FroNtier: High Performance Database Access Using Standard Web Components in a Scalable Multi-tier Architecture

S. Kosyakov, J. Kowalkowski, D. Litvintsev, L. Lueking, M. Paterno, S.P. White, Fermilab, Batavia, IL 60510, USA

Lauri Autio, Rovaniemi Polytechnic, Rovaniemi, Finland*

B. Blumenfeld, P. Maksimovic, M. Mathis, Johns Hopkins University, Baltimore, MD 21218, USA

Abstract

A high performance system has been assembled using standard web components to deliver database information to a large number of broadly distributed clients. The CDF Experiment at Fermilab is establishing processing centers around the world imposing a high demand on their database repository. For delivering read-only data, such as calibrations, trigger information, and run conditions data, we have abstracted the interface that clients use to retrieve data objects. A middle tier is deployed that translates client requests into database specific queries and returns the data to the client as XML datagrams. The database connection management, request translation, and data encoding are accomplished in servlets running under Tomcat. Squid Proxy caching layers are deployed near the Tomcat servers, as well as close to the clients, to significantly reduce the load on the database and provide a scalable deployment model. Details the system's construction and use are presented, including its architecture, design, interfaces, administration, performance measurements, and deployment plan.

INTRODUCTION

The CDF experiment has a widely distributed environment for data processing and analysis. Access to their centralized database repository is critical, and a model using database replication[1], while successful, was difficult to sustain while meeting the ever-increasing load. Long distance network transactions with the database encountered very high latencies for processing farms located far from the Fermilab site. An effort was initiated to find a solution that would provide a multi-tier delivery system to distribute the load on the central system, and provide much improved performance for both local and distant clients. Experience in D0 with a multi-tier approach [2] seemed inappropriate for CDF due to its CORBA-based client interface and other implementation details specific to D0.

Requirements and Technology Choices

The requirements for the system include many aspects from design to performance and support. The system must be easily installed, maintained, and administered. It must fit easily within the existing experiment framework, and provide a library that will link seamlessly into CDF C++ client code. The system must be highly available with no single points of failure, and readily scalable to thousands of simultaneous clients while minimizing the number of open connections to the database. It should provide a caching mechanism that will enable remote clients to operate even while decoupled from the central Fermilab database. Remote caches must be easily managed and support features like cache purging or refresh. Database schema changes should not affect the client API or client access and adding new table access should not affect basic server code. In other words, old clients do not need to be rebuilt to accommodate a database or schema change. The system must be capable of operating on private networks and behind firewalls.

In addition, it is required that the system includes tools for deployment and administration, and monitoring facilities so the overall health of the system can be assessed. It is also highly desirable that the system be built with as many commodity components as possible to reduce the development time, improve reliability, promote reusability, and reduce maintenance costs. For a more complete discussion of the use cases and requirements, as well as additional details of the design refer to the Frontier Roadmap document[3].

Several existing technologies were examined to understand which might be appropriate for our needs. Tomcat[4] was chosen as the servlet container engine because it is under active development and provides many features satisfying our needs, including database connection pool management, and JDBC as the database API. HTTP was the obvious choice as the server-client transport protocol because of its ubiquity in web applications, and cURL was originally employed in our client library, although it has been replaced with our own simpler implementation of the needed functionality. Several existing approaches were explored for the framework for the client-server exchange including SOAP, Apache Axis[5], and Java JDO[6]. It was decided

* Through collaboration with Fermilab and The University of Helsinki

that a simple framework could be built to provide an efficient capability for requesting and delivering very large data objects.

Including a proxy-caching server layer in the system brings many of the systems most important features, including low latency, high scalability, ease of deployment and maintainability. Several proxy caching products were examined, but squid[7] meets the large majority of our needs. It is widely used, highly configurable, and freely available. It provides extensive access control, a variety of cache sharing protocols, and an array of monitoring options. Although such a service is generally not used for caching dynamic content pages, i.e. content coming from web service such as Tomcat, it is very effective in providing read-only access to the static database information we are serving.

Design and Implementation

The overall view of the system is shown in Figure 1. The principal components are a server hierarchy which application clients contact with requests for desired data objects. The server layer, in turn, translates the client request into a data query and returns to the client the desired information in a serialized form. The Frontier client library receives the encoded object, de-serializes its contents, and delivers it to the client.

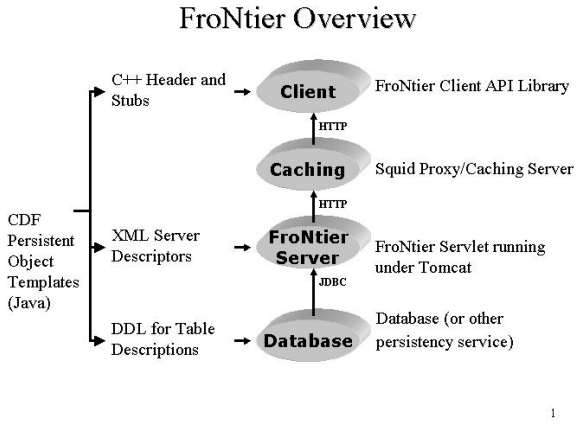

Figure 1 Overall view of the Frontier system.

CDF had an existing framework that starts with a template, written in JAVA, specifying the persistent objects stored in their database. With this template, they build their database tables, client C++ Headers, and interface to the database through OTL, MySQL, or more recently ODBC. Frontier converted CDF's existing tools for generating the client components to now generate the Frontier client interface, and what is needed in the middle tier servlet to map the client request to the database schema.

Client Request Protocol

The request, which the client sends to the server layer, uses a standard URI with name-value parameters we refer to as the *client request protocol*. The simple protocol includes a description of the needed data object and includes a type, encoding format, and key or keys. It has the form:

type (''string_name:version_number'' & encoding=BLOB|CVS|XML & key1=value1 & key2=value2 …

The *string_name:version_number* is the type name and its version number appended into one string. This forces the type and versioning information to ride together and prevents conflict with other versioning that will be present in the requests and results. The *encoding* parameter expresses the format of the returned result. There is no default, it must be supplied for each request in the URI and may be different for each. The keys are used to identify particular instance of the data objects. Each of these keys is specific to a type, such as "CID" for a calibration type and "DataRun" for a CDF query for a particular set of calibration runs.

There is an implicit, or hidden, parameter in this style of request, which is the *method name*. The request can be viewed as a method call and the method name is implicit in this request - it is always assumed to be "Retrieve Data". This query works for locating class definitions and catalog information as well as for the data itself. If a definition of a type or class is viewed as an instance of a type called "Description", then the instance could be the name of the *type*. Using the query for type information and by using the attributes argument, one can construct a generic browsing tool that allows one to transfer the information into a statistical analysis tool such as R [8] or ROOT [9].

Structure of Reply and Returned Data Format

The Frontier server reply to the client consists of metadata describing the enclosed data payload(s), and a reply can consist of a sequence of zero or more individual payloads. Different types or instances of data objects are never coalesced into a single payload bundle; they are received as distinct items. The reply is an XML datagram in which the XML serves as a descriptive wrapper around the data payload. The datagram XML's protocol identifies the data being returned, detailing the contents of each section of data being returned and the quality of the data section.

The datagram provides identifying information about the product including name, version, and XML protocol version. There is a wrapper around data being returned which describes the number of payloads being returned, their types, versions, and encoding method. The actual data payload follows, then a summary of the quality, which identifies any errors encountered in producing the data, including syntax errors, and the number of records in the payload. An MD5 checksum is included so the integrity of the data can be verified by the client.

Frontier Servlet Design

The Frontier servlet's responsibility is to translate client requests into data queries, and return the resulting information in serialized form. The overall design is

shown in Figure 2 with a sequence illustrating the flow of a request through the servlet. First, the client sends its request to the servlets' URI (1). The servlets' *Command Parser* parses the request and sends the information to a *Servicer Factory* (2), which gets an *XML Server Descriptor* (XSD)(3) from the database, and uses its content to create a *Servicer*.(4). The Servicer, in-turn, queries the database for the desired object information, and forwards it to an *Encoder*. The *Encoder* serializes the information with the wrapper, and sends a response back to the client.

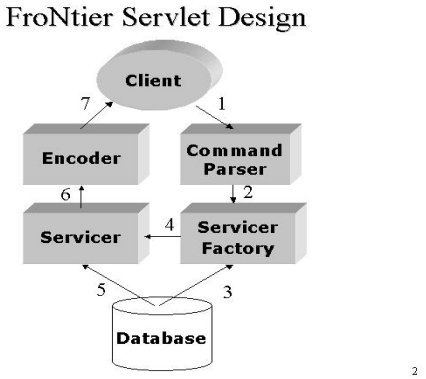

Figure 2 Frontier servlet design and operation.

The servlet is built using ANT and each module has an associated JUnit test. The servlets are deployed using the standard Tomcat administration deployment and application management tools.

An important feature provided by the XSD is data objects can be described and made available to the system without modifying the servlet code itself. The Frontier server can obtain data from virtually any data source for which there exists a JDBC driver. This also includes a wide range of ODBC sources, including flat files, which can be accessed through a JDBC-ODBC bridge. In fact, the XSD does not limit the server to read-only access - it could be easily extended to support object creation and updates.

XSD - XML Server Descriptor

The XSD itself contains a complete set of information describing 1) the object structure along with hints for marshalling, de-marshalling, and instantiation in the client address space, 2) the source of the object, for example table name, and 3) how to get the object from the source, i.e. a set of parameters or keys. The format of the current version of XSD was chosen to be optimal for use with JDBC API compatible data sources. The actual XSD's are stored in the database for consistency and version management.

The Frontier server architecture was designed to be open for adding new methods of describing and obtaining objects. Those methods could include descriptor-based methods (like XSD) or plugin-based methods if there would be requirement for very complex server-side data processing. Plugins are Java classes combined in a single or multiple Jar files. Those Jar files are stored in a database in the same way as XSD, and are dynamically loaded into JVM upon request.

The XSDs provide flexible way of writing schema and database technology-independend applications. In the case of CDF, XSDs are auto-generated based on the their primary data template description of each object. However, XSDs are flexible enough to describe complex forms of data retrieval. In the case of relational databases (specifically Oracle for CDF) it includes complex joins, sub-queries, stored PL/SQL function and procedure calls. In all cases, XSDs take full responsibility for obtaining the persistent objects for user applications.

The format of the XSD is shown below, followed by a description of each element.

```
<descriptor type="CalibRunLists"
            version="1" xsdversion="1">
   <attribute position="1"  type="int"
              field="calib_run" />
   <attribute position="2"  type="int"
              field="calib_version" />
   <attribute position="3"  type="string"
              field="data_status" />
   <select>
     calib_run, calib_version, data_status </select>
   <from> CalibRunLists </from>
   <where>
    <clause> cid = @param </clause>
    <param position="1" type="int"
           key="cid"/>
   </where>
   <final> </final>
</descriptor>
```

- descriptor - Top level tag describing the data; type - Name of the specific object type, version - Version number of the object, xmlversion - The version of XML which is being used to process the descriptor.
- attribute - Describes a datum which is being returned; position - The location of the datam in the select tag this attribute is decribing; type - How the data will be marshalled out. This is also the value returned when the client requests a description. Valid values are: int, long, double, float, string, bytes, date; field - The name of the field provided to the client when asked for a description.
- select - The fields returned from a query.
- where - A wrapper around tags which describe a specific where clause or clauses.
- clause - The SQL for the where clause to be used in the query; arameters may be passed in by using the keyword "@param".
- param - Identifies which "@param" keyword to replace with what value; position - Which keyword to replace with this parameter; type - How that keyword string is to be translated. Valid values are: int, long, double, string, date; key -

What key, supplied on the URL, which is being substituted into the parameter.

- final - Any final SQL clause which in the query.

Frontier Client Library API

Frontier provides a convenient C/C++ client API that clients can use to communicate with the Frontier service. The API provides a uniform, portable, reliable, and transparent way to obtain data from Frontier. The API supports a basic set of datatypes employed in a typical database, and also allows user applications to extend the datatype set to support application specific data structures. In addition, the API provides multiple ways to specify the Frontier servers and squid proxies to be contacted, and facilitates automatic failover if a server or proxy is unavailable. It allows requesting many objects of any type in a single query.

The API automatically parses and de-multiplexes responses into object instances, validates responses, and verifies the MD5 checksum of each object instance to eliminate possible transfer errors. The interface accommodates hardware architecture specifics, such as byte order, and operand 32/64 word bit widths. It provides typed access methods to the object data (de-marshalling), and warns, or signal errors, when a type mismatch occurs. A forced refresh of any object in squid cache can be requested and a fresh copy of the object obtained directly from the Frontier server. The API is compatible with C++ and C programs, and the C++ API can be compiled with or without C++ exceptions support.

TESTING

Extensive testing was performed to verify that the system would satisfy the desired functionality, reliability, and performance requirements. Many configurations of servers and caching proxies were assembled to test various features of the system, cache stability, and overall data throughput. Tests were done to stress the Tomcat server and squid proxy by running multiple clients and filling the cache. In one set of tests all the CDF calibration data ,representing 10.9 GB, was loaded into a squid cache with no performance degradation.

In another set of tests CDF reconstruction jobs were run on a processing farm at the San Diego Super Computing Center. In the test, 100 clients ran and requested data objects. In one case the data was accessed directly from the Oracle server at Fermilab, and in a second case the calibration data was obtained through the Frontier system with a squid cache server located at San Diego. Access durations for the 75 object types needed in the processing job were compared, and a factor of nearly 1000 in decreased access time for many objects is observed for the Frontier case relative to direct Oracle.

DEPLOYMENT

The Frontier system is being deployed for CDF at the present time. A general overview is shown in Figure 4. A high availability system of two or more server machines is being installed at Fermilab, each machine running a Tomcat-Squid pair of services. A network load balancing and failover box provides access to the servers from CDF systems throughout the world through a single domain name. We refer to the installation at Fermilab as the *launchpad*, as it represents the starting point for all objects. Squid caching servers are established at remote processing facilities and configured to allow access for clients local to them, to the Fermilab launchpad. The Squid installation procedure is straightforward and we anticipate many more in the near future, as the Fronteir client is propagated through the CDF code-base and used at CDF collaboration sites.

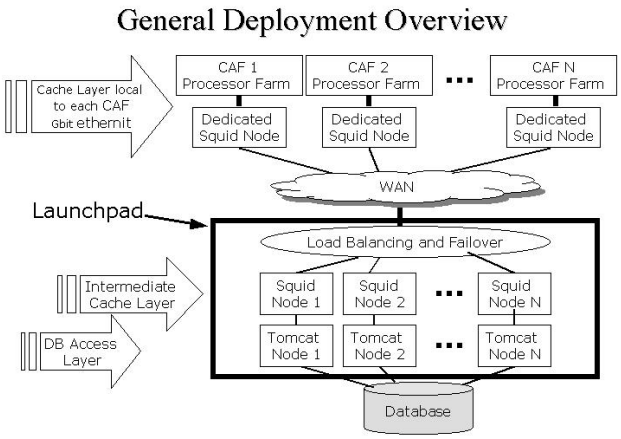

Figure 3 Overview of Frontier Deployment.

ACKNOWLEDGMENTS

We would like to thank the CDF Experiment and the Fermilab Computing Division for their support and cooperation throughout this project. Special thanks go to Frank Weurthwein, Elliot Lipeles, and the Run II hardware support team for their contributions in our testing on the CDF CAF facilities at Fermilab and UCSD.

REFERENCES

[1] D. Bonham, et al,"Database Usage and Performance for the Fermilab Run II Experiments," CHEP04, Interlaken Switzerland, Sept. 27 – Oct 1, 2004.

[2] J. Kowalkowski, et. al., "Serving Database Information Using a Flexible Server in a Three Tier Architecture," CHEP03, UCSD, La Jolla CA, March 24-28, 2003, THKT003.

[3] The Frontier Roadmap: http://whcdf03/ntier-wiki/ProjectDescription?action=AttachFile&do=get&target=TheNewFroNtier1_2.pdf

[4] The Jakarta project http://jakarta.apache.org

[5] The Axis project http://ws.apache.org/axis

[6] JDO http://java.sun.com/products/jdo/ .

[7] Squid home page http://www.squid-cache.org/ .

[8] The "R" project page http://www.r-project.org .

The ROOT home page http://root.cern.ch .