Web Based Database Applications Architecture

Igor Mandrichenko
David Dykstra
6/24/2014

Introduction

The fundamental idea behind this proposed architecture is to use Representational State Transfer (REST) [1] and specifically its HTTP implementation as a unified mechanism of network-based data management. Unification of the application-independent data transport mechanism allows independent implementation of the communication peers, which in turn simplifies and makes more efficient design and development of the software and operational support of software systems.

HTTP/REST has become a de-facto standard for web-based application development and is very attractive to us in HEP for the following reasons:

- The internet fuels world wide development of HTTP-based standards, technologies, applications and frameworks, many of which are suitable for data management and transfer. Internet tools and web application development frameworks such as Apache httpd, squid, Varnish, WSGI, Tomcat, etc. provide readily available powerful building blocks for application development;
- HTTP is a very simple yet powerful protocol. It is flexible enough to be used in a wide variety of applications. As an abstract transaction representation layer, REST provides a perfect set of operations to represent the variety of data management operations used in data intensive applications;
- Web development frameworks and tools such as Apache httpd provide necessary throttling and resource management functionality better than direct database or data storage interfaces in large part because the Internet industry faces scalability challenges all the time and has made great progress in developing scalable and manageable tools and frameworks;
- Using HTTP as a well known base protocol standard for data exchange allows decoupling of the implementation details of the

server and the client, which decreases overall software development and support cost by eliminating unnecessary interdependencies between system components. Also, as long all the components of the system communicate using a common protocol, it is possible to use a modular approach to the system design and treat many components as optional and plug them in when necessary.

- Introduction of a web services layer between the client and the data storage often reduces the amount of client-to-storage communication by confining complexity of the data organization to the web service – storage segment.
- The web service layer allows managing the communication as a resource by introducing a single point where the data requests come through that manages the resources in a storage independent way.

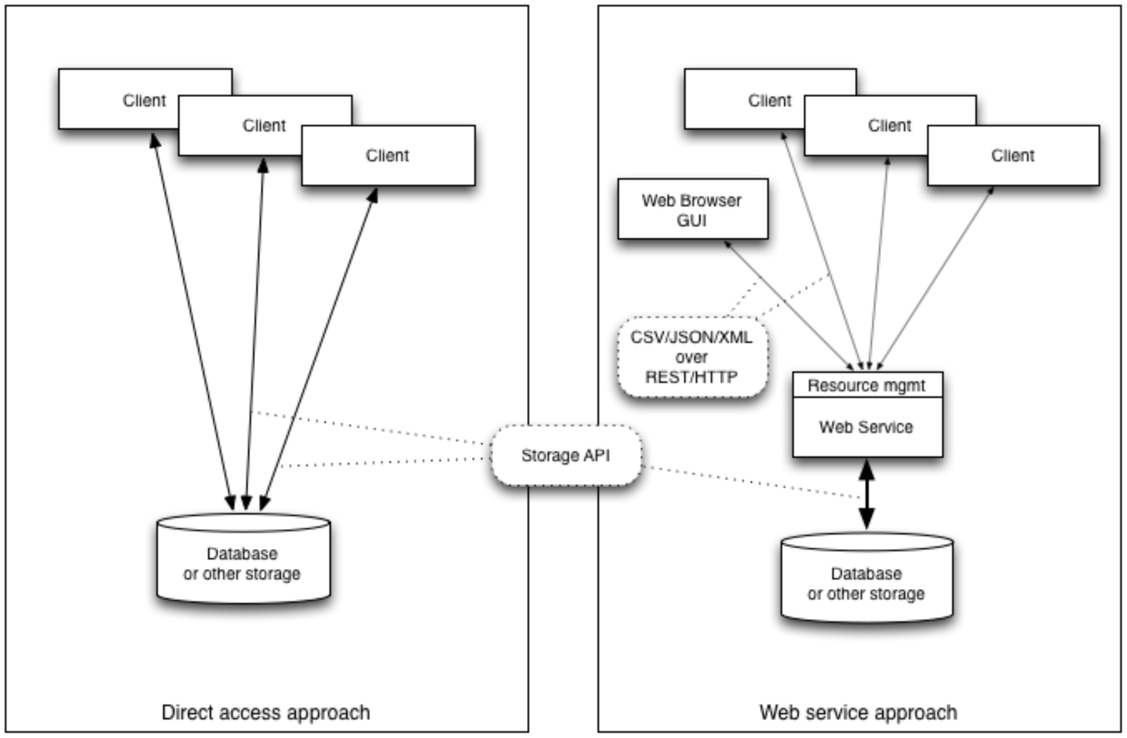

Figure 1. Direct Access vs. Web Services Approach

At FNAL, we have a long successful history of using web services to build data management applications.

Web Services Architecture

General architecture for a web-based data application consists of several components. The picture below shows the current architecture. Some of these components are optional, and not necessarily included in every application. Others are common.

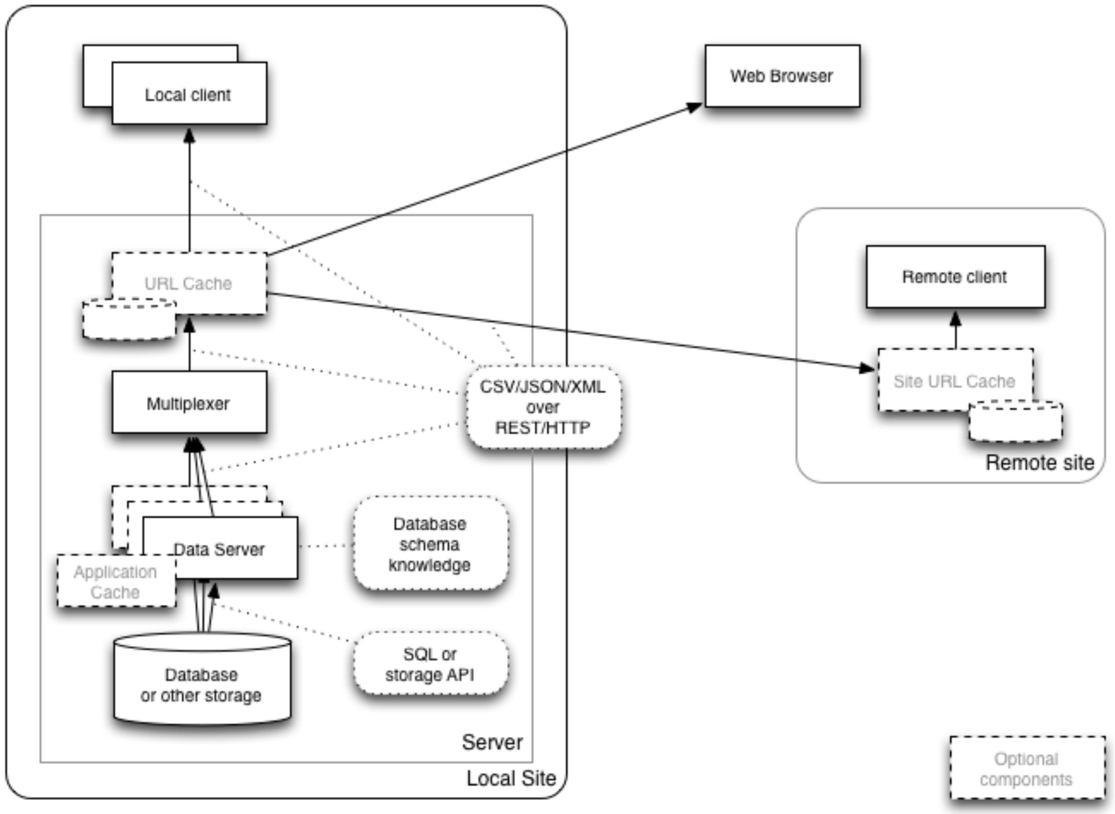

Figure 2. General application architecture

Currently our architecture uses the following platforms and standards:

- Python as the server side programming language
- We support Python, C and C++ client side libraries
- Apache httpd as the HTTP server
- WSGI and mod_wsgi as the Python/Apache interface and abstract API
- HTTP and HTTPS as the data transport protocol
- CSV, JSON and XML as data representation standards
- Squid as general purpose HTTP caching proxy
- Jinja2 as the templates package
- Jango as the framework for interactive applications

- Google Charts as the client side plotting package

We use the same architecture for interactive and data applications. This allows our interactive components to interact with data portions in the same way data clients do.

Data Storage

Typically, the data is stored in a database, but the architecture is well suited to work with other types of storage, and in fact some of our applications store some data in memory or files on disk and make the data available via the architecture.

Data Server

Data Server is an application specific component, which essentially translates HTTP/REST data representation into the Data Storage representation and back. Data Server communicates with its client in terms of HTTP/REST and with the underlying storage using the data storage API. Data Server sends data to the client encoded in one of several Internet standard data representation formats such as CSV, JSON or XML.

Depending on the application, Data Server can provide not only read functionality (HTTP GET) but also write functionality (HTTP POST).

Following the REST model, Data Server itself is a state-less component. Data requests do not change the context of the server. The only exception is the case when the Data Server has its own data cache.

Because all the data communication in front of the Data Server is done in the HTTP protocol, the Data Server does not make assumptions about which component of the architecture its client is: web browser, client application or web cache, etc.

For interactive applications, the corresponding component is called Application Server. The fundamental difference between Data Server and Application Server is that generally Application Server can not be considered stateless and usually the client and the server maintain some session context information.

Server Cache

Depending on the application, sometimes it is possible and beneficial for the server to keep some sort of cache of data retrieved from the data storage so that subsequent requests do not always cause new interaction with the database.

Request Multiplexer

Typically, we use multiple redundant Data Server instances, running independently on multiple computers. Running multiple servers serves 2 purposes:

- It increases service availability as failure of one or more individual servers does not necessarily lead to the failure of the whole system.
- It provides an easy mechanism for adding (or removing) resources to the system to meet performance requirements.

The stateless nature of the server makes it possible to use multiple redundant copies of it because the client does not ever need to be able to interact with a specific instance of the server. Request Multiplexer is the component that chooses an available instance by checking its availability and then forwards the request to the selected data server instance.

Request Multiplexer smooths out load peaks by queuing excessive requests and executing them at a later time.

Request Multiplexer monitors all active connections and is used as a resource monitoring and management tool.

A single instance of Request Multiplexer works with multiple "services". Typically, a service is a single application. Also a service can represent resources dedicated to specific activity within the same application, or a group of users of the same application. The Request Multiplexer allocates resources to multiple competing services and monitors and manages their utilization. The service configuration includes the limit on the number of simultaneously active requests, the size of the queue, timeouts, etc.

The multiplexer also has data caching capabilities.

For interactive applications, we do not use the Request Multiplexer. Instead, we use the HTTP Request Redirector. One reason for this is that most interactive applications maintain some session context information between the client and the server, which makes the server not stateless, and therefore the client needs to communicate with the same real application server through the lifetime of the session.

URL Cache

URL Cache is a general purpose HTTP caching proxy. It can be deployed either on the server side (reverse proxy) or on the client side, or both. In cases when the client runs at a remote site, client side cache can significantly reduce WAN traffic.

The reason why this component is called URL cache is because it uses the URL as the cache key to identify the object in the cache. Currently we use Squid [2] as the URL Cache. URL Cache sits between the client application and the multiplexer, essentially shielding the application from repeating data requests and effectively increasing the application performance.

This component is optional. Not every application provides an opportunity to use URL cache, and not every application would benefit from it.

Squid is one of several HTTP caching proxy products available. There are other popular products such as Nginx [3] and Varnish [4], although these seem to be optimized to be reverse proxies, while squid is the most popular forward proxy.

URL cache is not used for interactive applications.

Low level client interface library

We provide a low level C library, called libwda, which provides the functionality of downloading a document by URL, decoding CSV-represented data and presenting it as a list of tuples. This low level library is built on top of the popular public domain libcurl package which actually implements the HTTP communication. libwda has a controllable delayed retrial functionality.

Data Caching

Data caching is widely used to significantly increase performance of web servers specifically and data applications in general.

The idea of caching is based on the assumption that if the client or multiple clients repeatedly ask for the same piece of information, the information can be retrieved only once, saved somewhere and then returned to the client the next time it asks for it.

There are several conditions that are required for any caching mechanism to be beneficial:

- The client asks for the same information often enough. Often enough means:
    - Requested item is still in the cache and was not purged to make room for some other information.
    - The cached information is still valid.
- It is significantly cheaper to save and return the data than to re-retrieve or re-compute it.

Not every data request is cacheable. For example, requests like “what is the current state of the detector” are not cacheable at all because presumably the state of the detector changes all the time, whereas requests like “what was the state of the detector for run N?” usually can be safely cached.

In our architecture, there are 3 locations where caching can be performed:

- Client side caching. The client application can
    - save data “locally” and use it later instead of issuing a new request for the same data
    - pre-fetch more data than it needs immediately and use the received extra data later

  Client side caching is used in several applications we developed: Minerva Conditions, IFBeam DB, NOvA Conditions.

- URL caching. In some cases, when data deterministically depends on the requested URL and the time dependency is slow enough, application-independent, URL based cache (e.g. squid) can be used.

URL caching works well for several applications we developed, including NOvA Conditions and certain components of IFBeam DB.

URL caching is the most common type of caching and is the most attractive type because it is the easiest to implement.

- Server side caching. Sometimes different client requests translate into requests for the same data item or dataset from the data storage. In these cases, while the URL caching may not work due to lack of request correlation at the URL level, it is possible for the server to save some datasets so that next time it needs the same dataset it can get it from its own local cache instead of the data storage.

  Minerva Conditions is the example of an application where URL caching is not working due to lack of correlation at the URL level, (~5%), but server side dataset caching works very well (typical cache hit ratio ~70-90%).

While caching is generally considered to be a beneficial practice, the decision whether to use it or not needs to be based on the specifics of the application. If the cache hit ratio is low, using a cache just adds another layer to the application, consumes resources and in fact decreases overall reliability and performance of the system. And when data changes over time, caching needs to be done carefully because there is a chance that the client will get stale data. In those cases, cache coherency needs to be managed.

Frontier

The idea of using the Internet document caching technology to optimize data communication was implemented at FNAL for CDF around 2004 [5]. The framework developed at that time was named Frontier. Its main goal was to help develop data access applications, which can take advantage of the fact that multiple clients issue repeating requests for the same information. Frontier proposed to convert application data requests from SQL into a URL and use the URL as a key for the data cache. Frontier used squid (general purpose HTTP caching proxy software) as the caching component. CDF still uses Frontier as proposed back in 2004, in a somewhat simplified implementation.

Since then, Frontier development continued and it has been modified substantially [6]. Currently it is in active use by CMS and ATLAS [7].

Frontier remains a web application development framework, designed for data communication.

While there are a lot of similarities with the existing architecture, Frontier has some fundamental differences in its approach to scalability and performance. The current architecture can be called server-based in the sense that most of the infrastructure is located at the server side, while the client is very light weight. In comparison, Frontier can be called client-based. A significant portion of its infrastructure and functionality is located on the client side or at the site where the client runs.

Frontier puts a big emphasis on the URL caching as the main scalability mechanism. It has a sophisticated infrastructure for using layers of forward and reverse caching proxies.

Frontier takes different approaches than the existing architecture in some areas, including:

- Additional protocol layer - Frontier implements its own protocol layer on top of standard HTTP. The additional layer is used to implement such features as data compression and error communication in addition to what is included in the HTTP standard.
- Client Side Multiplexing - Frontier client can be configured to select one of many available servers.

Also, as an option, Frontier provides the ability to send SQL over HTTP, which essentially allows the database-aware client to access the SQL database over HTTP and provide schema elasticity without any modifications to the Frontier server.

Frontier as a Web Applications Framework

As a web applications development framework, Frontier does not seem to be as unique and attractive as it was at the time of its introduction because the Internet industry has provided many powerful and easy to use frameworks and made it extremely easy to publish a piece of code on the web. Here is an example of what it takes to create a “hello world” web server in Python using standard library WSGI module:

```
from wsgiref.simple_server import make_server

def simple_app(environ, start_response):

    status = '200 OK'
    headers = [('Content-type', 'text/plain')]

    start_response(status, headers)

    ret = [“Hello world”]
    return ret

httpd = make_server('', 8000, simple_app)
httpd.serve_forever()
```

In this code snippet, out of 9 lines of code, 6 are application “business” code (the “simple_app” function) and remaining 3 lines are “infrastructure” code, that actually turns the business function into a web application. With mod_wsgi, used to plug a WSGI application into Apache httpd, these 3 lines would not be even needed, so there would be actually 0 lines of “infrastructure” code:

```

def application(environ, start_response):

    status = '200 OK'
    headers = [('Content-type', 'text/plain')]

    start_response(status, headers)

    ret = [“Hello world”]
    return ret
```

In other words, modern technologies make it so easy to build a web application that it becomes irrelevant which particular framework is used, and the choice of the framework becomes a matter of preferences of the individual developer or the group of developers.

Of course in reality things are a little more complicated than in the demo example above, and the “infrastructure” is not exactly zero, but still the amount of effort spent on it is negligible, because it is being reused from one application to the next.

Here is a breakdown for 3 typical database applications showing the number of lines of Python code per component:

	Minerva Conditions	NOvA Conditions	IFBeam DB
Application specific data management code	1200	1450	N/A
Data Server	230	650	7500
"Infrastructure" layer	400		

Our "infrastructure" web application layer is mostly a convenience package. It is used to map the URL to an object on the tree of Python objects and call one of its methods with arguments extracted from the URL. Also, it provides server-side sessions functionality used mostly by interactive applications but not data applications. It was initially developed in 2005-2006 as a replacement for a CGI layer, used by the D0 Trigger DB. It is used by all applications developed by our group since then with minor modifications.

Frontier implements several features, which can be useful in some cases and worth mentioning here:

SQL Communication

Frontier can be used where it is necessary to expose a relational database schema to the client application and it is not practical to let the client access the database directly. This is usually the case when the database schema is very simple and static and when scalability is needed, e.g. with grid production.

Data Compression

Frontier implements data compression. Based on our experience, in most cases, it takes much more time and CPU resources to retrieve large amounts of data and to convert them into the format suitable for data transfer (CSV, JSON, etc.) than to actually transfer the data over the socket. Data compression and decompression reduces data transfer time at the expense of even further increasing data processing time. So the benefits of data compression are debatable.

However, there are cases when data compression can be beneficial. In particular, when the data is stored in many grid site caches, data compression can reduce their disk and bandwidth requirements and may be worth the extra CPU resources.

If necessary, data compression can be easily implemented without Frontier.

Client Side Multiplexing

Frontier implements client side multiplexing, which is the ability for the client to choose one of many available servers. The same functionality is provided by the Request Multiplexer described above, but it does it on the server side.

Compared to the client side, server side multiplexing is more practical and convenient than client side. With server side multiplexing, the client knows of only one server address instead of a list of them. Server side multiplexing is much easier to manage by the service provider, especially in the Grid environment because the multiplexer configuration is located in single location controlled by the service provider and can be changed instantly without clients even knowing. Also, server side multiplexing can be combined with resource management and collapsed request forwarding, which is not possible with client side multiplexing. Frontier server itself has some resource management capabilities.

However, there are cases where client side multiplexing could be useful, for example when the client can choose to talk to multiple sites, or to talk to multiple local caches, or to use remote backup cache when local site cache is unavailable.

Note that client side and server side multiplexing do not contradict each other.

Client side multiplexing could also be implemented outside of Frontier.

Site URL Cache Discovery

Locating the URL Caches at many sites is not simple. CMS and ATLAS each maintain their own complete list of them at all the sites they use, but not in a way that is easily shared with other projects. This is being changed to make it easier, based on an internet standard called Web Proxy Auto Discovery that the Frontier client supports. Implementation of the WLCG proxy auto-discovery service is under way [8].

Error Caching

Generally, URL caches do not cache error responses. Under high loads, it may be beneficial to prevent repeating requests which cause errors from all reaching the database. The Frontier server can instruct URL caches to cache error responses for a short period of time, thus reducing potentially harmful request traffic.

On the other hand, it is also important to not pollute the cache when a response body is incomplete due to an error during transmission, but the response has valid HTTP headers. The Frontier server handles this by signaling an error at the end of the response, after the HTTP headers specifying caching time have already been sent, and the Frontier client then retries and requests a short cache time.

Error caching could be managed by the Data Server using cache control HTTP headers without using Frontier. Caching of incomplete responses could be avoided without Frontier if the Data Server pre-generated the whole response so it can include an HTTP content length header.

Monitoring

Frontier has its own proxy monitoring tools, which can be used to monitor request traffic, and to detect cases when the requests bypass local site caches and go directly to the server or central backup cache. The tools can automatically notify site administrators that they have a problem so database operations personnel don't have to spend much time on it.

When to Use Frontier

Frontier should be used when:

- the existing architecture does not scale and
- it is possible to use URL caching and
- the caching infrastructure requires the client side multiplexing

Another use case for Frontier is when it is necessary for the client application to access a remote relational database on SQL level.

Adding Frontier to the Architecture

The current Frontier system that CMS and ATLAS use to access conditions databases at the SQL level also has the ability to read from any HTTP server as a backend instead of directly to a database. For cases where Frontier should be used as described above, the proposal is to add Frontier infrastructure between the existing infrastructure and the client as shown in the in Figure 3.

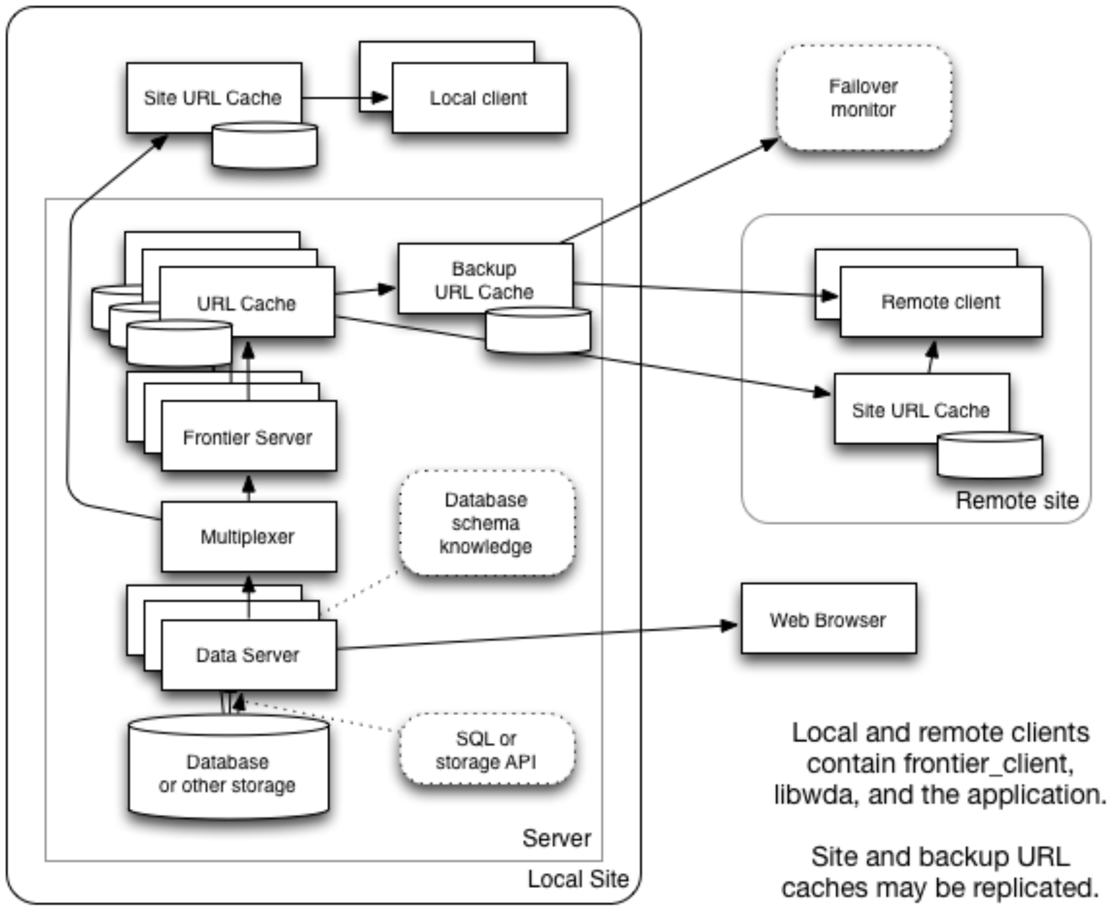

Figure 3. Architecture including Frontier

The entire server side could be replicated at another distant site for more reliability and for better access near the distant site. A monitored backup URL cache (actually a pair of them) is introduced in order to allow clients to keep operating when their own site caches are failing.

On the client side, libwda would be extended so that if an application requested a server URL beginning with "frontier://" instead of http://, it would invoke the Frontier client instead of directly connecting to the

Data Server or multiplexer of the current architecture. In this way the application would only need to change the URL string to use either type of server. All the parameters can be passed to the Frontier client through the URL string or through environment variables [9].

The Data Server is the same as in the current architecture, so all of the existing benefits of its implementation would be preserved. This includes enabling web browsers to continue to access the Data Server directly.

Recommendations

1. Use HTTP/REST as the protocol of data communication and common standards like CSV, JSON, XML for data representation.
2. Build application specific data server using one of widely available web applications development frameworks.
3. Whenever possible and beneficial, use data caching as combination of client side, URL and server side caching, in that order of preference, depending on the specifics of the application.
4. Use redundant web services infrastructure to increase reliability and performance of the application.
5. Unless required, hide data storage implementation details behind the Data Server.
6. If it is necessary to expose the database schema to the client application, CMS style Frontier should be used.
7. Use Frontier infrastructure on top of the current architecture when it is necessary to take advantage of more sophisticated client side cache services infrastructure.

References

[1] http://www.squid-cache.org/
[2] http://nginx.com/
[3] https://www.varnish-cache.org/
[4] http://www.pha.jhu.edu/~mmathis/seminar/
[5] https://indico.cern.ch/event/0/session/7/material/paper/1?contribId=204
[6] https://twiki.cern.ch/twiki/bin/view/Frontier/FrontierOverview
[7] http://frontier.cern.ch
[8] https://twiki.cern.ch/twiki/bin/view/LCG/HttpProxyDiscoveryTaskForce
[9] http://frontier.cern.ch/dist/FrontierClientUsage.html