"Politehnica" University Bucharest
Faculty of Automatics Control and Computers

Implementation of the Active XML Peer for the J2ME platform

Undergraduate
Cosmin Cremarenco

Scientific Counselors
Prof. Dr. Ing. Irina Athanasiu
Dr. Ing. Ioana Manolescu

2003

Abstract

Embedded systems become more and more sophisticated. Today's mobile devices have the necessary hardware to host complex applications especially designed to speculate advantages of a mobile platform.

Subject of this thesis is the implementation of an Active XML[1] peer whose deployment target is the mobile platform.

A classic Active XML peer used to be implemented as a fix application (as regard to its physical location) and required a lot of computing resources. What is the most important benefit that arises from using the Active XML technology? ease of managing data placed in physically different locations. The Active XML Mobile Peer amplifies this. It makes the location totally transparent and it ought to be the next evolutionary step in P2P networks.

[1] Patent of Inria France

1. Introduction

With the advent of the Internet access to information becomes more and more easy and cheap. However the Internet is heterogeneous, driven by software that doesn't facilitate Intercommunication.

1.1 About the Active XML technology

The heart of the Active XML is represented by Web Services, open standards (SOAP, WSDL, XML-RPC, and UDDI) that facilitate communication between peers. This project integrates Active XML document model and web services onto a wireless platform.

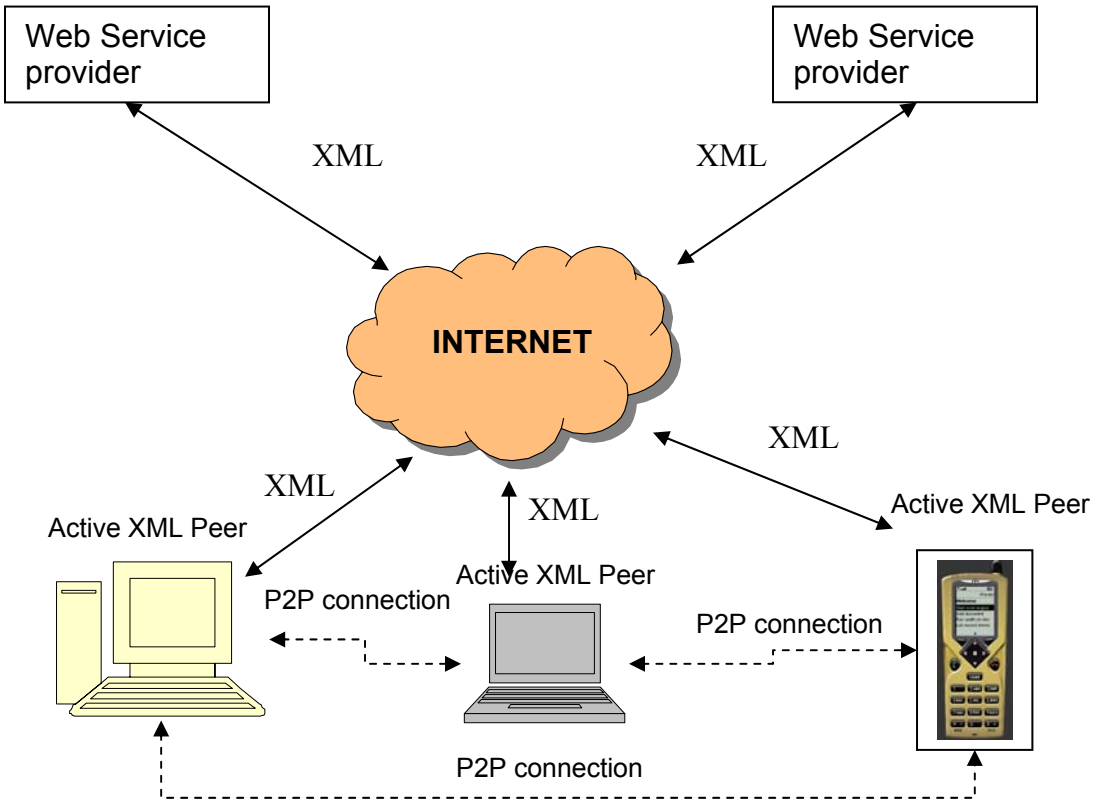

Figure 1

An important thing to notice in fig. 1 is that communication Active XML peer – Active XML peer and Active XML peer – Web Service doesn't use any proprietary protocols, but plain XML messages.

Scenarios of use include:

- electronic biddings; in this particular scenario every peer holds a list of items it's interested in. By means of communication between Active XML peers and other Active XML peers or third party web services bidding data is exchanged and every peer has its own view over the auction.
- distributed data warehouses. In this case Active XML can serve both as repository for data but also as data integration tool. In the latter case it allows querying over multiple heterogeneous sources.

1.1.1 Active XML document model

The Active XML document model is structured as an XML which contains references to web services.

References towards web services can be translated to web service invocation results which are included in the Active XML or can be passed to another Active XML peer. The latter is also free to call or not the web service that's reference it has received. In other words some data is explicitly included in the source document whilst for other data a definition is included as a reference to a web service. If the data source that is referenced in the definition changes its contents, of course the Active XML document will also synchronize.

As opposed to other technologies that imply mixing data with executable code the Active XML contains seeds of distributed computing. One peer can translate the web service reference that it evaluates or it can send it to another peer. Hence one is able to receive tasks in the form of a web service reference to be executed or it can delegate to another peer a task by passing it a web service reference.

Center of the Active XML technology is the <sc> element which contains all the information necessary to instantiate a web service. It is placed in a proper XML namespace to differentiate it from other elements with the same name which may appear in the same Active XML document. The xml namespace is "http://www-rocq.inria.fr/verso/AXML".

Example of an Active XML document

Irrelevant details have been removed.

```
<airports>
  <airport name=”Otopeni”>
    <weather>
      <sc service=”capeconnect.com/getWeather()”>
        <sc service=”director_indicative.ro/obtineIndicativ(Otopeni)” />
      </sc>
    </weather>
  </airport>
</airports>
```

In the above xml document fragment the service endpoint at capeconnect.com accepts as arguments only airport id’s. Unfortunately the airport id is unknown so a new service is nested in the first. The latter obtains the airport id and offers it as a parameter to the outer service.

In the next step the nested service is executed:

```
<airports>
  <airport name=”Otopeni”>
    <weather>
      <sc service=”capeconnect.com/getWeather(,BOTP’)”>
        <sc service=”director_indicative.ro/obtineIndicativ(Otopeni)” />
      </sc>
    </weather>
  </airport>
</airports>
```

Now the outer service can be translated into results:

```
<airports>
  <airport name=”Otopeni”>
    <weather>
      <sc service=”capeconnect.com/getWeather(,BOTP’)”>
        <sc service=”director_indicative.ro/obtineIndicativ(Otopeni)” />
```

```
    </sc>
      <location>Otopeni International Airport, Bucharest</location>
      <temperature>100 F</temperature>
      <humidity>9</humidity>
   </weather>
  </airport>
</airports>
```

Results from materializing the outer web service reference are placed underneath the <sc> element.

In the above examples important pieces of the <sc> element were not mentioned.

1.1.2 The <axml:sc> element

A real-life <sc> element looks like this:

```
<axml:sc serviceURL="http://staros.inria.fr:8080/axis/servlet/AxisServlet"
serviceNameSpace="Sleep" methodName="sleep" frequency="every 300000">
    <axml:params>
      <axml:param name="millis">
        <xpath>../duration/@value</xpath>
      </axml:param>
    </axml:params>
</axml:sc>
```

Parameters can be split in two main categories:

1. parameters to locate and call a web service;
2. data that the specific web service accepts as parameters;

1. The parameters to locate a SOAP web service are:

- serviceURL – compulsory, identifies the SOAP endpoint as a URL;
- serviceNameSpace – compulsory, frames the web service method in a specific namespace;

- methodName – compulsory, the method name which we want to be remotely executed in order for proper results to be returned;
- frequency – optional, states when the web service should be instantiated, thus also specifies the validity of the returned results. Has two forms that the Axml Mobile Peer recognizes: “once” and the service will be instantiated at the first pass of the inspector thread, “every x”: the service is periodically instantiated after x mili seconds;
- signature – optional, this attribute’s function is not fully implemented. It should allow dynamic invocation based on the WSDL of the target service;
- useWSDLDefinition – optional, specifies if we want to use or not the “signature” or the WSDL of the service;
- followedBy – optional, specifies an order of execution for the services. This service will be forced to execute before the one who’s name is stated in the followedBy attribute;
- id – optional, identifies the service with a unique sequence not only on the local machine, but also in the P2P network. If it is not specified, the peer automatically assigns one to the service;
- name – compulsory, a conventional name for this service. It is only used in the “followedBy” attribute;
- mode – optional, specifies what to do with the previous results. It only has two possible values: “replace”, clears the previous results and replaces them with the new ones, “merge” concatenates the new results with the old ones.

2. Parameters to the web service method.

The parameters section is element <axml:params> and is a child of the <axml:sc> element. A single parameters is placed inside a <axml:param> element.

Parameters can be specified in two ways:

- as value, child of the <value> element;

- as xpath expression, child of the <xpath> element.

Because when using the xpath mode to specify a parameter, one is able to specify as parameter the result of another service execution, this introduces a problem: deciding which service is responsible for the results that are of special interest to us.

This calls contain can be explicitly inserted in the document as <axml:sc> elements or can be the result of calling a web service. The results of web service call are embedded in the original XML.

This document model addresses problems such as distribution of data (because a document may contain only links to data physically deployed on another machine) and replication of data (because copies of a document are available on other computers).

2. Similar technologies

Mixing data with executable code is by no means a new idea. Some of the existing products that support this are:

- SUN's JSP – Java executable code is mixed with HTML markup data, everything is translated into a servlet later compiled into bytecode;
- PHP – php interpreted code is mixed with HTML markup;
- Apache Velocity – a replacement for the JSP technology uses an expression language instead of the well-known java scriptlets. It is used in the presentation layer of an application and clearly delimitates presentation and application logic;
- XSLT – XML format that allows special nodes that specify how to generate elements in the target XML document.

Web service references inside data are not a new idea also. Macromedia Coldfusion integrates data definition by using references to web services. Hence the Coldfusion document is client to web service data, but more interesting, it can play the part of data provider through SOAP. Coldfusion's purpose of existence is a rapid development of dynamic web pages that are also sensitive to data changes in the outside world, because it incorporates hooks to exterior data sources (web services).
Apache Jelly is a tool that allows translation from XML to executable code. This makes it a wise choice for inserting web service call references. It resembles JSP and Velocity, the difference is that Jelly executable code is inserted as XML nodes and not as scriptlets.

An important observation is that all this products trigger web service references translation to results or executable code execution the first time the document is inspected, thus all interpretable code is evaluated.

3. Encountered problems

The target of this project is the implementation of the Active XML mobile peer on the Java 2 Micro Edition platform.

J2ME is the java version meant for devices with limited resources.

Causes for the problems that were encountered when designing and implementing the mobile peer can be split into three big categories:

1. the sophisticated architecture of a Active XML peer;
2. restrictions due to the limited resources on the J2ME platform;
3. lack of software for XML processing designed for J2ME.

1. Active XML Peer is autonomous in a P2P network thus it has a complex structure and much functionality built in. The most important and delicate part is the Web Services management. Problems have been solved by a careful design and a modularized architecture.

2 . A mobile device has limited capabilities because of its size primarily. It has a limited amount of working memory, weak CPU performance (due to battery power conservation considerations). It works mostly offline and when online the bandwidth is small, thus exchanged messages should be small and should hardly occur.

3. Implementation of this project enriches the set of J2ME compatible tools for processing XML files.

It includes an xml repository that is in charge with managing the Active XML documents that are manipulated by the peer. This is the first implementation of such a feature for the J2ME platform and modest when comparing to implementations for other platforms: dbXML, Apache Xindice, eXist.

An XPath processor was implemented. Other products that process Xpath results (unfortunately not available for the J2ME plathorm) are:

- Jaxen implemented in Java only available for J2SE edtion;
- Pathman available in Java and C++ versions.

At the time this project was implemented there were no XPath processors for Java 2 Micro Edition.

In the same category of XML processing tools a XSL transformer was implemented that supports a rather limited instruction set of the XSLT language. Other products that implement XSLT transformations are Xalan, Saxon.

In the presentation module of the project a WML type file renderer paginates the results and lays them on the user screen. Similar WML renderers can be found in the mobile devices that support WAP. Because the implementation is in fact a midlet it is subject to a sum of limitations. One of this is that one cannot access the device's micro-browser from the midlet. Hence one cannot use the micro-browser to render an application generated WML.

4. General architecture

A general Active XML peer (and the mobile peer in particular) can communicate with two classes of targets:

- another Active XML peer by calling the services that the server part of it implements;
- a SOAP web service that implements a generic web service.

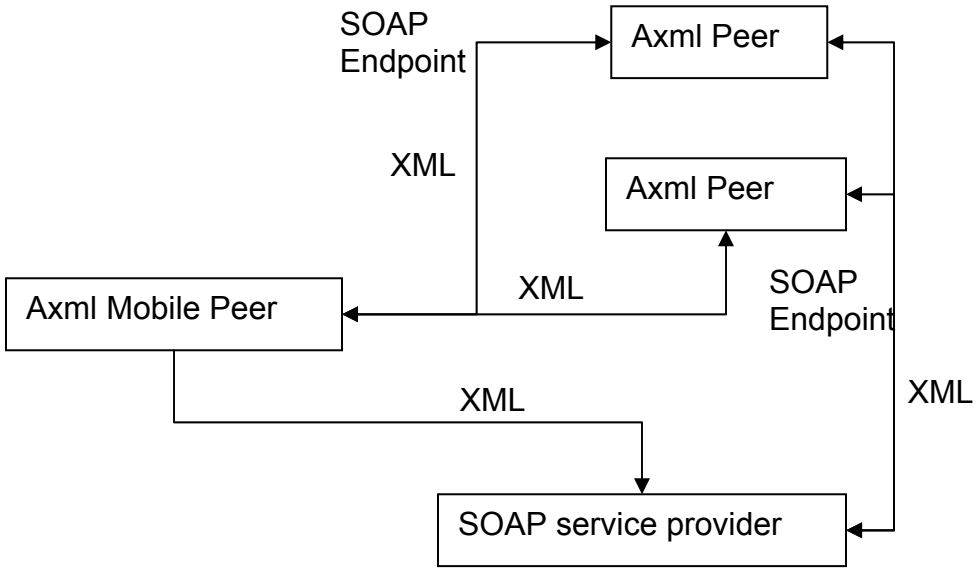

Figure 2

Communication between Axml Peers (one of which can be, in particular, an Axml Mobile Peer) is based on a communication infrastructure.

4.1 Infrastructure

The implementation of Active XML mobile peer uses web services. However, Active XML is not about web services (although interesting things are allowed, like web services chaining, nesting...) but it is all about the data in the document. Web services are just a tool.

It isn't that important the fact that we use XML-RPC and/or SOAP. Web services are part of the infrastructure in order to achieve the goal of distributing

and/or replicating documents onto multiple peers and ultimately integrating data from multiple sources into one bigger document.

Since XML-RPC and SOAP are both based on the HTTP protocol for transport (although SOAP can also use SMTP and others as transport protocols), this is what is used.

4.2 The Active XML Mobile Peer

It is composed of two main parts:

- the axml mobile peer itself, software deployed on the mobile device;
- a proxy in charge with mediating the communication between the mobile device and the outside world.

Why was a proxy needed in the project's architecture? A proxy is usually used in these situations:

- it serves more than one client and is in charge with storing data (cache) in order for more than one client to access data already retrieved, thus minimizing costs;
- hides parts of an internal architecture for security purposes;
- takes the burden of extra processing when dealing with a client. The client might have limited resources or not. In both cases processing is distributed among two entities.

The proxy is introduced in the architecture mainly because of the latter reason. All its functions are:

1. it intermediates service calls, communication between the mobile device and the proxy conforms to the XML-RPC protocol and communication between the proxy and the outside world conforms to the SOAP protocol;
2. it intermediates file transfer; in this case as opposed to the others a proxy might not have been strictly necessary. However interesting possibilities like caching the file for use by other clients can be exploited. Formally correct is that all the communications between the mobile client and the outside world are mediated by the proxy;

3. it acts as a directory for finding out which clients are currently online; of course the proxy and the clients directory could have been decoupled, but then overhead would have occurred because of messages exchanged between the proxy and the directory (possibly implemented as a web service itself).
4. intermediates queries which are destined to the mobile peer.

4.2.1 The mobile component (proxy is left out)

The mobile component (software physically deployed on the mobile device) has three main components:

1. an xml *repository* manages Active XML documents that are persistently stored on the mobile device;
2. *client* materialiases web services references into their results and generally handles web service calling;
3. *server* sets up a communication way such that a mobile proxy is able to receive and respond to queries from the outside world.

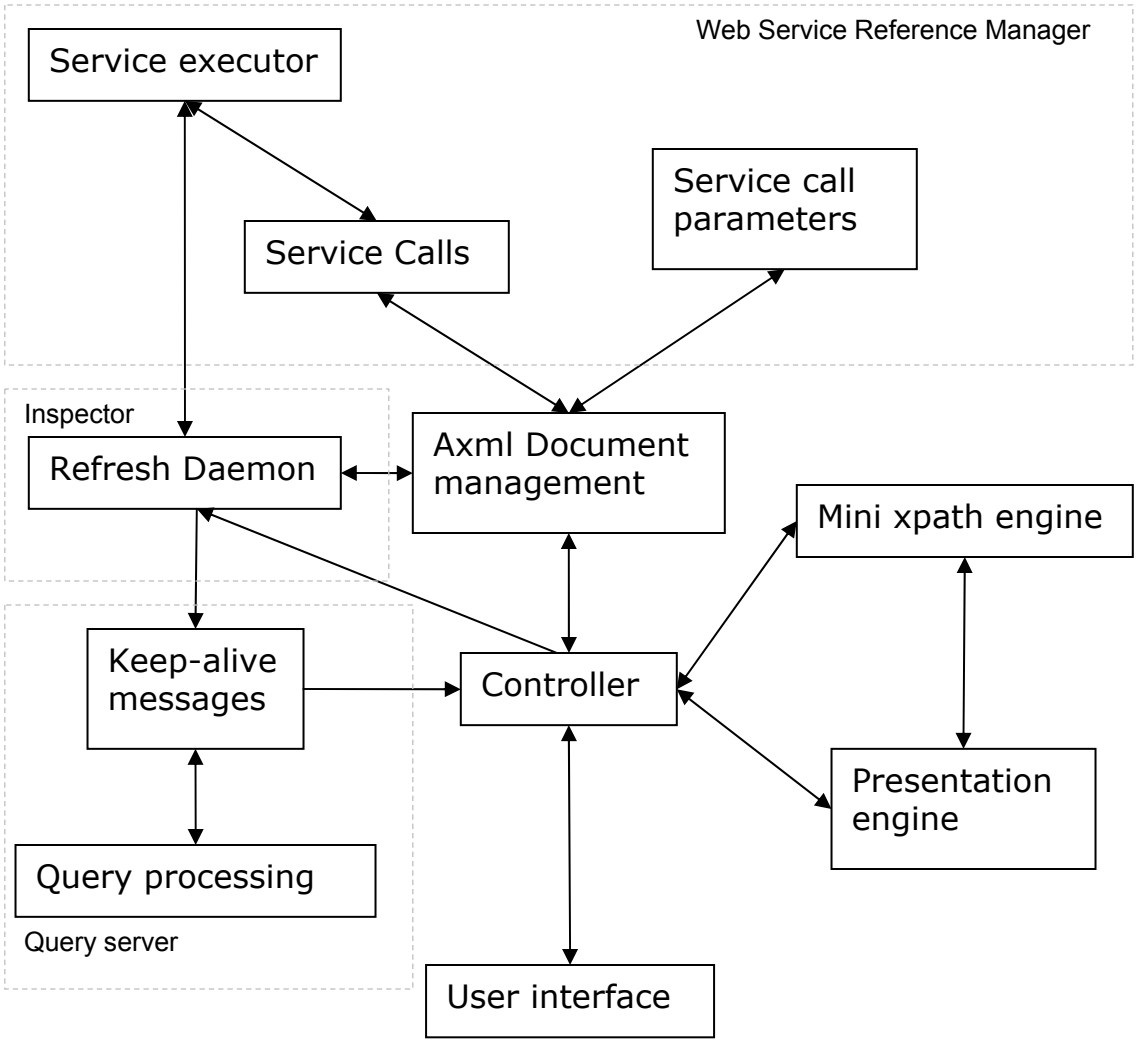

Figure 3

In fig. 2 *client* components are connected with each other. The *Query server* although part of the *server* component is included because is provides a way for the *server* to interact with the stored documents. It acts like a plugin.

4.2.1.1 Repository

The repository component is in charge with managing the physical data that makes up, at a logical level, the XML specific structure. It is made of these parts:

- “Persistent memory manager “ makes public methods for managing a stored XML in the phone specific Record Store;

- “File retriever” tries to obtain an xml file (axml, xsl or wml) by inspecting different locations;
- “DOM Cache” used to cache XML DOM fragments for later usage. It automatically adapts to device memory limitations.

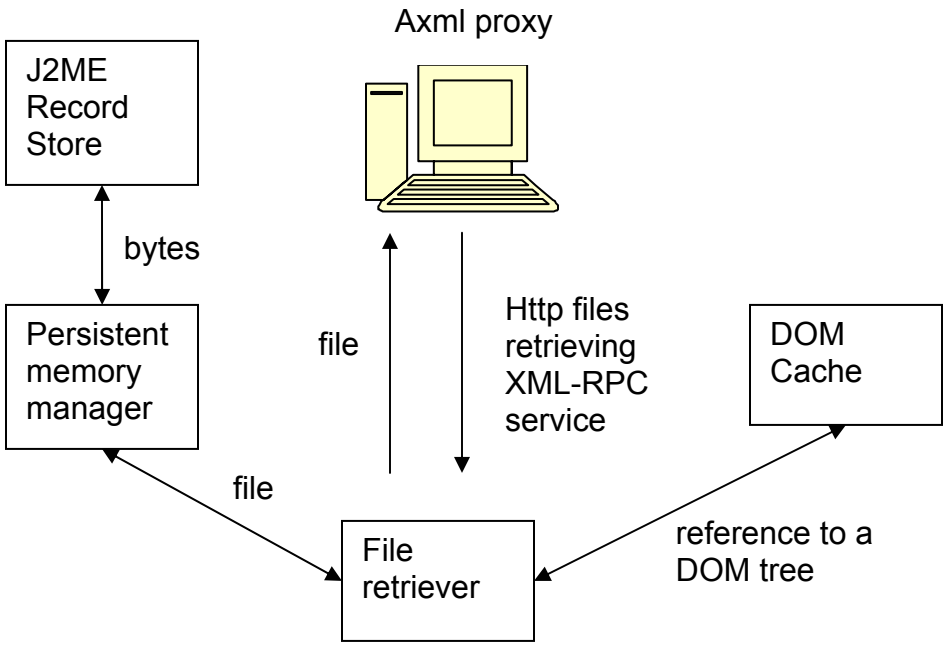

Figure 4

XML Repository’s purposes for existence are:

- Provide a unitary way of retrieving files that are to be processed as Active XML documents or presentation XSL;
- Facade to the J2ME Record Store System;
- Reduce overhead by using (when possible) a cache of DOM objects (XML files already parsed).

4.2.1.2 Client

The client part of the Axml Mobile peer is split into a few distinguishable components:

1. Active XML Document Manager;
2. Web Service Reference Manager;

3. Inspector;
4. Presentation Module;
5. Mini XPath Engine;
6. User interface.

1. Active XML Document Manager

Table containing references to all the Active XML documents currently present on the mobile peer. In charge with proper instantiation of an axml document and serialization into a storage ready string.

2. Web Service Reference Manager

A web service reference consists of parameters for calling the actual and its parameters. It is the logical representation of a <axml:sc> element present in the axml document. The Service Executor takes this reference and assembles a web service request to be addressed to the proxy.

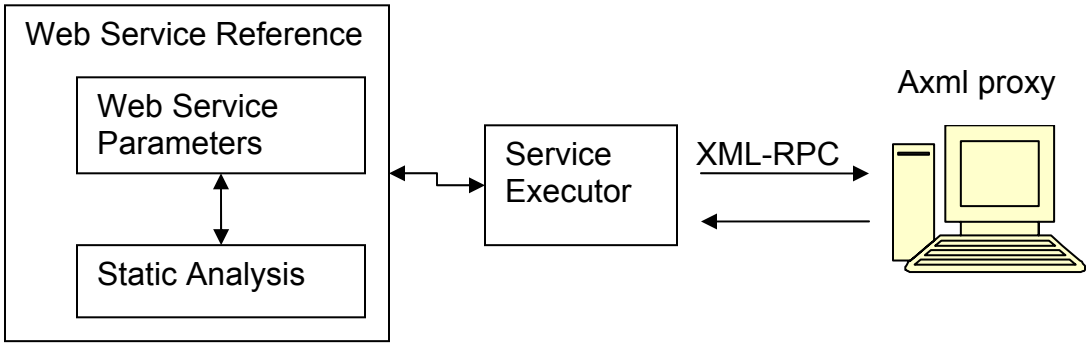

Figure 5

The parameters can reference the results of the materialization of another web service reference. To determine the service responsible for those results a "Static Analysis" method was implemented. It provides dependencies services which should be materialized in order to provide correct parameters to this service. Information about the web services to be materialized is fed to the Service Executor which in turn translates them to a XML-RPC style XML envelope. The *Service Executor* is triggered in two ways:

- by the Static Analysis module which corresponds to the lazy instantiation mode;

- by the Inspector module which corresponds to the immediate instantiation mode.

3. Inspector

Is in charge with triggering the Service Executor for different services when they expire and their data needs to be refreshed. It is executed at specific intervals and checks all the web service references.

4. Presentation module

Its purpose is to translate the results of calling different web services into a form user-friendly that can be displayed on the device screen.

5. Mini XPath Engine

Delivers an easy way of referring fragments of an xml, all this conforming to the XPath open standard. It is decoupled from the rest of the modules so that it can be used with more than one purpose in mind.

6. User interface

Uses the lcdui midlet package and employs visual components specific to the J2ME platform to represent application data. Uses MVC design pattern for easier screens manipulation and maintenance.

4.2.1.3 Server

This component is meant to provide a way for any client to be able to query the Axml Mobile peer through a SOAP service.

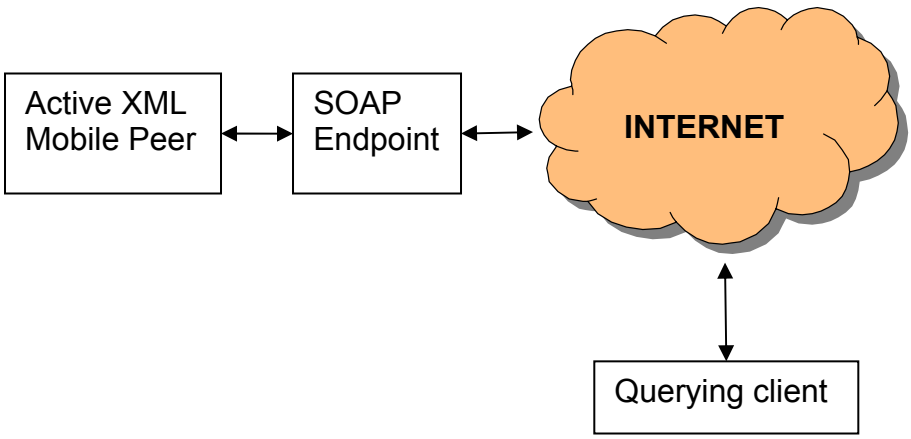

Figure 6

The server component of the Axml Mobile Peer is essential for the data integration feature of the technology. The SOAP endpoint is decoupled from the mobile component. It is usually installed on the same server that the proxy runs on. However this is not necessary since the SOAP service that listens for queries makes an XML-RPC connection to the proxy in order to transmit the message.

4.2.2 The proxy

It runs inside an application container that is in charge of managing its life cycle and dispatching outside requests to different threads.

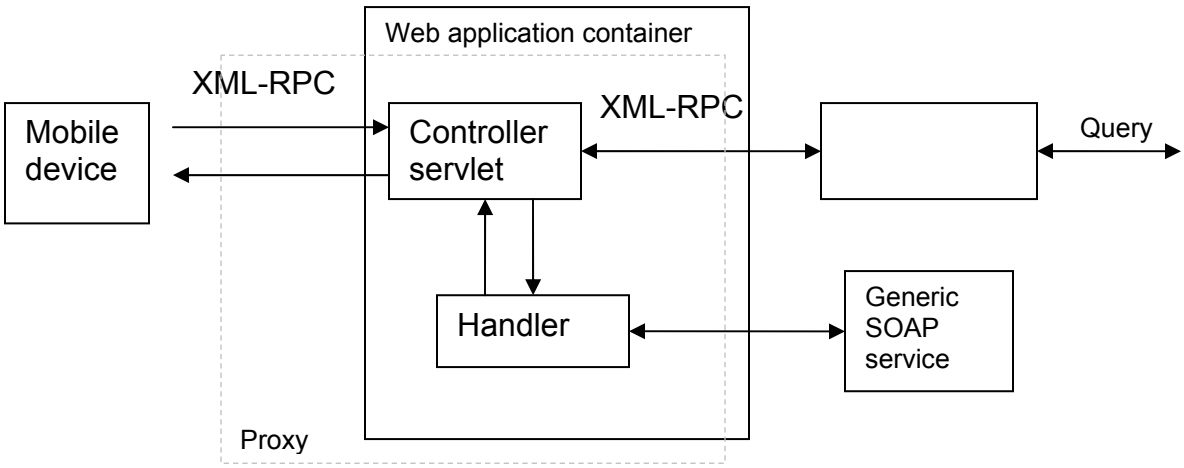

Figure 7

Mediates transaction between the mobile component and the outside world. It is built as a web application and this brings a few advantages:

- it can use the standard HTTP protocol to receive and send XML-RPC envelopes;
- uses the default HTTP port 80, this way no problem with firewalls can arise;
- The Web application server is in charge with multithreading and dispatching various requests to different threads;
- it is portable to another operating system and/or other application server;
- ease of debugging.

5. Implementation

With a Java technology solution, software applications run locally on the client device.

Within the framework of J2ME technology, the various types of consumer devices are grouped into basic categories -- set-top boxes, screen phones, wireless devices (pagers and cell phones), and so on -- with a "profile" for each category, specifying a set of category-specific APIs and a configuration that consists of a minimum set of APIs and a Java Virtual Machine.

The Connected Limited Device Configuration (CLDC) is composed of the K Virtual Machine (KVM, described in detail in the next paragraph) and core class libraries that can be used on a variety of devices such as cell phones, pagers, PDAs, and retail smart-card readers. The Mobile Information Device (MID) profile includes APIs covering the user interface, persistent storage, security and messaging specifically for cellular phones and pagers.

CLDC refers to a configuration specific for the mobile phone, the user only has to interact with a known set of classes that don't change no matter the specific hardware issues of the mobile device. A device that fits the CLDC configuration typically has 160Kb-512Kb heap available and more than 8Kb persistent memory.

The AXML Mobile Peer is built on Java 2 Micro Edition (J2ME), using CLDC as configuration and MIDP as base profile.

5.1 Mobile component.

The mobile component of the Active XML mobile peer is made of *client*, *xml repository* and *server.*

5.1.1 Client

The client component depicted in the "General architecture" is made out of "Inspector", "Web Service Reference Manager", "Active XML Document Manager", "Mini XPath Engine", "Presentation module" and the "User interface".

All these components are linked together and managed by a controller (the name has nothing to do with the MVC controller) implemented in the MController object.

The MController's functions are:

- keeping track of all the documents currently available in the system. This table is implemented as a Vector;
- it is the only link to the user interface classes; this minimizes complexity and provides a simple design. Especially needed because, unnaturally, many object fields are not provided with accesors (in order to minimize class size) so they must be declared either friendly or public;
- installs the Inspector and manages its functioning, provides methods for stopping and restarting it ready to be called from the user interface.

5.1.1.1 Inspector

Decides when execution of services is necessary, checks if one service is no longer valid because based on its "frequency" attribute it should be executed.

The RefreshDaemon class is responsible for triggering service calls when necessary.

There are two possible approaches here:

1. using a timer for every service call such that service calls are independent in the moment of calling. Theoretically all services can be called in the same time;
2. using only a timer, the services are called sequentially. The daemon is activated periodically and checks the discovered services.

Every time a service is checked by this daemon a countdown is decremented. When it reaches 0 the service is executed.

Given the resource requirements of the target platform only one timer is used. Should more than one timer be created, how does this affect the existent resources? At each timer creation a thread is created "stopped" on a monitor condition (Object.wait(milliseconds)). When the sleep period expires, the timer

executes the job (the class which extends TimerTask). But this takes cpu time and resources which we don't have.

Another fact should be taken into account.

Only 5 HttpConnections are available (an exception is thrown if we try to use more). XmlRpcClient wraps around this connections so we cannot access them. Having only one Timer is simpler.

The Timer activates periodically a class (RefreshDaemon.class) which extends TimerTask. The daemon is activated periodically and checks the discovered services. Why I chose this approach rather than scheduling every service?

- ease of implementation;
- the synchronization mechanisms are not so sophisticated as we would need if we were to schedule every service;
- ease of debugging.

For every document a new thread is started that inspects every service that exists in a particular axml document. A barrier is installed that waits for all working threads to end. This has two important consequences:

If one of the threads that inspects a document is blocked waiting for the document's lock to be unlocked for a long time everything stalls for that thread.

- services are no longer checked in a new cycle because the barrier expects all threads to end before moving forward. This issue has been fixed and it is more thoroughly explained in the synchronization section;
- the inspector is run by a thread every one second. However if one of the services takes more than one second to execute the timing of service execution is dephased forward. A work-around that I've applied is to measure how much time it takes for all threads to reach the barrier. If that time is longer than 1 second, more than a unit is substracted from every service's countdown-to-execution counter. The measurement is performed like this:

```
long startTime = System.currentTimeMillis();
```

```
performOperations();
long duration = System.currentTimeMillis() – startTime; int
decrement = duration/1000>0?duration/1000:1;
```

5.1.1.2 Web Service Reference Manager

To every web service which has a reference in an axml document an MService object is available which the logical description of that service is. An MService object contains both the attributes and parameters of a web service mapped to object fields and methods to work with them.

Types of services

Because in an Active XML document more than one service reference can coexist at a time it comes natural that some sort of schema to enforce an order of execution should exist.

Regarding the moment of their execution the services can be split in:

- immediate mode – this are scheduled for execution using the frequency attribute;
- lazy mode – this kind of services upon execution return results that are needed by another service to execute correctly; for example a service may reference through xpath the results of another service. This kind of service is executed only when its results are needed.

Concrete services

A concrete service is one that only has value parameters and no xpath parameters. Why is important to differentiate between concrete services and not concrete services? Because the concrete type can be executed without checking for any service that should be executed before it. Its name (not id) can be referenced by other services via the “followedBy” attribute which forces the latter to be executed before the former. However in the concrete service case there is no need for more in-depth analysis of the document.

In order to come up with a scheme for a static analysis of the document we start with these assumptions:

- results of a web service invocation are placed underneath the <axml:sc> element;
- nodes referenced by an xpath expression that is parameter of a web-service might exist or not in the axml document;
- the xml fragment that is result of a web service invocation is yielded by the closest web service on the horizontal (its node index is lower than the last referenced existent node in the xpath expression, but the highest when comparing to other web service nodes on the same level) and vertical (its node is placed above or on the same tree depth level than the last referenced existent node in the xpath expression).

From what is stated above it is obvious that a cycle can occur when processing service references. However there is little we can do about it.

Lazy analysis of document

Taking into consideration the assumptions stated above an algorithm was implemented to detect all dependencies of a particular service.

Definition: a service A is dependent on another service B if parameters of service A refer results of executing service B.

The dependency relation between services can be of two types:

- explicit – enforced by the axml:sc “followedBy” attribute; for example

<sc name=”A” followedBy=”B”/>

<sc name=”B”/>

In the above example the semantic of the “followedBy” attribute is that service B is to be executed immediately after A finishes. This explicit relation is determined the first time the service is executed. Why here and not when the service is first parsed? Because when the service is parsed there is no consistent image of all the services present in the Active XML document;

- implicit – service B refers through its parameters potential results of service A execution.

Lazy analysis of document is the algorithm used to determine implicit dependencies. Non-concrete parameters are given as xpath expressions’, thus determining dependencies is done by extending the xpath algorithm.

Lazy analysis algorithm applies to every xpath parameter of a web service.

```
Input: xpath parameter
Output: Vector of service references on which the current
depends.

0. Parse the xpath parameter into location paths
1. For every location path
    2. Save previous vector of result objects;
    3. Filter current vector of result objects based on
current location path;
    4. Search in the vector of result objects saved at 2.
the <axml:sc> element;
        5. If found
            6. Filter the vector of service references
based on the current xpath evaluation result
    7. Current vector of result objects is assigned the
result of step 3.
8. End for.
9. return last vector of valid service references
determined at 4.
```

In words, while evaluating the xpath expression we are also searching the current context for matching web service references. Then we check the latter to fulfill the conditions stated in the *Types of services* paragraph against the xpath evaluation.

Because an xpath parameter is already evaluated once when doing the *lazy analysis* in order to optimize web service execution I have come up with a scheme to not evaluate the parameter when it is not necessary. This condition is true when, although the service has xpath parameters, it doesn't depend on any other service. In the latter case the result of xpath evaluation is taken directly from the *lazy analysis* algorithm. Because the policy is conservatory if the analyzed service depends on at least on other service (implicitly or explicitly) its xpath parameters are always evaluated.

Algorithm for discovering and calling new services

1. the source xml (axml) document is parsed for services;
2. every time a new service element <axml:sc> is discovered it is analyzed and after all its parameters are parsed a new entry is added to the document's service table;
3. the RefreshDaemon inspects periodically the service table. If a new service has to be executed, it delegates job to the ServiceExecutor module.
4. Once the ServiceExecutor has done its job the result, of calling the service is again parsed. As a result a new web service entry might be added to the service table (a service that was obtained as a result of calling another web service).

2. In this step a logical image of the web service reference is built in memory.

Following attributes are parsed:

- name – a conventional name for the service. It is mostly when referring a service by using "followedBy" attribute;
- followedBy – used to enforce an explicit dependency relation;
- serviceUrl – endpoint location;
- serviceNamespace – xml namespace to assign to service reference;
- methodName – operation to be invoked;
- signature – URL to service WSDL;
- doNesting – *true* or *false*;
- mode – *replace* or *append*;
- id – it is a unique alphanumeric combination (not only on the local peer, but also on the Internet). When the service is parsed if it doesn't have an *id* attribute one is assigned to it. Class ServiceIdGenerator manages service id's, taking care that a unique one is generated every time;
- frequency – two important flags are managed by the code sequence that parses the execution interval (frequency). If the parsed format is unrecognizable (due to syntax errors) the *isSchedulable* flag is set to

false. In all other cases *isSchedulable* is true. *callOnce* forces one and only one execution of the service whatever the value of *frequency*. If an error occurs *frequency* is set to *null*.

3. When a service is ready to be executed the following steps are performed:

```
If this is the first time the service is executed and the
service is not concrete
    Search discovered services for one with
    followedBy==this.@name

    If found
        put this service reference in the nextToExecute
        field of ancestor;
    End if

    Perform Lazy analysis to determine implicit dependency
    relations;

    If any dependencies are found or current service has
    ancestor
        Clear xPathStatic field from every xpath parameter
         (reevaluate every time)
    End if
End if
Run all dependencies of this service;
Build an xmlrpc envelope to be addressed to the proxy;
Invoke xmlrpc service and receive results.
```

4. In this step results from a previous xmlrpc call are handled. First of all the document lock is set.

If previously executed service had *mode* attribute set to *replace* than two things happen:

- all elements with *axml:origin* (read below) attribute set to previously executed service *id* are deleted;
- all services that resulted from parsing the results of calling the previous service are deleted recursively (a tree like structure is maintained, every service has a reference to his father).

The results are included in the Active XML document underneath the <axml:sc> element. From now one the document lock is unlocked.

Finally if this service has a *followedBy* attribute that has materialized in a *nextToExecute* field, the latter is immediately executed.

5.1.1.3 Active XML Document Manager

Every Active XML document is stored in an MDocument.

The MDocument has a few important functions:

- synchronization: concurrent access from different threads to the same document either in main memory or in persistent memory;
- keeps a table of all services. This is implemented as a Hashtable in which the key is the id of service and the element is the reference to the service;
- manages inclusion of web service materialization results in the source document;
- implements a layer of abstraction to writing and reading the document from the persistent memory (uses the facade provided by the “XML repository” component of the client).

Synchronization

Because more than one thread accesses the axml documents and the services that they contain some mechanism for synchronization is in order. A lock object implemented using monitors is used. Lock object has two methods lock() and unlock().

Every MDocument object contains two Lock type objects: memoryLock and storeLock(). The former protects the memory image of an axml document and assures its consistency, while the latter does the same thing to the persistent storage of the axml document. Related to the storeLock there are two ways in which to store the documents currently found in device’s volatile memory:

- writing data to persistent memory at specific intervals of time;

- writing data only when the midlet application is either terminated of paused.

Because I chose the second method the storeLock is obsolete since no two threads access the store at the same time.

The memoryLock is locked whenever some operation is performed on the axml document. Because some operations like modifying the xml are user dependant they may take quite a while before them end. This is why a flag was introduced that announces the Inspector that one document is under way to be locked. If the flag is set, the Inspector gives up and doesn't try to lock that document (operation that would block indefinitely, until the user finishes modifying the xml, for example). To manipulate this flag a synchronized getter and a synchronized setter are provided.

```
public void run()
{
    //first check if another thread performs a long
    //operation and we shouldn't block waiting for it
    //to end.
    if(!mDocument.isWillBlock()) {
        //todo: lock interval should be made as
        mDocument.memoryLock.lock();
        check(mDocument);
        mDocument.memoryLock.unlock();
    } else {
        System.out.println("Another thread performs a long
        operation on "+mDocument.name);
    }
}
```

5.1.1.4 Mini XPath Engine

XPath is an open standard. Its desired purpose is to serve as a language for addressing parts of an XML document, it was designed to be used by XSLT.

The path to an XML node is fully specified by an XPath expression (a path). This path is very similar to a file system path.

XPath models an XML document as a tree of nodes. In the standard XPath there are different types of nodes, including element nodes, attribute nodes and text nodes. However the version built into this project has only support for elements and text nodes (the attribute node is cast to a text node). The remaining of this paper "xpath" refers to the version this project implements and not the XPath standard if not otherwise specified.

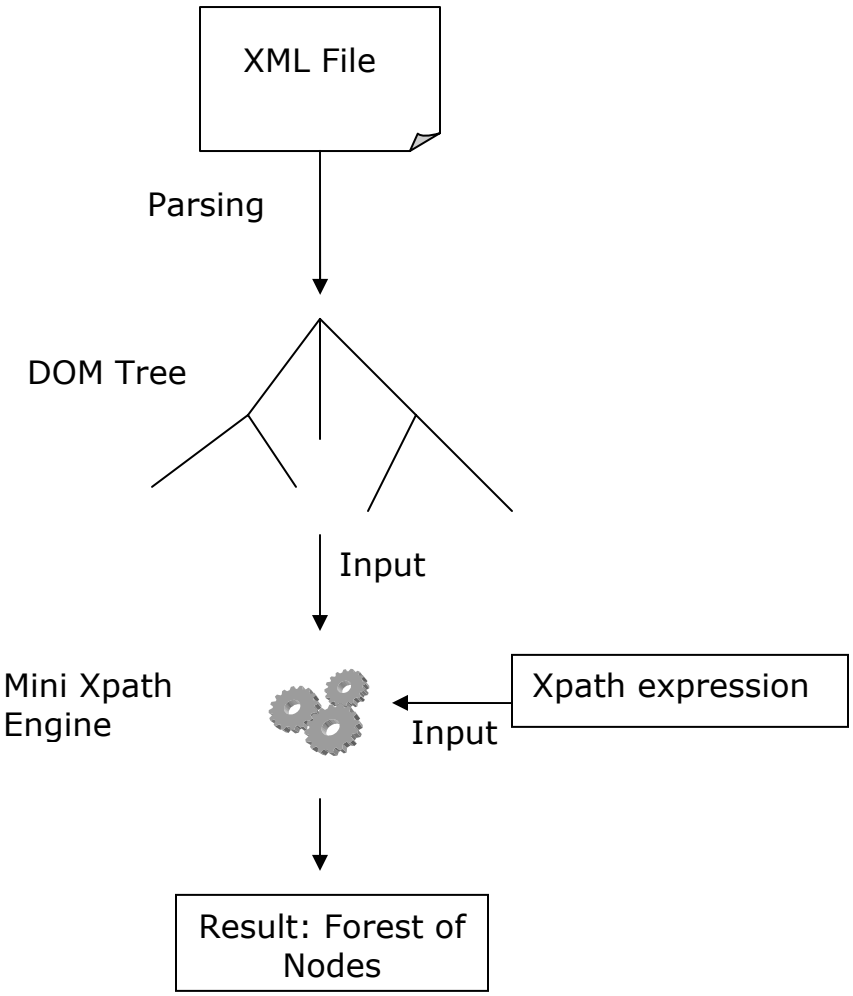

Figure 8

The basic construct in XPath is the expression.

An expression is evaluated to yield result objects, which may have one of the following types:

- node (actually an element);
- string (a sequence of characters).

Every xpath expression has a context within it executes. When evaluating an xpath expression by its own its context is the document's root node.

However an xpath expression can be evaluated as part of an XSL, hence its context is defined by the xsl transformer and it is made of a set of start nodes.

The current xml namespace is not passed to the xpath processor although the latter supports xml namespaces.

Location path

One important kind of expression is a location path.

A location path selects a set of nodes relative to the context node.

The result of evaluating an expression that is a location path is the node-set containing the nodes selected by the location path. This xpath implementation specifies that an xpath location path cannot contain recursive expressions. If this feature were to be inserted in the project, the xpath expression parsing process would have been more costly.

Every location path can be expressed either straightforward or using an abbreviations for the most common cases.

There are two kinds of location path:

- relative location paths;
- absolute location paths.

A relative location path consists of a sequence of one or more location steps separated by /. The steps in a relative location path are composed together from left to right. Each step in turn selects a set of nodes relative to a context node. An initial sequence of steps is composed together with a following step as follows. The initial sequence of steps selects a set of nodes relative to a context node.

Each node in that set is used as a context node for the following step. The sets of nodes identified by that step are unioned together.

The set of nodes identified by the composition of the steps is this union.

An absolute location path consists of / optionally followed by a relative location path.

Location step

A location step has the following parts:

- an axis, which specifies the tree relationship between the nodes selected by the location step and the context node;
- a node test, which specifies the node type and expanded-name of the nodes selected by the location step;
- a predicate, which uses arbitrary expressions to further refine the set of nodes selected by the location step.

The syntax for a location step is the axis name and node test separated by a double colon, followed by at most one expression in square brackets.

Axes

The following axes are available:

- the child axis contains the children of the context node;
- the descendant axis contains the descendants of the context node; a descendant is a child or a child of a child and so on; thus the descendant axis never contains attribute or namespace nodes;
- the parent axis contains the parent of the context node, if there is one;
- the attribute axis contains the attributes of the context node; the axis will be empty unless the context node is an element.

Node tests

Every axis has a main node type. If an axis can contain elements,
then the principal node type is element; otherwise, it is the type of
the nodes that the axis can contain:

- for the attribute axis, the principal node type is attribute;

- for other axes, the principal node type is element. A node test * is true for any node of the principal node type. Currently this works only for axes that have as principal node type the element.

The node test text() is true for any text node.

For example, child::text() will select the text node children of the context node. A node test node() is true for any node of any type whatsoever.

Predicates

A predicate filters a node-set with respect to an axis to produce a new node-set. For each node in the node-set to be filtered, the PredicateExpr is evaluated with that node as the context node.

A pseudo-grammar of the accepted xpath syntax

```
xpath_expression ::= locations
locations ::= (locations '/' location) | location
location ::= (axis '::' node_test '[' predicate ']') |
                ( axis '::' node_test ) |
                ( node_test '[' predicate ']') |
                ( node_test ) |
                ( '.' ) | ( '..' ) | ( '*' )
axis ::= 'attribute' | 'descendant' | 'parent' | 'child'
node_test ::= ( element_namespace ':' element_name ) |
                ( element_name) |
                ( 'node()' ) | ( 'text()' ) |
                ( '@' attribute_name )
predicate ::= node_index |
                ( expr_member '=' expr_member ) |
                ( expr_member '>' expr_member ) |
                ( expr_member '<' expr_member )
node_index ::= [0..9]+
expr_member ::= ( '@' attribute_name ) ||
                ( "" string "") ||
                ( string )
```

Given a DOM tree and an xpath expression the mini xpath processor computes a set of objects. These objects are either kdom Element or String. Processing instructions and other entities are ignored.

The minixpath engine is used throughout the project in the following ways:

- inside an xsl to select relevant nodes of the axml document;
- it performs a query on the document when such a query is sent to the phone.

This part of the project has evolved into a subproject and is now hosted on http://minixpath.sourceforge.net.

Algorithm

The used algorithm is split into three units. It handles the expression as a hole, the location path and the predicate.

Expression

1. the xpath expression is parsed into location paths;
2. the result set is assigned all the nodes in the DOM tree;
3. for every location path contained in the expression start refining the result set by calling the Location path algorithm with parameter the result set from the previous step.

Location Path

```
Parse the location path into axis, node test and predicate;
If the axis is equal to "child" or  "descendant"
    For every node in current context
        For every childe of the current node
            If node test matches the current child node
                add it to the result set
            endif
            If axis equals "descendant"
                add all matching descendants to the result
                set
            endif
        endfor
    endfor
endif
if the axis is equal to "parent"
    For every element in context node set
        Add node's parent to result set
```

```
    Endfor
Endif
If the axis is equal to "attribute"
    For every element in context node set
        Extract attribute from node test and put its value
        to result set
    Endfor
Endif
```

Every location path acts as an additional filter to the already found results.

The *context node set* is a collection containing nodes found in a previous iteration (processing of a location path). Every item is processed with regards to the current location path. If it is accepted it is put in the *result node set*. Evaluation of the next location path will receive as *context node set* the current *result node set*.

The *context node set* is initialized to a set of nodes. In an xslt reference to an xpath expression, for example, is influenced by the results of evaluating previous expressions. Standalone functionality of the mini xpath engine implies initializing the *context node set* the root of the xml document.

5.1.1.5 Presentation engine

In order to present the results to the user I developed a presentation layer which is composed (as you can see in the diagram) of two main components: the transformer engine and the rendering engine.

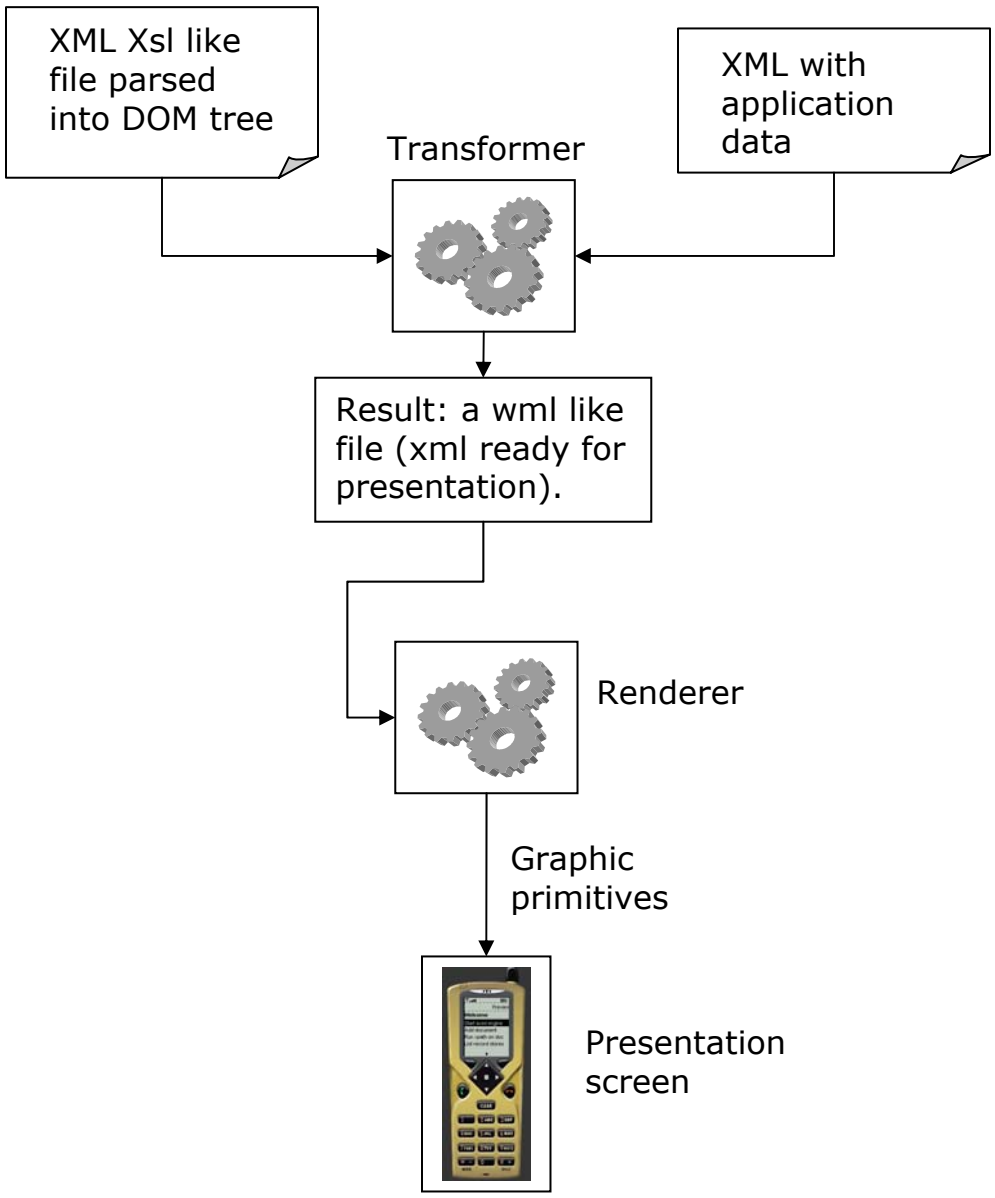

Figure 9

Benefits of a presentation engine

At the first glance these architecture is much too complicated and a bit of overhead is involved. Presentation modules present in this project implementation only a subset of the standards used. Therefore there isn't too much overhead present.

The goal is to have results properly extracted and presented on the user interface (mobile phone or pda screen). The advantages that this approach yields are noticeable:

- very flexible, references to data to be presented are not hardcoded in the program;
- xslt transformer engine is general and its output xmls are not necessarily for presentation purposes;
- xslt can be used to transform an xml into another one.

When arriving at this step the axml which is stored in memory contains relevant data. But its format is inappropriate for presentation on the device screen.

5.1.1.5.1 The transformer

Its input are a "stylesheet" (a model of how data should be organized) and the source AXML which contains application data. Its output is another xml (this time it is called a wml) which contains markup ready to be interpreted in order to render the results to the device screen.

The input is a stylesheet called XSL (eXtended StyLesheet). Its language is is a reduced set of the original XSLT language which is generally used at transforming XML documents into other XML documents.

XSLT is designed for use as part of XSL, which is a stylesheet language for XML. In addition to XSLT, XSL includes an XML vocabulary for specifying formatting. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.

We use a simplified version of the XSLT language because these are the needs and because the device lacks resources for additional XSLT processing.

This implementation cuts standard xslt down to two instructions: **generating text** and **repetition**. Although it sounds poor comparing to standard XSLT it is all we need.

Generating text

In order to generate text a special element is inserted that belongs to the xsl namespace:

<xsl:value-of select = string-expression />

The xsl:value-of element is instantiated to create a text node in the result tree. The required select attribute is an expression; this expression is evaluated and the resulting object is converted to a string as if by a call to the string function. The string specifies the string-value of the created text node. If the string is empty, no text node will be created. The created text node will be merged with any adjacent text nodes.

Example

For this xml:

```
<albums>
    <album>
        <title>Dark side of the moon</title>
        <author>Pink Floyd</author>
    </album>
    <album>
        <title>Greatest hits</name>
            <author>Queen</author>
     </album>
</albums>
```

the instruction

<xsl:value-of select = "/albums/album[1]/title"/>

would yield: "Dark side of the moon".

Repetition

In order to repeat fragments of xml another special element is inserted:

<xsl:for-each

select = node-set-expression>

<!-- Content: (<xsl:value-of>, markup) -->

</xsl:for-each>

When the result has a known regular structure, it is useful to be able to specify directly an xml pattern for selected nodes.

The xsl:for-each instruction contains a pattern, which is instantiated for each node selected by the expression specified by the select attribute.

The select attribute is required. The expression must evaluate to a node-set. The template is instantiated with the selected node as the current node, and with a list of all of the selected nodes as the current node list.

Example

For this xml:

```
<albums>
    <album>
        <title>Dark side of the moon</title>
        <author>Pink Floyd</author>
    </album>
    <album>
        <title>Greatest hits</name>
        <author>Queen</author>
    </album>
</albums>
```

the xslt instructions:

<xsl:for-each select = "/albums/album">

Title: <xsl:value-of select = "title"/>

Author: <xsl:value-of select = "author"/>

</xsl:for-each>

would yield the following xml:

Title: Dark side of the moon

Author: Pink Floyd

Title: Greatest hits

Author: Queen

5.1.1.5.2 The renderer

The result of the transformer is another XML, a WML. WML stands for Wireless Markup Language and it is an open standard. It is a mark-up language inherited from HTML, but WML is based on XML, so it is much stricter than HTML. WML is used to create pages that can be displayed in a WAP browser.

Pages in WML are called DECKS. Decks are constructed as a set of CARDS. To render a WML you would normally need a wap micro-browser.

A Micro Browser is a small piece of software that makes minimal demands on hardware, memory and CPU. It can display information written in WML, a restricted mark-up language.

The j2me application (midlet) cannot access the device micro-browser, even if one is built into the phone. Hence this project includes a proprietary minimal renderer for WMLs.

One has two available methods of accessing the device screen:

- low level, which means the programmer has control over the screen at pixel level, hence more flexibility is available;
- higher level, appropriate for forms, or longer screens of text that need some form of navigation.

The renderer uses the second approach, also a much simplified syntax as opposed to the usual WML.

WML syntax implemented

As enumerated above two essential components of the WML syntax are a) the Decks and b) the cards.

a) WML pages are often called "decks". A deck contains a set of cards. A deck is wrapped by a <wml> element.

b) A card element can contain text, markup.

The card element can contain one attribute: "title". This will appearon the first line of the screen in which the current card is rendered. Every card is rendered in its own screen. However the renderer takes care to set up a sequential navigation system between cards.

Text is rendered on screen by including it either in a <p> element or in a <table>. A table has rows wrapped inside <tr> and rows have columns wrapped inside <td>. Important to note here is that the "columns" attribute to the<table> element is not needed (otherwise, its presence is compulsory should the renderer be a usual micro-browser).

DTD of the supported WML is:

```
<!ELEMENT card (table | p)>
<!ATTLIST card title CDATA #REQUIRED >
<!ELEMENT p (#PCDATA)>
<!ELEMENT table (tr+)>
<!ELEMENT td (#PCDATA)>
<!ELEMENT tr (td+)>
<!ELEMENT wml (card+)>
<!ATTLIST wml xmlns:xsl CDATA #REQUIRED>
```

5.1.1.6 User interface

The user interface is based on the MVC design pattern.

The Model-View-Controller (MVC) is a design pattern which links efficiently the user interface with object oriented programming. This architecture is frequently used when programming in Java, C++ or Smalltalk because it allows reusing the source code, thus reducing the development time of application which have a use interface.

The model-view-controller architecture is made of three main components:

- the Model component, which is the business logic of the application and high level classes associated with it; the core of the application. This maintains the state and data that the application represents. When significant changes occur in the model, it updates all of its views;
- the View component is a collection of classes which represent the user interface (every object that the user sees on screen and are interactive); The user interface which displays information about the model to the user. Any object that needs information about the model needs to be a registered view with the model;
- the Controller component which represent the classes which allow communication between classes in Model and View components.

Application flow is mediated by a central Controller whose role is to pass the requests to the specific logic classes.

The Model component is made up of classes which store the logic and system state. After the logic is executed on the Model component layer, control is passed back to the Controller which transmits it to the View layer.

This way the MVC model makes possible separation between business logic and results presentation logic. This separation allows every component to be reused and assures an easier maintenance of the whole application.

General MVC diagram:

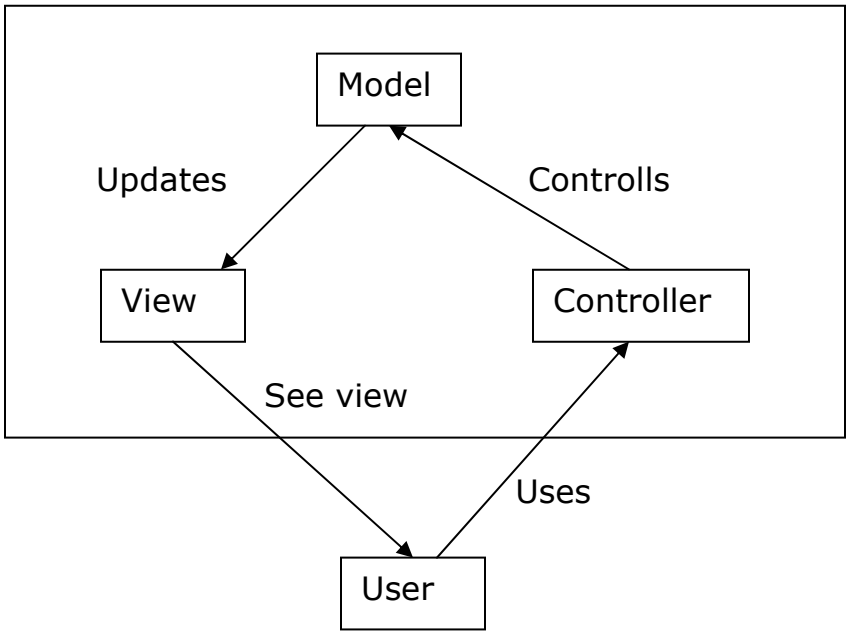

Figure 10

MVC components in Active XML Mobile application

Any class can be the controller. The condition is that it implements UIController interface from the org.axml.mobile.midlet package.

The view is played by different classes which extend components of a display (see UML diagram) and their constructors are passed references towards the class which implements UIController (the controller).

The model is the class MController which controlls what happens in background (how and when axml documents are updated, e.t.c.).

The controller

The controller implements the UIController interface.

By implementing the UIController interface, the MainMidlet can expose certain methods to the individual screens. However we could instead pass the MIDlet as a parameter to the screens that are part of the View component, we don't want to interfere with the normal MIDlet lifecycle.

Within the MainMidlet, the nextScreen() and lastScreen() methods have been used to maintain the information about the currently visible screen. These methods use the java.util.Stack object to maintain the display state.

The application will push the currently displayed Displayable object to the Stack prior to displaying the next screen when nextScreen is called.

When calling lastScreen() the application will pop the previously displayed Displayable object from the Stack and set the display to show that screen. currentScreen() returns the current displayed screen.

getAxmlController() returns a reference to the MController, instantiated in MainMidlet. This reference is used to access different components of the business logic layer. The getListIndex and setListIndex methods are used to communicate the screens. Here's how: we use this to share the index of the element chosen from a list across multiple screens. For instance the xpath inspector calls getListIndex in order to find out on which document to work. That index is set in the "list documents screen".

The view component architecture

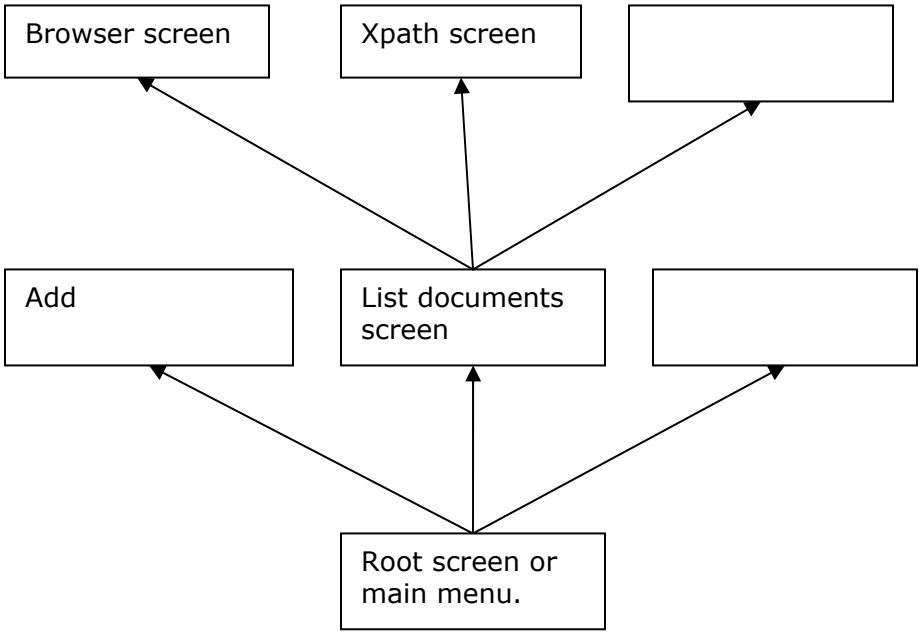

Figure 11

A Stack container is used to store the already visited screens.

This can be done because, as you can see from the above diagram our user interface architecture is a tree and not a graph. That means that there is only one entry point into each screen.

Root screen

The first screen that is visible to the user is the RootScreen. The RootScreen extends from the javax.microedition.lcdui.List object and implements the CommandListener interface.

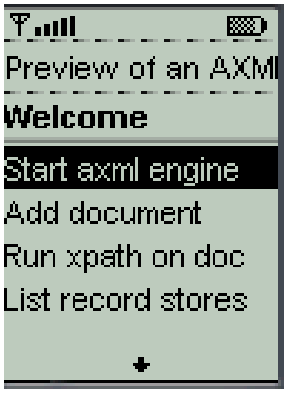

Figure 12

The constructor is passed a parameter that contains a reference to the Controller as an argument, which will be stored as a private variable for future use.

public RootScreen(UIController controller)

The RootScreen extends the lcdui List object which presents a basic way of organizing items in a user menu. The RootScreen will listen and handle any commands that are initiated when it is displayed. Therefore, it calls the setCommandListener() method with itself as a parameter.

this.setCommandListener(this);

The event-handling infrastructure in the MIDlet architecture, briefly discussed in the RootScreen example, lets one handle events generated from the MIDlet Screen objects. For the RootScreen example, the user is presented with a list containing these items:

- “Start axml engine” if the engine wasn’t not started and “Stop axml engine” if the engine was started;
- “Add document”;
- “Run xpath on doc”;
- “List record stores”;
- “Dump document”;
- “Presentation”.

When the user selects one of the items by using the up/down arrows and then clicking the action button, the commandAction() method will be called.

public void commandAction (Command c, Displayable d);

The commandAction() method captures the events generated by an IMPLICIT list. The first step involves retrieving the list from the display, which in this example is done through the UIController interface. The selected item can be retrieved from the list to allow the application to perform logical operations to determine the proper course of action. In this case, the next display object will be instantiated and passed to the controller's nextScreen() method.

AddDocScreen

This screen is reachable from the main menu via the “Add document” option.

Figure 13

The AddDocScreen class (which is the View in the MVC design pattern) extends (or inherits) another *lcdui* component: Form. The label “Enter url” is a

StringItem. After the correct URL is entered and the Load button is pushed the new document is loaded and parsed hence new service calls are added to the specific tables. After that method lastScreen of the UIController interface is called. These leads to popping the last screen from the stack, thus, in this case we return to the main menu.

XpathScreen

This screen is for debugging purposes because the results of evaluating an xpath expression will be shown in the system console hence visible only if the application runs in an emulator. This is one of the screens which are preceded by the “ListDocsScreen”. This is very natural since we need a document against which to run our xpath expression. Xpath screen corresponds to the “Run Xpath on doc” option from the main screen (RootScreen). Below, a capture of the ListDocsScreen:

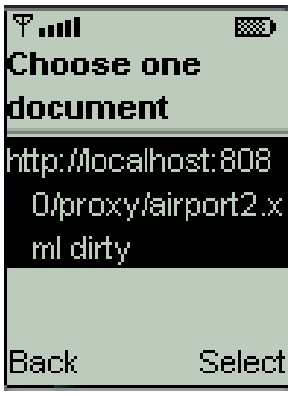

Figure 14

A reminder: a document’s name is its URL. The ListDocsScreen class extends a List, a very similar method of building an option list to the one used in RootScreen.

Other screens that also use the ListDocsScreen:

- Dump document screen;
- Browser screen;
- Modify screen.

To facilitate code reusage of the ListDocsScreen, the constructor is:

public ListDocsScreen(UIController uiController, String dataUser);

The first parameter is obviously the reference to the UIController. However the second parameter is more interesting. We use it in order for the ListDocsScreen to delegate control (using the controller of course) to a specific screen of our choosing. The “dataUser” is the name of the screen class. A switch statement decides to which screen to delegate. To note here that in the switch statement, the screen names are hardcoded.

Another way of doing this would have been:

```
try {
    Class.forName(dataUser).newInstance();
} catch (InstantiationException ie ){}
//catch (IllegalAccessException iae) {}
```

BUT we want to pass the uiController parameter to that instance. The only way to keep it general would've been to make a new interface with one method public void setUIController(UIController ....) but this means wasting precious resources for a midp device.

Finally the XpathScreen:

Figure 15

The actual XpathScreen only contains one interactive component, a TextField whose constraint is ANY which means that any characters are valid for input.

After pushing the “Evaluate” button the desired xpath expression is evaluated and its results are shown in the system console.

This screen uses as model (the Model component in MVC design pattern) the xpath evaluator module.

ListStoresScreen

This screen corresponds to the “Run Xpath on doc” option from the main screen (RootScreen).

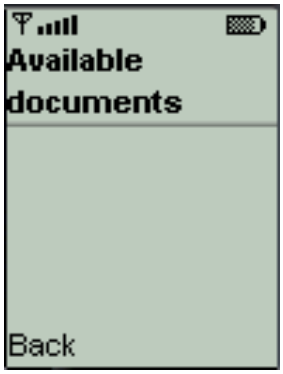

Figure 16

This screen provides a way to see which documents are stored in the Record Store System

DumpDocScreen

Its purpose is to show the contents of an axml document in the system console, hence is only useful in the emulated environment. This screen is also based on the ListDocsScreen.

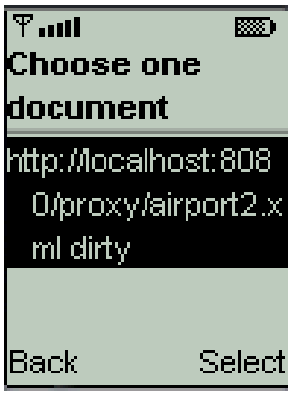

Figure 17

After selecting a particular document, the DumpDocScreen is:

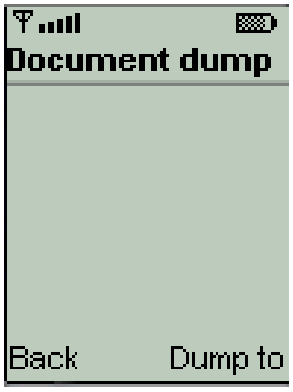

Figure 18

To dump the contents of a file, one has to push the “Dump to” button.

BrowserScreen

This is the central part of the presentation. Its entry point is the “Presentation” item in the main menu (RootScreen). As usual the document to present is chosen in the ListDocsScreen.

After choosing one document the next screen is:

Figure 19

Once this screen is shown that means that the presentation is ready. After pushing the “View presentation” button the actual presentation starts as a sequence of slides.

Example:

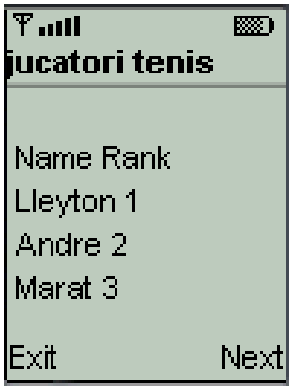

Figure 20

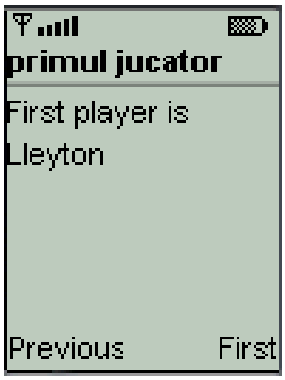

Figure 21

The presented slides are the first and the last. Every slide (screen) is actually a form. By taking into account their relative position in the Vector, buttons are assigned to every form. The key thing is that there is only one CommandListener for all the frames (in fact a controller).

This means that the MVC pattern is applied even in the presentation layer.

Modifying the XML screen

The user has to be provided with a way to modify the existing Active XML document in order to interact with the existent data. Because we have to deal with an xml, that is tree-like structured data, a convenient method to access the node were interested in is in order. The ModifyXmlScreen class fulfills all this desiderates. However only attributes can be modified because of the way kDOM stores the elements of the XML.

Navigation is “explorer” like. A node (actually an element) that can be expanded more is prefixed “+“, a leaf is prefixed by “-“.

Every line in the unfolding tree is logically represented by a TreeEntry inner class. This provides the essential elements for rendering that entry to the screen.

5.1.2 The XML repository

On the implementation level the XML repository's central part is the *Persistent Memory Manager* which also gathers around it the *File retriever* and *the DOM Cache* as depicted in the *General architecture* section.

5.1.2.1 Persistent Memory Manager

An xml repository is essential in order for the application to work even if the mobile device is not connected to the Internet.

The Mobile Information Device Profile provides a mechanism for MIDlets to persistently store data and later retrieve it. This persistent storage mechanism is modeled after a simple record oriented database and is called the Record Management System.

The Record Management System is organized into "stores" which are split into "records".

Two types of xml files are stored in the Record Store:

- Active XML documents which contains web service references;
- XSL files which drive the presentation and are essential if a presentation of the stored data is to be made while the device is offline.

A reminder is in order here: every axml document is named by its URL initial location. However Record store names are case sensitive and may consist of any combination of up to 32 Unicode characters. This means that we cannot store every document in a store named with its conventional name because there are simply too many characters in a URL string.

Another reason for which we need a central place in which to store document names is that we need a mechanism to open just those stores that we are interested in.

A "repository" named store is created that contains all the names of the documents currently stored on the peer. The method "retrieveDocStoreIndex" from the NamesRepository class returns an index that acts as a pointer to the record store where the actual document is stored. For example if name

"http://localhost:8080/airport.xml" is stored in the "repository" store in record "3", then the document is available in record store "3", record 1.

To find a specific record, the Record Management System has a mechanism named RecordFilter. A filter class must implement the RecordFilter interface and define the match method.

This is used when searching for a specific record store.

Example of a filter:

```
public static class MyFilter implements RecordFilter
{
    String name;

    public MyFilter(String name)
    {
        this.name = name;

        public boolean matches( byte[] recordData )
        {
            return(name.equals(new String(recordData)));
        }
    }
}
```

The above filter is used to test if a document exists in the repository, hence it already has an instance stored in one of the record stores of the application. Another filter is used to find all the records that end in a user defined pattern.

Synchronizing access to repository

No locking operations are provided in the Record Store Management API. Synchronization is provided by the Persistent Memory Manager. Record store implementations ensure that all individual record store operations are atomic, synchronous, and serialized, so no corruption will occur with multiple accesses. However the specification states that if more that one thread tries to access the same Record Store then it is the programmer's responsibility to set up proper synchronization mechanisms.

In every MDocument class a "store" Lock field exists that is directly related to the Record Store operations. Every time operations with the Record Store

Management system are under way the lock is set and it is unset only when all the operations have ended.

A lock is necessary because the record store is susceptible of being accessed by more than one thread.

All accesses to the same record store are serialized, although, for example, read-read operations could have been parallelized.

Besides protecting every record store that contains an axml document, the repository itself is protected by a Lock. That is the NameRepository class contains a Lock field which enforces serialization when dealing with the "repository" store that contains the names of all stored documents.

5.1.2.2 File Retriever

The File Retriever is a facade for the Xml Repository, has public method that facilitate access to an entire file stored in the Xml Repository whilst the latter accesses data at byte level. The entry method is:

```
public static synchronized String retrieve(String name,
     boolean putInCache);
```

The locations searched by the file retriever are: record store and http. If the file is found in the record store, the method returns that file. Else a request is sent to the proxy for the URL location of the file. The proxy is used as an intermediary of file transfer. If the file can be found neither in the record store nor in the URL location the method fails and the result is null. An option is provided to indicate that we would like for the cache manager to try putting the newly retrieved document in cache.

Because String content (the output of the latter described method) is of no use to us the file retriever presents us with an additional method that acts as a wrapper for the Xml parser that might occur and returns the DOM tree of the xml document:

```
private static Node buildDom(InputStream is) throws
     Exception
```

A higher level wrapper for this method is:

```
public static Node getDom(String xmlString)
```

which handles all exceptions thrown during parsing time.

5.1.2.3 DOM Cache

This application module stores an already used DOM tree for later usage. Closer to the truth it doesn't store the DOM tree but keeps a reference to it such that the memory if occupies cannot be collected by the Garbage Collector. Hence a reference to it is still available to whoever is interested in using it.

What are the advantages that result from having a DOM cache?

- once the presentation for a specific axml document was generated it can be cached such that if neither the document nor the xsl don't change, it doesn't need to be computed again;
- the DOM tree that is generated from parsing the XSL file can be stored in cache so that it doesn't need to be reparsed;
- a single point of entry for all management of presentation xml or generic xml, BUT not source of Active XML documents;
- generically any xml fragment can be cached.

Important issues come into question when using a cache:

- WHAT to cache?;
- the cache's policy.

First of all only xml DOM trees are cache although the reference to any object can be inserted in the Hashtable that holds the references.

The heap memory on the J2ME platform is already limited hence unused objects should be discarded as quickly as possible. Because we want to avoid getting the dreaded OutOfMemoryException the proxy's policy is a conservatory one. The policy is based on these rules:

- never cache when the free memory is less than a quarter of the total memory;
- never cache when the size of the object we want to keep in memory is larger than a half of the total free memory;

There is no quick method to determine exactly the size of the object we intend to cache. The method I use is to assume that the object's size is approximately the size of the stringified xml. In reality this size varies between ¾ and 1/10 of the real tree object size (figures determined by running the profiler and the memory monitor).

Still part of the cache's policy is the answer to the question: when the contents of the cache become stale?

Every MDocument has two fields that act as flags: dirty and presentationDirty and indicate that different resources require synchronization with the new document content. "dirty" indicates that the document has changed as regard to its stored in the record store system. The first time the document is instantiated this flag is *true* because we want to force a write to the record system even though no service has been instantiated yet. "presentationDirty" may switch from *false* to *true* should any of these things happen:

- the Active XML document has changed as result to web service materialization;
- a new XSL stylesheet was downloaded.

If only the Active XML source document has changed and the XSL is cached, the same XSL reference is used as the one from cache, otherwise the XSL reference is discarded.

In both above cases the generated WML whose reference is stored in cache is discarded and a new presentation is generated.

The data structure that holds the cache is a Hashtable in which the key is a String that contains the axml document name and the element is the reference to the DOM tree.

5.1.3 Server: Querying the axml mobile peer

This proves somewhat difficult since one cannot initiate access to a particular j2me mobile device. An ordinary j2me mobile device has no ip address hence doesn't listen for connection, thus it cannot receive incoming connections.

A work-around is in order here. Basically we let the mobile device initiate the connection and transmit a message. As response to the message it has already sent, the device gets the proper query.

I propose the following approach (see diagram).

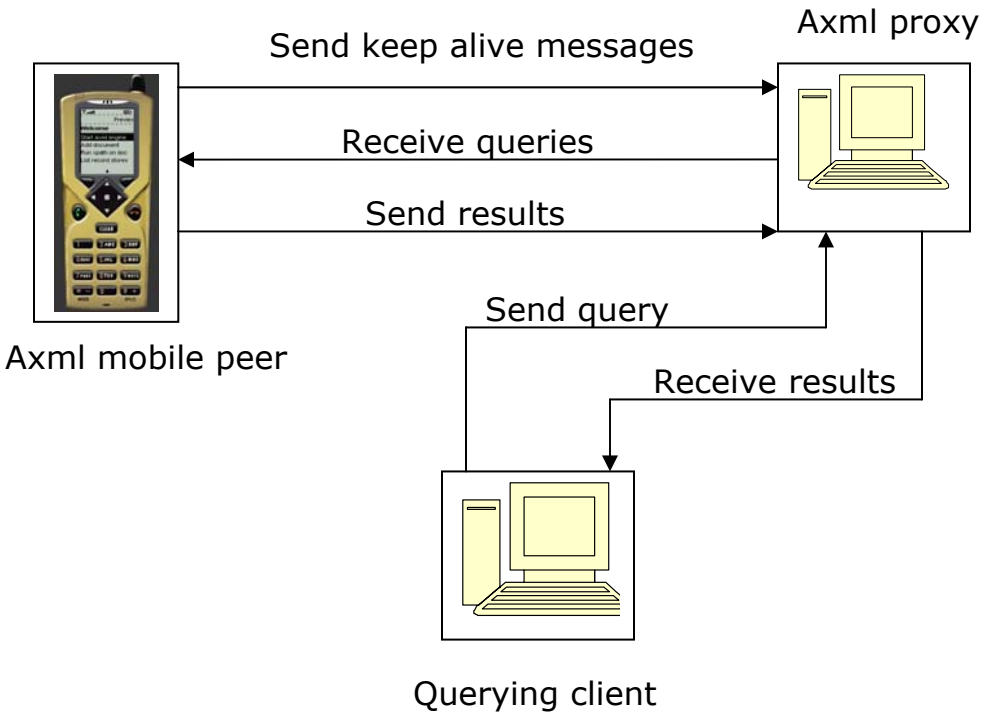

Figure 22

However more problems occur here. How does the proxy know to which device to address the query?

When the axml mobile sends the "query request" it actually sends a keep-alive message which contains its name. This name is chosen by the user and should be a descriptive name, will see later why. Whenever such a keep-alive message reaches the proxy, the device name is put in a table so that we can see it is online.

But we want the proxy to be able to intermediate more transactions at once (many-to-many). We need some sort of session mechanism and a queue of unresolved messages.

5.1.3.1 Query algorithm

The algorithm is distributed between the mobile client and the proxy. It is multithreaded on different machines: *proxy*, *actor* (the mobile component), *querying client*.

5.1.3.1.1 Proxy

There are three threads that execute in the same time on the proxy. Their execution is triggered by calls to proxy's xmlrpc services.

1. ping service	2. nodeRequest service	3. nodeResponse service

remember actor name;
 	
search queries for this
actor;
send query to actor;
STOP
	
	if actor is online
	initialize request;
	stamp request with unique id
	put request to queue;
	P_timeout(request.semaphore);
	endif
	
	if timeout send error message
	else
	send query results to client;
	STOP - SUCCESS
		
		Receive query response
		Match message id to
		a request's id from queue
		if match found
		set this as request's
		response
		V(request.semaphore)
		else dump response,
		requester
		is not listening anymore;
		STOP

```
1. ping: When a keep-alive reaches the proxy:
the proxy checks the internal table of messages in queue
if a query is destined to this device
    the query is sent as response to the keep-alive message
else do nothing.

2. nodeRequest: When a query message reaches the proxy.
the proxy checks to see if the desired server for the query
is online.
if it is online (that means it has sent at least one keep-
alive message) then
    the message is stamped with a unique sequence;
    then the message is put in a queue;
    the querying service is blocked (with timeout).
    after awakening:
        if timeout has occurred an error message is sent to
            the client;
        else if a response was received it is sent to the
            querying client.
        otherwise
            an error message is sent to querying client.

3. nodeResponse: When a response to a query reaches the
proxy:
the proxy checks for match the stamp of message (session
id) to every message in the internal message queue
if a match is found
    the proxy sets the response to the message
    the thread that waits for answer is awaked.
else the message is simply dumped because its requester
is not listening for an answer anymore.
```

5.1.3.1.2 Actor (mobile component)

This is the actual *server* of the mobile component.

Only one thread is needed. Periodically a ping is sent announcing the proxy that the actor is alive. If there is a query for this device it is received as response to the ping. The query (in fact an xpath expression) is performed and the results are sent back to the proxy.

5.1.3.1.3 Querying client

Here also a single thread is involved.

The client invokes either the xmlrpc of proxy directly of a SOAP endpoint that is able to command more than one proxy.

The process is synchronous. The client blocks until it receives a response of any nature.

A more in-depth look at the implementation overall

Usually a querying client will call the outside SOAP service in order to place a query.

The SOAP service receives the following parameters:

- proxyUrl - because our soap service can be used with more than one proxy we need to know to which to relay the query. In fact this proxyUrl is the URL to the proxy controller servlet;
- clientName - string parameter which indicates the name of the server to which the query is addressed;
- docName - string parameter, name of the axml document which is to be queried;
- xpath - string expression, the actual query.

This external SOAP service connects to the proxy through an xmlrpc service which the proxy implements. This xmlrpc service only takes as parameters the clientName, docName and the xpath expression. All this parameters are packed on the proxy in a Request class.

```
public class Request
{
    /**
     * the name of the mobile device which is
     * to be queried
     */
    p
    /**
     * axml document name that is subject to query

    private String docName = null;
    /**
     * actual query
     */
    p
}
```

A request is packed into a Message:

```
public class Message
{
            que id of this request.
     *

    private String stamp = null;

    /**
     * @see Request
     */
    private Request request = null;
    priv
    private Semaphore sem = null;
}
```

On the proxy the pseudo code at 5.1.3.1.1 is applied and, finally, the correct mobile device receives this:

- a session id (or message stamp);
- the docName;
- the xpath expression.

Because these parameters are sent in response to an xmlrpc service call we have to pack them into a single string to be sent to the client. Of course we could instantiate a new xml to contain all this parameters, but the optimum approach is to pack them into a single string whose elements are separated by some distinctive character (I chose newline '\n').

From now on the phone knows how to parse these parameters, execute the query and initiate a new xmlrpc call. This call has as its parameters the session id and the results of the query (the result will be explained below).

Different threads must synchronize inside the proxy in order for this process of exchanging parameters and query results.

Everything happens synchronously (blocking policy).

A timeout of 2 minutes is hardcoded in the proxy. This means that when one queries a mobile device through our SOAP service waits maximum 2 minutes for a message (either result to the query or an error report).

For a working example on how to call the query soap service:org.axml.proxy.synchronize.outside.soap.TestSoapRequestNode The result is a SOAPBodyElement whose root node is called either <result> (everything went smoothly) or <error>.If it is <result> its children are:

- if the xpath evaluation yielded an element it is wrapped in <element>;
- a text node is wrapped in <text>.

If the root element was <error> it should have a child text node, error description.

Synchronization on the proxy

In order to synchronize the three different threads that execute on the proxy semaphores are used. As seen above, from every request a Message is built. This message includes a Semaphore object. On proxy the message is registered in a collection and the thread (that handles the request) performs a P(timeout) operation on the semaphore from the message, blocking until the request is solved. The thread that registers the response in the message performs a V() operation on the semaphore.

P(timeout) calls the *wait* method with argument timeout in milliseconds.

```
public synchronized void P(long timeout)
{
    semValue--;

        while(true) {
            try {
                wait(timeout);
                break; //notify has occured
            } catch(InterruptedException e) {
                if(semValue >= 0)
                    break;
                else continue;
            }
        }
    }
}
```

Although *notify()* was not received the current thread will stop waiting after *timeout* milliseconds. This mechanism is useful, for example, if one of the

queried actors is no longer online. If not for timouted P the proxy thread would block forever waiting on a response.

How do different threads have access to the queue that contains unresolved Messages? Because the proxy is implemented as a web application (see section 5.2 for more information) we have access to a common area to all servlets of a specific application: ServletContext. All objects placed here can be retrieved by any thread in the same application.

External SOAP service

An external SOAP service exists that acts like an interface that receives queries for the mobile peer and hands back results to those queries.

This service was implemented using Apache Axis. Axis proposes two ways of publishing a user SOAP service: either through the JWS system which means that Axis automatically compiles the java classes and installs the service or through the WSDD descriptor. The service is implemented by using the second way because it offers more flexibility and the java sources don't have to be available. The service's WSDD is:

```
<deployment name="NodeRequestService"
xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">
  <!-- note that either style="message" OR
provider="java:MSG" both work -->
  <service name="NodeRequestService" style="message">
    <parameter name="className"
value="org.axml.proxy.synchronize.outside.soap.Handler" />
    <parameter name="allowedMethods" value="nodeRequest" />
  </service>
</deployment>
```

Installing the service described by this WSDD is done using the AdminClient: java org.apache.axis.client.AdminClient deploy.wsdd. Also there is available a WSDD that undeploys the service: undeploy.wsdd.

The method

```
public Element[] nodeRequest(Element[] params)
```

contacts the proxy and obtains the results that are then sent to the client that invoked the external soap service in the first place.

Examples of different xmlrpc envelopes transited between the mobile device and the proxy

Ping message sent from mobile device to proxy:

```
<methodCall>
 <methodName>proxyService.ping</methodName>
 <params>
  <param>
   <value>
    <string>Mary</string>
   </value>
  </param>
 </params>
</methodCall>
```

It is obvious that the only parameter is the device name.

Response to a query message sent from mobile device to proxy:

```
<methodCall>
 <methodName>proxyService.nodeResponse</methodName>
 <params>
  <param>
   <value>
    <string>0.8516749219678416</string>
   </value>
  </param>
  <param>
   <value>
    <string>
&lt;result&gt;
 &lt;element&gt;
  &lt;firstname xmlns:axml="http://www-
rocq.inria.fr/verso/AXML"&gt;Lleyton&lt;/firstname&gt;
 &lt;/element&gt;
&lt;/result&gt;</string>
   </value>
  </param>
 </params>
</methodCall>
```

5.2 The proxy

Roles

In this project's architecture the proxy has four roles:

5. it intermediates SOAP service calls;
6. it intermediates file transfer;
7. it acts as a directory for finding out which clients are currently online;
8. intermediates queries which are destined to the mobile peer.

The proxy is implemented as a web application. It interfaces with the outside world through a *servlet.*All calls are dispatched to the Controller servlet by the web application container. Although it is named Controller this servlet only delegates handling of the XML-RPC stream to the ServiceHandler class. This is the actual controller which integrates code to call the appropriate methods for all kinds of requests.

Six methods are available for different messages.

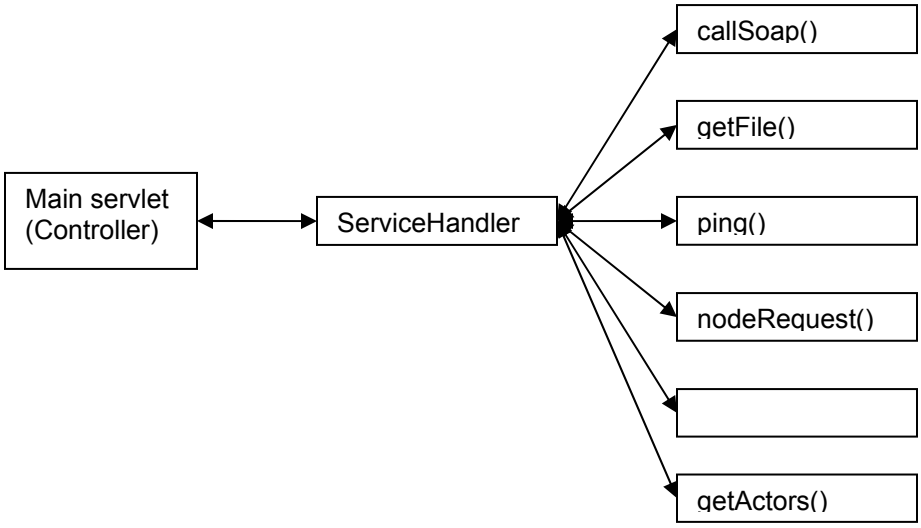

Figure 23

1. Proxy as an intermediary for SOAP service calls

One possible approach would be that the mobile device calls itself a SOAP service. But why do we need a proxy for our SOAP service calls:

- first let's remember that composing a SOAP envelope requires a lot of processing; it's not only that we have to use an xml parser, but also the serialized xml contains a lot more nodes than a xmlrpc requests;
- because a soap envelope is larger than a xmlrpc one, more data is required to transient the http connection, hence lower response times and bigger bills;
- the result of the service call can be additionally processed. For instance we can validate the returned types;
- results of a web-service call are available for use to more than one client;
- should the phone call soap services itself it is more difficult and requires more cpu time to use messaging style.

The client asks the proxy to execute a certain web service.

The xmlrpc request message has this structure:

```
<methodCall>
 <methodName>proxyService.callSoap</methodName>
 <params>
  <param>
   <value>
    <struct>
     <member>
      <name>serviceNameSpace</name>
      <value>
<string>capeconnect:AirportWeather:com.capeclear.weathersta
tion.Station</string>
      </value>
     </member>
     <member>
      <name>serviceURL</name>
      <value>

<string>http://live.capescience.com/ccx/AirportWeather</str
ing>
      </value>
     </member>
     <member>
```

```
      <name>methodName</name>
      <value>
       <string>getSummary</string>
      </value>
     </member>
    </struct>
   </value>
  </param>
  <param>
   <value>
    <struct>
     <member>
      <name>arg0</name>
      <value>
       <string>KLAX</string>
      </value>
     </member>
    </struct>
   </value>
  </param>
 </params>
</methodCall>
```

The proxy supports two ways of calling a SOAP service:

a. the messaging style (default);

b. RPC style, dynamic based on the WSDL of the service.

a. The messaging style

It is more natural to use since we can't always locate the WSDL. It follows these steps:

1. a Call object is instantiated and instantiated with information for locating the SOAP service;
2. initialize a SOAPBodyElement (which extends a DOM Element) that will be the envelope for the whole message;
3. for every parameter that the service accepts build a SOAPBodyElement and add it to the main one that represents the whole envelope;
4. perform the call and get the resulting SOAPBodyElement as the first element of the returned vector;

5. the SOAPBodyElement is translated into a DOM tree that the mobile device knows how to include in the original axml document.

b. RPC style

It takes up more resources than the messaging style because an additional analysis has to be performed. As the parameters received from the mobile peer have no types, the application consults the service's WSDL. Hence, in the axml document, the <sc> element should contain the "signature" attribute filled with the location (URL) of the WSDL. Usually the WSDL can be obtained from the concatenation of the service's endpoint URL by adding "?wsdl". However, this is not a standard and most of times, the .NET built services don't respect this. The next steps are performed to compose a valid SOAP request envelope:

1. the wsdl is downloaded locally;
2. a proprietary wsdl parser is fed the wsdl document;
3. the selected method is selected from the wsdl file;
4. a new service is built based on the wsdl;
5. in, out and in/out parameters are analyzed from the wsdl and correct types are established for the given parameters;
6. the service is called;
7. returned parameter types are verified against those that are expected;
8. an xml response message is sent to the mobile device.

To perform this type of call the wsdl4j.jar is required to be in the classpath. One important difference between the two types of SOAP service call is that the latter performs type verification. This method of performing a SOAP call is similar to generating a stub for that service with the wsdl4java axis tool.

Communication with the proxy

The communication with the proxy conforms to the XML-RPC open standard, thus it uses the HTTP protocol as its transport protocol. The initial version used the XmlRpcClient that came together with the kXmlRpc package from enhydra.org. This version works well in the J2ME environments. When testing on the PersonalJava environment, however, an exception is thrown when

trying to open a new HttpConnector after the previous has been successfully closed.

There are two ways of dealing with this:

- every time open a new HttpConnector and never close the previous. Let the midlet container manage the connections;
- loop until a connection is successfully opened. I have noticed that the exception is thrown every two connection attempts. Although there is no formal demonstration of this fact, it is almost certain that some time a successful connection is established (tests showed that the second try is every time successful).

Because the J2ME environment limits the number of possible connections to five I was forced to go with the second alternative (should we try to exceed the maximum number of allowed connections an exception is thrown). I have modified the XmlRpcClient and included it as a new class because:

- the process of connection setup, connection opening and connection closing is encapsulated in the XmlRpcClient.execute method, hence inheritance is not an option to achieving the goal of extending the XmlRpcClient class; the var that holds the connection is declared as local variable in the XmlRpcClient.execute method;
- design of XmlRpcClient is bad; some exceptions are handled whilst others are thrown to the parent try/catch block. This is bad because, for example, the exception thrown when there is no longer a network connection available should be handled by the programmer (action: stop trying to open a connection...). Other example of bad design is the fact that the "debug" boolean field is declared as friendly and the class lacks getters/setters for it, hence there is no way to remove the debug output in the production version.

Another reason for modifying directly the XmlRpcClient is that I made ita singleton. The singleton design pattern state that only one instance of an object is available to all client objects at one time.

That is only one instance of the XmlRpcClient exists and a method is provided that gets the reference. Limitations occur:

- all client code of the XmlRpcClient singleton connects to the same URL (actually there is a way to change the target machine but more complicated synchronization would be in order). This presents no problem because the mobile client talks to only one host: the proxy;
- only one thread can use the XmlRpcClient.execute method at one method because it is declared synchronized. This means that the process of calling a service is serialized. There are rare the moments when two services have to be executed at the same time moreover a lot of processor time is lost when handling the results of calling a web service. The latter happens multithreaded, every document is handled by a separate thread. All threads join up at a barrier.

However, the latter limitation is rather important and cannot be allowed. More than one thread should be allowed to talk to the proxy using xmlrpc protocol at one time. Too manage the available connections I have implemented a resource pool. Usually resource pools are developed because:

- the resources are very costly to instantiate; reusing an old one that is available is the prefered way;
- the maximum number of available resources is restricted.

In our case the pool resembles more a mutex because it lacks the first feature. An HttpConnector cannot be reused because once an input stream has been opened for obtaining results, we cannot send any more data through the output stream (an exception is thrown that signals the fact that we are violating the Http protocol). In fact the pool is a mutex that allows only five threads in the critical execution zone. Other arriving threads must wait for connections to be released.

To implement this conditional exclusion zone I have used semaphores. Pseudo code is:

```
XmlRpcClientSingleton.execute ()
{
    semaphore.acquire();
```

```
    open connection send and receive data;
    semaphore.release();
}
```

The semaphore object is declared as final (we don't have to synchronize for accessing it although it is used by more than one thread) and its constructor initializes the "value" variable to the maximum number of available connections (5).

Implementation of semaphore's acquire and release:

```
public synchronized void acquire()
    throws InterruptedException
{
    try {
        while(value <= 0)
            wait();
            --value;
    }catch(InterruptedException ie) {
        notify();
        throw ie;
    }
}

public synchronized void release()
{
    value++;
    notify();
}
```

To note in the above code that in the acquire method the semaphore value is not decreased if already lower or equal to zero. An ordinary semaphore would allow us to descend with the P() method below 0. These would lead to deadlock as more than the maximum number of threads request access to the conditional exclusion zone.

Constructing a xml-rpc request for a service call to be addressed to the proxy

The request is made up of two Hashtables serialized as <struct>s.The first Hashtable contains parameters for calling the service. It has as its keys:

- serviceNameSpace;

- serviceUrl - soap service endpoint;
- methodName - desired operation to be performed by the soap service;
- signature - URL of soap service WSDL (this is optional).

Every key has a value of type "string" serialized into a <string> element.

Once the proxy has received a response from the called SOAP service, it builds a response to return to the client. However, the relevant information is made up of a set of nodes. To make parsing possible at the client site there has to be a root node in order for this response to resemble a DOM Document. To make this possible all the response nodes are wrapped in a <result> element.

A possible response would look like this:

```
<result>
    <location>Los Angeles International Airport
(KLAX)</location>
    <temperature>15</temperature>
    <pressure>700</pressure>
    <visibility>5 Miles</visibility>
    <sky>sunny</sky>
</result>
```

2. Second role of proxy : intermediate file transfer.

This is useful if one wants to download a certain file from the Internet onto his/hers device.

First time the axmlmobile project is started on the mobile device it doesn't have any documents to work with so a source document must be retrieved. In order to accomplish this the file has to be downloaded some way.

A common way would be to open an HttpConnection to the source site (we assume that the file is set up on a web server).

This approach has, however, some short comings when the target platform is j2me. First of all only about five tcp/ip connections are available. But remember that we might have to call web services the same time we are trying to retrieve a file. Requesting an additional connection (the sixth in this case) would cause an exception to be thrown.

The second advantage is that by letting the xml-rpc module handle all the connections some sort of connection pooling can be implemented.

The third is that by having a proxy to intermediate this transaction we open the door to many possible optimizations. Caching is the most obvious one. The file transfer service of proxy receives one parameter of type String, which is the URL of the file to be downloaded.

Example of an xmlrpc envelope to request a file transfer:

```
<methodCall>
 <methodName>proxyService.getFile</methodName>
 <params>
  <param>
   <value>
    <string>
        http://www.cnn.com/new.xml
    </string>
    </value>
  </param>
 </params>
</methodCall>
```

Result of calling this service is the requested file serialized as a string.

3. Proxy as directory of mobile clients (or actors)

This functionality is more thoroughly explained in the “Querying the axml mobile peer” section of the “Implementation” chapter.

“Actor” is a synonym for a mobile client of the proxy.

It is important to note that a mechanism must exist through which current clients (actors) must register as active users of proxy services and in the same time volunteer to answer queries from outside. As such, two xmlrpc services were implemented to deal with these situations: ping() and getActors(). The former facilitates registering of a mobile client while the latter is used in order to obtain the current set of actors.

Current online actors represented as a Vector of String are serialized in the standard xmlrpc method as an “array”. A soap endpoint is provided which uses directly the getActors() service of proxy.

Unit tests

Unit testing is one of the requirements of Extreme Programming and states that tests should be written for any piece of code that is susceptible to fail. Ideally these tests should be written before the application code is actually written. It ensures all functionality works and enables easy refactoring because after each small change the unit tests can verify that a change in structure did not introduce a change in functionality.

JUnit is a framework for developing unit tests for Java classes.

The base class used for testing in this project is TestCase. Classes that provide unit testing extend TestCase. Conventionally all methods that provide testing of pieces of code have the prefix “test”. It is due to the fact that unit testing relies heavily on reflection.

Mainly the proxy code is tested.

Following tests are provided:

- HttpFileGrabberTest – method testGrabFile() tries to retrieve an arbitrary file by using the file grabbing xmlrpc service of the proxy. It tests that the result of the transfer is not null;
- TestGetActors – method testGetActors uses the getActors xmlrpc service to discover all online clients of proxy. It assumes that the returned result is not null;
- TestPing – method testNodeRequest() sends xmlrpc envelope to the ping service of proxy and tests that the response is valid. The response indicates whether or not a query was received for the current device;
- ServiceCallerTest – method testServiceCallUsual builds an xmlrpc envelope used for testing the xmlrpc-to-soap service of proxy. The test succeeds if the answer is not null. Another test method testServiceCallDotNet is provided in order to test a specific .NET service;

- TestNodeRequestInside – method testNodeRequest() injects a query for an xml fragment from a mobile device. This is done by invoking directly the xmlrpc service of proxy. The test is passed if the proxy returns a result and null is not returned. In order to use this test following fields should be changed: PROXY_URL, DOC_NAME, CLIENT_NAME, XPATH;
- TestNodeResponseInside – method testNodeResponse() injects a query response to the nodeResponse xmlrpc service of proxy. The message appears as coming from a mobile device. This test purpose is to be used in conjunction with the other query tests to simulate a conversation between the mobile device, the proxy and the querying client. Thus this test never fails;
- TestSoapGetActors – method testSoapGetActors uses the getActors soap service to discover all online clients of proxy. It assumes that the returned result is not null;
- TestSoapHandler – method testSoapServiceExists checks if the SOAP handler of the outside SOAP service is correctly installed and that is communicates with the proxy. Tries to call, through the xmlrpc-to-soap service of proxy, the standard echoElements endpoint of Axis. Fails if no response is received of if the latter is null;
- TestSoapRequestNode – method testRequestNode injects a query for an xml fragment from a mobile device. This is done by invoking soap endpoint of the querying subsystem. The test is passed if the proxy returns a result and null is not returned. In order to use this test following fields should be changed: NODE_REQUEST_SOAP_HANDLER_ENDPOINT,PROXY_URL, DOC_NAME, CLIENT_NAME, XPATH;
- TestConvPing, TestConvRequest, TestConvResponse, TestConvRequestSoap are simple wrappers around test cases in order to simulate a conversation who's purpose is to satisfy an exterior query. These are test suites which are run by a test runner. Example:

```
public class TestConvPing
{
     publ
     {

     }
     publ
     {
                                ew TestSuite();
          //tests the ping service deployed on proxy
                              new                          class

     }
}
```

All tests are gathered in Test Suites. A Test Suite is provided which runs much of the above tests. The query conversation test can be run only manually. This test is difficult to perform because one has to change the message stamp (which should be unique every time) in org.axml.proxy.synchronize.inside.TestNodeResponseInside.

This stamp can be obtained by taking a peek at tomcat stdout.log or by performing a second ping (ant testConvPing), the message stamp should be located on the first line.

To perform the test two consoles are needed:

First console	Second console
ant testConvPing #register as client	
	ant testConvRequest #uses xmlrpc service of proxy or ant testConvRequestSoap #uses soap service
ant testConvPing #to see query or ant testConvRequest #to responde	
	#query results start coming unless 2 min timeout has occurred

Building the project (ant tool)

Ant is a Java based build tool. It resembles the “make” tool in many ways. The proxy is built using ant. All targets for building the proxy are in the “build.xml” file. Targets are:

- compile – compile source code and move output to the ./bin directory;
- deploy – constructs directory tree and builds new tomcat context “proxy”; moves compiled bytecode to the new context;
- test – performs all junit tests by running the main test suite;
- testConvPing – part of querying conversation, pings the proxy;
- testConvRequest – part of querying conversation, registers new query;
- testConvRequest – same like above, but talks to the outside soap endpoint and not directly to the proxy;
- testConvResponse – part of querying conversation, responds to a query that was received earlier;
- deployNodeRequestService – deploys the front SOAP service used by regular clients when talking to the proxy. It moves compiled files to the “axis” tomcat context. Uses org.apache.axis.client.AdminClient to deploy a wsdd that describes the SOAP endpoint;
- undeployNodeRequestService – undeploys the front SOAP service by using the org.apache.axis.client.AdminClient;
- listServices – lists currently installed Axis service.

6. Performance and measurements

6.1 Target platform performance measurement

Test platform: Compaq IPAQ CPU StrongArm 233Mhz, 64Mb Ram, Java Virtual Machine: PersonalJava v1.1, Micro Edition Emulator: ME4SE.

First off all we need to find out the number of bytecodes/ms that are executed by the target device. In order to accomplish this I prepared two tests.

First test:

```
int a = 1;
long start = System.currentTimeMillis();
for(int i = 0; i < 1000000; i++) {
    a = a+i;
}
long duration = System.currentTimeMillis()-start;
System.out.println("time: "+duration);
```

On execution of the first test the Sun Wireless Toolkit reported 8058338 bytecodes.

The execution of the test on the emulator took about 875 ms.

Results of executions on the Compaq IPAQ platform:

Trial no	1	2	3	4	5	Average
Ms	1001	998	998	1000	1002	999.8

Average bytecodes/ms: 8059

Second test:

```
int a = 1;
long start = System.currentTimeMillis();
for(int i = 0; i < 1000000; i++) {
    a = a+i;
    a = a*10 - a;
}
long duration = System.currentTimeMillis()-start;
System.out.println("time: "+duration);
```

The second test took 14058379 bytecodes for execution.

On emulator it took about 1312ms to execute this test

Execution on Compaq IPAQ

Trial no	1	2	3	4	5	Average
Ms	1987	1990	2000	1995	2003	1995

Average bytecodes/ms: 7046.

My conclusion is that actual speed is around **7552 bytecodes/ms** when using the PersonalJava platform on Compaq IPAQ.

6.2 Performance tests regarding various features of AxmlMobile

Tests were performed by using a special class: org.axml.mobile.Benchmark.

This class only contains two methods: mark() which initializes a new timing measurement and measure(String str) which closes the measurement. The PersonalJava console is not ergonomic and output is difficult to follow. Because of this measurement results are written in a special Record Store named “bench”.

Retrieving files through HTTP connection

Active XML documents and presentation stylesheets are received by using the proxy xmlrpc service for file transfer. At physical layer the link between the IPAQ and the proxy is a USB cable.

Transfer times where two times measured and an average value was computed.

Id	File name	Size	Trial 1	Trial 2	Average
1	hello.xsl	997b	1025ms	1002ms	1013.5ms
2	hello.xml	1.5Kb	1040ms	1100ms	1070ms
3	Stock_trade.xml	2Kb	1105ms	1050ms	1077.5ms
4	OrgChart.xml	7Kb	1156ms	1137ms	1146.5ms
5	OrgChartDebug.xml	15Kb	1220ms	1320ms	1270ms

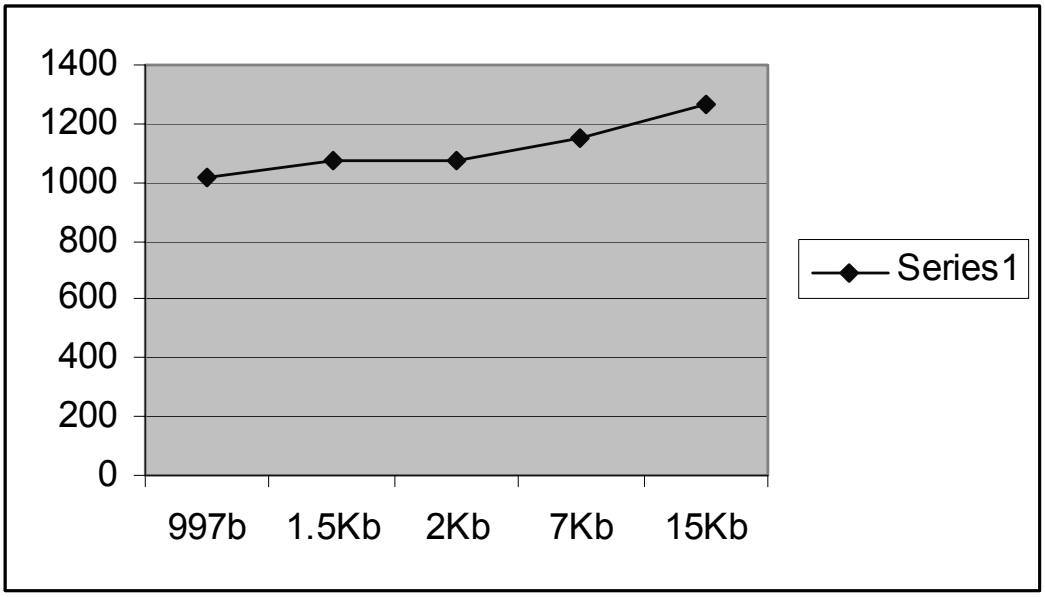

Figure 24

Although transferred files become larger and larger (eg. OrgChartDebug.xml is fifteen times bigger than hello.xsl) the transfer time doesn't increase proportionally. The explanation is that most of time is used to compose the XML envelope, the actual request to be addressed to the proxy. This computational effort is comparable for all transferred files. Because files take a certain amount of time to be transferred across the wire this is the difference noticeable in Figure 1. Probably these differences would grow higher if a small bandwidth connection is used in place.

Parsing a xml into a DOM tree

It is an important and costly operation. Whenever possible it is optimized by using a cache.

Filename	Size	Timing
transactions.xml	1Kb	470ms
hello.xml	1.5kb	536ms
stock_trade.xml	2kb	770ms

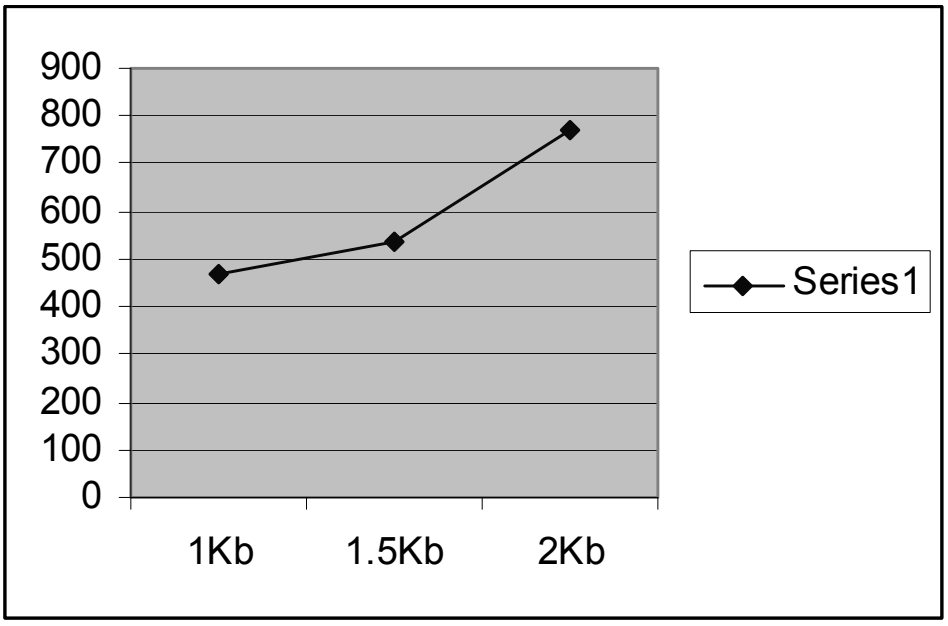

Figure 25

The measured time is directly proportional to the size of the parsed xml and the number of nodes.

Evaluating XPath expressions of different complexities applied to xml documents

XPath expression	Target xml	Timing
"/stock_trade/trade_data"	stock_trade.xml	50ms
"/salute_players/tennis/player[@country="russia"]/../player[@country="norway"]/atp/rank"	hello.xml	68ms
"/salute_players/tennis/player"	hello.xml	57ms

Timing for evaluating different XPath expressions is directly influenced by the expression's complexity (mostly number of location paths and the presence of complex predicates). However a big chunk of CPU time is allotted to building and instantiating necessary objects serving to run the mini xpath engine.

Parsing a web service

It is important because it is the most costly step (excluding the DOM parsing) of building a logical image of the axml document. Duration of initializing a document and the memory footprint is influenced by the number of web service references in the document.

This also supposes parsing the web service parameters.

Active XML document	Service	Method	Concrete parameters	Non-concrete parameters	Timing
hello.xml	Hello.jws	sayHello	1	0	230ms
		sayHello	0	1	205ms
stock_trade.xml	StockService	calcExchange	0	1	245ms
		cheaperThan	0	1	235ms
transactions.xml		getTransactions	0	0	215ms

Hello.jws is an Axis JWS service. Uses RPC style. It only has one method -> sayHello. This method receives as parameter a String and returns a String.

StockService is an Axis WSDD deployed service. Uses messaging style. It has three methods:

- calcExchange: receives a parameter (sum of money in local currency) and translates it into us$;
- cheaperThan: receives a parameter and returns an xml envelope containing information about all the stocks cheaper than that;
- getTransactions: has no parameters, returns all performed transactions.

Concrete parameters are passed by value, do not contain xpath references.

Parsing services takes a comparable amount of time because much the same operations are performed. Non-concrete parameters are not materialized when first initiating the web service. They are materialized every time the service is executed.

Generating and rendering the presentation

Following steps are performed to generate a presentation:

1. check if a presentation is present in cache;
2. if the xsl file is not present in DOM Cache it is retrieved (either through http connection or from the RecordStore);
3. generate presentation.

Axml document	XSL file	XSL file size	Generate presentation timing	Render presentation timing
hello.xml	Hello.xsl	1Kb	510ms	1012ms
stock_trade.xml	Stock_trade.xsl	7Kb	710ms	1230ms

Speed of presentation generation depends on the number of XSL instructions present in the .xsl file.

To note that, in case of a cache hit, timing of this step of presentation generation can be reduced to 0.

When rendering the previously generated presentation a WML file is translated into viewable objects. This operation has a consistent memory footprint, but it runs in reasonable time. Presentation is rendered synchronously, it happens at the start, after all *lcdui* objects have been put to the screen, the user is free to browse without further interruptions.

Retrieving data from the Record Store

Involves accessing the Record Store Management System.

Axml document	Size	Timing
transactions.xml	1Kb	854ms
hello.xml	1.5kb	911ms
stock_trade.xml	2kb	1000ms

Operation duration is determined mainly by the file size, no processing is done.

Above measurements serve as an informal demonstration to the fact that application speed depends heavily on the XML parser which consumes most of the memory and CPU time. This includes operations that make use of the XmlRpcClient. In order to invoke an xmlrpc service an xml envelope must be built, that is the xml parser is used.

Purpose of the Active XML implementation is not only educational, to show that it can be done. It is meant to also demonstrate practical uses. Hence, the implementation should be more production-ready rather than prototype like.

It is optimized to be deployed onto profile MIDPv1.0 (minimum requirement for any J2ME implementation) on which the performance is acceptable.

7. Conclusions

This thesis together with the implementation that accompanies it proves that it is possible to port a resource demanding application like a classic Active XML peer to a critically resource limited mobile device.

The Active XML Mobile takes a step further and starts to pay more attention to the needs of the individual user. Because of this it is a viable alternative that may be used, someday, as a full-fledged commercial application.

Active XML Mobile is more than an educational experiment. It can make a difference in the world of Embedded applications by extending P2P networks beyond the boundaries of desktop computers. Embedded systems are resource limited as individual devices. But their number is huge. Active XML Mobile provides a way to link them all together in a cooperative P2P network.

The implementation level of the Active XML Mobile is 100%. In order to better test the implementation a demonstration scenario was produced.

7.1 Further development

- a better algorithm should be implemented to detect dependencies amongst services;
- the user interface should be more friendly;
- a better method of interacting with the Active XML document is needed;
- extended testing is needed.

References

[1] Serge Abiteboul, Omar Benjelloun, Tova Milo, Ioana Manolescu, Roger Weber: Active XML: **A Data-Centric Perspective on Web Services**, BDA 2002, ftp://ftp.inria.fr/INRIA/Projects/verso/gemo/GemoReport-213.ps.

[2] Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo, Roger Weber: Active XML: **Peer-to-Peer Data and Web Services Integration**. VLDB 2002 (demo) ftp://ftp.inria.fr/INRIA/Projects/verso/gemo/GemoReport-226.ps.

[3] Serge Abiteboul, Angela Bonifati, Gregory Cobena, Ioana Manolescu, Tova Milo: **Dynamic XML Documents with Distribution and Replication**, SIGMOD 2003, ftp://ftp.inria.fr/INRIA/Projects/verso/gemo/GemoReport-272.ps

[4] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, Frederic Dang Ngoc: **Exchanging Intensional XML Data**, SIGMOD 2003, ftp://ftp.inria.fr/INRIA/Projects/verso/gemo/GemoReport-271.ps

[5] Irina Athanasiu: **Java ca limbaj pentru programarea distribuită** – Editura Matrix Rom 2002;

[6] Irina Athanasiu, Bogdan Costinescu, Octavian Andrei Drăgoi, Florentina Irina Popovici: **Limbajul Java: o perspectivă pragmatică –** Editura Agora, ediția a II-a.

[7] Bruce Eckel: **Thinking in Java** – http://www.mindview.net/Books/TIJ/;
[8] Bruce Eckel: **Design patterns in Java** - http://www.mindview.net/Books/TIJ/;
[9] Doug Lea: **Concurrent Programming in Java – Design Principles and Patterns –** Addison Wesley;

[10] Project Apache Jelly: http://jakarta.apache.org/jelly;
[11] Project Apache Velocity: http://jakarta.apache.org/velocity;
[12] Apache XML-RPC project: http://xmlrpc.apache.org;
[13] Proiectul PHP: http://www.php.net;
[14] Macromedia: http://www.macromedia.com;
[15] Enhydra: http://www.enhydra.org;
[16] kXml: http://www.kxml.org;
[17] kXmlRpc: http://kxmlrpc.enhydra.org;
[18] Web Services standards: http://www.w3.org/2002/ws;
[19] XPath standard recomendation: http://www.w3.org/TR/xpath;
[20] XSL Transformations (XSLT) standard recomendation: http://www.w3.org/TR/xslt;
[21] Active XML web site: http://www-rocq.inria.fr/verso/Gemo/Projects/axml/;