I have been a ham for 15 years and one of the issues that has been a pain has been 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.
* DXGate – DX Cluster service with push notification using Microsoft SignalR service.

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

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

The 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.

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.

**Benefits**

**Amateur Radio user**

This will mean that every application to 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.

Everything works together.

For the application developer this means you can spend more time on the features that is important to your users and less time on figuring out which DB or radio protocols you need to write one off code.

**ReST or ReSTful standard**

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

xmlns:i="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://schemas.datacontract.org/2004/07/Wa1gon.Models">

<RadioComConnConfig>

<AdditionSetting

xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays" />

<Bps>19200</Bps>

<Command i:nil="true" />

<ConnectionName>Flex</ConnectionName>

<DataBits>7</DataBits>

<Default>true</Default>

<Dtr>false</Dtr>

<IsConnected>false</IsConnected>

<Parity>NONE</Parity>

<Port>COM5</Port>

<RadioType>PowerSDR</RadioType>

<Rts>false</Rts>

<Status i:nil="true" />

<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. Note that this is in XML, but the default format is JSON. 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 old version would still be accessible.

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 BaseValidator Validator { 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 string BuildUriControllerOnly(string controller) {…}

public string BuildUriControllerDisplay(string controller) {…}

}

*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.*

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;

}