I have been a ham for 15 years and one of the issues that has been a pain ispa the lack of interoperability between ham applications. This has been due to non-standard interoperability between software programs in general.

So this year I have started a project called “StarGate”. The purpose of this project is to create working copies of 4 Amateur Radio Web Services:

* RigGate – a service to control one or more radios. This is the flagship service and should be usable by press time
* LogGate – a service to log QSO into a database.
* DXClusterGate – DX Cluster service with push notification using Microsoft SignalR service.
* BeaconGate – Provide a cloud location for Reverse Beacon Reports.

At the time of writing all except RigGate hasn’t been started. I thought that RigGate would be the easiest to write. I am fairly sure I was incorrect. ;-) My plan is to get Flex PowerSDR, ICOM and Kenwood support for RigGate, but not sure how much else I can put into it. Being that it is open source I am open others will contribute to the project.

These services and projects are not a replacement for logbook programs, but to provide a powerful but easy flexible framework for all Amateur Radio applications to use common resources. The StarGates takes care of a lot of hard and dull parts of building an application so developers don’t have to reinvent the wheel.

**Benefits**

**Amateur Radio user**

This will mean that every application that uses the service will cooperate together. For example:

* Digital software can log contacts to same database is the logging application.
* Cluster applications, logging applications, digital applications can talk to the radio at the same time.
* Server and the client doesn’t have to be on the same computer or country.
* Server will be backwards compatible with older versions of the client.
* With a single click log your contacts to one or more servers. No more exporting an ADIF file on one logbook app and important it on another.
* More than one user and/or one computer could be logging to the same database at the same time. This would be great for multi-Op contest like Field Day.
* Applications are no longer required to be desktop applications. There can and will be browser logbook programs that can control the radio and log to the same database as a desktop application.
* Upgrade friendly. Servers will be backward compatible with older versions of applications.

Version support is really important. Here is why: Let’ say you like Acme PSK that uses LogGate V1, however you upgrade to the latest version of WizBang logger which wants V3.1 of LogGate. Since the latest version of LogGate will support V1-Vn everyone will be happy. There still could be a version problem if say QSL-Me.com writes there web services and write previous version services, then it won’t work.

**Application developer**

* This means you can spend more time on the features that is important to your users and less time reinvent the wheel.
* Your applications will work hand in hand with others using the same gateways regardless of language or operating system.
* Making connections to databases, clusters, radios
* Installs become a lot easier since you don’t have to worry about database locations, drivers, etc.
* Each server should have an URI which will display usage of the services.
* Source code for all the StarGate projects will be hosted on GitHub.com
* Versioning reduces the pressure to upgrade your application because a new StarGate server has been released. You users can upgrade that StarGate server without feature it will break your application.

Each projects will come with a reference client application and client library for .NET to show how to implement the calls to the servers.

Each service can run as a standalone webserver or as a part of an application server such as IIS, TomCat or Jetty. The first version will be a Windows self-hosted console application or Windows service. However the StarGate project is more about defining protocols using ReST than defining which system or language is used to write the servers and client. Since ReST is language and operating system agnostic client and servers can be written and/or ported to Linux, Mac, etc. very quickly.



**History**

There have been standards ways to connect software, but for the most part they have been low level or very expensive. In the late 1990 web services came into being. It was based upon the idea that web service could not only display web sites to a browser but also be a server of information to applications. A new standard was created called SOAP (Simple Object Access Protocol). When you are talking about computer protocol if you see the word “simple” it means very very complex. SOAP is no exception.

In 2000 Dr. Roy Fielding coined the term Rest for Representation State Transfer (ReST), however it didn’t really catch on until after 2010. Now Rest is being used by all the cool kids from Twitter, Amazon, Wal-Mart, Google, etc., etc.

**ReST or ReSTful standard**

What makes ReST so great is that it is truly simple. It uses the standard HTTP protocol as a transport, just like your favorite web browser does.

ReST uses HTTP actions GET, POST, PUT and DELETE to map to database CRUD (Create, Read, Update and Delete) operation. Therefore an URL like:

<http://localhost:7301/api/v1/connection> to a RigGate server could return something like this.

<ArrayOfRadioComConnConfig>

<RadioComConnConfig>

<Bps>19200</Bps>

<ConnectionName>Flex</ConnectionName>

<DataBits>7</DataBits>

<Dtr>false</Dtr>

<Parity>NONE</Parity>

<Port>COM5</Port>

<RadioType>PowerSDR</RadioType>

<Rts>false</Rts>

<StopBits>1</StopBits>

</RadioComConnConfig>

</ArrayOfRadioComConnConfig>

This is a standard http GET action which is the same action that a normal web site uses to return a page. I got this from using the Chrome browser. Chrome and Firefox converts JSON to XML. There are JSON parsers for just about every programming language you can think of.

Notice the “v1” on the URL. If there is a change to the content that would break a client, then a new version is added to the server. It would then become V1.1 or V2, however the older version could still be supported.

Doing POST and PUT action is almost the same. A JSON or XML object is passed in the body of the http request.

To set anyone one or more radio properties such as frequency or mode the client would issue at a post statement. A .NET client wrapper has been provided so the C# code would look something like this:

*public class RadioProperty*

*{*

*public string Property Name { get; set; }*

*public string PropertyValue { get; set; }*

*public string Status {get; set; }*

*public string EnumItemNum { get; set; }*

*public string RadioId { get; set; }*

*}*

*public class RadioPropComandList*

*{*

*public List<RadioProperty> Properties {get; set; }*

*public string Status { get; set; }*

*public int Success { get; set; }*

*public int Failed { get; set; }*

*}*

public class Connection

{

    public string DisplayName { get; set; }

    public string HostName { get; set; }

    Public string Port { get; set; }

    public bool DefaultServer { get; set; }

}

*public void PostSetModeFlexTest()*

*{*

*string baseUrl = server.BuildUriControllerOnly(RadioConstants.RadioController);*

*var cmdReq = new RadioPropComandList();*

*var setting = new RadioProperty();*

*setting.PropertyName = RadioConstants.Mode;*

*setting.PropertyValue = RadioConstants.USB;*

*setting.EnumItemNum = RadioConstants.VfoA;*

*cmdReq.Properties.Add(setting);*

*var respCmd =**RadioControl.**SetRadioProperty(cmdReq, server);*

*}*

Notice that RadioPropCommandList is a list, so many properties can be set with a single call. The client doesn’t know anything about how to control the radio. It just has *a Connection Name* for the com port and radio. To keep it short for paper publication I removed the comments that were mainly for the *Visual Studio Code Insight.*

To retrieve a list of property requires only a change from *SetRadioProperty to GetRadioProperty*. Assigning a property value also wouldn’t be required.

With *C# Web API* the method behind *RadioControl.SetRadioProperty* is also very simple:

static public RadioPropComandList SetRadioProperty(RadioPropComandList cmd, Connection serv)

{

    string baseUrl = serv.BuildUriControllerOnly(RadioConstants.RadioController);

    var client = new HttpClient();

    HttpResponseMessage response = client.PostAsJsonAsync(baseUrl, cmd).Result;

    var results = response.Content.ReadAsAsync<RadioPropComandList>().Result;

    return results;

}

This would only have to be done writing if you are using a language that didn’t have a JSON or XML serializer built into the http client class.

**Suggested TCP Ports**

|  |  |
| --- | --- |
| Type of Application | TCP Port Number |
| LogGate | 7300 |
| RigGate | 7301 |
| DXClusterGate | 7302 |
| BeaconGate | 7303 |
| CallLookUpGate | 7304 |
| PropagationGates | 7305 |

**License**

These projects will be open source under the Apache License. **http://www.apache.org/licenses/**

**Links**

StarGate Wiki: <https://github.com/wa1gon/RigGate/wiki>

RigGate GitHub Repository: <https://github.com/wa1gon/RigGate>

LogGate GitHub Repository: <https://github.com/wa1gon/LogGate>

DXClusterGate GitHub Repository: <https://github.com/wa1gon/DxClusterGate>

Yahoo Group: <https://groups.yahoo.com/neo/groups/StargateRadio>

**Final Words**

This has to be a team project. I hope that I have made a good case why standardizing on ReST web services make a lot of sense and will help everyone. However ever if I wrote all the services I would still need application author to adopt the protocols.