Scaling HEP to Web Size with RESTful Protocols: The Frontier Example

Dave Dykstra
Computing Division, Fermilab, Batavia, IL, USA

E-mail: dwd@fnal.gov

Abstract. The World-Wide-Web has scaled to an enormous size. The largest single contributor to its scalability is the HTTP protocol, particularly when used in conformity to REST (REpresentational State Transfer) principles. High Energy Physics (HEP) computing also has to scale to an enormous size, so it makes sense to base much of it on RESTful protocols. Frontier, which reads databases with an HTTP-based RESTful protocol, has successfully scaled to deliver production detector conditions data from both the CMS and ATLAS LHC detectors to hundreds of thousands of computer cores worldwide. Frontier is also able to re-use a large amount of standard software that runs the Web: on the clients, caches, and servers. I discuss the specific ways in which HTTP and REST enable high scalability for Frontier. I also briefly discuss another protocol used in HEP computing that is HTTP-based and RESTful, and another protocol that could benefit from it. My goal is to encourage HEP protocol designers to consider HTTP and REST whenever the same information is needed in many places.

1. Introduction
The software architecture style known as Representational State Transfer (REST) was originally defined by Roy Fielding in his PhD dissertation [1]. It is a general architectural style that was derived from using a subset of the Hyper Text Transfer Protocol (HTTP) strictly according to the HTTP standard as defined in the RFCs from the Internet Engineering Task Force. Roy was a principal author of both the HTTP 1.0 [2] and 1.1 [3] RFCs. One of the major goals of REST is massive scaling, so it was designed to do that well. Scaling is achieved primarily by making caching very efficient and effective so caching can be easily deployed at many levels. REST also makes replication of servers very easy.

The most common use of REST and HTTP is for transferring pages to web browsers, but they have been designed to be broadly applicable to any network-based service that needs to massively scale, especially those that need to send the same information to many readers. Modern High Energy Physics (HEP) computing has many applications where this is the case because the applications often use computing resources scattered across tens or hundreds of thousands of computer cores distributed around the world.

2. How REST scales
There isn't space in this paper for a full introduction to REST principles, but these are the aspects that I have found to be essential or important to making HTTP-based RESTful protocols scale well.

2.1. Stateless
The fact that RESTful protocols are stateless is the most essential component to make REST scale. It is essential for caching. Every client request has to be independent, and every request for the same

information is exactly the same. This means among other things that techniques such as the use of cookies stored in web browsers must not be used in REST because cookies are different for each client. Encrypted protocols such as HTTPS have state and are uncacheable so they are also not RESTful. Digital signatures may be used instead, and can reliably authenticate servers to clients (and possibly even clients to servers if the clients use a shared key). If encryption is absolutely required, it can still be helpful to tunnel a RESTful protocol over HTTPS to a large cluster of replicated servers that have frontend caching servers which remove the decryption and then forward the unencrypted RESTful protocol to backend servers over a secure private network. The RESTful protocol will then enable the backend cluster to scale by adding more servers.

2.2. HTTP methods

Another essential component of making an HTTP-based RESTful protocol scale well is strictly following the HTTP methods as intended by the HTTP RFCs. The four HTTP methods are GET, PUT, POST, and DELETE. POST is the only method of the four that takes input from the body of the message sent from the client to the server, so many protocols (for example all SOAP-based protocols) use it incorrectly [4] to pass in complex parameters for an operation that is read-only. Instead, the GET method should be used with as long and complex of a URL as it takes to pass in all the information needed to define every unique “resource” (defined below). That makes the responses cacheable. The GET URL should not contain a question mark ('?') because the HTTP standard says URLs with question marks do not need to be cached.

2.3. Caching proxy servers

The primary means by which REST scales is with caching proxy servers. The protocol is designed to be able to nest caching proxies as deeply as needed in order to transparently achieve a desired throughput. As long as the application complies with the HTTP standard, any of a number of different standard-compliant caching proxies may be deployed interchangeably.

2.4. Cache consistency

Whenever an application accesses resources whose values can change, cache consistency becomes an issue. This is very simply managed in HTTP by servers including a response header (either **Expires** or **Cache-control: max-age)** that defines how long a response may be cached before it is out of date. The length of the expiration varies greatly by application; for example, Frontier in CMS (more details about it are below) has cache expiration times varying from 30 seconds to a year, and most are 6 hours.

2.5. Cache revalidation

HTTP-based REST includes very powerful features to cheaply revalidate expired cached items. If a server includes a **Last-Modified** or **ETag** header in the original response, HTTP-compliant caching proxies automatically send an **If-Modified-Since** or **If-None-Match** header, respectively, with the next request to the server for the same item. (The header sent from the server to the cache includes a parameter that the cache sends back to the server with the **If-*** header). If the server determines nothing has changed, it sends a very small **NOT MODIFIED** response and the cache is revalidated for another expiration period. If something has changed, the server sends the new response with no additional protocol overhead. These features are especially valuable for applications that need short expiration times because some small parts of the data changes often, but yet has a majority of the data organized into relatively large parts that don't change often. For more details including diagrams of how this works see my 2009 CHEP paper [5].

3. Frontier as an example RESTful protocol

Frontier [6] is a RESTful protocol. It is middleware that distributes read-only database SQL queries over HTTP. It is used by the CDF, CMS, and ATLAS High Energy Physics experiments primarily to distribute particle detector “Conditions” data that describe the calibration of the detectors as they change over time. There are many different readers of the data around the world, and many readers at the same geographical location read the same data over a relatively short period of time. Updates to the database are done with a different protocol. This application greatly benefits from caching.

3.1. REST Data Elements

Roy Fielding's dissertation defines “Data Elements” to describe the different parts of the REST protocol messages. The six different types of Data Elements are these:

1) resource – any information that can be given a name.
2) resource identifier – the name for a resource. In HTTP this is the URL (Uniform Resource Locator)
3) representation – the current value of the resource. This is the "RE" in REST.
4) representation metadata – information describing the representation. In HTTP these are in the response headers.
5) resource metadata – information about the resource. In HTTP these are in the body of the response.
6) control data - information that defines the purpose of the message or changes behavior.

Table 1 shows examples of each in Frontier.

Data Element	**Frontier example**
resource	Data returned from a database query
resource identifier	SQL query encoded in URL (by gzip and base64)
representation	XML document including data (encoded by gzip and base64)
representation metadata	**Last-Modified** time header
resource metadata	Database error messages
control data	**If-Modified-Since**, **Cache-Control**

Table 1: Frontier examples of REST Data Elements

3.2. REST Components

Roy Fielding's dissertation also defines "Components" to describe the different parts of a deployed REST-based installation. They are:

1) origin server – the interpreter of the resource identifiers in the user requests and the original source of the responses (which include the representation and associated metadata/control data)
2) gateway – an intermediate server inserted by the network or origin server, typically for caching or security enforcement.
3) proxy – an intermediate server selected by the user agent.
4) user agent – the initiator of resource requests and the final receiver of the response

Table 2 shows the REST components in Frontier.

Component	**Frontier example**
origin server	Tomcat with Frontier servlet
gateway	Squid in "accelerator" or "reverse proxy" mode
proxy	Squid in its default mode
user agent	frontier_client

Table 2: REST Components in the Frontier example

These components can be combined in a variety of ways depending on the application needs. The CMS experiment deploys it in two different ways for its Online and Offline Conditions applications. The CMS Offline case in Figure 1 uses a set of 3 origin servers at one location, co-located with database servers, with a gateway Squid on each of the same machines as the origin servers. The URL that the clients use specifies a round-robin DNS of the gateways, and each gateway automatically forwards requests to the origin server running in the same machine. Every site, both local (Tier 0) and remote (Tiers 1, 2 or 3), also has one or more Squids as proxies. Each user job on every worker node in the worker node farms runs a copy of the frontier_client software to initiate requests.

This deployment architecture is highly effective at delivering the data to many user jobs without putting a very large load on the servers for this application. User jobs start out at different times throughout the world. Recent usage statistics show an average of about 250 job starts per minute making 500,000 requests/minute and loading a total of 500MB/second, but on the 3 central gateway Squids there are only 6,500 requests/minute and 0.7MB/second. That is a factor of 77 improvement on requests and 715 on bandwidth. The load on the origin servers is another factor of 2 better in requests per second and a factor of 3 better in bandwidth.

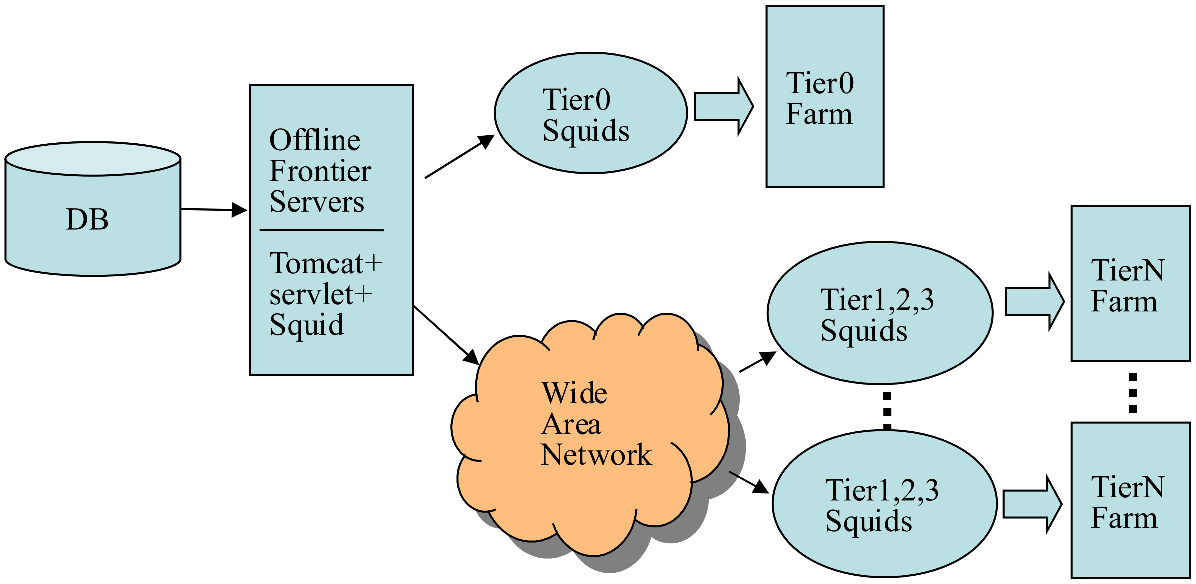

Figure 1: CMS Offline Conditions deployment architecture

The CMS Online case in Figure 2 has different requirements. It has 1400 worker nodes that it loads simultaneously with the same data. They all have to be loaded very quickly because during that time the expensive detector cannot record particle collisions. In addition to the gateway Squids co-hosted with the two origin servers (two machines for reliability, not load), there is a proxy Squid on every worker node. Each worker node feeds four others in a nested fashion so all the nodes are loaded in parallel without heavily loading any single network link (there are backup nodes configured in case some of them fail). The roughly 100MB of data is loaded to all nodes in under 30 seconds. The cache is set to expire in 30 seconds, so every time the data is loaded it is guaranteed to be up to date. Most of the time a majority of the data doesn't change, so very little needs to be transferred between squids other than cache re-validations, but even if all the data is reloaded it is very fast because of the parallel links.

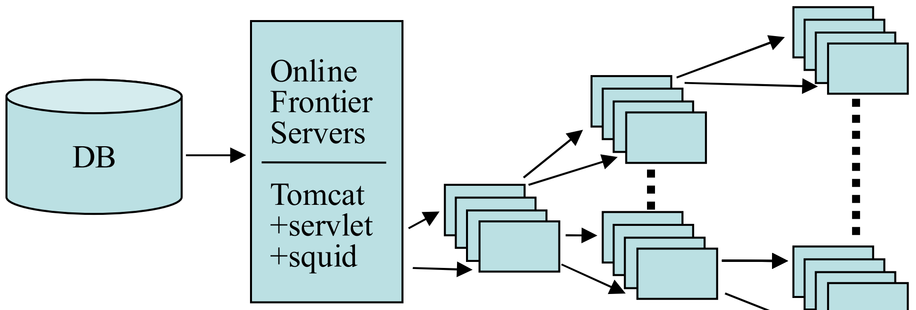

Figure 2: CMS Online Conditions deployment architecture

4. CernVM Filesystem example

The CernVM Filesystem [7] is another example of a RESTful protocol. It is optimized to distribute slowly changing filesystems containing software that is mostly added and not modified. Its URLs are based on secure hashes of the contents of files, and they have catalogs that map filenames to the hashes. An implication of this is that once the files have been loaded into the cache, they never change. The catalogs can change and so need to have cache expiration times, but they change slowly so the expiration times can be relatively long. Another advantage of using secure hashes as resource identifiers is that it is trivial for the client to detect tampering by just recomputing the secure hash of the data received. The catalogs are digitally signed by the origin server and verified by the client to prevent man-in-the-middle attacks.

5. REST for authorization protocols

Authorization protocols used in scientific grid computing [8] are not RESTful. They are based on SOAP and are not cacheable for 3 reasons:

1) The protocols require HTTPS to be secure.
2) SOAP uses POST for parameters rather than encoding the parameters in the URLs.
3) They include the client node name in parameters to every request, so they can use different sets of authorizations for different groups of client nodes. This results in every request being unique.

However, when many related jobs start at a grid site close together, there are many duplications of the same authorization request that happen close together, so this is an application that could benefit if a RESTful protocol and proxy caches were used. This could be done if the following changes were made:

1) Use digital signatures instead of encryption to authenticate the source, and put a timestamp in the responses to prevent replay attacks.
2) Encode all the parameters in the URLs.
3) Put a group name in the URL instead of an individual client name.

There is no inherent reason why authorization protocols cannot be RESTful.

6. Conclusions

Applications that need to transfer the same information to very many computers in a short period of time, as is the case for many High Energy Physics applications, should strongly consider designing their protocols to follow REST principles and use HTTP. This provides an easy way to achieve high scalability through the deployment of popular open source caching programs such as Squid. Much other heavily tested open source software is also readily available for building the clients and servers.

7. Acknowledgements

Fermilab is operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.

References

[1] Fielding R 2000. Architectural Styles and the Design of Network-based Software Architectures *University of California, Irvine* PhD Dissertation

[2] Berners-Lee T, Fielding R and Frystyk H 1996. Hypertext Transfer Protocol – HTTP/1.0 *RFC 1945*

[3] Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, and Berners-Lee T 1999. Hypertext Transfer Protocol – HTTP/1.1 *RFC 2616*

[4] Prescod P 2002. Roots of the REST/SOAP Debate *Extreme Markup Languages Conference*

[5] Dykstra D and Lueking L 2010. Greatly improved cache update times for conditions data with Frontier/Squid *J. Phys.: Conf. Ser.* **219** 072034

[6] See http://frontier.cern.ch

[7] Blomer J and Fuhrmann T 2010. A Fully Decentralized File System Cache for the CernVM-FS *International Conference on Computer Communications and Networks (ICCCN)* DOI 10.1109/ICCCN.2010.5560054

[8] Garzoglio G, Alderman I, Altunay M, Ananthakrishnan R, Bester J, Chadwick K, Ciaschini V, Demchenko Y, Ferraro A, Forti A, et al. 2009. Definition and Implementation of a SAML-XACML Profile for Authorization Interoperability across Grid Middleware in OSG and EGEE *Journal of Grid Computing* DOI: 10.1007/s10723-009-9117-4