ISBN: 978-3-95450-168-7 doi:10.18429/JACoW-IPAC2015-MOPWI049

OPEN XAL SERVICES ARCHITECTURE*

T. Pelaia II#, ORNL, Oak Ridge, TN 37831, USA

Abstract

Open XAL is an accelerator physics software platform developed in collaboration among several facilities around the world. It includes a powerful new services extension that allows for natural remote procedure calls. The high level services interface is based upon custom implementations of modern standard protocols such as JSON-RPC and WebSockets. This choice of modern protocols allows for flexibility such as seamless communication with web clients free of plugins plus rich object type support. The JSON parser was designed for convenient data type transformations with easy extensibility, high performance and low memory overhead. The Open XAL services architecture features a simple application programming interface, high performance, memory efficiency and thread safety.

INTRODUCTION

Open XAL [1, 2] is a Java based platform for building accelerator physics software. The software products can be categorized as either application or standalone service. Here, an application refers to software with a user interface that is launched by an end user and runs until the user quits the application. A standalone service is software that runs perpetually unattended and has no user interface. Other client applications may communicate with a service for management or to display data from the service. Services are useful for monitoring, logging and certain calculations that are best suited to be offloaded from the client.

The Open XAL services framework is designed to provide a high level API to discover and communicate with services over standard protocols implemented internally. The communication protocols were chosen to allow for both thick application and thin web browser clients.

FEATURES

The services framework is designed to provide a high performance, low overhead remote communication mechanism that translates local Java calls into remote messages without having to provide stubs. Furthermore, it provides for dynamic registration and lookup which eliminates the need for configuring service addresses and allows services to be launched from any server on the local network rather than tied to one specific server. The implementation is thread safe and can process multiple concurrent calls.

PROTOCOLS

The services framework is built upon three standard protocols: multi-cast DNS, JSON-RPC [3] and WebSocket [4]. The multi-cast DNS protocol is implemented using the external JmDNS [5] library. The JSON-RPC and WebSocket protocols have been implemented directly in Open XAL. All three protocols are wrapped in the services framework so as to allow the public API to be independent of these internal protocols. Most of the interaction with the services framework is through the ServiceDirectory class and specifically the default singleton of this class known as the “default directory.”

Multi-cast DNS

The open source JmDNS library provides the implementation of multi-cast DNS that is used in Open XAL. Multi-cast DNS allows for dynamic registration and lookup of services. A service is registered by name and type pair and are bound to the provided IP address and port. Clients can lookup services using the name and type pair and the IP addresses and ports of matching services are provided.

The services framework hides the details of the underlying JmDNS library. A service is simply registered using the default directory passing only the service name, a Java interface and the service provider implementing the Java interface. Internally, the fully qualified lowercase name of the Java interface is converted to a properly formatted multi-cast DNS type by replacing periods with underscores and appending “_tcp.local.” to provide a unique type that conforms to the requirements.

JSON-RPC

A variant of JSON-RPC is the messaging protocol that is used to encode messages with JSON constructs. This protocol is hidden from the caller as method calls are automatically converted to the messaging protocol. Internally, a message request is encoded as a JSON [6] object (also known as a dictionary) using the “message,” “params” and “id” keys. The message parameter consists of the service name followed by the method name separated by “#” such as “*MySeervice#doSomething*” for example. The params are an array of JSON encoded parameters to pass. The id parameter is an incremented integer that is used to uniquely identify the request.

A custom JSON implementation is used to efficiently encode and decode JSON to and from Java objects. This

* This manuscript has been authored by UT-Battelle, LLC, under Contract No. DE-AC0500OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for the United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).
# pelaiata@ornl.gov

implementation is highly tuned for both time and memory performance while allowing support for many Java classes beyond those defined in the standard JSON set of string, number, dictionary, array, true, false and null. Custom types are supported using the dictionary type with a special type key named "__XALTYPE" and corresponding value key named "value." Using this feature, the coder defines support for a rich set of predefined data types including all of the Java primitive types, common collection types, Serializable conforming types and more. Java method calls of an arbitrary number of parameters consisting of any combination of these types (even multidimensional arrays) and returning any combinatory of these types are supported. For example, a method could be called taking a string and an array of double precision numbers as parameters and returning an integer. The return value is returned using a dictionary with keys "result," "id" and "error" where the id matches the request id, the result is the encoded Java return value and the error is an encoded exception if any.

WebSocket

The JSON-RPC messages are packaged and passed through a critical subset of the WebSocket protocol. A custom implementation was developed to provide high performance and low memory consumption. WebSocket (as with JSON) was chosen since it is supported natively in modern web browsers allowing the option for web clients without the need for plugins. The service framework has been successfully tested using both Safari on Mac OS X and Firefox on Linux web browsers.

HIGH LEVEL API

The services framework hides the underlying protocols and provides a high level API that makes it easy to implement a service and a client.

Service Handler

Implementing a service involves providing a Java interface which declares the remote methods and a handler which implements this interface. The interface is specified in the service's extension since it is made available to any other application or service. The interface can optionally mark any method with the "@OneWay" annotation to indicate that the method will not return after execution and thus will be a non-blocking call. The handler class is internal to the specific service and should implement the interface.

Upon startup, the service should register itself with the interface, service name and handler. This is done with a single call to the default directory's registerService() method. Remote method calls to the service will automatically be forwarded to the registered handler's corresponding method for processing.

Client Calls

The client first needs to discover which services are available on the local network. It can do this either by setting up a monitor with callback or with a single call to the default directory's findServicesWithType() method passing the Java interface of the service to lookup and a timeout. For the latter approach, this method returns an array of service references that serve the requested interface. Upon picking a single service reference from the array, the client makes a call to the default directory's getProxy() method passing the service's interface and the desired service reference. The returned proxy implements the methods of the corresponding service's interface. The client can make calls to this proxy as if it were a local object and those calls will automatically be dispatched to the remote service. The proxy also implements methods that provide information about the service itself such as the service name, host and port.

When using the monitor approach, the client can use the callback to detect new services and determine if existing services have terminated.

Concurrency Support

A concurrency support class, RemoteDataCache, is provided to prevent waiting on a response from a remote service. It provides a mechanism to submit a request to a remote service and receive a callback when a response becomes available. Multiple pending requests result in a single coalesced response. This tool can be tested for an active connection.

FUTURE PLANS

Building out infrastructure for web applications [7] is the main focus of future efforts for the services framework. A simple web application has been demonstrated, but work needs to be done to formally provide support for three different launch mechanisms: Direct, Directed and Directory.

In the Direct launch mechanism, the service registers itself upon launch and immediately launches the web browser on the same node and the client connects to the service. This requires zero configuration and has already been demonstrated; however, on Linux an issue has been encountered in launching the browser automatically.

In the Directed launch mechanism, a launcher executable is run on the local console. The launcher can launch a service on a server on the local network and then it monitors to discover the services on the local network. The client web applications are then launched on the local console and the launcher configures it to be bound to the corresponding service.

The Directory launch mechanism has a directory executable that runs with a fixed IP address and port and is responsible for spawning the services in response to a client request. This approach is somewhat more similar to traditional web services and would be most practical for web clients which are intended to be run outside of the accelerator network.

Also, while the underlying JSON coder is easily customizable to support any Java type, the services framework does not expose this feature and thus only supports the rich set of predefined data types. Exposing extensibility of custom data types may be supported in the future.

REFERENCES

[1] Open XAL website: http://xaldev.sourceforge.net
[2] T. Pelaia II, "Open XAL Status Report 2015", MOPWI050, these proceedings, IPAC'15, Richmond, VA (2015).
[3] JSON-RPC website: http://json-rpc.org
[4] WebSocket Protocol website: http://tools.ietf.org/html/rfc6455
[5] JmDNS website: http://jmdns.sourceforge.net
[6] JSON website: http://json.org
[7] T. Pelaia II, "Future of High Level Apps at SNS", ESS Open XAL Workshop 2014; https://indico.esss.lu.se/indico/event/151/contribution/19