Flexible, service-based content presentation: The Hellenic Dissertations Presentation System

Christos KK Loverdos and Sarantos Kapidakis

National Documentation Centre,
National Hellenic Research Foundation,
48 Vas. Constantinou Avenue, 11635 Athens, GREECE,
{loverdos,sarantos}@ekt.gr

Abstract. We describe an architecture for the presentation of the digital content of the Hellenic Dissertations database, consisting of about twelve thousand records and two million pages. Our architecture is based on a flexible service registration and discovery mechanism, which can be used to reconfigure the Presentation System even at runtime. Services are responsible not only for presentation but also for various document conversions, format transformations, authorization policies enforcement and watermarking. The architecture encourages modular programming, by allowing services to cooperate.

Keywords. service versioning, incremental development, dynamic reconfiguration, servlets

1 Introduction

For a complete handling of a dissertations digital database, work at four distinct directions is required: 1) Submission, 2) Repository management, 3) Retrieval and 4) Presentation. There are systems like DIENST [3] which try to implement different aspects of the above directions. This work focuses on presentation. The proposed architecture can handle the data organisation and other issues – such as authorization – quite flexibly and transparently.

A dissertation may come in a variety of content types. Microsoft Word Document, Postscript and PDF are probably the most widely used formats. Images in TIFF or JPG format are also common. The National Documentation Centre (NDC), for example, hosts about two million TIFF images representing distinct dissertation physical pages. The situation with formats gets even more complicated by the introduction of "bundles": each chapter may be a DOC or HTML document and all the chapters are packed into a tar.gz or zip file.

Also, a client may request the dissertation in a format not originally submitted. For example, we may have TIFF images, but they are not diplayable by today's most widely known Internet browsers. We need a mechanism to automatically transform to supported formats, such as JPG. Also, watermarking may be desirable, in order to identify the origins of the distributed images and to prevent untraceable distribution. This transformation property may be needed for policy reasons: we may want not to distribute high resolution formats. Instead, lower quality formats can be freely given. Such policy enforcement has to be taken into account.

Supporting new formats, making transformations and enforcing policies in a flexible and as transparent a way as possible are our main concerns.

In Sec. 2 we describe the proposed architecture, in Sec. 3 we outline an existing implementation based on state-of-the art web technologies and finally, in Sec. 4, we conclude.

2 Architecture

2.1 Requests, services, attributes

Each Dissertation, which is our basic digital object, is uniquely identified by its Hellenic Dissertations Identifier (HDID), typically a number assigned to the dissertation during the process of its submission to NDC.

A request to the presentation system has the form of a pair which consists of an HDID and a service description: *Request = [HDID, Service]*. A service is further broken up into a service name and attributes: *Service = [Name, Attributes]*.

Services started as characterizations of what to present, but turned out to be a more general idea. For example, initially we supported services like:

- *getPage:* To support one page retrieval as an image.
- *getContentsTable:* To support the retrieval of the Table of Contents with hyperlinks to the contents.

Soon though, we created more services to support:

- *Authorization*: Not everyone is allowed to retrieve the highest quality images.
- *Content discovery:* To cope with the different storage organisation schemes.

Attributes play the role of service customization data. For example, concerning the *getPage* service, we need to supply the actual page number and that is the role of the *pageNumber* attribute. An attribute is explicitly given by the client or is implicitly given by the request as a whole – just like the HOST_ADDRESS and other “attributes” are silently passed to an HTTP server by the browser. Alternatively, the system itself may change the set of attributes at will, in order to fulfill certain needs.

An attribute name is unique for a particular service but a service name need not be unique in the presentation system. This means that we can have two services named *getPage*, but we cannot have two attributes named *pageNumber* in any of these two services. We chose the capability of multiple versions of the same service (this is actually what the same names represent), so as to incrementally support new functionality.

The key idea behind the registration and discovery of a service is the set of attributes it supports. More specifically, we distinguish between two kinds of

System Main Modules	
Service Manager	(*SM*)
Service Repository	(*SRp*)
Service Collector/Distributor	(*SCD*)

Service Manager Module	
Service Registration	(*SRg*)
Service Discovery	(*SD*)

Fig. 1. System modules.

attributes: mandatory and optional. Both types are registered along with the service name but only mandatory attributes are used to discover which service will handle a particular request. The ordered tupple $[L, m_1, m_2, ..., m_N]$, where L is the service name and m_1, m_2, ..., m_N are the names of the mandatory fields, is used to uniquely identify a service version. This is how we support multiple services with the same name and it also shows the way to evolve the functionality of a service: by adding one or more mandatory fields.

Using the previously mentioned technique, it is very easy to experiment with services. Using mandatory attributes to incrementally support new functionality, makes it relatively easy to cope with changing requirements without changing source code: all that is needed is to program a new version that may use the previous one, if it fits the needs of a particular request. This is actually the initial motivation behind the ideas presented here.

2.2 System modules

The several modules which are part of the system are shown in Fig. 1 and analyzed further on.

Service Manager (SM) This module is responsible for the registration and discovery of services. Each of these two functions is supported by a respective submodule.

Service Repository (SRp) All registered services are stored here. An implementation is the one to specify the exact nature of the repository. The only requirement imposed by our architecture is that it should support active elements, that is pieces of code which can be activated/executed at will.

Service Collector/Distributor (SCD) This module is the entry and exit point of our system, its interface to the outer world. The name Collector/Destributor has been chosen deliberately. Its role is to collect requests, distribute them to the appropriate services, collect the replies and distribute them back to the respective clients. Distribution, as used here, means that - depending of course on the overall implementation - services may be distributed over the network and even replicated over the network. This way we may gain performance and fault tolerance. Of course, since the entry and

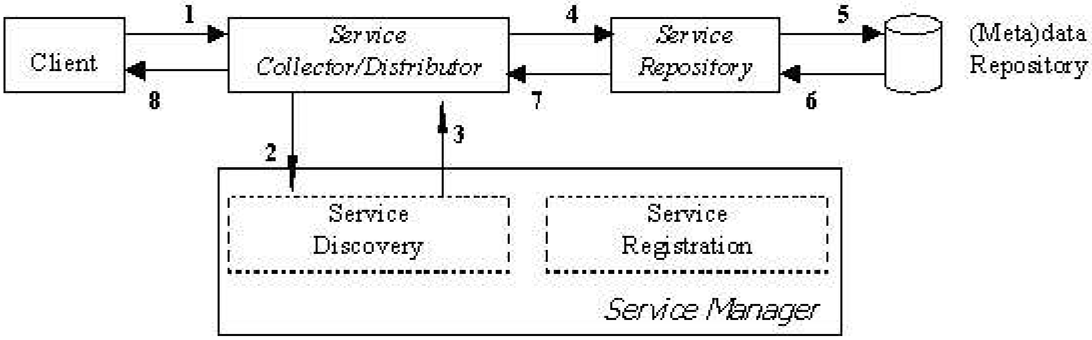

Fig. 2. Servicing Logic.

exit point is unique, it should be as powerfull as possible, in order to handle extensive request loads. Yet again, well established techniques for load balancing can be used to increase performance.

We note that no particular (meta)data databases are considered part of the system, since they can be conveniently abstracted by respective services.

2.3 Servicing Logic

In Fig. 2 we show the steps the system follows to fulfill a request:

Step 1 The client issues a request, which consists of the HDID and the desired service description. The service description is its name and the mandatory and a part or all of its optional attributes.

Step 2 The SCD module has received the request and broken it up into its constituent parts. Those are delivered to the Service Discovery (SD) module of the Service Manager. The responsibility of SD is to find a particular service version with the given service name that can handle the request.

Step 3 The SD module uses its internal service tables to discover the most appropriate service. The proposed and currently used algorithm for service discovery is a “greedy“ one, which tries to find - among all the transmitted attributes - as many mandatory ones as possible. The result is returned back to the SCD module.

Step 4 The SCD module passes the actual request to the correct handler (service), which is recalled from the Service Repository (SRp) and activated.

Steps 5, 6 The recalled handler services the request, probably using other services for intermediat results. The (meta)data repositories, are queried for the needed digital objects.

Step 7, 8 The SCD module receives the response and sends it back to the client.

3 Implementation

The aforementioned architecture has been implemented as a web application, using an Apache Web Server [2] and the Java Servlets technology. We have designed easily memorized URLs of the form http://thesis.{ndc,ekt}.gr/*HDID* to uniquely identify a dissertation over the web.

Our SCD and SM modules and each service provided have been mapped to servlets. Service attributes correspond to servlet parameters. Using the automatic servlet reloading capabilities of the servlet engine used (Apache JServ [1]), any change to a service is reflected to the runtime environment. Service registration is part of the application's (re)configuration process, which is based on configuration files. These are always processed when the application starts and every time its administrator requests a reconfiguration. The Service Repository is, in this implementation, the servlet engine and the Service Manager is backed by an API, which cooperates with the servlet API.

Typical service requests take the form of URLs. For example, if http://thesis.ndc.gr/1 is requested from a browser, it returns the dissertation with HDID=1. For this particular example, there are JPG images available and the user is given the opportunity to navigate through the whole set. If the dissertation with HDID=8775 is requested, a hyperlink to the Table of Contents is also given, which contains more hyperlinks to the actual page images. All the JPG images are automatically watermarked by a "Watermarking" service. The html version of the Table of Contents does not exist for all dissertations and is automatically discovered by a respective "Get Table of Contents" service.

4 Conclusions and Future Work

The aforementioned architecture is based on an extensible service-supporting mechanism. Easy reconfiguration and incremental development have been our main concerns in designing it. These, in turn, provide for easy administration, a shorter development cycle and flexibility in incorporating new requirements.

All the claims above have been proved in practice. The servlet-based implementation, outlined in Section 3, is now serving a database of two million dissertation pages. Services such as watermarking, automating image transformation and automatic generation of the Contents Table have been identified as requirements *during* the lifetime of the system and have, quite easily and succesfully, become a part of it with minimal effort.

Although the presentation of content has been our initial motivation, enforcing access policies and supporting other operations (like full-text searching) are in our immediate plans. The development up to this time shows us that the new functionality can be incorporated by programming new services.

It is noted that our research emerged from the very demands of our job responsibilities. It is our belief though that it can be generalized to areas requiring the presentation of digital content other than that of a Dissertions Database.

References

1. Apache JServ, http://java.apache.org/jserv/
2. Apache Software Foundation, http://www.apache.org
3. Carl Lagoze and Jim Davis, “DIENST: An Architecture for Distributed Document Libraries”, Communications of the ACM, 38(4), April 1995, p. 47