ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THMPA06

BUILDING CONTROLS APPLICATIONS USING HTTP SERVICES*

T. D'Ottavio†, K. Brown, A. Fernando, S. Nemesure
Brookhaven National Laboratory, Upton, NY, USA

Abstract

This paper describes the development and use of an HTTP services architecture for building controls applications within the BNL Collider-Accelerator department. Instead of binding application services (access to live, database, and archived data, etc) into monolithic applications using libraries written in C++ or Java, this new method moves those services onto networked processes that communicate with the core applications using the HTTP protocol and a RESTful interface. This allows applications to be built for a variety of different environments, including web browsers and mobile devices, without the need to rewrite existing library code that has been built and tested over many years. Making these HTTP services available via a reverse proxy server (NGINX) adds additional flexibility and security. This paper presents implementation details, pros and cons to this approach, and expected future directions.

INTRODUCTION

Application development has changed dramatically over the last 20 years. These changes encompass the computer languages that we use, the infrastructure that binds software modules together, and the development tools used to build the software. This paper focuses primarily on changes to the software infrastructure.

Figure 1: Traditional Monolithic Application.

Twenty years ago, most applications were built using a single computer language, with software tools bound into the application running as a single executable program. This is known as a monolithic application [1]. As seen in Fig. 1 above, a monolithic Controls application might be written in C++ or Java and consist of a custom user interface utilizing standard toolkits to access control system devices, interact with database systems, and/or extract data stored by archiving or logging systems.

Of course, this is still a viable way to put applications together and will remain so for many years to come. The process is well established and highly optimized. And the applications produced have very good performance and are relatively straightforward to test and troubleshoot. However, this type of application development does have some limitations. The next section describes the issues that drove our group to look for an alternative.

* Work supported by Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy.
† Email address dottavio@bnl.gov

Limitations of Monolithic Applications

Language and Code Reuse Twenty years ago, all of our applications were built as monolithic applications using C and C++. A few years later, we started investigating what it would take to use Java for application development. Java had in many ways surpassed C++ in terms of its basic tools and development environment and had become the language of choice for many software developers.

The problem was that we had invested many years of effort into building a set of modular C++ tools for streamlining application development. And we wanted to reuse these tools if possible. We explored, and then rejected as too complicated, the use of Java Native Interface (JNI) [2], which allows Java programs to call C/C++ code. We followed a similar path when exploring the Common Object Request Broker Architecture (CORBA) model [3]. Instead, we invested our time into rewriting many of our C++ tools in Java.

However, it was clear, even before we were finished with that effort, that supporting other types of applications (LabView, MatLab, python, web browser, synoptic displays) would be necessary as well. We needed a more flexible way to reuse the software tools that we had built using C++ and Java.

Remote and Mobile Clients A second arc in our application development needs revolved around using our applications outside of the BNL campus, especially from employee homes. This was driven mostly by the expansion of broadband to homes and the associated increases in broadband speeds.

Our first solution to this problem was to have users log-in and run applications on BNL computers, but display them on their local home computers. This is possible because most of our applications are written for a Linux/X Windows environment, which has remote display built into the X Windows protocol [4]. This solution works, but requires users to install special software on their home computers. In addition, remote displays can be slow, even with relatively fast home connections. Recently, we've improved display performance by using a commercial product specifically designed to speed up X Windows network communication [5].

Though the above solution works, and is still in use today, it ultimately limits users to running Linux/X Windows applications on remote computers that are specially configured for that purpose. It would be preferable if the computer could run applications in its native environment. This would allow a user to run an application from a computer not specially configured (for example, in a hotel) or run an application from a universally available environment like a web browser.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THMPA06

A final push came from the popularity of mobile devices like smart phones and tablets. How could our Controls applications be run on these devices? Certainly a mouse-driven, remote X Windows client solution wouldn't work for these touch-based devices.

USING HTTP SERVICES

Based on the above history, we began to explore the idea of making some of our core Controls system tools available as networked services. This involved wrapping existing toolkits with a network communication layer and a network API as shown below in Fig. 2.

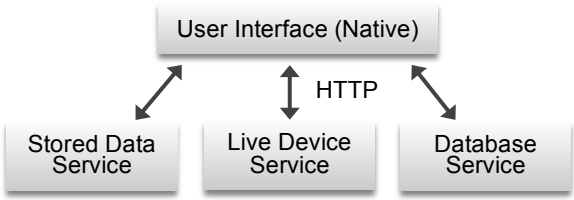

Figure 2: Application Using HTTP Services.

Using HTTP as the network protocol was an easy choice. There are many good HTTP development tools available for virtually all languages and operating systems and most of them are free. HTTP is the native protocol used by web browsers, which makes building web applications easier and allows the browser to be used as a testing tool. In addition, the basic verbs used by the HTTP protocol (GET, SET, POST, DELETE) easily map onto the operations needed by most Controls applications [6]. So creating an HTTP network API for a Controls toolkit is not difficult. And there are many good tools available for encoding and decoding data into JSON [7] and XML [8], which have become the standard ways to package data moving into and out of HTTP services.

There are many good choices for wrapping a toolkit in an HTTP shell. Typically, you would choose one written in the same language as the tools that you want to wrap. We chose a Java wrapper in most cases, both to match existing toolkits and to match our developer preferences. In particular, we expose many of our services using a Java Enterprise Edition (Java EE) server [9]. We started with one called GlassFish [10], but have recently moved these services to Payara [11]. Both are free. We have also used an HTTP wrapper for our C++ tools [12].

By moving services onto the network and exposing them with an HTTP interface, we open up applications to services that may have been previously unavailable. For example, a C++ application can now take advantage of code written in Java, and a Python script can now access tools written in C++. This solved some problems, but not all that was needed.

SHARING HTTP SERVICES

With a monolithic application, all "services" are built into the application and are available only to that application. In theory, the same could be done with HTTP services. An application, when it starts, could also start the HTTP services that it needs, use them while running, then stop those services when it quits.

In practice, though, the services model is much more powerful when the services are always available and can service multiple applications at the same time. In this case, you start to think of services as part of the infrastructure available to all applications, like the networked file system. Applications become "thin" – that is, they consist of nothing but native user interface code and calls to HTTP services. This makes constructing applications simpler and developers more productive. An always-on service infrastructure makes it possible to construct more types of clients (web, remote, mobile), as long as the clients can figure out how to talk to the needed services.

So how do clients know where services are located and how to communicate with them? For HTTP communication, this involves the client knowing the hostname (or IP address) and port number for each of the services that the client will be using. Initially, these can be hardcoded or located in a file or database. But, ultimately, you'll want a more flexible arrangement. The solution that we use involves running a Reverse Proxy Server as shown in Fig. 3 below.

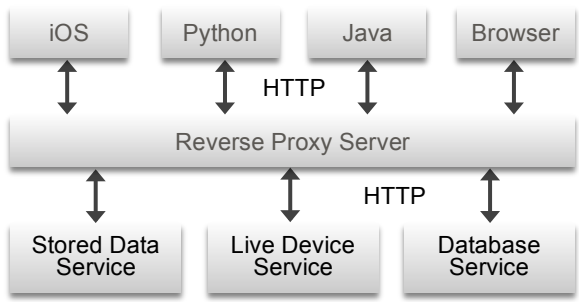

Figure 3: Applications Sharing HTTP Services.

A reverse proxy [13] retrieves resources on behalf of a client from one or more services. It then returns those resources to the client as if they originated directly from the services. All clients deal only with the reverse proxy and know nothing about the location of the services. The reverse proxy maintains an internal mapping of request type and service location that allows it to reroute requests to the right service. If the service location moves, only the reverse proxy needs to be informed. We are currently using reverse proxy software made by NGINX [14].

DISCUSSION

Other Advantages of Reverse Proxys

As noted above, a reverse proxy is primarily intended to assist in routing HTTP messages from clients to services. But it can be used for additional purposes:

- Load Balancing – At some point, resources may get stretched at one or more of the HTTP services. In this case, you can run more than one service of a given type and set up the reverse proxy to take care of how clients are routed to services.
- Connection Limiting – The reverse proxy can be used to limit both the number of connections and the

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THMPA06

frequency with which any one client can connect. This helps to prevent overloaded services and inhibit Denial of Service (DOS) attacks [15].

- Other – A reverse proxy can also be a focal point for additional security measures, data compression, and SSL management.

More on Moving to HTTP Services

After reading this paper, you may be thinking that transitioning to HTTP services is a big job. However, virtually all of the work can be done incrementally. We started with one set of tools (Stored Data) that we use to access data from our logging system, wrapping these Java tools inside a GlassFish server. Then we modified one C++ data viewing application to read data from that server.

The success of this initial work convinced us to keep going down this path, wrapping more toolkits as HTTP services, and modifying more applications to use them. Use of the Reverse Proxy Server became essential only when we started running remote applications, though we would recommend using one even for internal use.

Developing HTTP Applications

There are many good and free HTTP development tools. We, in no way, did an exhaustive search of the best possible tools. But for those that might be interested, here is a list of the tools that we have found useful.

The Integrated Development Environment (IDE) that most developers are using is Eclipse [16]. It provides good integration with the Java EE servers that we typically use to run HTTP services, has good facilities for editing and debugging, and can be extended with a number of useful plugins.

We have found it useful to have tools that can make HTTP requests and see their response. For HTTP GET requests, a standard web browser is often useful, properly formatting XML or JSON responses. Some browsers have plugins (eg. RESTClient for Firefox) that permit the testing of all HTTP requests. Other tools of this type we have found useful are the Curl command-line tool [17] as well as other standalone HTTP/REST clients that have user interfaces such as WizTools/RESTClient [18].

Additional Pros and Cons

We have a small group of programmers that is primarily responsible for building and maintaining most of the applications used by members of our Collider-Accelerator facility. Over time, individual programmers become expert in particular areas of the control system. Moving software tools to HTTP services has allowed those experts to more easily take responsibility for those software tools with which they feel most comfortable and have the most expertise. And it allows those experts more freedom to package, test, and release changes as needed.

We have noticed a couple of disadvantages when comparing the process of building and supporting monolithic applications vs. HTTP service applications. First, the extra network hop required to make remote procedure calls using HTTP can have an effect on performance. A minimum round-trip time for accessing an HTTP service is about a millisecond. Depending on the tool and the application, this time may or may not be significant. A second disadvantage is that it is sometimes more complicated to diagnose and debug software issues. With a monolithic application, one can easily attach a debugger to diagnose a problem. But an application built with HTTP services is split across many processes. Even if you can isolate the problem to one of the HTTP services, you will usually need to set up a private test environment to debug the problem. Neither of these problems is severe for most applications, but it is wise to keep them in mind.

Future Directions

Containers [19] allow for the packaging of HTTP services that contain code, configuration, and dependencies in a module that can run in virtually any operating system environment. This would help to reduce the time that is currently necessary setting up the HTTP service environment.

REFERENCES

[1] Wikipedia, Monolithic application, https://en.wikipedia.org/wiki/Monolithic_application

[2] acmqueue, The Rise and Fall of CORBA, http://queue.acm.org/detail.cfm?id=1142044

[3] Wikipedia, Java Native Interface, https://en.wikipedia.org/wiki/Java_Native_Interface

[4] Wikipedia, Network transparency, https://en.wikipedia.org/wiki/Network_transparency

[5] NoMachine, https://www.nomachine.com

[6] Wikipedia, Hypertext Transfer Protocol, https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[7] Wikipedia, JSON https://en.wikipedia.org/wiki/JSON

[8] Wikipedia, XML https://en.wikipedia.org/wiki/XML

[9] Wikipedia, Java Platform, Enterprise Edition, https://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition

[10] Wikipedia, GlassFish, https://en.wikipedia.org/wiki/GlassFish

[11] Payara, https://www.payara.fish

[12] CPP-NETLIB, http://cpp-netlib.org

[13] Wikipedia, Reverse proxy, https://en.wikipedia.org/wiki/Reverse_proxy

[14] NGINX, https://www.nginx.com

[15] Wikipedia, Denial-of-service, https://en.wikipedia.org/wiki/Denial-of-service_attack

[16] Eclipse, http://www.eclipse.org

[17] Curl, https://curl.haxx.se

[18] RESTClient, https://www.wiztools.org

[19] Containers, https://aws.amazon.com/containers/