff>
    </serverDirectives>
    <!-- Same continuous service parameters as above -->
  </axml:sc>
</test_compil_inline_cont>
```

3.5 Caching Service Definition

Used in continuous services on the server side, the Caching service allows saving and later retrieving the results that the clients missed. Because we have a loose coupling between clients and servers in Active XML we have no prior possibility of anticipating when a client will become unreachable and for how long it will stay that way. It is only when we have a successful callback that we know the client is online. Thus the problem is designing a result sending system that has a low overhead at network and peer level and that is configurable by the clients and by the server.

3.5.1 Client Options

The clients should be able to specify what they want to receive from their cached results. The first tendency could be to send them the entire set of results they have missed but this might prove un-practical in some situations.

Time Span of Results (TSR)

A first parameter that seemed appropriate to define was the *time span of desired results*. This means that a client can specify to receive only the results that are within the respective interval from the current date. The importance of being able to choose the time interval for interesting results is clearly shown if we consider our previous example with the NYSE stock quoting service. The frequency for this service is 10 minutes. Lets say that we have a client that does two-week statistics regarding stock price evolution and that he missed his results for 3 months. It is obvious that he would be interested only by the results corresponding to the last two weeks and not by their whole set.

Caching Mode

The client should be allowed to choose a Caching Mode, depending on his application. For instance we might have a client that wishes to cache only the last result or one that wishes to cache a result every N failed callbacks.

3.5.2 Server Options

On server side, the peer's owner should have the possibility of configuring the Caching service accordingly with the available resources and the type of services proposed.

Behavior on Successful Callback

We should let the service provider choose what to do with the cached results after they were transmitted to the client. Keeping cached results has an interest if we consider optimizations at peer level.

Maximum Size

A usefull option that the service provider should have is to limit the maximum size of his caching space, thus managing his total storage space on his peer.

Caching Mode

If the client can choose *which* one of its results to have cached, the service provider can choose *how* to do the actual saving of results. The caching mode on server side specifies whether we want to compress the results before saving them or to save them directly as we received them. The first alternative brings an overhead at computational level, by introducing processing of the data before it is saved, while the latter one brings an overhead at spatial level by saving uncompressed data.

3.5.3 Optimizations

We wanted to optimize a peer's behavior through an efficient use of the caching service. The first idea was to use the Caching service *not only to save* missed results but also *to share* results.

Context Invariance of a Service

We define the context invariance of a service as the time interval where calls to that service will return the same result. For better understanding of the concept we extend the example with NYSE stock quoting service and we assume that quotes change every 10 minutes. It is obvious that calls that happen in that time interval will return the same result. Thus we can say that the context invariance of NYSE's stock quoting service is of 10 minutes.

Grouping Service Calls

The service calls addressed to the same service, that have the same parameters and are inside the respective service's context invariance will obtain the same result. We say that these equivalent service calls define a group of likeliness. Having grouped equivalent service calls we observe that it is efficient to fire only one service call per likeliness group and then use its result in servicing the rest of service calls in the respective group.

Different Kinds of Sharing

We can assimilate the grouping of service calls defined above with the more general notion of sharing the same result. Because we use services' results either to perform callbacks or to cache them we can do a sharing of:

- already cached results
- wrapped service results

Sharing already cached results has a major impact to the storage space required by the Caching Service because we can save a result per likeness group. The second type of sharing is important because it reduces the overhead at peer and network level by performing only once the call per likeness group. This means that the service will be invoked only once and thus we will avoid computing multiple times the same thing. It also means that we avoid sending redundant information over the network (e.g we have an exterior service that we wrap into a continuous one. Every time that we execute callback we have to fire a call to the exterior service. If we have a likeliness group, we will call the wrapped service only once and then use its result in performing all callbacks for the respective likeness group)

3.5.4 Ongoing work

We are currently focused on implementing a coherent system of optimization at peer level, integrating the optimization performed at Caching service's level with the dynamic analysis of the peer's load and load balancing between peers.

Chapter 4

Composed Services in Active XML

Once we had defined continuous services and implemented them in Active XML, we realized that we could have done it much more easily if we had a way, a functionality that would have allowed us to compose existing services.

The interest was high because of the extensibility that we could have obtained at the language's level once we were able to take basic bricks that were already defined and use them to build complex services, better suited to our needs.

The advantage was that being able to form very easily new services, the entire system became highly modulable. Imagine that we provide a standard set of services that we deploy on every peer. The peer's owner is then able to define his own desired functionality according to his possibilities and the role he wishes to play on the web. As an example of a particular workflow we can consider the automaton defined in Section 3.2 that modeled the server's behavior. It is obvious that having a simple declarative language for workflow can ease the definition of such behaviors.

We can better grasp the importance if we consider the following straightforward example: lets say that we have two Active XML peers, a Personal Digital Assistant (PDA) and a regular desktop PC. Our basic set of services consists of a service that allows us to query an Active XML document (we'll call it from now on "query "), a service that gives us the differences between two XML documents or permits us to reconstruct an XML document starting from its differences with one of its previous versions and the respective version (the Diff service previously defined) and a service that integrates the results into an Active XML document (Callback service). What we would like to define is an application that queries a document on one side and sends the differences(delta) with previous results to the other side. Because of obvious limitations in power and space on the PDA peer, we will deploy our XML documents on the PC peer and send the results to the PDA. If we are able to compose services, we can quickly form new services that model the above described behavior, starting from the existing base kit. On the PC peer we will take the query service and compose it with the Diff service thus performing queries and sending over the connection only the delta. On the PDA we will use the Diff service in order to retrieve the new document starting form the current delta and the previous version and the Callback service that will integrate this document in the specified document.

There were two major directions that we considered when we added composed service functionality to Active XML:

- The first one was to define our own language for composing existent web services and to provide a workflow engine that would realize the actual composition of services.
- The second one was to reuse an existent language that described the services' composition, or eventually a subset of such a language and to reuse an existent workflow engine that implemented it.

4.1 Implementing our own WF language and WF engine

The advantages of having our own (restricted) WF language are the following:

- We can adapt it better to our needs, being us its architects.
- We are not dependent of a specification that might change overnight, thus forcing us to re-think our application's logic.

A first idea could be to realize just the language and to compile from it to another WF language (ie: BPEL4WS) that already has implementations of WF engines (ie: BPWS4J). Although this is faster to realize, it does not offer us total independence from proprietary specifications. It mearly means that every time the specification changes, we have to change the compiler that does the translation between the two WF languages.

If total independence from specifications is what we want then we are forced to provide our own WF engine that will implement the WF language that we defined.

4.1.1 Restricting the General Problem to Our Needs

There are numerous papers that address the general problem of a workflow language for composing web services [10], [11], offering its complete description and formalism. We do not wish to implement the entire functionality presented in the aforementioned papers but only a small set of it that is

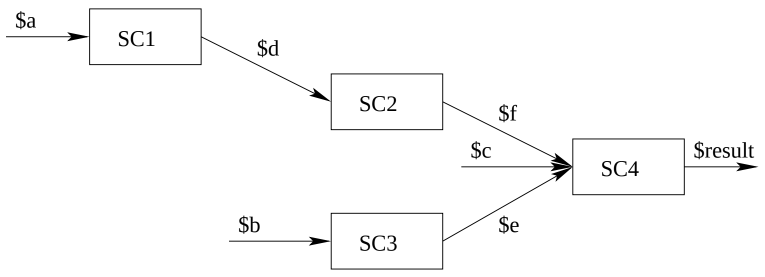

Figure 4.1: Composed Service Example

usefull for a simple service composition in Active XML. In the followings the modeling of the problem and the solutions we will provide are restricted to the service composition in Active XML, that are applicable to continuous services.

The first thing we wanted to do is to be able to define a new service as a sequence of existing ones. In order to make the new service accessible we have to define its interface (ie: what are it's methods signatures - parameters and their types, what is the structure of the returned result). When we are executing a sequence of services we might have dependencies between the output of one service and the input of another one. We must therefore accept that not all parameters of a service are known (evaluated) from the beginning and thus we have to expand the notion of parameter to that of a variable.

In order to use as much as possible from the existing functionality in Active XML, we can model our simple workflow by composing service calls. This way the execution of a service will be represented by an Active XML call to that service. The only differences with the usual manner of firing calls from an Active XML document are:

- the invoked service's result will replace the call that triggered its execution
- we are able to bind parameters at runtime

We need only to define a new syntax that will give us the order of calls and their nesting. Lets take a simple example of defining a web service that is composed by four different other services, situated on different levels (see Figure 4.1).The inputs for the four services are: S1(a), S2(S1), S3(b) and S4(S2, S3, c).

A sequence of service calls means that we have a fixed order for evaluating the respective calls and that implicitly the result from one service call is the input of the next one. Thus we model the sequential behavior of the system. It is obvious that we cannot limit only to a sequential treatment of service calls, because we will loose drastically in terms of performance. In the above example it would mean to disregard the fact that S2 and S3 can be treated in parallel. The order of evaluation is also very important (ie in our example we need to evaluate first S1, then S2 and S3 in parallel and at last S4). This order can be inferred by the examination of available parameters and by following an execution plan that will optimize the firing of calls situated on the same level and exterior to sequences. Once all parameters for a service call are known, we are entitled to fire the call.

Another problem that arises is that of cycles in the composed service's definition. We should not execute definitions that contain interdependent service calls because we would find ourselves in the situation of looping forever inside the cycle. To solve these problems we build a dependency graph, imposing the condition of aciclicity (we obtain a tree representation of service calls evaluation). What rests to be specified is the transfer (or correspondence) of parameters. For this we name the parameters and we'll do a name based coupling between outputs and inputs of services.

Although parameters are well coupled according to their names we can still have mismatches between them. A mismatch occurs when we are trying to give to a parameter a value with incompatible structure. We will perform a static detection of the mismatches based on the service's WSDL. This detection is done by statically analyzing the service's WSDL and deciding if the values that we'll use as input for the respective service match its input parameters. If parameters do not match we will add a service that will do the matching (e.g am XSLT - stylesheet one).

4.1.2 Simple Workflow Example

A straightforward example would be to define the composed service illustrated by Figure 4.1:

```
<serviceDefinition name="ComposedService" type="composed">
  <parameters>
    <$a,$b,$c>
  </parameters>
  <SC1>
    <axml:sc1...>
      <axml:params>
        <axml:param name="$a"></axml:param>
      </axml:params>
    </axml:sc1>
    <result name="$d">
  </SC1>
  <SC2>
    <axml:sc2...>
      <axml:params>
```

```
        <axml:param name="$d"></axml:param>
      </axml:params>
    </axml:sc2>
    <result name="$f">
  </SC2>
  <SC3>
    <axml:sc3...>
      <axml:params>
        <axml:param name="$b"></axml:param>
      </axml:params>
    </axml:sc3>
    <result name="$e">
  </SC3>
  <SC4>
    <axml:sc4...>
      <axml:params>
        <axml:param name="$f"></axml:param>
        <axml:param name="$c"></axml:param>
        <axml:param name="$e"></axml:param>
      </axml:params>
    </axml:sc4>
    <result name="result">
  </SC4>
  <sequence>
    <axml:sc1.../>
    <axml:sc3.../>
    <axml:sc2.../>
    <axml:sc4.../>
  </sequence>
</serviceDefinition>
```

As usual we can define the expected input parameters by our composed service. Sequences are contained in the ”sequence” tags. For all other calls we specify the inputs, doing a name based parameter binding and we name the result. Doing this way allows us to infer the evaluation order, as presented in the previous subsection.

The definition of a composed service is then parsed by our WF engine that will actually build the process representing the described functionality. The advantage here resides in the fact that we restricted the general task of doing WF on web services to one that's much more simpler and better adapted to most of our needs in respect to service composition.

The advantage is that we know the exact overhead that the service composition will introduce on an Active XML peer and thus we are able to do a better estimate of the current and future load at peer level. We can also change both the WF language and its underlying implementation without affecting the rest of the Active XML system.

In the followings I will present the other direction that we considered when we added composed services to Active XML.

4.2 Using BPEL4WS as WF language and BPWS4J as WF engine

4.2.1 Business Process Paradigm

An accurate description of business process paradigm can be found in [20].

Systems integration requires more than the ability to conduct simple interactions by using standard protocols. The full potential of Web Services as an integration platform will be achieved only when applications and business processes are able to integrate their complex interactions by using a standard process integration model. The interaction model that is directly supported by WSDL is essentially a stateless model of synchronous or uncorrelated asynchronous interactions. Models for business interactions typically assume sequences of peer-to-peer message exchanges, both synchronous and asynchronous, within stateful, long-running interactions involving two or more parties. To define such business interactions, a formal description of the message exchange protocols used by business processes in their interactions is needed. The definition of such business protocols involves precisely specifying the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal implementation. There are two good reasons to separate the public aspects of business process behavior from internal or private aspects. One is that businesses obviously do not want to reveal all their internal decision making and data management to their business partners. The other is that, even where this is not the case, separating public from private process provides the freedom to change private aspects of the process implementation without affecting the public business protocol.

Business protocols must clearly be described in a platform-independent manner and must capture all behavioral aspects that have cross-enterprise business significance. Each participant can then understand and plan for conformance to the business protocol without engaging in the process of human agreement that adds so much to the difficulty of establishing cross-enterprise automated business processes today.

What are the concepts required to describe business protocols? And what is the relationship of these concepts to those required to describe executable processes? To answer these questions, consider the following:

- Business protocols invariably include data-dependent behavior. For example, a supply-chain protocol depends on data such as the number of line items in an order, the total value of an order, or a deliver-by deadline. Defining business intent in these cases requires the use of conditional and time-out constructs.

- The ability to specify exceptional conditions and their consequences, including recovery sequences, is at least as important for business protocols as the ability to define the behavior in the "all goes well" case.

- Long-running interactions include multiple, often nested units of work, each with its own data requirements. Business protocols frequently require cross-partner coordination of the outcome (success or failure) of units of work at various levels of granularity.

If we wish to provide precise predictable descriptions of service behavior for cross-enterprise business protocols, we need a rich process description notation with many features reminiscent of an executable language. The key distinction between public message exchange protocols and executable internal processes is that internal processes handle data in rich private ways that need not be described in public protocols.

In thinking about the data handling aspects of business protocols it is instructive to consider the analogy with network communication protocols. Network protocols define the shape and content of the protocol envelopes that flow on the wire, and the protocol behavior they describe is driven solely by the data in these envelopes. In other words, there is a clear physical separation between protocol-relevant data and "payload" data. The separation is far less clear cut in business protocols because the protocol-relevant data tends to be embedded in other application data.

4.2.2 Choosing BPEL4WS for Modeling WS' Composition

The best option for a workflow language that models services' composition using the concepts of business process was BPEL4WS. It represents an evolution of Web Services Flow Language (WSFL) [19] which is an XML language for describing Web Services composition developed at IBM and of XLANG [18] which is Microsoft's language for formally specifying business processes as stateful long-running interactions.

BPEL4WS uses a notion of message properties to identify protocol-relevant data embedded in messages. Properties can be viewed as "transparent" data relevant to public aspects as opposed to the "opaque" data that internal/private functions use. Transparent data affects the public business protocol in a direct way, whereas opaque data is significant primarily to back-end systems and affects the business protocol only by creating non-determinism because the way it affects decisions is opaque. We take it as a principle that any data that is used to affect the behavior of a business protocol must be transparent and hence viewed as a property.

The implicit effect of opaque data manifests itself through nondeterminism in the behavior of services involved in business protocols. Consider the example of a purchasing protocol. The seller has a service that receives a purchase order and responds with either acceptance or rejection based on a number of criteria, including availability of the goods and the credit of the buyer. Obviously, the decision processes are opaque, but the fact of the decision must be reflected as behavior alternatives in the external business protocol. In other words, the protocol requires something like a switch activity in the behavior of the seller's service but the selection of the branch taken is nondeterministic. Such nondeterminism can be modeled by allowing the assignment of a nondeterministic or opaque value to a message property, typically from an enumerated set of possibilities. The property can then be used in defining conditional behavior that captures behavioral alternatives without revealing actual decision processes. BPEL4WS explicitly allows the use of nondeterministic data values to make it possible to capture the essence of public behavior while hiding private aspects.

The basic concepts of BPEL4WS can be applied in one of two ways. A BPEL4WS process can define a business protocol role, using the notion of abstract process. For example, in a supply-chain protocol, the buyer and the seller are two distinct roles, each with its own abstract process. Their relationship is typically modeled as a partner link. Abstract processes use all the concepts of BPEL4WS but approach data handling in a way that reflects the level of abstraction required to describe public aspects of the business protocol. Specifically, abstract processes handle only protocol-relevant data. BPEL4WS provides a way to identify protocol-relevant data as message properties. In addition, abstract processes use nondeterministic data values to hide private aspects of behavior.

It is also possible to use BPEL4WS to define an executable business process. The logic and state of the process determine the nature and sequence of the Web Service interactions conducted at each business partner, and thus the interaction protocols. While a BPEL4WS process definition is not required to be complete from a private implementation point of view, the language effectively defines a portable execution format for business processes that rely exclusively on Web Service resources and XML data. Moreover, such processes execute and interact with their partners in a consistent way regardless of the supporting platform or programming model used by the implementation of the hosting environment.

Even where private implementation aspects use platform-dependent functionality, which is likely in many if not most realistic cases, the continuity of the basic conceptual model between abstract and executable processes in BPEL4WS makes it possible to export and import the public aspects embodied in business protocols as process or role templates while maintaining the intent and structure of the protocols. This is arguably the most attractive prospect for the use of BPEL4WS from the viewpoint of unlocking the potential of Web Services because it allows the development of tools and other technologies that greatly increase the level of automation and thereby lower the cost in establishing cross-enterprise automated business processes.

In summary, we believe that the two usage patterns of business protocol description and executable business process description require a common core of process description concepts.

BPEL4WS defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners. The interaction with each partner occurs through Web Service interfaces, and the structure of the relationship at the interface level is encapsulated in what we call a partner link. The BPEL4WS process defines how multiple service interactions with these partners are coordinated to achieve a business goal, as well as the state and the logic necessary for this coordination. BPEL4WS also introduces systematic mechanisms for dealing with business exceptions and processing faults. Finally, BPEL4WS introduces a mechanism to define how individual or composite activities within a process are to be compensated in cases where exceptions occur or a partner requests reversal.

BPEL4WS is layered on top of several XML specifications: WSDL 1.1, XML Schema 1.0, and XPath1.0. WSDL messages and XML Schema type definitions provide the data model used by BPEL4WS processes. XPath provides support for data manipulation. All external resources and partners are represented as WSDL services. BPEL4WS provides extensibility to accommodate future versions of these standards, specifically the XPath and related standards used in XML computation.

4.2.3 Choosing BPWS4J as WF Engine

Once we had established that BPEL4WS is the WF language that we needed to use to support service composition in Active XML we had to find a working implementation of it. We call such an implementation as WF engine beause it actually "runs" the processes we defined with BPEL4WS. The most appropiate implementation we could find was BPWS4J, realized by AlphaWorks.

BPWS4J's Definition

The IBM Business Process Execution Language for Web Services JavaTM Run Time (BPWS4J) includes the following: a platform upon which can be executed business processes written using the Business Process Execution Language for Web Services (BPEL4WS); a set of samples demonstrating the use of BPEL4WS; and a tool that validates BPEL4WS documents.

BPWS4J's Functioning

For each process, the BPWS4J engine takes in a BPEL4WS document that describes the process to be executed, a WSDL document (without binding information) that describes the interface that the process will present to clients (partners in BPEL4WS terms), and WSDL documents that describe the services that the process may or will invoke during its execution.

From this information, the process is made available as a Web service with a SOAP interface. A WSDL file that describes the process's interface may be retrieved from the run-time. The BPWS4J engine supports the invocation, from within the process, of Web services that have a SOAP interface, that are EJBs, or that are normal Java classes.

BPWS4J's Interface

Unfortunatelly the access offered to BPWS4J was at the moment of writting this paper only through a web interface. This limited the options of using BPWS4J inside an application. In order to make it transparent to the user (the Active XML peer's owner) we had to define a complex wrapper that simulates invocations to BPWS4J's web interface from Active XML. We had to define a sort of a BPWS4J Dialogue API that managed all possible interactions with BPWS4J's framework. The functions offered by the BPWS4J's web interface are:

- List all processes currently deployed - displays the list of active and inactive processes.
- Deploy a process and its partners - a process might have partners and when the deployment is made for the respective process we have to specify also its partners' WSDLs locations, thus providing the BPWS4J's WF engine with all the necessary information for the respective process.
- Undeploy an existent process - terminates an instance of an existent process.

These functions allow us basic process creation, monitoring and termination.

4.2.4 Integrating BPWS4J into Active XML

What we did was to wrap BPWS4J WF engine into Active XML. A simplified view is represented by Figure 4.2

General View of the Interaction Between Active XML and BPWS4J

What is very important to specify is that there is a clear demarcation between an Active XML composed service and the BPEL process it wraps. We

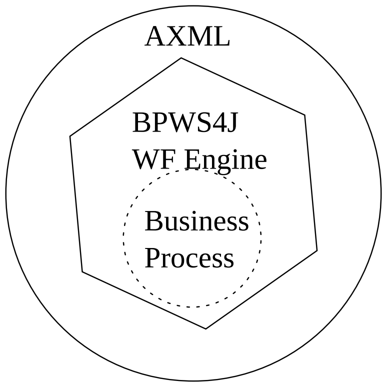

Figure 4.2: BPWS4J's Wrapping in Active XML

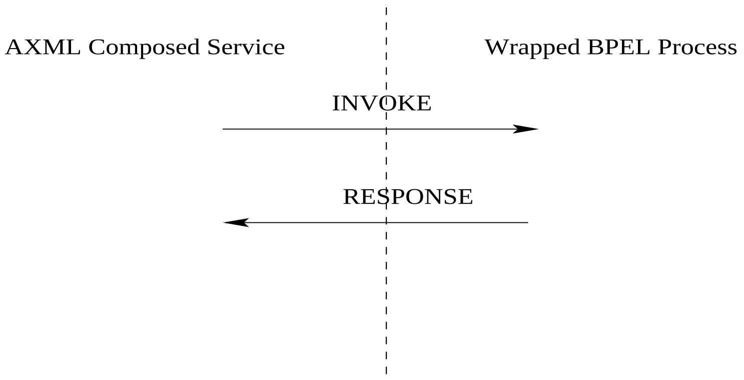

Figure 4.3: Interaction Between Active XML Composed Service and Its Wrapped BPEL Process

have one Active XML service per BPEL process and we use it to transmit to the process, which is running inside BPWS4J, requests made by Active XML peers and to retreive the results offered by the BPEL process and send them back to the Active XML clients that requested them. The interaction between an Active XML composed service and the BPEL process it wraps is illustrated in Figure 4.3 Because the Active XML composed service acts as a wrapper, it is natural that the parameters of the BPEL process be a subset of the Active XML service's parameters. Having more parameters for the Active XML composed service allows us to control the things that are orthogonal to the functioning of the wrapped process (ie: the maximum number of calls that we want to accept, the lifetime of the composed service).

The Creation of an Active XML Composed Service

The creation of an Active XML composed service starts from its definition file. This is a regular service definition file, having the type *composed* and specifying service's parameters' names and types at the begining. The *service definition* element contains:

- Process's parameters - a subset of Active XML's service's parameters.
- An optional WSDL for the process - if the user does not provide a WSDL for the process then one will automatically be generated for it.
- Caller identification - because BPWS4J uses an entry point called *soaprpcrouter* that routes calls to the deployed processes, we must provide a client identification for the repective call. The client is seen as a parnter of the BPEL process. The information needed by *soaprpcrouter* for correctly routing the call to the respective process is:
  - the client's name space.
  - the process's operation used for invoke.
  - the above operation's port type.
  - the above port's URI.
- Partners' WSDLs locations - if the BPEL process has multiple partners we have to specify their WSDLs locations, in order to be able to retreive their WSDL files when we will realize the deployment of the process to the BPWS4J's WF engine.
- Process's BPEL code - it contains the BPEL definition of the process. This code will be interpreted by BPWS4J and transformed into a composed process.

At Active XML's peer initialization, this file will be parsed by a dedicated *service factory* that will create the Active XML service corresponding to the respective service definition. The factory will first extract the process's BPEL definition. It will then check to see if a WSDL was provided for the respective process and if not, it will generate one for it. Then it will check to see if the process has some partners and if so, it will extract their WSDLs' locations. The composed service factory will then set the information necessary for correctly invoking a BPEL process through the *soaprpcrouter*. An Active XML service instance will then be created and the new service will be published. The operations described above are illustrated by the Figure 4.4.

The Deployment of the Wrapped BPEL Process to BPWS4J

From BPWS4J's point of view there are two disitinct phases in a process's life:

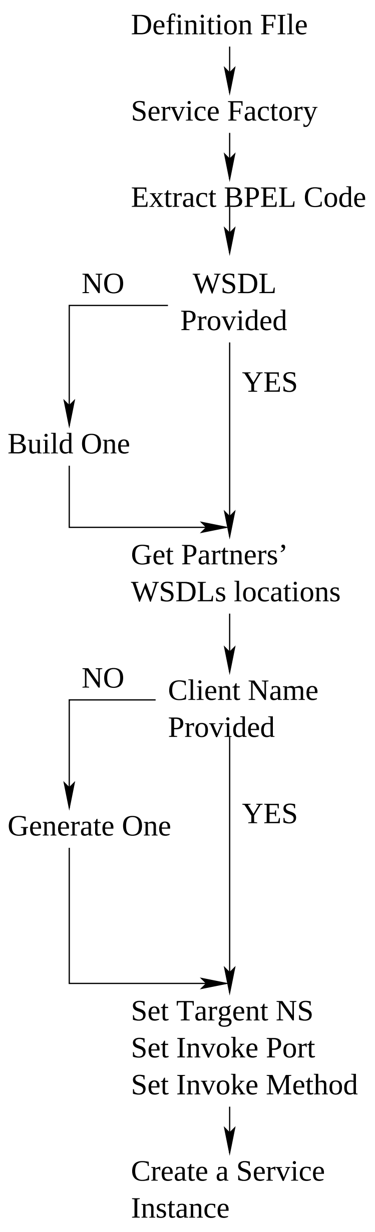

Figure 4.4: Active XML Composed Service Creation

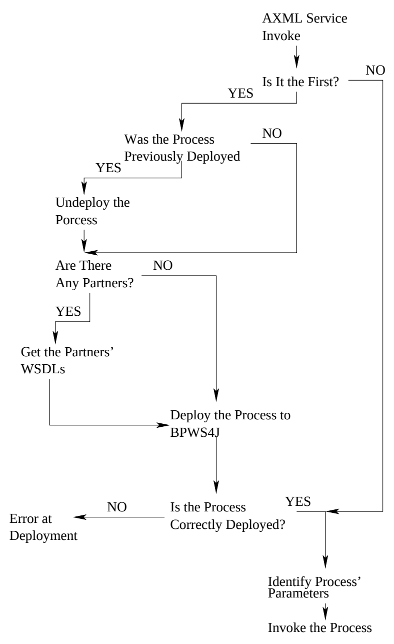

Figure 4.5: Lazy Process Deployment to BPWS4J

1. the creation - the process is deployed to BPWS4J's WF engine, but an instance for it does not yet exist.
2. the activation - on the first invoke for the process, an instance for it is created.

Active XML has a slighltly different philosphy concerning a service's life-cycle: once a service is created, an instance for it exists and there is no difference between the first call to that service and the following ones. In order to integrate BPWS4J to Active XML, we designed a scheme of *lazy deployment* for the wrapped BPEL process. This means that the BPEL process is deployed to BPWS4J and an instance for it is created only when a first call for the wrapping Active XML composed service is received. By doing this, we avoid deploying process's that aren't used and thus we reduce the overhead brought by BPWS4J's workflow engine to the system. The lazy deployment of a BPEL process is illustrated by the Figure 4.5.

4.3 Conclusions

We enhanced Active XML's functionality by supporting service composition. Basically what we did is to add a service composition workflow engine to the Active XML peer. Thus, the peer's owner is now able to define a desired functionality starting from existent basic units (services). We tried to render the definition of a composed service transparent to the user. To accomplish this we choosed BPEL as base language because of its wide aplicability and soon to come standardization. Thus we are sure that a working implementation of the language will always exist. For reasons linked to optimization (ie: an accurate load calculus for an Active XML peer) we considered implementing our own workflow engine. This engine has a restricted functionality, its main advantage being that knowing its architecture we can easily adapt it to our needs. Although workflow languages have been studied for more than 20 years, service composition in Active XML opened a new area of interesting applications.

4.4 Ongoing Work

We are currently working on our own implementation of a workflow engine, based on a restricted BPEL language. This engine will provide an alternative for the users that do not wish or can't afford to use BPWS4J as a workflow engine (ie: an Active XML peer that turns on a PDA has limited resources and a complete WF engine such as BPWS4J can't run on it). We are also considering service composition for continuous services and we are trying to define a system that will allow us to accomplish such a thing.

Chapter 5

Conclusions

By giving Active XML peers the possibility of pushing data to their clients, continuous services made the language more extensible and better suited for its users' needs.

We have added support for continuous services both on client and server side. On client side we defined a simple way of writing subscription calls. The client has only to specify that the respective service call is a subscription, the rest of the parameters needed by our framework being automatically compiled. On the server side we defined a general mechanism of performing callbacks and we rendered it reliable by implementing a fallback sequence.

Defining continuous services and adding support for them in Active XML made us discover new directions for future work:

- generic definition of a Caching Service
- simple composition of existent services
- generic error handling

5.1 Practical Applications

Because the push architecture we added to Active XML allows us to model asynchronous behavior we can easily integrate asynchronous services.

THESUS [8] is a system that deals with an initial set of web pages, extracts keywords from all pages' incoming links, converts them to semantics by mapping them to a domain's ontology and applying a clustering algorithm to discover document clusters. It is used in building a Set of Pages of Interest(SPIN) [9]. THESUS has an important execution time that does not allow us to use a synchronous request response architecture to retrieve its result. Thus, we integrated it into SPIN using asynchronous capability brought by our continuous services framework.

5.2 Thank You

Thank you **Omar Benjelloun** for your patience and understanding while helping me write this report. Your keen observations concerning the articulations of my sentences and logical reasoning brought coherence and structure into my work.
Thank you **Ioana Manolescu** and **Tova Milo** for your advices on writing a technical paper.
Thank you **Serge Abiteboul** for your living example.

Bibliography

[1] M. Rossopoulos, M. Baker *Practical Load Balancing for Content Requests in Peer-to-Peer Networks*, Department of Computer Science - Stanford University 2002

[2] S. Abiteboul, T. Milo, O. Benjelloun *Web Services and Data Integration*, International Conference on Web Information Systems Engineering 2002

[3] S. Abiteboul, M. Preda, G. Cobena *Computing Web Page Importance Without Storing the Graph of the Web*, IEEE-CS DataEngineering Bulletin 2002

[4] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, R. Weber *Active XML: Peer-to-Peer Data and Web Services Integration (demo)*, VLDB 2002

[5] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, R. Weber *Active XML: A Data-Centric Perspective on Web Services*, Confrence sur les Bases de Donnes Avances 2002

[6] S. Abiteboul, O. Benjelloun, T. Milo *Towards a Flexible Model for Data and Web Services Integration*, proc. Internat. Workshop on Foundations of Models and Languages for Data and Objects 2001

[7] S. Abiteboul, V. Vianu *Queries and Computation on the Web*, ICDT 1997

[8] M. Halkidi, B. Nguyen, I. Varlamis, M. Vazirgiannis *THESUS: Organizing Web Document Collections Based On Semantics And Clustering*, Gemo Report 2002

[9] S. Abiteboul, G. Cobena, B. Nguyen, A. Poggi *Construction and Maintenance of a Set of Pages of Interest*, BDA 2002

[10] F. Leymann *Web Services Flow Language (WSFL 1.0)*, May 2001

[11] T. Andrews, F. Curbera, F. Leymann et al *Business Process Execution Language for Web Services Version 1.1(BPEL4WS)*, May 2003

[12] World Wide Web Consortium *Extensible Markup Language (XML)*

[13] World Wide Web Consortium *Web Services Architecture*, Working Draft 14 May 2003

[14] World Wide Web Consortium *SOAP Version 1.2 Part 1*, Working Draft 26 June 2002

[15] World Wide Web Consortium *Web Services Description Language (WSDL) 1.1*, W3C Note 15 March 2001

[16] Apache Software Foundation *http://ws.apache.org/axis/*, 16 June 2003

[17] Apache Jakarta Tomcat Project *http://jakarta.apache.org/tomcat/*

[18] Satish Thatte *XLANG*, http://www.gotdotnet.com/team/xmlwsspecs/xlang-c/default.htm

[19] Frank Leymann *Web Services Flow Language*, http://www-3.ibm.com/software/solutions/ webservices/pdf/WSFL.pdf

[20] *Business Process Execution Language for Web Services Version 1.1*, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel