THE CLARENS GRID-ENABLED WEB SERVICES FRAMEWORK: SERVICES AND IMPLEMENTATION

Conrad Steenberg, Julian Bunn, Iosif Legrand, Harvey Newman, Michael Thomas, Frank van Lingen
California Institute of Technology, Pasadena, CA, 91125
Ashiq Anjum, Tahir Azim,
National University of Science and Technology, Rawalpindi, Pakistan

Abstract

This paper describes progress made in the development of the *Clarens* Web Services Framework, including a second Java-based server implementation, improved performance, a global lookup and discovery service leveraging of the MonALISA monitoring system, and adapting the framework to a secure message-based transport protocol.

INTRODUCTION

The Clarens Web Services Framework aims to provide the basis for a consistent, high-performance, fault tolerant system of distributed web services for data-intensive interactive and batch analysis in the CMS experiment.

By leveraging existing, widely implemented standards and software components, including the HTTP protocol, SSL/TLS (RFC 2246) encryption and X509 (RFC 3280)[1] certificate-based authentication, and SOAP/XML-RPC data serialization, *Clarens* also aims to be easily accessible to a wide variety of client implementations with the minimum of software dependencies. This approach lowers the barriers of entry to participate in the service network, re-use of existing developer skills, and a wide choice of development tools and languages.

During the last 18 months development focus has shifted from designing the basic services and security architecture [9], to implementing a first set of services for CMS, as well as taking a more global view of how these services would inter-operate in large services networks. In the following paragraphs the specific work done to achieve this goal will be described.

JAVA-BASED CLARENS SERVER

In response to a strong preference for developing *Clarens* services in the well-known Java language expressed by colleagues at NUST, a collaborative effort was started to develop a second server implementation. The Java language and runtime environment has several desirable characteristics, including implementations on several platforms, a large developer community, and mature web service development tools.

The resultant *JClarens* implementation is based on so-called *servlets* implemented inside a commodity *container*, in this case the open source Apache Tomcat[11] server.

Compared with the Apache/Python-based server, the JClarens implementation shown in Figure 1 is able to leverage the existing file access (HTTP GET) functionality of Tomcat, as well as being able to connect to the JINI-based MonALISA[3] distributed monitoring system, where it has access to a wealth of monitoring information, including service description information published by other *Clarens* servers.

JClarens provides fine-grained access control list (ACL) secured access to all the core *Clarens* services[10], namely the `system` administrative service, the `file` access service, `group` VO management service, and `proxy` escrow service for storing and retrieving proxy certificates. It does not currently implement file upload or file ACL functionality.

JClarens is able to host the standard *Clarens* browser-based interface (web portal) unmodified, presenting a unified service interface, and more importantly, eliminating the need to write a second browser based client interface. This interface is implemented as a client-side Javascript application that is loaded from the *Clarens* host server as with any standard web page, and makes web service calls from within the browser.

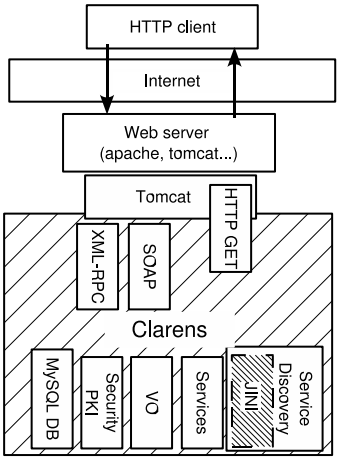

Figure 1: JClarens architecture.

DISCOVERY SERVICE

In a dynamic, global service environment the so-called *bootstrap* problem of finding service endpoints needs to be

[1] For Internet Engineering Task Force Request For Comment (RFC) documents, see `http://www.ietf.org/`

solved if such an environment is to scale to large numbers of servers and users without incurring prohibitively large amounts of administrative overhead.

In the next section a description is given of an attempt to create a distributed service registry based on the so-called *web spider* model, followed by a description of a more efficient implementation based on MonALISA.

Service Spider

Well known web search engines use a method known as *web crawling*, with agents know as *spiders* to download and collect as many publicly accessible web pages as possible. This architecture was used as the basis for a first attempt at a service discovery mechanism, where a set of Apache/Python-based *Clarens* servers acted as spiders to automatically access service descriptions of other *Clarens* services, and make these descriptions available to other web service clients, as shown schematically in Figure 2.

A web service API also provided methods to register and search for service endpoints identified by a service name, URL, server certificate distinguished name (DN) and serialization protocol (currently SOAP or XML-RPC).

Additionally, all `rendezvous` service instances would also register themselves to a configurable number of other such instances that are deemed to be 'close', in network access time, upon startup. In the case of the *active* servers would periodically repeat this process to try and ensure that even the passive servers contained reasonably up to date information of nearby servers, as shown schematically in Figure 2.

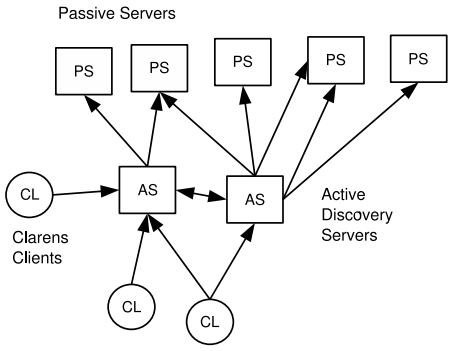

Figure 2: Discovery service implementation based on a distributed architecture of active and passive servers. The active servers continuously update their own and others' service registries.

As an optimization, each active spider was limited to making no more than a configurable maximum amount of client connections per time unit in order to minimize CPU and network usage overhead. Uncontactable servers would be remove from the list of know servers, and information about these servers would not be provided to `rendezvous` clients, including the service spiders in an attempt to provide a graceful expiry of obsolete service references.

This architecture ultimately proved to be too slow and inefficient for several reasons: a) the overhead of managing large numbers *Clarens* client sessions proved difficult to do efficiently, b) access controls implemented by servers did not always allow the service spider to even initiate a client session c) the information provided by any particular `rendezvous` service tended to be incomplete, leaving some discovery still to be done by clients themselves and d) obsolete service references expired from the registry too slowly.

MonALISA Service Publication

In the process of developing services that rely on the MonALISA monitoring framework as a data provider, it was realized that the publish-subscribe network implemented using the JINI technology also provides an ideal mechanism for publishing service descriptions. MonALISA has proven itself as an extremely scalable global monitoring system, being able to publish data from a large number of providers on short timescales.

Information provided to MonALISA is usually arranged roughly as described by the so-called GLUE schema, as a hierarchy of servers, farms, nodes and key/numerical value pairs. While not the ideal for organizing service description data, the other desirable qualities of the publish-subscribe network were too good to ignore.

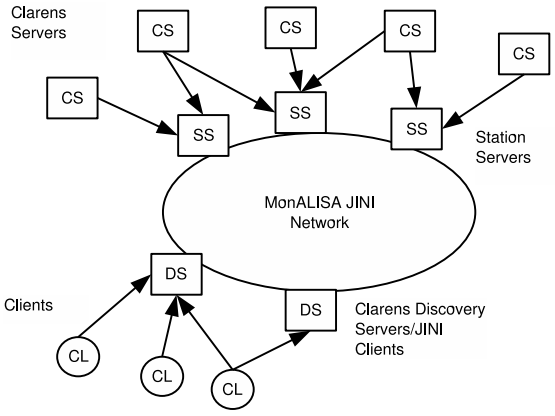

Figure 3: The MonALISA-based service discovery architecture. *Clarens* servers publish their service names and version numbers to MonALISA station servers via UDP, the latter then publish the service descriptions using the peer-to-peer JINI network. Some JClarens servers contain embedded JINI clients, which subscribe to the service information, and makes that available to *Clarens* clients via a discovery service API.

Using this hierarchy, farm descriptions named `ClarensServers.`*`VO`* are published for different VOs by each server, with node names corresponding to the service endpoint URL prefixed with the data serialization protocol, e.g. `soap.https://myserver.net`. Key

names providing service meta-data are constructed at the next lower level, e.g. `service.name` for different service names, with version numbers contained in the corresponding value entries. Server certificate distinguished names are published using a `provider.DN` notation.

Clarens servers can publish this information using the very lightweight UDP-based *ApMon* mechanism to so-called station servers that in turn republish it to the MonALISA network.

A first implementation of the above contacted a MonALISA global repository via an unauthenticated SOAP service for every `rendezvous` search request. As with the spider approach, this proved to perform very poorly, with latencies of more that a few seconds seen for many queries.

A second implementation is illustrated in Figure 3 where the JClarens server becomes a fully fledged JINI client, gathering information in a similar way to the global repository. The server is consequently able to respond to service searches far more rapidly by using information aggregated in local memory.

A recent change in MonALISA to provide for non-numerical data values to be published in the above hierarchy will allow richer service descriptions to be handled by the system. Also, *Clarens* servers currently need to have their URLs set manually by an administrator. A method `echo.hostname` has been implemented that will enable servers to look up their own URLs so that accurate information can be reported to the service registry.

In future server releases the `rendezvous` service would be renamed to the `discover` service to align the naming more closely with that proposed by the Open Science Grid[5] consortium for such a service.

PERFORMANCE MEASUREMENTS

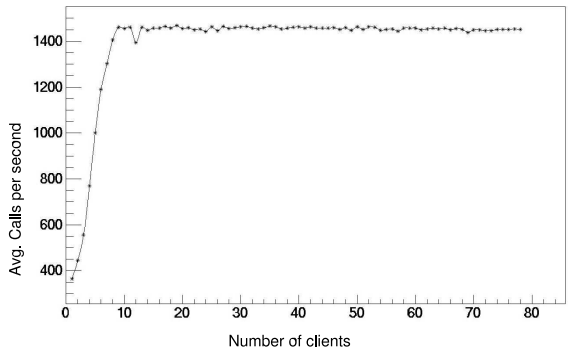

Figure 4: Transaction rate summary for up to 80 concurrent LAN-based clients

Performance is an important consideration for any server that is aimed at interactive use by a large user community. Poor performance would either lead to high hardware costs for larger servers or unacceptably long response times.

A performance and scalability test was recently performed using a CMS proto Tier-2 system consisting of a dual 2.8 GHz Xeon server with 1GB of memory, accessed using a 100 Mb/s local area network. In this test a configurable number of unencrypted client connections were opened and set to access the `system.list_methods` web service method as rapidly as possible. The client was run on a 2.6 GHz Pentium 4 workstation as a single process opening connections to the server and completing requests asynchronously, for a total of 10,000 requests per connection. The time taken for all the requests to finish was then used to compute a request rate number.

I.e. for one client connection a total of 10,000 requests would be issued, 20,000 for two clients, and so forth, up to a total of 800,000 for 80 connections. A grand total of 3.6 *billion* requests were successfully completed without any client or server errors.

Each request passed through two access control checks involving several database accesses, namely checking whether the client credentials are associated with a current session, and whether the client has access to the particular method being called. No caching was performed on the server, with each request incurring a database lookup for all registered methods in the server, and serializing the resultant list of more than 30 strings as an array response in XML-RPC. The Python client de-serialized each response to a native list object that could be used in the rest of the script.

In effect this test reports the overhead that the *Clarens* server system imposes on service requests, with control passing through all parts of the server used by a typical service.

This test under-reports the actual server performance for at least two reasons: in a more realistic environment multiple client machines would be accessing the server, and the Apache server configuration was used unmodified on a Linux 2.4-based kernel which is known to be a sub-optimal setup.

A final summary of the results of this test is given in Figure 4, showing an average of 1,450 requests per second served.

During the test the controlling Apache server process that is responsible for accepting new requests and opening new connections constantly used all available CPU time on one of the two CPUs of the test server. This is probably due to the way that the file descriptors used for network connections are handled by the Linux 2.4 kernel.

Future tests will be repeated using Linux 2.6 kernels as well as more optimized Apache configurations and SSL/TLS-encrypted network connections. Informal tests shows the latter to reduce performance by up to 50%.

MESSAGE-ORIENTED SERVICES

During the early development of the *Clarens* framework, a need was expressed by the CMS experiment for the ability to interact with Physics analysis jobs running on compute nodes. These compute nodes are generally part of private networks protected by network address translation (NAT) firewalls. Furthermore, the HTTP protocol used for

the current *Clarens* web services implementations was designed for a request response mode of operation, making it ill-suited for the type of asynchronous bi-directional communication required.

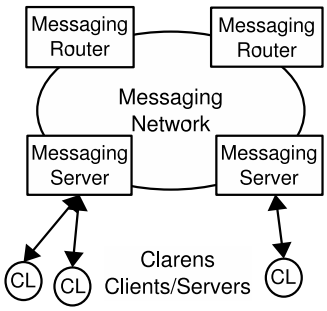

Figure 5: Architecture of the messaging service network. Clarens clients and servers exchange messages using a secure backbone consisting of servers that host messaging client connections, as well as dedicated message routers.

Instead, an approach using the ubiquitous instant messaging (IM) architecture was decided upon. This architecture, shown in Figure 5 consists of a series of high-performance servers acting as an overlay network backbone, with large numbers of clients connected to each server using persistent TCP connections. In contrast to the HTTP (web) server model, the IM server's functionality is quite simple: it needs to accept and send messages to individuals of groups of clients. These clients may be connected to the same server or to different servers forming part of the same network.

Additionally, IM servers usually allow peer-to-peer file transfer, as well as so-called *presence management*, where users can be notified whether other users are connected to the network. This has the desirable side-effect that a discovery service becomes particularly simple to implement, since presence notification is handled by the IM servers already.

The development of an IM-based *Clarens* service layer currently uses the SILC[7] protocol and tools, which was designed with security and PKI-based message encryption in mind. The RSA public/private keys used by SILC is not currently RFC 3258 compatible, requiring a translation step between the two formats at authentication time. Messages sent and received using the SILC protocol are fed into the standard Python *Clarens* server code and processed in the same way as HTTP messages (requests). This allows the server code to be re-used, and existing *Clarens* services to be made available to the IM-based clients.

Since messages can be sent and received by analysis code asynchronously, analysis jobs can be instrumented to act as *Clarens* servers, or clients sending information to monitoring systems or remote debugging tools. The most common form of such remote debugging information is expected to be the standard output of the job. Services implemented by an analysis job might include the ability to provide histogramming of certain program parameters, job control, and the ability to change job parameters interactively.

CONCLUSION

Clarens continues to be developed as a high-performance, standards-compliant and easy to use framework for implementing web services and clients. Two compatible implementations provide service writers a choice of implementation language and computing platforms, with a choice between the XML-RPC and more complex SOAP serialization formats available on both servers.

A particularly exciting development is the use of a message-oriented protocol, made possible by the protocol-agnostic nature of the *Clarens* framework.

ACKNOWLEDGMENTS

This work supported by Department of Energy contract DE-FC02-01ER25459, as part of the Particle Physics DataGrid project [6]. Clarens development is hosted by SourceForge.net [8].

REFERENCES

[1] The Clarens Web Service Framework, http://clarens.sourceforge.net.

[2] JINI Network Technology, http://wwws.sun.com/software/jini/

[3] Monitoring Agents in a Large Integrated System Architecture, http://monalisa.caltech.edu.

[4] LeGrand, I., Newman, H., "MonALISA: An Agent Based, Dynamic Service System to Monitor, Control and Optimize Grid based Applications", Paper 89, This Volume, Computing in High Energy Physics, Interlaken, 2004.

[5] The Open Science Grid consortium, http://www.osg.org.

[6] Particle Physics DataGrid, http://www.ppdg.net.

[7] Secure Internet Live Chat, http://www.silcnet.org.

[8] SourceForge.net Open Source Software Development web site, http://www.sourceforge.net.

[9] Steenberg, C.D., Aslakson E., Bunn, J.J., Newman H.B., Thomas, M., van Lingen, F., "The Clarens Web Services Architecture", Proceedings of CHEP2003, La Jolla, Paper MONT008, 2003.

[10] Steenberg, C.D., Aslakson E., Bunn, J.J., Newman H.B., Thomas, M., van Lingen, F., "The Clarens Web Client and Server Applications", Proceedings of CHEP2003, La Jolla, Paper TUCT005, 2003.

[11] The Tomcat Servlet Engine, http://tomcat.apache.org.