Proposal for Optimizing the Design of the National University of La Plata Service Cloud

José Nahuel Cuesta Luengo[1], Miguel Carbone[1], Claudia Banchoff Tzancoff[2], Claudia Queiruga[2], Christian Rodriguez[1]

[1]High Center for Information Processing (CeSPI), UNLP, 50 and 115, La Plata (1900), Argentina
[2]Laboratory of Investigation in New Information Technologies (LINTI), Computer Science School, UNLP, 50 and 120, La Plata (1900), Argentina
{ncuesta, mcarbone, car}@cespi.unlp.edu.ar
{cbanchoff, claudiaq}@info.unlp.edu.ar

Abstract. One of the great difficulties facing the informatic systems development team of the Department of Development of the CeSPI of the National University of La Plata (UNLP) is the use and maintenance of the service cloud. This issue triggered a theoretical-practical analysis of the state of the art around service oriented architectures and a proposal for a new design for the service cloud.
A real use case was implemented from this design proposal, as well as a reduced cloud architecture, which included everything from a Gateway API routing the incoming requests to services organized according to the guidelines of the microservices pattern. This approach encourages the decoupling for the architecture components, simplifying its development, maintenance, deployment and scalability.
This paper presents the advances achieved in the redesign of the UNLP service cloud and a case study implemented on it.

Keywords: Cloud, Web Service, API, REST, Microservices, Software Architecture Design Patterns, ESB, API Gateway, SOA, Loose Coupling.

1 Introduction

CeSPI[1] is the computing center of the National University of La Plata (UNLP) and its mission is to encourage, implement and manage ICT[2]. The participants in the work here presented include developers and systems analysts of the Direction of Development of the CeSPI and teachers of the Computer Science School, who have been involved in the survey, implementation, maintenance and production of multiple daily use Web applications in the different academic units (high schools and university schools belonging to the UNLP) and other administrative dependencies of the University. In the course of over 7 years of work, multiple architectural paradigms have been used for the development and integration of these applications.

The implementation of the current UNLP architecture is an application based on a REST architecture, which we call “Integrador”, and whose main function is to concentrate the information, making it easier for different applications to unify and correlate data. Although its implementation was adequate and functional for the needs of the moment in which it was developed, the change in requirements and technological advancement have posed the need for a new analysis that led to the solution proposed in this paper.

1 High Center for Information Processing of the UNLP – http://www.cespi.unlp.edu.ar/
2 Information and Communications Technology

Section 2 describes the current UNLP service cloud, the "Integrador", as well as the difficulties of its use and maintenance that triggered the proposal offered in this paper. Section 3 enumerates the technologies evaluated to achieve the final redesign proposal described in section 4. Section 5 describes the case study and finally conclusions are exposed in section 6.

2 The current UNLP cloud

"Integrador" is the implementation of the UNLP cloud service currently in use. It is a Web-based system that concentrates information, allowing multiple Web applications to unify data, combine and correlate the information each of them has. It contains information such as unique identifiers for different types of documents (e.g., 1 equals National Identity Number, 2 equals Personal Identity Booklet Number, 3 equals Passport Number), values identifying the academic units or dependencies of the University (e.g., 33 is the Computer Science School, 26 is the CeSPI, etc), and data related to persons linked to the UNLP, among others.

The services provided by "Integrador" can only be consulted, i.e., it is not possible to modify or destroy them. The applications that consume this information, cloud clients, access it by means of different Web services.

When analyzing the current state of the cloud service, we find a great amount of monolithic applications containing strongly coupled components, which makes changing, testing and deploying them difficult [1, p-29].

The design of "Integrador" does not allow efficient escalation. In order to achieve that, it is necessary to replicate the virtual instance of "Integrador" and implement a load balancer that allows redirection of requests to their replicated instances. The same happens when escalating the APIs that compose the cloud – they escalate poorly. The strong coupling of the applications and their respective APIs makes it necessary to replicate each application and its API, as they have an API implemented in the same virtual instance. The generation of so many applications with their respective APIs results in a proliferation of point-to-point connections, generating dependencies between them.

As regards caching, each client implements their cache without server directives, i.e., there is no centralized caching policy or shared cache, which generates unnecessary processing in the APIs.

Given the amount of applications that use the services of this cloud for its basic functioning, it is highly relevant to optimize its performance, minimize maintenance and achieve a scalable and fault tolerant architecture.

3 Analyzed technologies

The aforementioned issues triggered the research and generation of a proposal for their solution. The following section describes the evaluated technologies that were used to base decisions made regarding the optimization of the UNLP service cloud architecture.

3.1 Service-Oriented Architecture

A Service-Oriented Architecture (SOA) establishes a design framework for integrating distributed and independent applications, allowing network access to its features, offered as services. SOA is generally implemented by means of Web services, a standard based technology and independent from the platform that

provides the data. Thus, SOA can decompose monolithic applications into a set of services [2, p. 2].

This architectural model includes practices and processes based on the fact that distributed systems are not controlled by the owners themselves. Different equipments, dependencies or even different organizations may manage these distributed systems. In the past, a great amount of methods has been proposed to solve the problem of the integration of distributed systems by means of the elimination of heterogeneity, but experience has shown that these approaches do not work. [3, p. 14].

Distributed systems from medium to large scale usually have different owners, which are usually heterogeneous. The approach followed by SOA accepts this heterogeneity based on the principle that it is necessary to deal with the fact that most legacy systems that are already in production will remain in production [3, p. 15].

3.2 Microservices

One of the ways to implement SOA is by means of the microservices architecture pattern. Monolithic applications generally consist of strongly coupled components, which are part of a single deployable unit, which is unpractical and generates difficulties as regards changing, testing and deploying the application. For this reason, big IT organizations with applications with these features tend to use monthly cycles for the deployment of their products.

The microservices pattern deals with these matters, separating the application into multiple deployable units called service-components[3], which may be developed, tested and deployed independently from another service-components [4, p. 27].

As shown in Fig. 1, all client requests are performed through a layer called *user interface layer*, which is in charge of accessing service-components.

Taking into account that the main components of an application are divided into smaller deployable parts individually, applications built using the microservices architecture pattern are generally more robust, provide better scalability and can provide support for continuous delivery, enabling production environment updates by means of real time deployments, eliminating the need for monthly or weekly updates [4, p. 33]. All this is possible given that the change is generally reduced to a single service-component and only the units that change must be updated. As previously mentioned, this is a noticeable improvement in the face of monolithic application development, where the strong coupling of its components leads to fragile applications that tend to fail with each new deployment.

3 A service-component is a service logic grouping basic unit, its granularity may vary from a simple module to a great part of an application.

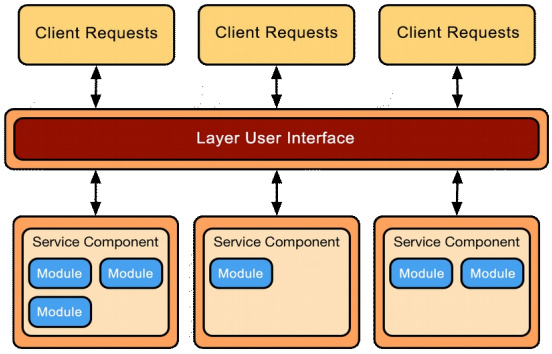

Fig. 1. Basic microservices architecture.

Although there are multiple ways to implement the microservices architecture pattern, the three main topologies are: REST-based API, REST-based application and centralized messaging [5, p. 29]. According to the analysis performed, the centralized messaging topology presented in Fig. 2 is the one that will be implemented, due to its adequacy to the needs of the project. This topology is similar to the REST-based application topology, only that instead of using REST to access a service component remotely, it uses a centralized and light message broker.

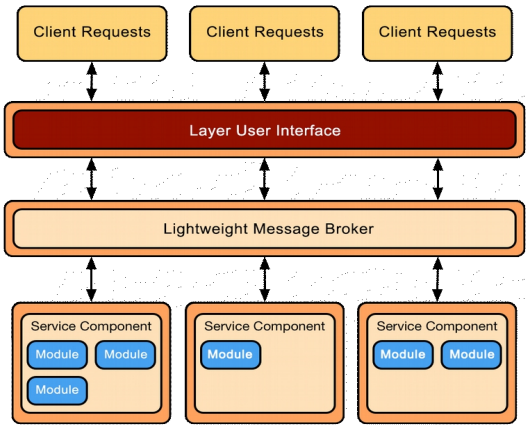

Fig. 2. Centralized messaging.

3.3 Enterprise Service Bus

Although UNLP service cloud applications are integrated, this integration generates a great amount of point-to-point connections, resulting in a high degree of coupling. One of the main SOA design principles is service loose coupling and a usual way to implement it is through an Enterprise Service Bus (ESB): a means of communication that connects and abstracts service providers and consumers.

ESBs themselves do not implement service oriented architectures, but provide the features by means of which it is possible to implement one, i.e., they provide an

abstraction layer for the endpoints, thus achieving flexibility and easier service connection.

There are multiple different opinions regarding the exact role and responsibilities of an ESB, mainly due to the existence of multiple technical approximations to implementing an ESB [3, p. 47]. According to the technical and organizational approaches adopted for the application of the ESB, it may imply one or more of the following tasks:

- Providing connectivity.
- Information transformation.
- Intelligent routing.
- Management of security aspects
- Dealing with service reliability.
- Service management, activity monitoring and logging.

4. UNLP services design optimization proposal

The previously exposed analysis resulted in the proposal of a service-based architecture that is more decoupled than the current service cloud, thus minimizing maintenance and development costs and simplifying its deployment in production environments. The proposed solution is based on standards that allow the integration of heterogeneous systems, accepting the fact that most legacy systems that are currently in production will remain in production, thus making the underlying infrastructure facilitate the incorporation of changes that may be needed in the CeSPI and UNLP.

The service model facilitates information access and consumption through the network. Given that services are independent and autonomous, they can be combined as many times as needed in a simple manner, generating new applications that respond to the constantly evolving needs of the UNLP [2, p. 3]. The possibility of adding and combining services makes this strategy a highly beneficial option, with the goal of creating complex services and applications that are independent from underlying technologies. Therefore, the service cloud design optimization has taken into account features such as redundancy, scalability, decoupling, simplicity and standard use. The new name for the proposed service cloud is “Cloud”.

4.1 Redundancy and Scalability

The application scalability model called scale cube [6] classifies the different ways of scaling applications in 3 axes: X-axis scaling, Y-axis scaling y Z-axis scaling. The “Integrador” and the APIs integrating the cloud do not scale efficiently (X-axis scaling), which is the reason for the proposal of a new UNLP service cloud architecture that must be replicable and scalable.

In order to achieve this, each API must be executed in an independent virtual instance, so it can be replicated as many times as necessary (X-axis scaling). The abstraction layer of these replicated instances will be a load balancer, implemented with NGINX, that will serve both for load balancing and for failover, giving continuity to services in case that one of the replicated instances fails. The load balancer will be able to distribute incoming requests to a group of servers (backends) according to a decision and weighing algorithm called scheduler. This structure that is transparent to the client avoids direct accesses between service providers and consumers.

A shared cache was implemented in front of the load balancer using Varnish[4], which avoids unnecessary access by keeping an in-memory copy of the replies generated by one of the replicated instances, achieving better response times.

4.2 Decoupling

As previously mentioned, one of the principles of SOA design is loose coupling, frequently implemented by means of an ESB, in charge of providing interoperability among different platforms.

The integration achieved in the current service cloud results in the proliferation of point-to-point connection between systems, which is often known as a "spaghetti architecture" (as exemplified in Fig. 3), generating dependencies between the applications. Although the problem of interoperability between applications is solved, it is difficult to maintain [7, p. 4].

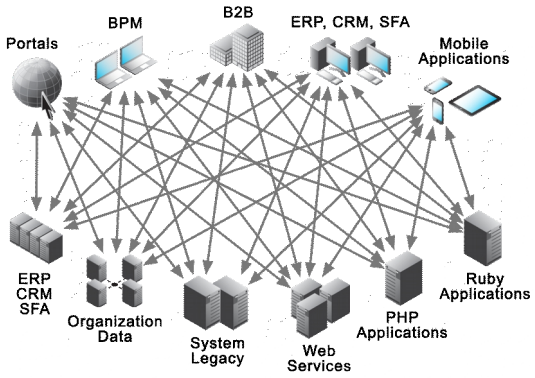

Fig. 3. Scheme of point-to-point connections.

In an architecture implementing ESB, applications communicate through this central bus acting as a message broker among them. This architectural system is called "mediation". The number of point-to-point connections necessary to allow communications between applications is minimized, simplifying the maintenance and deployment of a system while decreasing the degree of direct dependency that may exist between the instances. As shown in Fig. 4, consumers still use the same endpoint where the task is delegated, when messages arrive, the mediator distributes them to the instances that provide this service [3, p. 52].

The new architecture will use Tyk[5] as an ESB in front of Varnish, adding mechanisms for authentication, rate limiting, monitoring and a single access point to all the cloud. Moreover, including the ESB avoids the need to develop extra functionalities in the APIs.

In the interest of working on the decoupling of the platforms, the proposal is to develop an API dedicated to serving reference data in which to implement the necessary services that will allow access to this information from different applications. This API will comprise the endpoints of the Integrador that will be in use.

4 Varnish is an HTTP reverse proxy, sometimes refered as an HTTP accelerator or Web accelerator, that stores files and file fragments in memory which reduces response times and bandwidth for the same requests [4, p. 20].

5 API Gateway Open Source - https://tyk.io/about-tyk

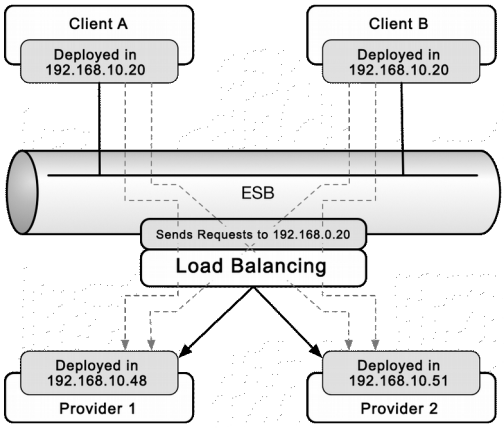

Fig. 4. Scheme of connections with mediator ESB.

Additionally, it is necessary to decouple the APIs from their respective applications and rewrite those that are currently in production using Ruby on Rails as a development framework. Version 5.0 of Ruby on Rails incorporates a new Web application creation mode: api mode. This mode launches applications configured with a reduced subset of components obtained from the elimination of those unnecessary for Web API development, and prepared to serve content in JSON format by default, seizing the main caching techniques for this type of applications.

Applying the microservices pattern will result in different service-components that used to be implemented in each application and coupled to them, and will now be implemented in a new independent and dedicated application. Thus, the logic of the application is decoupled from the data generated by it, allowing it to scale horizontally in a simple manner. This solution will allow independence from the language used for the application development: application can be developed in Ruby, PHP, JavaScript, Java or any other language different from that used to develop the APIs, generating an independent service layer where applications will delegate to the APIs the access, management and persistence of data that they themselves generate.

4.3 Simplicity

Monolithic applications consist of strongly coupled components that are part of one single deployable unit, which makes it difficult to apply changes, test and deploy without service failures. The microservices pattern deals with these matters, separating the application into multiple deployable units, which can be developed, tested and deployed independently from one another [1].

Dividing the application into smaller, deployable components that are independent facilitates development, testing and production due to the change being isolated from a service-component, enabling better control.

4.4 Standard use

Standardization is the process by means of which commonly accepted norms are established that allow communication between different applications[6]. Therefore, for each application that requires so, an API must be implemented to give access to the

6 Dictionary of the Royal Spanish Academy

data it generates, based on the JSON API[7] specification. This standardization facilitates the development of clients that consume information from different services of the APIs, since they define rules that specify how the data can be accessed, what will be their reply structure and even assist in achieving independence from the language in which these clients are written, which must only respect the specifications to implement access to services.

5. Case Study

In order to test the proposed architecture, the process of registering a new Single Sign-on (SSO) user for the UNLP service cloud was used as a case study. All the agents that are part of the UNLP staff have a single user to access the SSO integrated applications, among them, to consult their pay stub, use an application as part of their daily tasks, present extension projects or access any application that may be developed as a UNLP service in the future.

In the process of self-management of a new user, the agent must complete the requested data in a series of previously-established steps leading to access to the applications using the SSO scheme. We can summarize the registration process in the following steps:

1. The agent enters the self-managed application and indicates they wish to register a new user. For this, they must enter their institutional email account in order to receive an access link to effectively begin the process of registration of a new user.
2. Upon accessing the link received by email, the application will ask the agent to identify with the type and number of Identification Document, asking to confirm that they are still among the University agents without a single access user.
3. Once the agent is identified, they are suggested a username following the username policy, which can be modified as part of the process of entering information. This username is confirmed as vacant and as abiding by the policy.
4. Once the username is selected, the agent is asked to enter an alternative email account for a second means of contact.
5. To finish entering personal data, the agent is asked to indicate the dependency of the UNLP where they are employed. This is asked with a captcha to avoid bots attempting mass registration of new users and to keep ill-intentioned users from registering users in the name of real agents.

Decomposing the services of the self-managed process, we identified the following dependencies as services of "Cloud":

- **Reference services**: to visualize and mark dependencies (or academic units) and lists of identification document types.
- **Staff information services**: to identify the person and later consult the academic units in which they are employed.
- **User services**: to consult the existence of a username for the selected person, suggest usernames that abide by the defined name policy and the creation of the new name.
- **Notification service**: to send emails.

Developing the functional prototype implied:

- Implementing the aforementioned services (reference, staff information, user and notification), in which we only included the endpoints necessary for the logic of the case study.
- Implementing a new version of the registration client application, respecting the aforementioned steps, that will consume all the information of the services of "Cloud".

7 Specifications to build APIs in JSON - http://jsonapi.org

- Developing a gem[8] that encapsulates the service access logic (client) from authentication to query and abstraction of objects from the response of their APIs, based on the JSON API standard.

For a more detailed explanation, it is advisable to consult the thesis "Proposal for the Redesign of the UNLP Service Cloud (Propuesta de rediseño de la nube de servicios de la UNLP)" [8].

6. Conclusions

The proposed architecture is scalable and more robust, with high availability, organized in layers (Gateway, Caching L1, Balancing, Services, Caching L0 and Persistency), all this independent from the technologies used in each layer. The technology of any of the layers can be changed transparently without it affecting service providers or consumers.

With the concept test we obtained a notion of the cost entailed in implementing services following the microservices pattern and taking these changes to our applications, which allowed for an estimation of the time and resources necessary for the implementation of this change in existing and future developments.

The key point and benefit of the proposal is the decomposition of application into microservices that simplifies development, testing and deployment, similarly achieving looser coupling and easier extension of the service cloud.

Using an ESB allows for the replacement of a point-to-point connection topology by a star topology, simplifying maintenance and error detection. At the same time, the functionality is deleted from the APIs (authentication) and new functionalities are achieved (rate-limit, monitoring, and logging, among others), centralized and delegated to another layer.

7. References

1. Richards Mark. Software Architecture Patterns - Understanding Common Architecture Patterns and When to Use Them. O'Reilly, Febrero 2015.
2. Microsoft Corporation. La arquitectura orientada a servicios (soa) de microsoft aplicada al mundo real, Diciembre 2006.
3. Josuttis Nicolai M. SOA in Practice: The Art of Distributing System Design. O'Reilly, paperback edition, 2007.
4. Francisco Velázquez; Kristian Lyngstøl; Tollef Fog Heen; Jérôme Renard. The Varnish Book. Manning, 2016.
5. Richards Mark. Software Architecture Patterns - Understanding Common Architecture Patterns and When to Use Them. O'Reilly, Febrero 2015.
6. akfpartners.com. Splitting applications or services for scale, Mayo 2008.
7. D'Emic David, Dossot; John. Mule in Action. Manning, Febrero 2010.
8. Cuesta Luengo, José Nahuel; Carbone, Miguel. Propuesta de rediseño de la nube de servicios de la UNLP. Junio 2016.

8 Name given to reusable libraries in Ruby