A CPL to Java compiler for dynamic service personalization in JAIN-SIP server

Paolo Falcarin
Paolo.Falcarin@polito.it
Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi, 24
I-10129 Torino, Italy

Abstract

Service personalization is a key feature of next generation networks: different standards and technologies have been proposed to meet this requirement in an easier and more flexible way. Among these, using standard APIs to abstract behavior of different network protocols and defining service behavior with scripting language can drive to a more flexible way to realize user-personalization of call-handling policies. To reach this goal, this paper presents an efficient architecture and implementation of personalization policies, using standards like CPL (Call Processing Language) and JAIN-SIP. The novelty of this approach is the improvement of performances, enhancing a JAIN-SIP compliant proxy server with advanced Java features, like the dynamic remote class loading of profile objects. These are asynchronously generated by a CPL to Java compiler, in order to reduce overhead due to CPL interpretation and remote calls.

1. Introduction

Service creation in NGNs addresses the main feature of service programmability. This means the ability of implementing new services following the customers' needs, and to differentiate the offer faster than competitors. Another important key point is the ability to offer to customers the same service everywhere, providing a seamless access from different terminals (mobile phones, soft-phones, UMTS phones…).
These goals can be obtained opening up levels of programmability to third parties or also to the users that want to personalize their own services. As stated by Eurescom P1109 project [1], SIP [3] is the preferred technology to address NGN communications.
Application development in a NGN context is in many aspects very close to Internet application development. As a matter of fact, the main development skills required from NGN application developer are related to Java and XML (eXtensible Mark-up Language) [10].
Using XML and its dialects like a scripting language to specify service behaviour has been recognized as the first step to ease service configuration and personalization both from the developer and user point of view. This approach has motivated the creation of other standards like CCXML [8], used to extend VoiceXML [13] functionalities, and SCML [4] , designed to be a generic service specification relying on Parlay [14] and JAIN APIs [9].
Scripting Languages are lightweight, highly customizable, and typically interpreted languages, appropriate in the area of rapid application development, acting as glue to provide connections among existing components. These characteristics allow them to be used to code or modify applications at runtime, and interact with running programs. These qualities and features make scripting languages applicable to the field of application programmability next to the classic approach based on Application Programming Interfaces (APIs).
Scripting languages represent, in an XML-based file, the service behavior that can be changed at run-time; they act like a dynamic reconfiguration of the script interpreter that follows a pattern of registering the static events and criteria that can be matched by events by the underlying network components, followed by declaration of service logic that should be executed in response to such an event. Typically, scripts are created, edited, and validated using regular editors or as a result of applying transformation techniques.
For example the Call Processing Language (CPL) is such scripting languages that connect existing components with a particular API, depending on the script file content. This approach enables easy end-user customization; if users are allowed editing their scripting files, they can directly personalize the service behavior.
A whole variety and type of APIs are emerging in NGN products providing different levels of functional abstraction, and initiatives like JAIN, and SIP-servlet [5] are examples of emerging standards in the Java Community.

In this paper it is described the CPL2Java project architecture, a personalization service relying on a SIP server, based on CPL language, following these objectives:

- Low response time, in particular the part where the behavior of incoming call policies is evaluated;
- Extensibility: the system should be easy to modify and enhance with new functionalities; CPL allows using extensions to be adapted to different contexts.
- Interoperability: system should be easily integrated with other standard components.
- Dynamicity: system allows dynamic download of code containing incoming call-handling policies.

JAIN-SIP API has been chosen because of its maturity and availability of reference implementations, like the one of NIST [15] that has been used in this project.
In the next section the CPL2Java service architecture is presented; after that some scenarios are described, and finally performance considerations are discussed.

2. CPL2Java architecture

The main goal of CPL2Java service architecture is to build a personalization service relying on a SIP server, based on CPL language, but introducing some advanced Java features like dynamic code downloading, and CPL to Java compilation.

CPL is used by the service user to define his/her own personalization policies and to insert these policies in the CPL2Java architecture. Its service architecture (see figure 1) is based on two independent components:

- Profile Server Manager (PSM) is external to the SIP server and it receives CPL scripts from a web application; then it converts the script in Java objects (called Profile objects) and these are stored in a database.
- ProfileManager (PM) is an extension of the JAIN-SIP proxy and it executes Profile objects, handling dynamic downloading of this code whenever an invite comes from the proxy and thus from the SIP network.

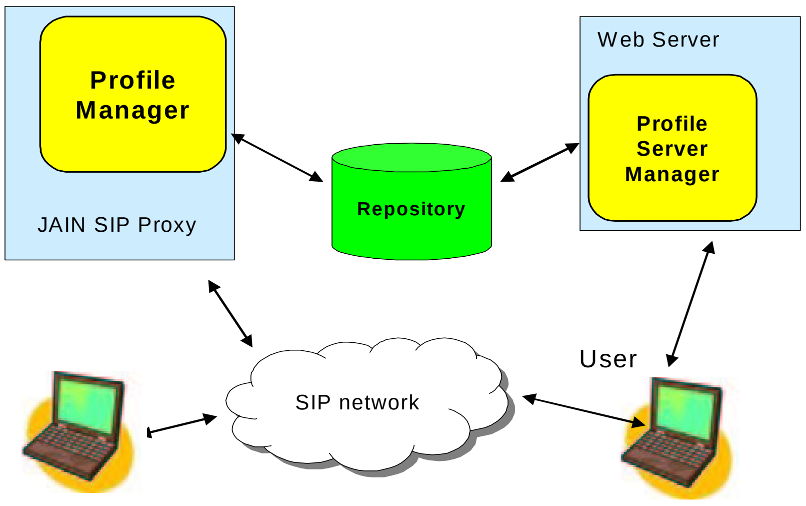

Figure 1: CPL2Java service architecture

Conversion of CPL files in Java objects is obtained through the creation of a new Java class for each new CPL script inserted by the user. This means that the PSM analyzes the CPL profile and basing on this, it creates a Java class, compiles it and store it in a repository. After that the PM needs to dynamically load (at runtime) a class and to instanciate an object. Usually, CPL server works following two different strategies:

1- every time a call comes to the SIP server, a service logic within it, reads the current CPL script and it directly interprets it, executing different SIP commands on the underlying SIP stack.

2- The CPL interpretation is delegated to a remote server that is in charge of handling CPL profiles and give back a response to the SIP server

Both the solutions meaningfully slow down set-up time of the call. In the first case the slow performance of XML parsers and the dimension of complex CPL scripts. In the other solution the necessity to make a remote procedure call is implicitly a bottleneck specially with high level of network traffic. In order to avoid these types of bottlenecks a better solution is to create, starting from a profile in CPL format, a pure Java object, whose behavior is equivalent to the specific profile. This Java object will be used by the proxy to directly manage set-up of a call. In this way interpretation of the CPL document is performed once when the profile is updated. The PSM is just a compiler from CPL to Java, and it generates a directly usable Java class: this operation is longer than the single interpretation of a CPL profile, but the advantage is that it is executed once, in a differet process host of the proxy that can continue handling calls, without losing time to wait for the interpretation or compilation of user's CPL file. Therefore the proxy, as soon as it receives a call, it has only to download the profile object from the repository and to istantiate it. This will be in charge of managing all the incoming calls for this user, and it will be reused for all the following calls that will have the performance of a local call, because the object has already been created and there is no need to download it fro mthe repository, unless a new CPL script is provided by the user. This type of solution guarantees an increased speed of call set-up, compared to traditional solutions: in fact the most time-expensive operations are executed not during the arrival of a new call, but during the definition of the new CPL file.
Before describing CPL2Java service details, a brief deepening of JAIN-SIP and CPL standards is necessary for a better comprehension: in the following subsections these standards are described, specially for the issues related with CPL2Java project.

2.1 JAIN-SIP architecture

In order to understand how the profile manager extends the Proxy implementation relying on JAIN-SIP, its architecture provided is depicted in figure 2.

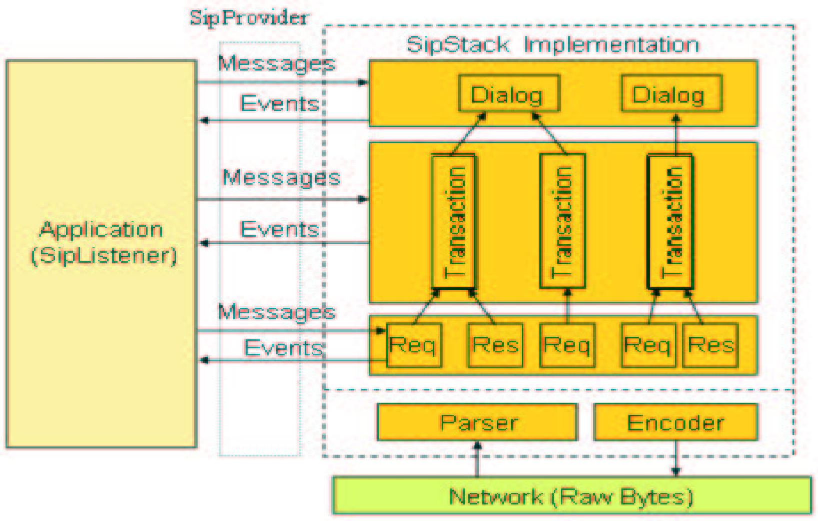

Figure 2: Architecture of JAIN-SIP implementation

The JAIN SIP API [9] is a Java API and it is only aimed at SIP User Agent type applications, which clearly define the kind of network capability exposed. Its methods expose SIP User Agent capabilities while hiding a few protocol details.
The NIST implementation of JAIN-SIP also contains an implementation of a SIP proxy server that has been extended in this work to enable user-defined call control features through CPL scripts.
The Profile Manager extends the Proxy behavior using Profile Objects to handle calls coming from the network, through the underlying SIP protocol stack. The SIP stack parses incoming SIP

messages and encodes outgoing messages, handling sessions and transactions in a transparent way with respect to the application level.
JAIN-SIP implementation wraps on the SIP stack implementation and it offers to applications the SipProvider interface, in order to receive messages coming from the application. On the other side the application must implement the SipListener interface to be notified of events coming from the SIP network. The main methods of the SipListener interface are the following:

```
 -  public void processRequest(RequestEvent requestEvent)
 - public void processResponse(ResponseEvent responseEvent)
 - public void processTimeout(TimeoutEvent timeoutEvent)
```

An application, like the Profile Manager, that wants to be notified of SIP events, implements this interface. In particular the NIST implementation of the SIP Proxy has been modified and extended with the Profile Manager, in order to directly handle CPL-based user profiles: this means that the SIPListener interface has been implemented by the Profile Manager in order to be notified of SIP events coming from the network.

2.2 CPL (Call Processing Language)

CPL is a scripting language defining how to handle outgoing and incoming calls in NGN networks. CPL was developed as an XML-based scripting language tailored for SIP proxy server.
CPL is mainly intended for non-trusted end users to upload their services on SIP servers. CPL scripts created by end users can be uploaded to SIP servers for call set-up in a secure environment. CPL is lightweight, efficient, easy to implement, extensible because it is possible to add customized features in a way that existing scripts continue to work. CPL exposes a main network capability: the call control feature, defined with CPL scripts that could be retrieved from the Application Server via HTTP GET/POST (even, dynamically generated from Server Side components). CPL is a high-level abstraction language because also users can easily write and edit their applications. As CPL is XML-based, the kind of interface offered to the application level is the scripting language itself. There is a range of commercially available CPL products and it should be noted that probably all Application servers deployed in SIP/H323 network have support for CPL. CPL could be used for implementing services in a number of different scenarios: using scripts created by the end users and uploaded to a server, or using scripts created by the server administrators on behalf of the users, or using scripts created by web applications that translate it in CPL. Because CPL is a standardized language, it can be used to allow third parties to create or customize services for clients. These services can then be run on servers owned by the end user or the user's service provider. CPL is a quite mature language and it is fully specified. However, CPL cannot originate calls towards two or more users because it is activated only through call related events, and cannot be used to create complex scripts. This is because CPL structure is defined on DTD (Document Type Declaration) [11], that is one of the language used to define and validate XML files structure, but it is not so flexible like XML-Schema that is more extensible and it is easier to learn because is similar to XML. A basic example of CPL script is depicted in the following figure; this script can be read as: "for each incoming call arriving in the time interval starting the 3rd of July 2003 at time 18.00, lasting 8 hours, for each day, then forward the call to the SIP address *sip:paolo@voicemail.polito.it*".

```
<cpl>
  <incoming>
    <time-switch>
      <time dtstart="20030703T180000" duration="PT8H" freq="daily">
        <location url="sip:paolo@voicemail.polito.it" clear="yes" />
      </time>
    </time-switch>
  </incoming>
</cpl>
```

3. CPL2Java Architectural Components

ProfileManager (PM) is an integrating part of the JAIN-SIP proxy and it retrieves the object managing a determined profile and then it gives it back to the underlying section of the proxy handling incoming calls. The PM is made by various components: the Repository Manager handling the profiles repository, the Decoder Manager, the local Cache storing the used profiles. When the set-up of a call is started, the PM checks for the related user's profile in the local cache; if it is not present it requests the profile object to the repository in order to download the new profile object. As this object is coded, PM has to decode it using the profile decoder, then it saves it in the local cache and finally it passes it to the underlying proxy. If the profile is already present in the cache, it is directly used without accessing to the repository. The cache management is important because this affects memory allocation and response time of the system.

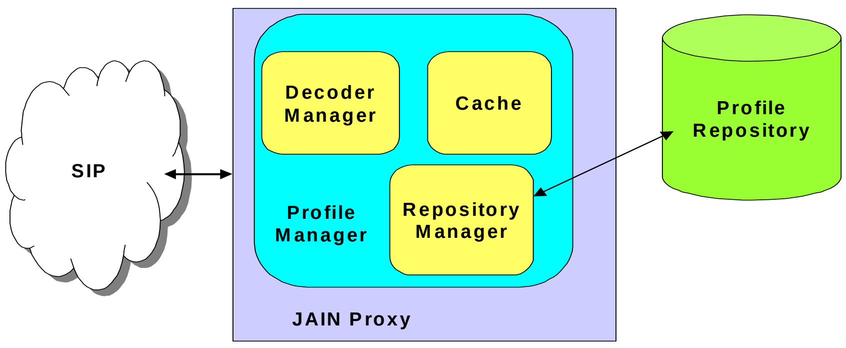

Figure 3: Internal architecture of Profile Manager

The basic cache policy used is dependent by profile object's timestamp and the number of stored profiles per user. In particular when the number of profiles crosses a defined threshold, the cache activates a cleaning procedure which deletes profiles with oldest timestamp. The class implementing the cache policy can be easily replaced with a more specific class in order to define new criteria for cache handling; this allows performance tuning of the system, according to specific requirements of workload and traffic, related to deployment context of the system.

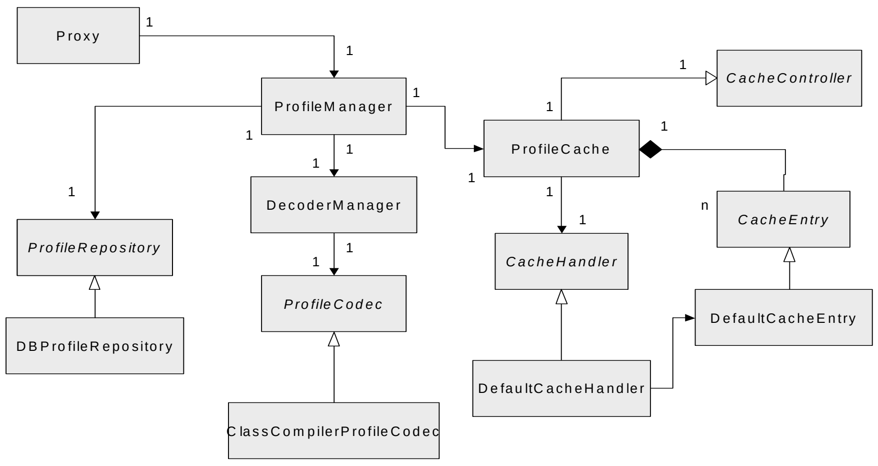

Figure 4: Class Diagram of Profile Manager

The ProfileServerManager (PSM) is the profile administrator and it runs as a web application supporting JSP (Java Server Pages), because users' CPL files and service configuration are managed through a web interface. PSM is made of these components (see figure 5):

- Web interface

- Encoder Manager, made of CPL compiler and Profile encoder
- Repository manager

The web interface is made of Java Server Pages and it is used by service subscribers to create and manage their own existing profiles.
The creation of CPL scripts is performed in two ways: uploading an existing CPL file made with an external editor, or with a graphical wizard applet handling a subset of CPL tags.

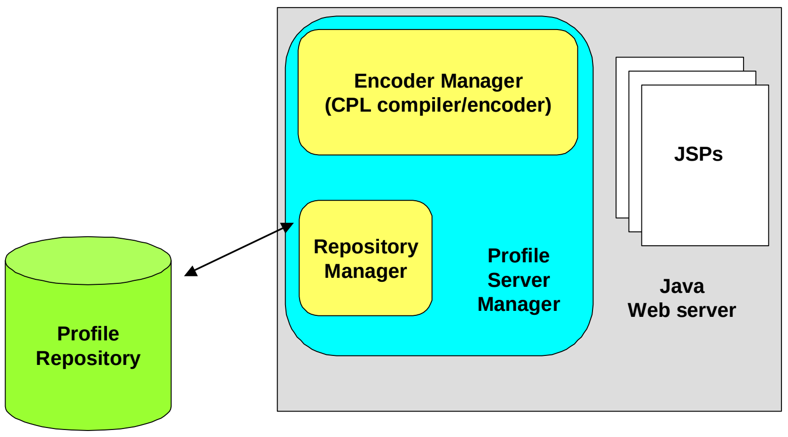

Figure 5: Profile Server Manager architecture

Repository manager's duty is the profiles management in a database; these data will be used by PM when it needs to download a profile object in the SIP server.
The repository contains relevant data for each user profile: the user name, last modification date, the compiled profile object and the related CPL file.

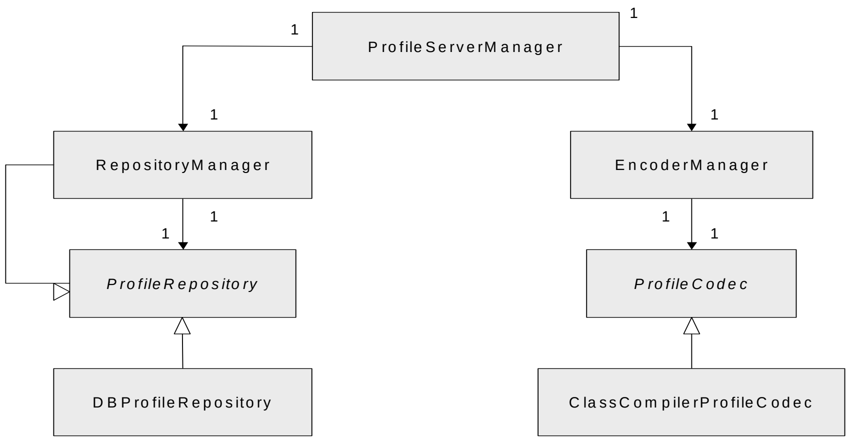

Figure 6: Class Diagram of Profile Server Manager

The main task of PSM is transforming CPL scripts in the corresponding Java classes usable by the SIP server; thus it acts like a CPL to Java compiler, parsing a XML file to Java source code and then compiling to bytecode.
The ProfileRepository (see figure 6) is an interface that allows inserting, retrieving, deleting, and listing profile objects and the corresponding CPL files. This approach separates repository generic

interface from repository implementation details. In the CPL2Java implementation there is only one reification of ProfileRepository, called DBProfileRepository; this uses JDBC (Java DataBase Connectivity) is used to communicate with a database (mySQL [17]), but it is possible to create an implementation to communicate with a LDAP (Lightweight Directory Access Protocol) server or another one to use HTTP with a remote repository behind a firewall.
A profile object must implement the *Profile* interface and its implementation depends on the kind of supported codec; in general, a codec is a component able to encode and decode generic data in a defined format: in this case the data to encode are Java objects but the way in which they are encoded depends on the selected format.
The interface ProfileCodec is then used to abstract different possible codec implementations and in this work, the default codec translates a CPL script in a Profile Object, which must contain at least one ProfileCodec implementation.
In this manner the architecture is more independent from the kind of strategy used to create and encode profiles, and specifying the default codec in a configuration file, allows changing coding policy without recompiling service classes.
Pushing this pattern a step further, DecoderManager and EncoderManager classes are utilized to manage different codecs at the same time.
Two solutions have been considered and detailed in the next section:

- Dynamic (e.g. run-time) creation of Java classes depending on CPL profile ad downloaded by PM through dynamic Class-Loading mechanism.
- Creation of objects persistent in the system and transmitted through object serialization.

3.1 Considerations on dynamic Classloading in Java

In CPL2Java project the first approach of dynamic creation of Java classes has been used. With this approach a new class, implementing Profile interface, is created for each different profile of each user: then a single instance of this class is created by PM.
Encoding phase is as follows:

- The CPL document is parsed and the corresponding Abstract Syntax Tree (AST) is generated. An AST is the hierarchical representation of the syntax structure of language elements in the program. During AST construction syntax correctness can be verified and an object model of the document is built to further verify semantic validity of the CPL script.
- Then a Java source file is generated implementing the profile interface. The implementation reflects behaviour of CPL instruction and then is compiled with a standard java compiler.

The criterion used to set class name is based on concatenation of SIP username and a pseudo-random alphanumeric string dependent from date and time: this is necessary to univocally identify a user profile depending on creation instant ad user identifier.
Profile instantiation is done during decoding phase, more precisely:

- A custom class-loader is created to download a ProfileObject class;
- An instance of this class is created using Java reflection APIs.

In Java architecture a Classloader class is responsible to load in memory a related Java bytecode file, usually stored in the underlying File-System; once the bytecode is in memory the JVM links it when needed to the running application.
Downloading additional classes by the network is a typical characteristic of applets, but Java language's inheritance features allows defining a customized Classloader to be utilized when needed. This flexibility is bound by some constraints set by the JVM: for example a Classloader cannot load two different classes with the same fully qualified name. This limit is needed to protect JVM consistency in case of misuses of this mechanism, e.g. two different implementations of the same class running in the same application or new version of the class lacking of some methods present in the former implementation. This means that once a class is loaded in memory its name cannot be dismissed and a new version of this class implementation cannot be loaded with the same name. Thus, changing the class name for each new version of the profile object is a workaround to avoid namespace conflicts in the JVM.
This conflict is only possible within the same Classloader; in fact it is possible to load two different classes having the same name but with two different Class-loaders. Anyway in this work the decoding mechanism is based on a single Classloader, because of memory issues. In fact for each user there are many profiles, one common custom Classloader, and for each profile there is a class its related single instance; in case of one Classloader per class there will be a greater waste of memory.

The main advantage of this approach is the relevant lowering of execution time of the profile behaviour, because of its implementation in pure Java bytecode.
On the other side, memory occupation slightly increases because each profile requires a different class definition and a related object instance.
In order to avoid a faster memory occupation it is necessary to set a different Classloader management policy in order to trigger garbage collection when there is a need of profile objects cleaning.
Another evaluated approach, but not used in the CPL2Java implementation, is based on creation of a class for each CPL language element, so profile object is an aggregation of these objects, reflecting the structure of the corresponding CPL script: in this way the navigation of a CPL script corresponds to objects collaboration.
This approach would lead to a process starting with CPL translation to the corresponding AST like in the previous approach; then, after a semantic check, the profile is assembled following AST's structure; finally the object obtained is serialized and stored in a ProfileObject.
In this way the CPL translation to Java would be simpler but the overhead introduced by multiple method calls in the profile object, would increase response time that would be also dependent by height of CPL document tree.
Regarding the memory, in this case the number of objects would depend on CPL script complexity, more precisely, on the number of tags occurring in the CPL document.
This problem does not allow identifying a clear maximum bound of the number of objects allocated in memory and this is the main reason why the serialization approach has been discarded.

3.2 CPL to Java compilation scenario

The Profile Server Manager contains the CPL to Java compiler realized by the ProfileEncoder component, whose relevant classes are depicted in the diagram of figure 7.

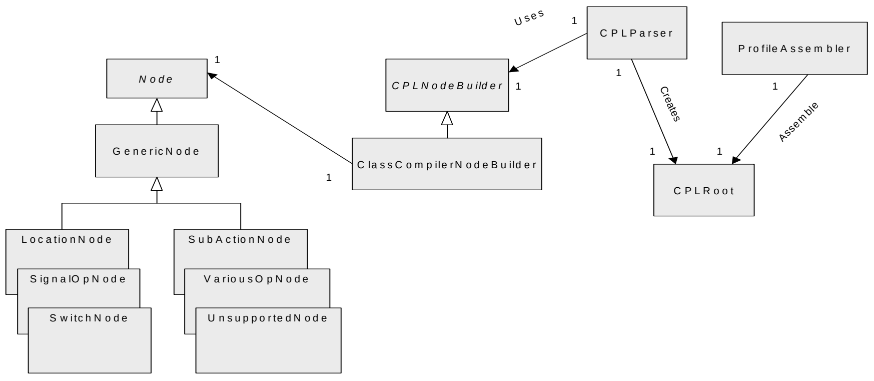

Figure 7: Class Diagram of the Profile Encoder

The Profile Encoder uses the CPL Parser to read the CPL document, to validate it with the corresponding DTD (Data Type Definition), and to build the related AST (Abstract Syntax Tree).
Moreover the Profile Encoder uses a ProfileAssembler, which translates the AST in Java source code, and the standard Java compiler to create the bytecode.
The CPL Parser analyzes the XML document and it validates it with respect to the CPL DTD.
A DTD is a document defining structure of an XML dialect, where all constraints on tags and attributes are specified. This task simplifies the following syntactic check of the CPL document. In addition, CPL parser checks parameters validity and absence of loops that are forbidden by CPL specification.
CPLParser uses a ClassCompilerNodeBuilder to generate AST nodes used by codec. In general it is possible to write a different CPLNodeBuilder and a different set of nodes in order to use the same parser even if the kind of codec changes.
ProfileAssembler takes the generated AST and it creates a Java source file for the user profile. In particular this class recursively explores the nodes tree; during the tree navigation, each marked node generates a related piece of Java code. Then the ProfileAssembler uses a file called "profile template" (that is a generic and incomplete Java implementation of the profile behaviour) as a basic skeleton of the new Profile class to be created: putting the pieces of Java code in the profile

template the user's new profile class is then generated. During this phase eventual errors in the CPL script and in the consequent Java source code are notified, stopping the CPL to Java bytecode conversion. Decoding phase is done in the PM component included in the JAIN-SIP proxy server. The previous scenario is detailed in figure 8.

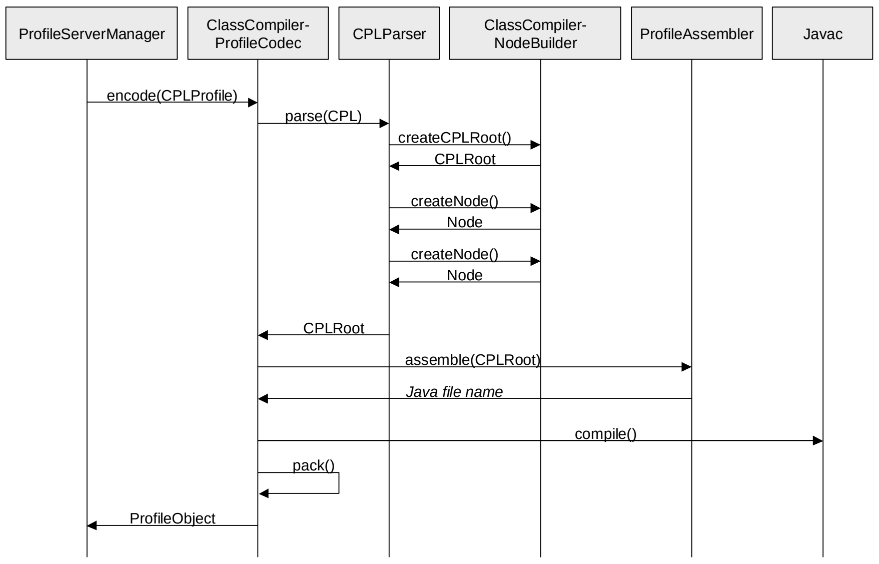

Figure 8: Sequence diagram of CPL to Java compilation

The scenario of the creation of a new profile for a user is depicted in figure 9 and it mainly involves the PSM component.
Using a web interface a user can directly insert the CPL script in a textbox or uploading a CPL file stored in the local file system.
Moreover a graphical applet wizard is available to select tags and to set values, producing the corresponding CPL script in the textbox.
Once the CPL profile is defined, CPL can be uploaded by the user to the ProfileServerManager which will activate the current ProfileCodec (in this case ClassCompilerProfileCodec) in order to generate the corresponding Java class; then the obtained bytecode is stored in the repository with the related CPL script; this can be subsequently modified by the user.
Then PSM will notify a new user profile insertion to the PM in the SIP server, through a remote method call with Java RMI (Remote Method Invocation) on the CacheController interface of the PM. If another profile of the user is currently used in the SIP server, this will be deleted, forcing the PM to call the ProfileRepository, whenever a new call for this user arrives.
This behaviour is necessary to avoid useless calls to the repository every time a user profile object is not currently present in the SIP server. The Profile modification scenario is similar to the profile creation scenario, but the latest CPL script is transferred from the repository to the user's browser.

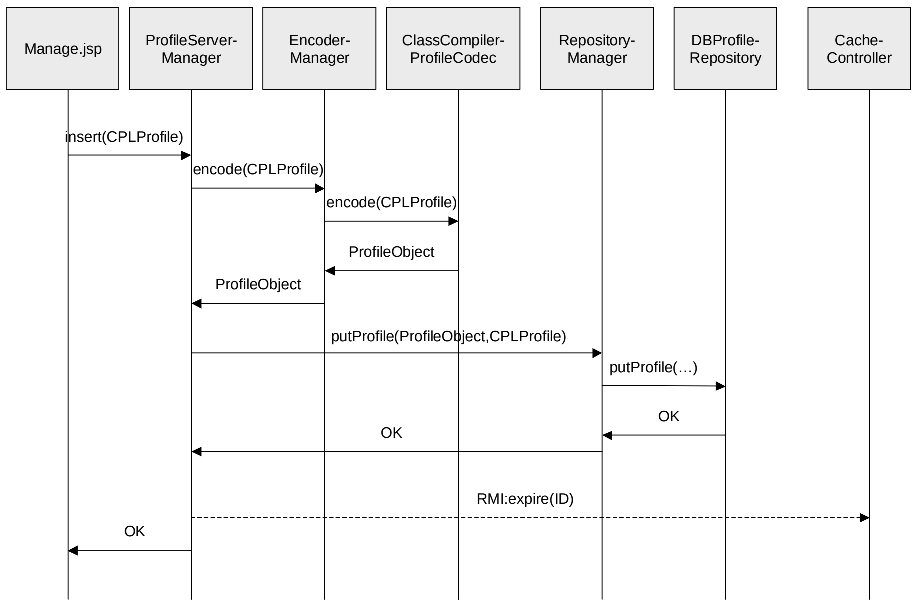

Figure 9: Profile insertion scenario

3.3 Call scenario of Profile Manager

Profile Manager has to execute the user's personalized service in the SIP server. As depicted in figure 10, when the SIP proxy server receives a SIP call for a user the PM looks for the current profile object of the called user. The ProfileManager checks out the local ProfileCache, and as it is the first call for this user, the request is forwarded to the ProfileRepository. This will return a ProfileObject, and the ProfileCodec will decode it in the PM.

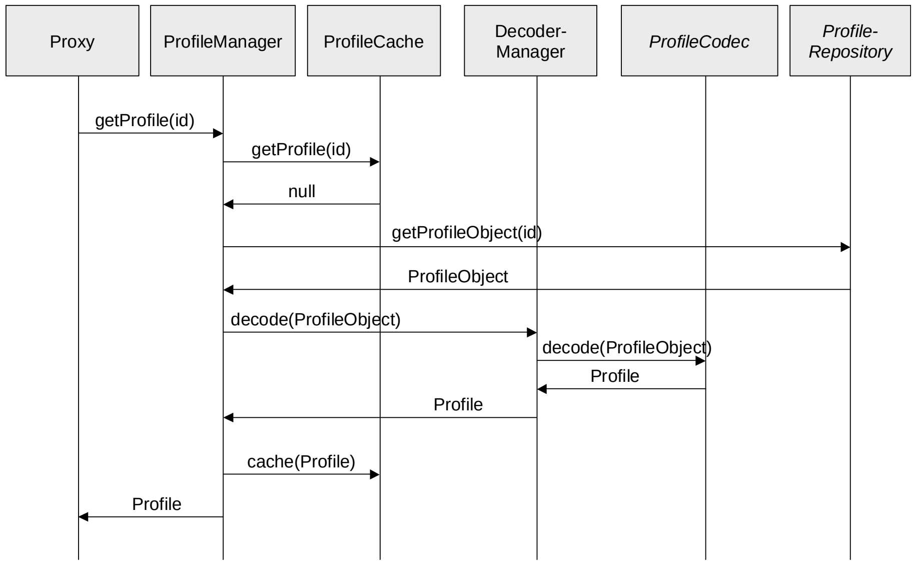

Figure 10: Sequence diagram of the first call

If the profile is not present in the repository, a marker is inserted in the cache: the default call handler will be invoked whenever a subsequent request arrives in order to avoid useless calls to the repository that is still lacking of a user-defined CPL profile.
When a CPL profile is inserted in the repository, this event is automatically notified to the PM and uploaded to the SIP server; this scenario is depicted in the following figure.
If a new profile has been inserted in the meanwhile, PSM notifies this event to the cache (through a Java-RMI call) so it can delete current profile object in the PM while PSM is compiling the new CPL into a new Profile object. As the cache has no entries for the current user profile, the PM retrieves a new profile object from the repository, like in the first-call scenario.

4. Conclusions

The system has been deployed and tested on Windows and Linux platforms with JDK (Java Development Kit) 1.4.x, on machines with Pentium processor with a clock frequency of 2 GHz and 512 MB of RAM memory, in a 100 Mbps Ethernet LAN.
PSM has been deployed on Tomcat and the repository on the MySQL open-source DBMS server because of its usability and performance.
The supported scenarios depend on the implemented part of the CPL specification, which are:

- Simple call scenario: this occurs when the call is forwarded to the SIP URL defined in the "proxy" tag.
- Choice scenario: this occurs when a CPL "switch" element is present; the "switch" can be of type "address", "string", and "time", and the related SIP tags are taken by the SIP INVITE message. Nested switch conditions have also been implemented.
- Redirection: call redirection can be realized using the "redirect" tag or using "location" and "proxy".

Multiple and conditional redirection scenarios: the call is redirected to another user if the first SIP URL is not available (e.g. busy), using "proxy" tag outputs.
The main goal of the system is reducing response time even if a CPL policy is used to give more flexibility to the user with service personalization, without penalizing performances.
The set-up time of a call does not relevantly increase using the approach (see figure 11), even for the first call to a new defined profile.

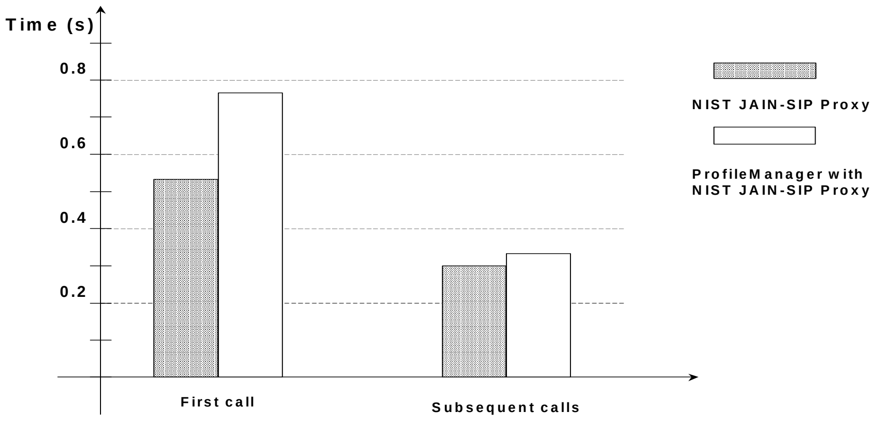

Figure 11: Response time comparison

In this case the response time slightly increases because of two factors: downloading time from repository and synchronization between PM and PSM; as the CPL compiler in the PSM works in a separate network node, the compiling does not interfere with call handling in the Profile Manager of the SIP proxy, because the PM cache is asynchronously updated by the PSM with the new Profile Objects.
Moreover the CPL compilation seldom happens because the insertion of a new CPL script is usually made by user during first system accesses, whilst after some time, the user can reuse his/her previous CPL scripts stored in the repository.

The benefit of the approach is due to dynamic code insertion in SIP server implementation and the extension to other scripting languages would impact on a part of the architecture (PSM), requiring less effort than other implementations deeply integrated in the SIP server.

5. Acknowledgements

The author wants to thank Marco De Nittis for his implementation efforts in part of the CPL2Java project, during his collaboration with Politecnico di Torino.

6. References

[1] Licciardi, C.A., Falcarin, P., Analysis of NGN service creation technologies. In Annual Review of Communications vol. 57, IEC, December 2003.
[2] Licciardi, C.A., Falcarin, P., Next Generation Networks: The services offering standpoint. In Comprehensive Report on IP services, Special Issue of the International Engineering Consortium, October 2002.
[3] Handley, M., Schulzrinne, H., Schooler, E. and Rosenberg, J., SIP: Session Initiation Protocol, RFC 2543, March 1999.
[4] Bakker, J.L., Jain, R., Next Generation Service Creation Using XML Scripting Languages, 2002. On-line at www.argreenhouse.com/papers/jlbakker/bakker-icc2002.pdf
[5] SIP Servlets specification. On-line at http://www.ietf.org/internet-drafts/draft-peterbauer-sip-servlet-ext-00.txt
[6] Lennox, J. and H. Schulzrinne, “CPL: A Language for User Control of Internet Telephony Services”. On-line at http://www.ietf.org/internet-drafts/draft-ietf-iptel-cpl-04.txt
[7] JAIN Java Call Control (JCC) API, Java Specification Request (JSR) 21", 2001. On-line at http://jcp.org/jsr/detail/21.jsp.
[8] Voice Browser Call Control: CCXML Version 1.0. W3C Working Draft 21st February 2002: http://www.w3.org/TR/2002/WD-ccxml-20020221/
[9] The JAIN APIs: Integrated Network APIs for the Java Platform. (White Paper), Jun. 2000. On-line at http://java.sun.com/products/jain
[10] XML (eXtensible Mark-up Language) specification. On-line at http://www.w3.org/XML/
[11] DTD (document Type Description) language specification. On-line at http://xml.coverpages.org/XMLSpecDTD.html
[12] JAIN SIP Lite specification. On-line at http://jcp.org/jsr/detail/125.jsp
[13] VoiceXML website. On-line at http://www.voicexml.org
[14] Parlay website. On-line at http://www.parlay.org
[15] NIST reference implementation of JAIN-SIP. On-line at https://jain-sip.dev.java.net/
[16] CPL2Java project website. On-line at http://softeng.polito.it/SoftEng/Projects/cpl2java/
[17] MySQL project website. On-line at http://www.mysql.com/