AppiaXML

A Brief Tutorial

José Mocito
Universidade de Lisboa
jmocito@lasige.di.fc.ul.pt

Liliana Rosa
Universidade de Lisboa
lrosa@lasige.di.fc.ul.pt

Nuno Almeida
Universidade de Lisboa
nalmeida@lasige.di.fc.ul.pt

Luís Rodrigues
Universidade de Lisboa
ler@di.fc.ul.pt

September 24, 2004

Contents

1 Introduction

Appia is a Java framework to support the composition and execution of communication protocols. An important feature of *Appia* is that the user may create different protocol compositions by stacking protocols. An instance of a protocol composition is called a *channel*. An unique property of *Appia* is that different channels may share the same *session* of a common protocol. Using this mechanisms is possible to coordinate different, but related, data flows.

This report describes an extension that we have implemented in the *Appia* system that allow the user to configure one or more communication channels using a description in XML. We have named this set of components *AppiaXML*. *AppiaXML* is able to interpret XML strings and files and automatically create and initialize, *Appia* communication channels. Using *AppiaXML*, the users may easily create and modify channel configurations, defining which layers should be used and setting initialization parameters without any Java programming effort.

This document attempts to be as much self contained as possible but a basic knowledge of the *Appia* framework [1, 2] and of XML syntax is assumed.

Document Structure

The rest of the document is structured as follows. Sections 2 and 3 present a brief overview of *Appia* and XML, respectively. Section 4 introduces an example application used throughout this document. In section 5 the main features introduced by *AppiaXML* are presented. Section 6 presents a detailed explanation on how to create XML configuration files. Section 7 describes the API provided by *AppiaXML*. In section 8 a demonstration example implementing a *Messenger*-like application is presented to illustrate the use of *AppiaXML*. Finally, this document concludes with an appendix A that simply shows a listing of the DTD used in the XML configuration files.

2 A Brief Overview of *Appia*

Appia is a protocol composition and execution framework that has been implemented in the Java programming language. *Appia*'s composition model is based on three fundamental abstractions: *layers*, *sessions*, and *events*:

- *Layers* are singleton entities, responsible for declaring the protocol behavior. A stack of layers defines what is called in *Appia* a *QoS* (Quality of Service). The QoS specifies which protocols must act on the messages and the order they must be traversed.

- *Sessions* are protocol instances that maintain the state required to execute the protocol. *Channels* are stacks of sessions. Each channel is defined on behalf of a QoS. The set of sessions composing a channel must respect the set of layers used in the corresponding QoS. Typically, the top of the stack is an application layer and the bottom of the stack is a layer that interfaces the network.

- *Events* are data structures used by sessions to exchange information. The framework supports an open event model, that is, the set of events supported is not defined *a priori*, and can be extended by the programmer. The framework ensures that events exchanged between two sessions in the same channel are delivered respecting FIFO order. Events may flow in the stack in both directions: upwards (from the network to the application) or downwards.

To exchange information among two or more processes, *Appia* uses a specific event: the `SendableEvent`. This event has two additional fields, the `source` and `dest` that are used to indicate the source and destination endpoints. A `SendableEvent` also has a `Message` field, which contains the information that will be sent to the communication link.

To build an *Appia* protocol, two classes must be created: a `Layer` and a `Session`. The `Layer` contains static information about the events that the protocol will *accept*, *require* and *provide*. The `Session` has the state of the protocol. An event is delivered to a protocol (or Session) in the `handle(Event e)` method. When a protocol wants to send an event to the next protocol, it must call the `go()` method of the event.

Appia provides also a set of common utility services to simplify the task of protocol development, such as the management of data buffers for messages (with methods to add, extract and inspect headers), management of timers, automatic generation of events to initialize the channels, etc.

More information about *Appia* can be found in the *Appia* home page (`http://appia.di.fc.ul.pt`) and in the following reports [1, 2].

3 A Brief Overview of the XML Language

XML stands for Extensible Markup Language, providing the capacity to store, structure and exchange information in a cross-platform, software and hardware independent manner. XML allows users to define information structure through a *DTD* (Document Type Definition) or a XML schema. A DTD describes the structure of a XML file by defining a list of elements and their parameters. A XML schema is an alternative to DTD but has no interest in the scope of this tutorial.

The simplicity of XML is related with data description: the user is allowed to define her own tags, being able to create any kind of data structure. The basic block of data in a XML file is called an element. An element can have an attribute list or more elements encapsulated (nested elements). XML tags are used to markup elements, whose representation is quite similar to HTML with the format presented in Listing 1. Tags like <`person`> mark up the beginning of an element and tags like <`/person`> mark the end. Analogous to elements tags can also be simple, as <`age`>, or can be nested as <`name`> with elements <`first-name`> and <`last-name`>.

```
<person>
        <name>
                <first-name>John</first-name>
                <last-name>Doe</last-name>
        </name>
        <age old="40"/>
</person>
```

Listing 1: XML file.

The use of XML is becoming increasingly widespread, adopted by many vendors of manufacturing software, and familiar to many programmers. Because of all the relevant features offered and wide acceptance XML was chosen for usage in *AppiaXML*.

4 The *Ecco* Application

A tiny application was developed with didactic purposes, in order to help the reader better understand all the issues related to the usage of *AppiaXML*. This application, called *Ecco*, provides a simple tool for exchanging messages among two peers. The behavior is very simple: a user types a message, then hits the *return* key and the message gets sent to the other user; by other words, the message is *echoed* in the receiver's console.

This application is provided with the *Appia* distribution and is named demo.xml.Ecco. To execute it the reader should have *Appia* installed and correctly configured[1].

Throughout this tutorial the reader will be exposed to different aspects of *AppiaXML*, that will be illustrated with the help of short pieces of code from this application. For this reason, it is of utmost importance that the reader obtains a clear understanding of how the internals of the application works.

Ecco has two distinct initialization modes. The first one does not use XML for channel configuration and uses Java programming to create the communication channel. The second instance, not surprisingly, uses a XML file that handles all channel configuration and creation. The command line for the execution of the application in one of the two modes is, respectively:

```
java demo.xml.Ecco <local_port> <remote_host> <remote_port>

java demo.xml.Ecco <xml_file>
```

In the first mode the values needed by some sessions are passed in the command line, and the way they are retrieved must be coded by the programmer of the application. The second mode gets these values from the configuration file, and the programmer only needs to know the name of the specific paremeters and respective sessions.

The comparison of the source code for these two initialization modes is valuable to the reader, as she may get a better insight on the differences between using *AppiaXML* and invoking *Appia*'s Java APIs.

The Ecco application uses a channel with two protocols as illustrated in the following communication stack:

Ecco
TCP

[1]The README file in the *Appia* distribution provides step-by-step instructions on how to install and configure *Appia*.

This stack corresponds to the default configuration coded in the application and provided in the accompanying configuration file ecco.xml. The layer containing the Ecco protocol should always be the topmost. Below this layer the reader can experiment with any protocol composition she finds adequate (i.e., UDP+FIFO or simply UDP). The reader should also note that in order to use this file she should correct the values of the ports and hostname parameters to suit his network configuration.

5 *AppiaXML* Features

AppiaXML has many features that try to address all the different configuration possibilities available to *Appia* users when composing communication channels. The most important feature provided is the possibility to define the structure of a channel and then instantiate as many channels as needed with the same properties defined in the configuration. All the available features will be described in detail in the next few sections.

5.1 Describing Channels

Channels are described in a configuration file. Describing a channel is the same as defining the channel properties using XML. This is accomplished by defining a structure, called a **template**, that specifies these different protocols, in the form of references to the *layers* that implement them. As the reader will see later on this tutorial, describing a channel is not the same as creating a channel. A template is an entity that has all the information needed to create channels with the same QoS.

Shared Sessions

One of the most interesting features *Appia* provides is the ability to share a session over any number of channels. *AppiaXML* is also able to create channels with different combinations of shared sessions. To allow these combinations *AppiaXML* introduces the concept of **sharing scope**. A sharing scope determines the way a session is shared between channels. Three sharing scopes are considered: **private**, **label** and **global**.

private sessions are never shared;

label sessions are shared by all channels that have the same session and the same label;

global sessions of a given protocol are shared by all channels that use them.

To better illustrate the usage of the different scopes Figure 1 shows a communication stack with three channels that combine different session scopes.

5.2 Creating Channels

Channels can be created either at configuration time or execution time. In both cases a channel is created based on a given template defined in the XML configuration file.

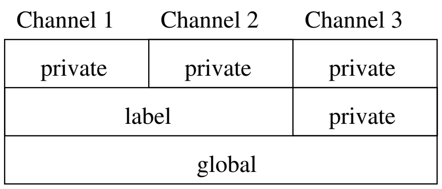

Figure 1: Tree channels using private, label and global sessions.

Session Parameters

When configuring *Appia* sessions using Java programming, parameters are passed to sessions using some method provided by the session for that purpose. It is up to the protocol programmer to provide that specific method and to let everyone know about it. This kind of flexibility is good in most situations and is very appreciated by developers as they are not bound to the use of a pre-defined method to pass the parameters.

As we have noted in the previous section, *AppiaXML* channels can be created at configuration time. If any session in the channel needs to be parameterized, the supporting framework for XML configuration needs to know which method to call in order to pass the specified parameters. As the reader will realize later in this tutorial, this will be achieved by requiring the session programmer to implement a pre-defined interface and thus, a specific method, to handle the initialization parameters specified in XML files.

Channel Initialization

Even if the channel has parameterizable sessions and is created at configuration time, it is up to the user to decide if it wants the channel to be initialized automatically by *AppiaXML*. It is possible to create an uninitialized channel. This might be useful when one knows *a priori* the configuration of channels (thus creating them at configuration time) but is only able to obtain some initialization parameters in run-time (for instance, from user input) or wants to obtain channel cursors in order to navigate through all the sessions that compose the channel and do all the operations *Appia* allows. The programmer may afterward perform the channel initialization with the common procedures that *Appia* provides.

6 XML Configuration File Structure

The XML file used to configure channels in *AppiaXML* has the structure described by a DTD file provided by the *Appia* developers team at the following URL: `http://appia.di.fc.ul.pt/appiaxml.dtd`. It is a short DTD with simple elements and is shown in Appendix A.

From the *AppiaXML* DTD it is clear that a configuration file is divided in two parts. The first one is where channels are described; the second one is used to declare which channels will be created. Every configuration file must start with the <`appia`> tag which aggregates all the other elements.

The default XML configuration file used in Ecco is shown in Listing 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE appia PUBLIC "-//DTDName//DTD//EN"
        "http://appia.di.fc.ul.pt/appiaxml.dtd">
<appia>
 <template name="ecco_t">
  <session name="tcp_s" sharing="global">
   <protocol>
    appia.protocols.tcpcomplete.TcpCompleteLayer
   </protocol>
  </session>
  <session name="ecco_s" sharing="private">
   <protocol>
    appia.test.xml.ecco.EccoLayer
   </protocol>
  </session>
 </template>
 <channel name="ecco_c" template="ecco_t" initialized="yes">
  <chsession name="ecco_s">
   <parameter name="localport">2000</parameter>
   <parameter name="remotehost">foobar.di.fc.ul.pt</parameter>
   <parameter name="remoteport">2000</parameter>
  </chsession>
 </channel>
</appia>
```

Listing 2: *Ecco* XML configuration file.

In the next couple of sections we will illustrate how to make use of these tags.

6.1 Defining Templates

Creating a template consists of describing the channel properties. Templates are created using the <`template`> tag. Each template must have an unique identifier associated with the attribute `name` and is composed by any number of sessions. Each session starts with the tag <`session`> and must have two attributes: `name` and `sharing`. The first one is the internal identifier of the session and the second is the corresponding sharing scope (see section 5.1).

Inside a <`session`> must exist a <`protocol`> tag containing the identifier of the protocol used in the session. The protocol is identified by the complete name of the Java class containing the layer implementation (e.g. appia.protocols.tcpcomplete.TcpCompleteLayer).

Suppose one would like to change the channel for the Ecco application so it would use UDP instead of TCP. One would only have to change line 8 substituting it by the following line: appia.protocols.udpsimple.UdpSimpleLayer as shown in Listing 3.

```
 5                                       ...
 6   <session name="udp_s" sharing="global">
 7     <protocol>
 8       appia.protocols.udpsimple.UdpSimpleLayer
 9     </protocol>
10   </session>
11                                       ...
```

Listing 3: *Ecco* using UDP.

6.2 Creating Channels at Configuration Time

Channels are instantiations of a given template. A channel has the exactly same properties as the template it was created from. Channels can be created either at configuration time or at execution time. This Section deals with the former case. The later will be presented in Section 7.3.

Putting it in simple terms, creating a channel consists in assigning a unique name and reference to the respective template. More specifically, the tag <`channel`> starts the creation of a channel. This tag defines three attributes: `name`, `template` and `initialized`. The first two are the identifier of the channel and the identifier of the template that rules the channel instantiation. The third one is a boolean that tells if the channel is to be initialized (see Section 5.2).

The channel creation at configuration time is illustrated in Listing 2 at lines 17 to 23. In this example, the channel is named *ecco_c* and uses the configuration template named *ecco_t*. It is also an initialized channel as stated in the last attribute (`initialized="yes"`).

Moreover it is also possible to pass any parameters to specific sessions that belong to the channel. This is done inside the scope of the <`channel`> tag using the <`chsession`> tag to identify to which session the specified parameters should be passed. Inside the scope of <`chsession`> lies every parameter and each one is described using the tag <`parameter`> with an attribute `name` that holds the parameter name, as defined by the protocol programmer. Inside the scope of this tag should then lie the value of the parameter.

To better illustrate this feature suppose one would like Ecco to connect to a different remote port, say port 2222. One would simply have to change line 21 of the configuration file. One would then get something like Listing 4.

```
16                          ...
17 <channel name="ecco_c" template="ecco_t" initialized="yes">
18  <chsession name="ecco_s">
19   <parameter name="localport">2000</parameter>
20   <parameter name="remotehost">foobar.di.fc.ul.pt</parameter>
21   <parameter name="remoteport">2222</parameter>
22  </chsession>
23 </channel>
24                          ...
```

Listing 4: *Ecco* using remote port 2222.

7 Java API for XML Configuration

AppiaXML defines an API to allow the user to load configuration files and to create channels based on the loaded templates. All these programming interfaces are provided in a single class called appia.xml.AppiaXML whose interface is presented in Listing 5.

```
class appia.xml.AppiaXML {
        static void load(java.io.File xmlfile)
          throws org.xml.sax.SAXException,java.io.IOException;
        static void loadAndRun(java.io.File xmlfile)
          throws org.xml.sax.SAXException,java.io.IOException;
        static apppia.Channel createChannel(
                  String name,
                  String templateName,
                  appia.xml.utils.ChannelProperties params,
                  boolean initialized)
           throws appia.AppiaException;
}
```

Listing 5: *AppiaXML* API.

7.1 Loading Configuration Files

Configuration files are loaded calling the static method load() from the AppiaXML class with a string argument containing the configuration filename. A little excerpt from Ecco's source code in Listing 6 illustrates this. The exception handling performed in the excerpt is left empty because it as no value in this section's purpose. Details on how these exceptions should be handled can be found in Section 7.4.

```
java.io.File xmlfile = new java.io.File(args[0]);
try {
        AppiaXML.load(xmlfile);
} catch (org.xml.sax.SAXException e) {
        ...
} catch (java.io.IOException e) {
        ...
}
...
appia.Appia.run();
```

Listing 6: Loading a XML configuration file.

When a file is loaded the parser loads into memory all the templates and creates all the channels contained in the configuration. From that moment forward the templates and channels are available for use.

7.2 Loading and Automatically Running Configurations

It is possible that for some applications there is no need for initializations at execution time; all the initialization steps are performed after loading the configuration. In this case *AppiaXML* provides a facility that allows the reader to perform the execution of a given configuration with only one Java statement.

For instance, the Ecco application doesn't need any initialization at execution time. All the parameters needed by the sessions are in the configuration file. This is a perfect example where this feature might come in handy. Listing 7 provides the full source code for an application that runs Ecco. As the reader can observe, there is no need for the extra statement that makes *Appia* start it's execution loop.

```
public class RunEcco {
  public static void main (String[] args) {
    java.io.File xmlfile = new java.io.File(args[0]);
    try {
      AppiaXML.loadAndRun(xmlfile);
    } catch (org.xml.sax.SAXException e) {
      ...
    } catch (java.io.IOException e) {
      ...
    }
  }
}
```

Listing 7: Loading and Running a XML configuration file.

Appia distribution contains a little demo application, demo.xml.LoadConfig, that loads and runs the configuration file whose name is passed as a command line argument. This little program serves pedagogical purposes only, as it does not do adequate exception handling, it just prints the stack traces of the exceptions. Nevertheless the reader might find it useful for testing configurations that fit the requirement stated above.

7.3 Creating Execution Time Channels

Channels are created with a call to the static method createChannel() from the AppiaXML class. This method takes five parameters, the name of the channel, the name of the template, the sharing scope label (if used by any session that belongs to the channel or null if not needed), the parameters to be passed to selected sessions (see Section 7.5), and a boolean telling if the channel is to be initialized, or more specifically, if a call to the start() method from the channel should be made.

Suppose Ecco did not have any channel creation section defined in the configuration file. After loading the configuration as shown above it would be easy to create the channel in run-time time. The code to do this is shown in Listing 8. To simplify the explanation, we will assume that this channel does not need any parameters to be passed during initialization.

```
try {
        appia.Channel ch = AppiaXML.createChannel("ecco_c","ecco_t",
          null,null,true);
} catch (appia.AppiaException e) {
        e.printStackTrace();
}
```

Listing 8: Creating a channel at runtime.

7.4 Exception Handling

In *AppiaXML* exceptions are handled a little different than in normal Java programming. This is due to the fact that the XML parser forces any exception that may occur during the parsing to be thrown as a SAXException. To circumvent this limitation the SAXException provides a way to wrap other exceptions inside it.

Instead of catching all sorts of exceptions the user only catches a SAXException. It is then able to test if there is any wrapped exception with the method getException() that returns the exception wrapped or null if none exists. With the return exception the user can do the treatment she wishes[2].

There are only four different exceptions that can be thrown by parsing a configuration file: ClassNotFoundException, InstantiationException, IllegalAccessException and AppiaException, the first three representing their usual meanings described in the *API Specification* included in the Java documentation, and the last one in [2].

To illustrate exception handling in *AppiaXML* we pick the previous example from Listing 6 and complete it with an adequate differentiated treatment of each possible exception. Listing 9 shows exactly this. The exceptions are treated by specific methods created to handle them. The implementation of these methods is not important in the current context and so is deliberately omitted in the example.

```
java.io.File xmlfile = new java.io.File(args[0]);
try {
        AppiaXML.load(xmlfile);
} catch (org.xml.sax.SAXException e) {
```

[2]Detailed information on how to deal with exceptions wrapped in a SAXException can be found on the Java API documentation, more specifically on the SAXException class description.

```
        Exception we = e.getException();
        if (we != null) {
          if (we instanceof java.lang.ClassNotFoundException)
            System.err.println(``Some layer class provided in the'' +
            ``configuration was not found!'');
          else if (we instanceof java.lang.InstantiationException)
            System.err.println(``Some class provided in the'' +
            ``configuration could not be instantiated!'');
          else if (we instanceof java.lang.IllegalAccessException)
            System.err.println(``Instance of some class (probably'' +
            ``a layer) could not be created'');
          else if (we instanceof appia.AppiaException)
            System.err.println(``Some channel could not be created'');
        }
        else e.printStackTrace();
} catch (java.io.IOException e) {
        System.err.println(``IO error! Probably the file was'' +
        ``not found or could not be red'');
}
```

Listing 9: Exception handling.

There are also two more exceptions that might be thrown: SAXParseException and IOException. The first one results from errors reading the XML file, like the structure not being in conformity with the DTD, or the DTD not being accessible. The second one results from errors accessing the file.

7.5 Passing Parameters to Channels

Passing parameters at execution time raises some issues on the way to provide access from sessions to their respective parameters. The solution devised is to encapsulate all the parameters of all the sessions of the same channel in one object. This object is an instantiation of the class appia.xml.utils.ChannelProperties. This class provides two methods that allow the user to add and get parameters for some selected session.

```
class appia.xml.utils.ChannelProperties {
        void putParams(String sessionName, SessionProperties params);
        SessionProperties getParams(String sessionName);
}
```

It was also necessary to introduce another object that encapsulates parameters for one session. This object is an instantiation of the class appia.xml.utils.SessionProperties and also provides several methods for adding and getting each parameter.

```
class appia.xml.utils.SessionProperties {
        void setProperty(String paramName, String param};
        boolean getBoolean(String paramName);
        byte getByte(String paramName);
        short getShort(String paramName);
        int getInt(String paramName);
        long getLong(String paramName);
        float getFloat(String paramName);
        double getDouble(String paramName);
        String getString(String paramName);
        char[] getCharArray(String paramName);
}
```

These two classes are very well documented in *Appia*'s *javadoc*. If the reader wishes to take full advantage of these features please refer to these documents. To illustrate the usage of these two classes we take the example shown in Listing 8 and complete it with the parameters needed by the *Ecco* layer.

```
appia.xml.utils.SessionProperties sp =
  new appia.xml.utils.SessionProperties();
sp.setProperty("localport","2000");
sp.setProperty("remotehost","foobar.di.fc.ul.pt");
sp.setProperty("remoteport","2000");
appia.xml.utils.ChannelProperties cp =
  new appia.xml.utils.ChannelProperties();
cp.putParams("ecco_s",sp);
try {
        appia.Channel ch = AppiaXML.createChannel("ecco_c","ecco_t",
          null,cp,true);
} catch (appia.AppiaException e) {
        System.err.println(''Error creating channel'');
}
```

Listing 10: Passing parameters to a channel at runtime.

The reader should now look carefully to the similarities between this code and the channel creation part of the XML configuration file shown in Listing 1.

7.6 Developing Parameterizable Sessions

In order to use the facilities provided by *AppiaXML* that allow the passage of parameters to sessions either at configuration time (Section 6.2) or at execution (Appendix 7.5), the session should implement the interface appia.xml.interfaces.InitializableSession.

This interface declares only one method. This method takes as argument a SessionProperties object that holds that parameters to be passed to the session. It is up to the programmer to retrieve this values as explained in Appendix 7.5 and do whatever is necessary with them.

```
interface appia.xml.interfaces.InitializableSession {
  void init(appia.xml.utils.SessionsProperties params);
}
```

8 Example

To demonstrate the features offered by *AppiaXML* an example application was developed. This application is a *Messenger*-like application used as a group communication tool. It allows an user to send messages to the other members of the group and also features a whiteboard drawing area shared by the group. The application can be found in demo.xml.Messenger. A more detailed description of its internals and behavior can be found in the respective section in *Appia*'s *javadoc*.

This application is provided as a pedagogical tool as it enables users to explore different channel configurations. For that purpose two useless protocols were developed: appia.test.xml.PeriodicSendSession and appia.test.xml.IntegritySession, which can be used for different configurations. Once more, details on these protocols can be found in their respective *javadocs* from *Appia*'s main *javadoc*.

8.1 Running the Test Application

A GossipServer is required by the application to support group communication [3]. In order to run this server the user has to issue the following command:

```
java gossip.GossipServer -port <port>
```

Finally, the application can be executed with the following command:

```
java demo.xml.Messenger <username> <gossip_host> <gossip_port>
<xml_file> <secret>
```

All the parameters have their usual meaning. The last one should only be used when the Integrity protocol is being used and represents the secret[3] that should be shared by all users of the group.

8.2 Experimenting with Messenger

AppiaXML is very easy to learn by example. As we said above in this section, we developed the Messenger application along with the two useless protocols as pedagogical value to combine well with this tutorial. We, thus, encourage

[3]This secret is a string that is shared by all the members of a group to ensure the integrity of the exchanged messages. For more details on this refer to the source code of the Integrity protocol provided.

the reader to experiment with these protocols or with any others available or developed for that purpose.

The user may find useful to compare the excerpt of a possible configuration shown in Listing 11 with the one provided by default with the Messenger demo named *messenger.xml* in the *demo/xml* directory. The complete configuration used in Listing 11 is also provided with the demo and is available in the file named *messenger_int.xml*.

```
                                    ...
<session name="integrity" sharing="global">
 <protocol>
  appia.test.xml.IntegrityLayer
 </protocol>
</session>
<session name="msg" sharing="label">
 <protocol>
  appia.test.xml.MessengerLayer
 </protocol>
</session>
                                    ...
```

Listing 11: Experimenting with the configuration.

The session introduces a new layer bellow the application layer that does integrity checking of messages. This layer is also shared by all channels that wish to use it. As the reader can see, by simply adding three lines in the configuration file, it is possible to add new layers that add extra properties to the channel.

9 Conclusions

This tutorial presents an extension to *Appia* called *AppiaXML* that allows the configuration and creation of channels using a description in XML. The usage of this extension simplifies the proccess of deploying and reconfiguring communication stacks.

Along the document the reader is invited to experiment with *AppiaXML* using a simple application called *Ecco* developed for this purpose.

A *AppiaXML* DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT appia (template*,channel*)>
<!ELEMENT template (session+)>
<!ATTLIST template
          name CDATA #REQUIRED>
<!ELEMENT session (protocol)>
<!ATTLIST session
          name CDATA #REQUIRED
          sharing (private|label|global) #REQUIRED>
<!ELEMENT protocol (#PCDATA)>
<!ELEMENT channel (chsession*)>
<!ATTLIST channel
          name CDATA #REQUIRED
          template CDATA #REQUIRED
          initialized (yes|no) #REQUIRED
          label CDATA #IMPLIED>
<!ELEMENT chsession (parameter*)>
<!ATTLIST chsession
          name CDATA #REQUIRED>
<!ELEMENT parameter (#PCDATA)>
<!ATTLIST parameter
          name CDATA #REQUIRED>
```

References

[1] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 707–710, Phoenix, Arizona, April 2001. IEEE.

[2] H. Miranda, A. Pinto, and L. Rodrigues. *Application Program Interface Specification of Appia (version 1.2).* Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, July 2001.

[3] A. Pinto. *Appia Group Communication Manual.* Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, February 2001.