A distributed infrastructure for publishing VO services: an implementation

Cepparo, Francesco, Scagnetto, Ivan, Molinaro, Marco, Smareglia, Riccardo

Francesco Cepparo, Ivan Scagnetto, Marco Molinaro, Riccardo Smareglia, "A distributed infrastructure for publishing VO services: an implementation," Proc. SPIE 9913, Software and Cyberinfrastructure for Astronomy IV, 991328 (26 July 2016); doi: 10.1117/12.2232686

SPIE.

Event: SPIE Astronomical Telescopes + Instrumentation, 2016, Edinburgh, United Kingdom

A distributed infrastructure for publishing VO services: an implementation

Francesco Cepparo[a], Ivan Scagnetto[b], Marco Molinaro[a], and Riccardo Smareglia[a]

[a]INAF - Osservatorio Astronomico di Trieste, via G.B. Tiepolo 11, 34143 Trieste, Italy
[b]Department of Mathematics and Computer Science of the University of Udine, via delle Scienze 206, 33100 Udine, Italy

ABSTRACT

This contribution describes both the design and the implementation details of a new solution for publishing VO services, enlightening its maintainable, distributed, modular and scalable architecture. Indeed, the new publisher is multithreaded and multiprocess. Multiple instances of the modules can run on different machines to ensure high performance and high availability, and this will be true both for the interface modules of the services and the back end data access ones. The system uses message passing to let its components communicate through an AMQP message broker that can itself be distributed to provide better scalability and availability.

Keywords: Astrophysical publishing environment, VO services, Distributed system, Modular architecture, Scalable architecture, Multithreaded, Multiprocess, Multiple instances, Message driven, High performance, High availability

1. INTRODUCTION

A complete astrophysical publishing environment, working as a helper system for an astrophysical data center, requires various components, from custom data back ends up to standardized front end solutions. VO technologies, standards and implementations, promoted by the International Virtual Observatory Alliance (IVOA), are a suitable solution for front end interfaces devoted to data discovery and access. A publishing system to generate VO services out of configuration steps can be a useful help to data providers and, for this reason, the Italian center for Astronomical Archives (IA2, INAF) data center developed a first prototype to perform this task. However, being the software monolithic and difficult to maintain due to standard's changes and scalability requirements, a new solution was needed. This contribution describes both the design and the implementation details of the latter, enlightening its maintainable, distributed, modular and scalable architecture. Indeed, the new publisher is multithreaded and multiprocess to be scalable both vertically and horizontally. Multiple instances of the modules can run on different machines to ensure high performance and high availability, and this will be true both for the interface modules of the services and the back end data access ones. The system uses message passing to let its components communicate through an AMQP message broker that can itself be distributed to provide better scalability and availability. An initial implementation of the system has been realized by a couple of IVOA Simple Cone Search[1] based services. The first one uses a servlet as the interface module and a JDBC based MySQL database as the back end one, thus relying completely and coherently on Java technology. The second one, even if it is using a similar servlet interface, passes its incoming requests through the broker to a Python coded back end module, accessing a JSON formatted catalogue. Both the services were made to work together in two different scenarios: a simple cloud environment and a single machine. In particular, the second deployment was carried out with the aim of comparing performances with the previous publishing solution, which will be discontinued. It is worth to notice that, even if this environment is not optimal (since it does not fully exploit the scalability features of our new proposal), the performance comparison with respect to the previous publisher resulted very encouraging to further pursue the development of this new publishing framework, which was born after a previous, failed, attempt to develop a modular VO oriented service deployer using Java EE technologies.[2]

Software and Cyberinfrastructure for Astronomy IV, edited by Gianluca Chiozzi, Juan C. Guzman, Proc. of SPIE Vol. 9913, 991328 · © 2016 SPIE
CCC code: 0277-786X/16/$18 · doi: 10.1117/12.2232686

2. USE CASE

The objective of this work is to implement a flexible and robust system for publishing VO services. The system needs to support HTTP parametric requests and provide tables in the VOTable format as response. However, the system also needs to be easily extensible for supporting new VO standards. Another requirement is the reliability of the system, so that a fault in one of the supported services does not hinder the whole system and all the other services deployed through it. Those requirements can be fulfilled if the system is designed to be modular. Moreover, by making all the modules of the system independent from one another, we can also support the usage of different programming languages when implementing the modules themselves. Thanks to the modular design, the system can also be broken down in smaller components, which can be distributed on multiple different machines, guaranteeing high availability and high performance. Those requirements, and thus the new system, were born to address all the shortcomings of the previous service publishing system used at the IA2 data center.

3. SYSTEM ARCHITECTURE

The developed distributed system is totally modular and scalable, and can be summed up by figure 1. The figure shows that the user, when performing the request, encounters a reverse proxy that redirects the request to one of the interfaces, balancing the load upon them. The interfaces then make a RPC request by using message passing, using the Advanced Message Queuing Protocol (AMQP), and send the message to the broker, which can be clustered on more than one node. At this point the broker delivers the message to the appropriate data access module, depending on the service called by the user, which is identified by the URL of the request. In addition to these modules, two additional components were developed. The first one is a component needed for logging, which listens for new log messages sent through AMQP. The second one is a component that is in charge of providing data access modules with the configuration of the corresponding service. The system was then distributed on three nodes, made of Xen virtual machines.

3.1 The reverse proxy

All the users' requests reach a reverse proxy whose task is to redistribute the requests to the appropriate host, to ensure high availability and load balancing. In our case, we used the software HAProxy. A simple configuration file lets you specify the servers that are listening for requests and which load balancing algorithm to use.

3.2 The interface

The interface that the system exposes to the user is implemented by a Java web application, which can be run on any application server. This web application listens on a specified base URL, to which the client-side application will append the ID of the desired service, and potentially other service specific HTTP GET parameters. The interface is generic and works as-is for every type of service to expose. Multiple instances of the same interface can be launched on different hosts in order to distribute the load with the help of the reverse proxy. The job of the interface is to process the user request and to generate an AMQP message for the appropriate data access module. The latter can be identified by the ID of the service, which can be found in the URL of the request. After this, the interface waits for the response message from the data access module, which gets redirected back to the user.

3.3 The message broker

The message broker provides a message oriented middleware in order to connect the interface to the data access layer. The communication happens through message queues, that are created from the URL of the request. Deployed services that are waiting for messages on the appropriate queue can retrieve the message and send back a response to the interface on a new response queue that gets automatically created. This mechanism is nearly equivalent to a RPC call, with the difference that message passing is used instead, so that the modules can be decoupled from each other.

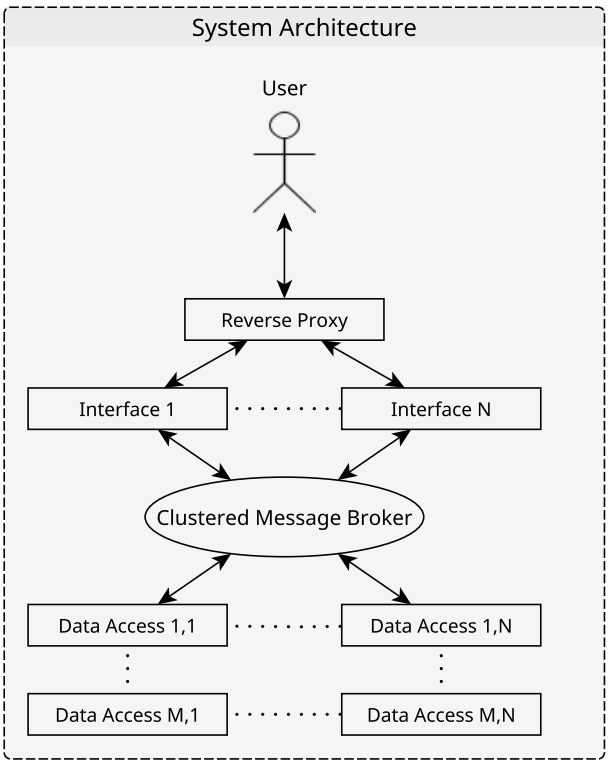

Figure 1. Architecture of the system

3.4 The data access layer

The data access layer is the component that is in charge of actually implementing the service: it extracts data from a data source, and processes it to return the result expected by the user. The data source is not predefined, as is not the programming language that can be used to implement a new data access module. This is due to the module separation allowed by the message passing paradigm. In our system, two different data access modules were written, in order to demonstrate the modularity of the system and the flexibility of this architecture. Both modules implement the Simple Cone Search[1] VO protocol. One of the two data access modules is written in Java and accesses a MySQL database through JDBC, while the other is written in Python and accesses a JSON formatted catalogue. Multiple instances of both modules can be running at the same time to generate responses for the same service or for different services. In case they are used to implement the same service, it is appropriate to ascertain that the data sources are equal and that both modules provide the same results.

3.5 The logging server

The logging server is an AMQP listener that waits on a message queue specifically created to handle log messages. It is a wrapper for the tinylog library, written so that log messages could be received through the AMQP protocol. In addition to the logging server there is also a logging client, which provides convenience methods to send log messages through AMQP.

3.6 The config repository

The config repository is another AMQP listener that waits on a specific queue. The config repository was conceived to contain the configurations of all the supported services and to provide them at runtime to all the data access modules that request them when they get started. The only mandatory parameter for the exchange of the service configuration is the ID of the service. The service configuration is specified in a Java properties file which gets serialized into the JSON format before being transmitted as an AMQP message.

4. SEQUENCE DIAGRAM

The way the system operates is described by the following sequence diagram. Put into words, when the data layer module :ConeDBAccess starts up it requests the service configuration to the config repository through the message broker. As soon as the configuration is received, the service is ready to operate. At this point, when a HTTP GET request is sent by the user, the interface :ConeRest logs the request (the communication always happens through the message broker) and forwards the request to the appropriate data layer module. After additional logging, the result is returned back to the interface, and then back to the user, in the VOTable format. Please note that while the interface was named ConeRest, the interface is in fact the same for all the types of supported protocols.

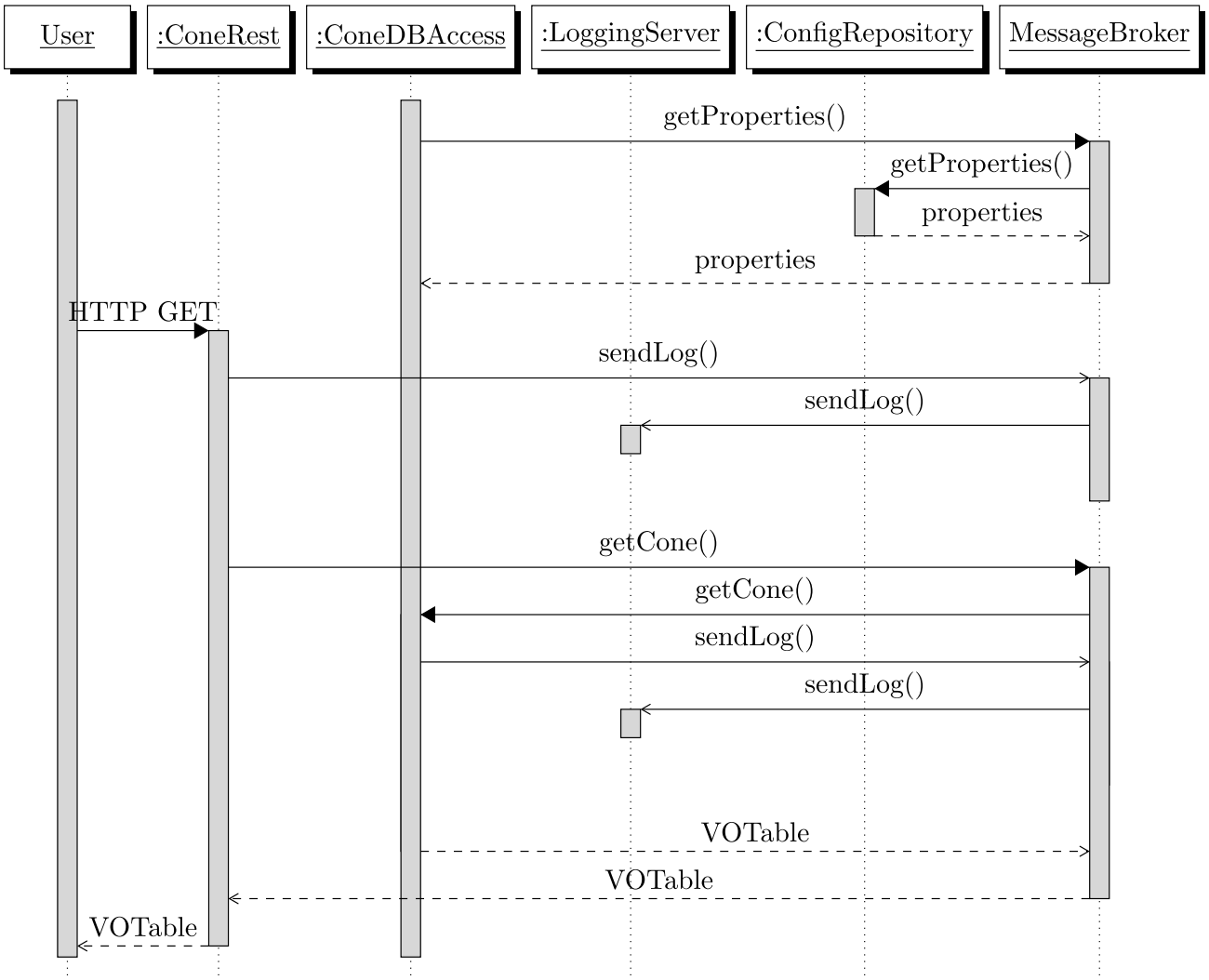

Figure 2. The sequence diagram of the system

5. ADOPTED TECHNOLOGIES

The programming languages used to implement the various modules are Java and Python. Java was used for writing the interfaces and one of the two data access modules, while Python was used for the second data access module. Python was chosen because of the possibility to use an open source project (Simple-Cone-Search-Creator)[3] written in that language. Java was instead chosen when writing the other modules because of the need

to integrate the system with the infrastructure of the IA2 data center and to use a language widely adopted by astronomical software.

To serialize the data into the VOTable format, we used the Starlink Tables Infrastructure Library (STIL).[4] This library easily let us generate a VOTable from a SQL-based source, in the Java data access module. On the other hand, the Python data access module does the serialization itself. The modification to the original open source project consists in the addition of AMQP messaging support for communicating with the Java-based interface modules, that is for obtaining the request parameters and sending back the response.

The reverse-proxy used within the system is provided by the software HAProxy, a free and open source solution which aims to be reliable and to support high availability and load balancing, in addition to proxying.

The messaging protocol used as middleware for the publish-subscribe layer is the AMQP protocol, a binary protocol used at application level and designed to efficiently support a wide variety of messaging applications and communication models. The AMQP standard describes the expected behaviour of servers and clients and the messaging format to the point of allowing interoperability between different implementations of the protocol. We chose AMQP as the messaging protocol because it has an excellent implementation and documentation, provided by the free and open source software RabbitMQ. RabbitMQ is a software written in the Erlang programming language, a language known to be well suited for distributed, fault-tolerant, highly available software. RabbitMQ also officially provides bindings to several other programming languages.

To support logging within the system, the tinylog library was used. Tinylog is a fast and easy to use, simple and lightweight solution. It is thread-safe and log entries are written atomically. It also supports the only critical requirement we had: runtime reloading of its configuration.

The system was run on virtual machines powered by the Xen Project. A solution based on KVM or Docker would surely be simpler to configure and distribute, but eventually Xen was used as the IA2 data center has a Xen-based infrastructure.

6. SYSTEM TESTING AND EVALUATION

The testing and evaluation of the new service publishing system was performed with the tool 'siege', an HTTP/HTTPS stress tester.[5] First of all, the correctness of the new system was verified by comparing the responses of the services to the responses given by the old service publishing system we used at IA2, VO-Dance.[6] After making sure that the responses were identical, we launched the benchmark. The new system was distributed on three nodes while the old system wasn't distributable and as such was deployed on a single node. As hoped, the new system had no problems in scaling and achieved almost three times the performance of the old system. The exact result of the benchmark follows. The new system was tentatively called VOBall.

```
$ siege <endpoint> -t 1 -b
```

	VO–Dance	VOBall	(Unit)
Transactions:	453	1257	hits
Availability:	100.00	100.00	%
Elapsed time:	59.63	59.26	secs
Data transferred:	0.35	4.80	MB
Response time:	1.95	0.70	secs
Transaction rate:	7.60	21.21	trans/sec
Throughput:	0.01	0.08	MB/sec
Concurrency:	14.80	14.90	
Successful transactions:	453	1257	
Failed transactions:	0	0	
Longest transaction:	3.08	1.68	
Shortest transaction:	0.38	0.15	

7. CONCLUSIONS AND FUTURE PLANS

We developed a new distributed system for publishing VO services. In particular, we developed the interface module and two different data access modules, both implementing the Simple Cone Search[1] VO protocol. To connect the different modules and guarantee high availability and scalability we added a reverse proxy and a message broker. Last, we developed a logging module and a module for handling service configurations. This architecture enabled us to overcome the shortcomings of the previous VO service publishing system. On top of this, the new publishing system provides a solid foundation upon which to add support for new VO protocols, a possible future development. We are also thinking to implement additional support modules, such as a VOTable generator and a centralized interface for administering the system. In addition to improving the publishing system itself, we are also planning to adapt two of our other software to make use of the architecture introduced with this system. The first is a data ingestion, cutting, and retrieval software written as part of the VIALACTEA project;[7] the other is IA2TAP,[8] a software implementing the IVOA Table Access Protocol.[9]

REFERENCES

[1] Plante, R., Williams, R., Hanisch, R., and Szalay, A., "Simple Cone Search Version 1.03." IVOA Recommendation 22 February 2008 (Feb. 2008).

[2] Molinaro, M., Cepparo, F., De Marco, M., Knapic, C., Apollo, P., and Smareglia, R., "Modular VO oriented Java EE service deployer," in [*SPIE Astronomical Telescopes+ Instrumentation*], 91520C–91520C, International Society for Optics and Photonics (2014).

[3] "Simple Cone Search Creator." https://github.com/tboch/Simple-Cone-Search-Creator/.

[4] Taylor, M. B., "TOPCAT & STIL: Starlink Table/VOTable Processing Software," in [*Astronomical Data Analysis Software and Systems XIV*], Shopbell, P., Britton, M., and Ebert, R., eds., *Astronomical Society of the Pacific Conference Series* **347**, 29 (Dec. 2005).

[5] "siege - An HTTP/HTTPS stress tester." https://github.com/JoeDog/siege.

[6] Molinaro, M., Knapic, C., and Smareglia, R., "The VO-Dance web application at the IA2 data center," in [*SPIE Astronomical Telescopes+ Instrumentation*], 845105–845105, International Society for Optics and Photonics (2012).

[7] Becciani, U., Bandieramonte, M., Brescia, M., Butora, R., Cavuoti, S., Costa, A., di Giorgio, A. M., Elia, D., Hajnal, A., Kacsuk, P., Liu, S. J., Mercurio, A., Molinari, S., Molinaro, M., Riccio, G., Schisano, E., Sciacca, E., Smareglia, R., and Vitello, F., "Advanced Environment for Knowledge Discovery in the VIALACTEA Project," (2015).

[8] "IA2 IVOA Data Access Layer service generation applications." http://ia2.oats.inaf.it/vo-services/vo-dance.

[9] Dowler, P., Rixon, G., Tody, D., Andrews, K., Good, J., Hanisch, R., Lemson, G., McGlynn, T., Noddle, K., Ochsenbein, F., et al., "Table Access Protocol, Version 1.0," *IVOA Recommendation, March* (2010).