Robot Raconteur® for Web Browsers using Bridge.NET and HTML5/JavaScript

Version 0.8 Beta

http://robotraconteur.com

Dr. John Wason

Wason Technology, LLC

PO Box 669

Tuxedo, NY 10987

wason@wasontech.com

http://wasontech.com

May 3, 2016

Contents

1 Introduction

With the introduction of HTML5 and the associated improvements to JavaScript, Web Browsers have become power software platforms that can implement fully featured applications. Robot Raconteur®takes advantage of this improved capability in combination with WebSockets to implement a pure HTML5/JavaScript client solution. The HTML5/JavaScript clients do not require any dependencies beyond a modern Web Browser, and the software can be served directly from a web server without requiring any installation. The HTML5/JavaScript client implementation of Robot Raconteur has been successfully tested on Chrome, Firefox, Internet Explorer, Edge, and Safari.

The HTML5/JavaScript implementation is developed using a tool called “Bridge.NET”. This tools allows for C# code to be translated directly into JavaScript. The “translate to JavaScript” solution to Web Browser development has become increasingly popular due to the limitation of the JavaScript language. The main advantages of C# over JavaScript in the development of Robot Raconteur clients are:

- C# is strongly typed.
- C# is fully object oriented and does not require obtuse syntax to implement standard data structures.
- C# supports “async/await” syntax. This is the most important advantage, and leads to dramatically simpler code.

The exact usage instructions for Bridge.NET is covered in the documentation which can be found at `http://bridge.net`. It is recommended that the iRobot Create examples be studied for better understanding of how Robot Raconteur Bridge.NET projects are structured.

2 Project Configuration

Creating a new project for Robot Raconteur Bridge requires first creating a new project for Bridge.NET. Instructions to create a new Bridge.NET project can be found at `http://bridge.net/kb/getting-started/`.

Once a Bridge.NET project has been created, unzip the Robot Raconteur Bridge zip file. Add “RobotRaconteurBridge.dll” as a project reference. Next, copy the “RobotRaconteurBridge.js” and “RobotRaconteurBridge.min.js” files to the directory “Bridge/output” in the project folder. This directory should contain “bridge.js” and “bridge.min.js” after building the project. Next, add “../output/RobotRaconteurBridge.js” as a script in the `Bridge/www/demo.html` file.

At this point, Robot Raconteur is ready for use.

3 Service Types

Robot Raconteur for Python and MATLAB has the ability to dynamically generate proxy object references with types provided upon connection to a service. Robot Raconteur Bridge also has this capability, so there is no need to generate “Thunk” code. Note that almost all operations involving the network are “awaited” to allow control to return to the browser during operation.

The object reference is returned as a “dynamic” object. For instance, to connect to a service:

```
dynamic o=await RobotRaconteurNode.s.ConnectService("rr+tcp://localhost:2354/?service=Create")
```

To disconnect:

```
await RobotRaconteurNode.s.DisconnectService(o);
```

Robot Raconteur Bridge currently supports the following members: property, function, event, objref, pipe, and callback. It does not currently support wires or callbacks. These will be available in a future version.

3.1 property

Properties are accessed by appending “get_” and “set_” to the property name to create getter/setter functions, and then awaiting the function. For example:

```
int d=await o.get_DistanceTraveled();
await o.set_DistanceTraveled(10);
```

3.2 function

Functions are awaited and use standard C# syntax.

```
await o.Drive(10,10);
```

There may also be a return value.

```
int r=await o.SomeFunction();
```

3.3 event

Events handlers are added using the “addListener” function.

```
o.Bump.addListener((Action) () => Console.WriteLine("Bump!"));
```

Where `() => Console.WriteLine("Bump!")` is a lambda function. It could also be a standard method delegate.

3.4 objref

Objrefs use awaited “get_” style functions.

```
dynamic w1=await o.get_Webcams(0);
```

3.5 pipe

Pipes are “Pipe” fields in the object reference.

```
Pipe p1=w1.FrameStream;;
var ep1=await p1.Connect();
var frame1=ep1.ReceivePacket();
```

3.6 callback

Callbacks are set by setting the field in the object reference.

```
byte[] play_cb(int DistanceTraveled, int AngleTraveled)
{
      return new byte[] {69,16,60,16,69,16};
}


o.play_callback=(Func<int, int, byte[]>)play_cb;
```

4 C# $\leftrightarrow$ Robot Raconteur Data Type Mapping

Each valid Robot Raconteur type has a corresponding C# data type. This mapping is similar to Python, but is in some ways simpler because C# has stronger typing than Python. Table 1 shows the mapping between Robot Raconteur and C# data types.

As seen in Table 1, most of the data type maps are straightforward. Due to a limitation in JavaScript, 64 bit integers are not currently supported. Support will be available in a future version.

Maps are stored in Dictionaries from namespace `System.Collections.Generic`. The `MultiDimArray` type contains the array stored as flat arrays in column-major order, which is frequently called "Fortran order". This means that the columns are "stacked" on top of each other to create the flat array. This is opposite of C which uses row-major order. See Section A.3 for more details on this class.

Structures are stored as a dynamic type. To create a new structure, create a new object and add fields dynamically:

```
dynamic s=new object();
s.v1=(double)10.0
s.v2="This is a string"
```

The fields in a structure must all be specified and match the types defined in the service definition.

The `varvalue` type is stored in a special class named `RobotRaconteurVarValue`. This class stores both the data and the type.

```
var v=new RobotRaconteurVarValue("This is a string", "string");
var v_data=v.data;
var v_type=v.datatype;
```

Table 1: Robot Raconteur ↔ C# Type Map

Robot Raconteur Type	C# Type	Notes
double	`double`	
single	`float`	
int8	`sbyte`	
uint8	`byte`	
int16	`short`	
uint16	`ushort`	
int32	`int`	
uint32	`uint`	
int64	`long`	
uint64	`ulong`	
double[]	`double[]`	
single[]	`float[]`	
int8[]	`sbyte[]`	
uint8[]	`byte[]`	
int16[]	`short[]`	
uint16[]	`ushort[]`	
int32[]	`int[]`	
uint32[]	`uint[]`	
int64[]	*long[]*	Not currently supported
uint64[]	*ulong[]*	Not currently supported
string	`string`	Strings are transmitted as UTF-8 but converted to normal C# strings
T{int32}	`Dictionary<int,`*T*`>`	Map type, *T* is a template
T{string}	`Dictionary<string,`*T*`>`	Map type, *T* is a template
T{list}	`List<`*T*`>`	List type, *T* is a template
structure	dynamic	Use dynamic type structure.
N[*]	`MultiDimArray`	Multi-dim array of type *N*
varvalue	`RobotRaconteurVarValue`	
varobject	`dynamic`	

5 TLS Security

TLS security is available in Robot Raconteur Bridge using “forge.js”, a project that implements TLS in pure JavaScript. A slightly modified version of “forge.js” is distributed with Robot Raconteur Bridge. It is unfortunately a fairly large file. To utilize TLS, include the “forge.bundle.robotraconteur.js” file as a script in the HTML page, and use the “rrs+” transport schemes, for instance “rrs+tcp”, “rrs+ws”, or “rrs+wss”. Note that “rrs+wss” will use HTTPS security, while “rrs+ws” will use plain HTTP. The extra layer of HTTPS security is recommended but may not be necessary in all cases.

A Robot Raconteur Reference

A.1 RobotRaconteurVarValue

`class` **RobotRaconteurVarValue**

A class to represent `varvalue` types that also contains the type. The type is a string in the same format as service definitions.

RobotRaconteurVarValue(`object` *data*, `string` *datatype*)

Creates a new `RobotRaconteurVarValue` object.

Parameters:

- *data* (`object`) - The data
- *datatype* (`string`) - The type of the data in service definition format

Return Value:

A new `RobotRaconteurVarValue` object

`object` **data**

The data

`string` **datatype**

The type of the data in service definition format.

A.2 RobotRaconteurNode

`class` **RobotRaconteurNode**

`RobotRaconteurNode` contains the central controls for the node. It contains the services, client contexts, transports, service types, and the logic that operates the node. The `s` property is the "singleton" of the node. All functions must use this property to access the node.

`string` **RobotRaconteurNode.s.Version**

Returns the version of the Robot Raconteur library.

`string` **RobotRaconteurNode.s.NodeName**

The name of the node. This is used to help find the correct node for a service. It is not unique. The name must be set before any other operations on the node. If it is not set it remains blank.

`NodeID` **RobotRaconteurNode.s.NodeID**

The ID of the node. This is used to uniquely identify the node and must be unique for all nodes. A NodeID is simply a standard UUID. If the node id is set it must be done before any other operations on the node. If the node id is not set a random node id is assigned to the node.

`Task<object>` **RobotRaconteurNode.s.ConnectService**(`string` *url*, `string` *username=null*, `Dictionary<string,object>` *credentials=null*)

Creates a connection to a remote service located by the *url*. The *username* and *credentials* are optional if authentication is used. This function should be awaited.

Parameters:

- *url* (`string`) - The url to connect to.
- *username* (`string`) - (optional) The username to use with authentication.
- *credentials* (`Dictionary<string,object>`) - (optional) The credentials to use with authentication.

Return Value:

(`Task<object>`) - A task to the connected object. This is a Robot Raconteur object reference that provides access to the remote service object. It should be cast to the C# inteface type corresponding to the Robot Raconteur object type.

`Task` **RobotRaconteurNode.s.DisconnectService**(`object` *obj*)

Disconnects a service.

Parameters:

- *obj* (`object`) - The client object to disconnect. Must have been connected with the `ConnectService` function. This function should be awaited.

Return Value:

None

`Task` **RobotRaconteurNode.s.RequestObjectLock**(`object` *obj*, `RobotRaconteurObjectLockFlags` *flags*)

Requests an object lock for a connected service object. The flags specify if the lock is a "User" lock or a "Client" lock. This function should be awaited.

Parameters:

- *obj* (`object`) - The object to lock. This object must have been created through `ConnectService` or an objref.
- *flags* (`RobotRaconteurObjectLockFlags`) - The flags for the lock. Must be `RobotRaconteurObjectL` for a "User" lock, or `RobotRaconteurObjectLockFlags.CLIENT_LOCK` for a "Client" lock.

Return Value:

None

`Task` **RobotRaconteurNode.s.ReleaseObjectLock**(`object` *obj*)

Requests an object lock for a connected service object. The flags specify if the lock is a "User" lock or a "Client" lock. This function should be awaited.

Parameters:

- *obj* (`object`) - The object to lock. This object must have been created through `ConnectService` or an objref.

Return Value:

None

`Task` **RobotRaconteurNode.s.MonitorEnter**(`object` *obj*, `int` *timeout=RR_TIMEOUT_INFINITE*)

Requests a monitor lock for a connected service object. This function should be awaited.

Parameters:

- *obj* (`object`) - The object to lock. This object must have been created through `ConnectService` or an objref.

- *timeout* (`int`) - (optional) The timeout for the lock in milliseconds. Specify -1 for no timeout.

Return Value:

None

`Task` **RobotRaconteurNode.s.MonitorExit**(`object` *obj*)

Releases a monitor lock. This function should be awaited.

Parameters:

- *obj* (`object`) - The object to lock. This object must have been created through `ConnectService` or an objref.

Return Value:

None

A.3 MultiDimArray

`class` **MultiDimArray**

The `MultiDimArray` represents a multi-dimensional array. It stores the data as two separate flat arrays, `Real` and `Imag`. The data is stored is column-major order (Fortran order) which is different than row-major order (C order).

`int` **DimCount**

The number of dimensions.

`int[]` **Dims**

The dimensions in column-major order.

`Array` **Real**

The real data in column-major order.

`bool` **Complex**

`true` if the array has complex data.

`Array` **Imag**

The imaginary data in column-major order.

MultiDimArray(`int[]` *dims*, `Array` *real*, `Array` *imag=null*)

Creates a new array with the provided data.

Parameters:

- *dims* (`int[]`) - The dimensions of the array in column-major order.
- *real* (`Array`) - The real data in column-major order.
- *imag* (`Array`) - (optional) The imag data in column-major order or `null` if the array is not complex.

Return Value:

This is a constructor for use with the `new` keyword.

`void` **RetrieveSubArray**(`int[]` *memorypos*, `MultiDimArray` *buffer*, `int[]` *bufferpos*, `int[]` *count*)

Reads data from the source array into `buffer`.

Parameters:

- *memorypos* (`int[]`) - The start position in the array.
- *buffer* (`MultiDimArray`) - The buffer to read the data into.
- *bufferpos* (`int[]`) - The start position in the buffer.
- *count* (`int[]`) - The number of elements to read.

Return Value:

None

`void` **AssignSubArray**((`int[]` *memorypos*, `MultiDimArray` *buffer*, `int[]` *bufferpos*, `int[]` *count*)

Writes data from `buffer` into the array.

Parameters:

- *memorypos* (`int[]`) - The start position in the array.
- *buffer* (`MultiDimArray`) - The buffer to write data from.
- *bufferpos* (`int[]`) - The start position in the buffer.
- *count* (`int[]`) - The number of elements to read.

Return Value:

None

A.4 Pipe

`class` **Pipe**

The `Pipe` class implements the "pipe" member. The `Pipe` object is used to create `PipeEndpoint` objects which implement a connection between the client and the service. On the client side, the function `Connect` is used to connect a `PipeEndpoint` to the service. On the service side, a callback function `ConnectCallback` is called when clients connects.

`string` **MemberName**

Returns the member name of this pipe.

`Task<Pipe.PipeEndpoint>` **Connect**(`int` *index=-1*)

Connects and returns a `Pipe.PipeEndpoint` on the client connected to the service where another corresponding `Pipe.PipeEndpoint` is created. In a `Pipe`, `Pipe.PipeEndpoints` are *indexed* meaning that there can be more than one `Pipe.PipeEndpoint` pair per pipe that is recognized by the index. This function should be awaited.

Parameters:

- *index* (`int`) - (optional) The index of the `PipeEndpoint` pair. This can be -1 to mean "any index".

Return Value:

(`Pipe<T>.PipeEndpoint`) - The connected `PipeEndpoint`.

A.5 Pipe.PipeEndpoint

`class` **Pipe.PipeEndpoint**

The `Pipe.PipeEndpoint` class represents one end of a connected `Pipe.PipeEndpoint` pair. The pipe endpoints are symmetric, meaning that they are identical in both the client and the service. Packets sent by the client are received on the service, and packets sent by the service are received by the client. Packets are guaranteed to arrive in the same order they were transmitted. The `Pipe.PipeEndpoint` connections are created by the `Pipe` members.

`uint` **Endpoint**

Returns the Robot Raconteur endpoint that this pipe endpoint is associated with. It is

important to note that this is not the pipe endpoint, but the Robot Raconteur connection endpoint. This is used by the service to detect which client the pipe endpoint is associated with. Each client has a unique Robot Raconteur endpoint that identifies the connection. This property is not used on the client side because the client uses a single Robot Raconteur endpoint.

`int` **Index**

Returns the index of the `Pipe<T>.PipeEndpoint`. The combination of `Index` and `Endpoint` uniquely identify a `Pipe.PipeEndpoint` within a `Pipe` member.

`int` **Available**

Returns the number of packets that can be read by `ReceivePacket`.

`dynamic` **ReceivePacket**()

Receives the next available packet. The type will match the type of the pipe specified in the service definition.

Parameters:

None

Return Value:

(`dynamic`) - The next packet

`dynamic` **PeekPacket**()

Same as `ReceivePacket` but does not remove the packet from the receive queue.

Parameters:

None

Return Value:

(`dynamic`) - The next packet

`Task<uint>` **SendPacket**(`dynamic` *packet*)

Sends a packet to be received by the matching `Pipe.PipeEndpoint`. The type must match the type specified by the pipe in the service definition.

Parameters:

- *packet* (`T`) - The packet to send

Return Value:

(`uint`) - The packet number of the sent packet.

`bool` **RequstPacketAck**

Requests acknowledgment packets be generated when packets are received by the remote `PipeEndpoint`. See also `PacketAckReceivedEvent`.

`Task` **Close**()

Closes the pipe endpoint connection pair. This function should be awaited.

Parameters:

None

Return Value:

None

`Action<PipeEndpoint>` **PipeCloseCallback**

A callback function called when the `Pipe.PipeEndpoint` is closed. This is used to detect when it has been closed.

`event Action<PipeEndpoint>` **PacketReceivedEvent**

An event triggered when a packet is received by `PipeEndpoint`.

`event Action<PipeEndpoint,uint>` **PacketAckReceivedEvent**

An event triggered when a packet acknowledgment is received. Packet acknowledgment packets are requested by setting the `RequestPacketAck` field to `true`. Each sent packet will result in an acknowledgment being received and can be used to help with flow control. The `uint` *packetnumber* (second parameter) in the callback function will match the number returned by `SendPacket`.

A.6 NodeID

`class` **NodeID**

The `NodeID` represents a 128-bit unique ID and is synonymous with a UUID. Every node instance must have a unique NodeID. If two NodeID's are the same it can result in unpredictable behavior. In string form, the NodeID uses the standard UUID format {`xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx`} where the "x" is a hexadecimal digit.

NodeID(`string` *id*)

Creates a new `NodeID`.

Parameters:

- *id* (`string`) - The value of the NodeID as a string.

Return Value:

(`NodeID`) - The new `NodeID`.

NodeID(`byte[]` *id*)

Creates a new `NodeID`.

Parameters:

- *id* (`byte[]`) - The value of the NodeID as a 16 byte array.

Return Value:

(`NodeID`) - The new `NodeID`.

`string` **ToString**()

Returns the string representation of the NodeID.

Parameters:

None

Return Value:

(`string`) - The string representation.

`byte[]` **ToByteArray**()

Returns the `byte[]` representation of the NodeID.

Parameters:

None

Return Value:

(`byte[]`) - The `byte[]` representation.

operator ==
operator !=
Standard operators for use with `NodeID`.

A.7 RobotRaconteurException

`class` RobotRaconteurException

`RobotRaconteurException` represents an exception in Robot Raconteur. Every Robot Raconteur function may potentially throw an `Exception`, and the `RobotRaconteurException` represents an exception in Robot Raconteur. It has a number of subclasses that are used to represent specific exceptions:

- `ConnectionException`
- `ProtocolException`
- `ServiceNotFoundException`
- `ObjectNotFoundException`
- `InvalidEndpointException`
- `EndpointCommunicationFatalException`
- `NodeNotFoundException`
- `ServiceException`
- `MemberNotFoundException`
- `DataTypeMismatchException`
- `DataTypeException`
- `DataSerializationException`
- `MessageEntryNotFoundException`
- `UnknownException`
- `RobotRaconteurRemoteException`
- `TransactionTimeoutException`
- `AuthenticationException`
- `ObjectLockedException`

Most of these exceptions are clear from the name what they mean and have standard exception members. The main exception that is different is `RobotRaconteurRemoteException`, which represents an exception that has been transmitted from the opposite end of the connection. It has two fields of interest: `errorname` and `errormessage` which represent the name of the error and the message assosciated with the error.

The `RobotRaconteurRemoteException` class represents an exception that has been passed from the other side of the connection.

`MessageErrorType` **ErrorCode**

The error code of the error.

`string` **Error**

The name of the exception that was thrown remotely. This is non-standard between languages.

`string` **Message**

The message associated with the exception.

A.8 RobotRaconteurRemoteException

`class` **RobotRaconteurRemoteException**

The `RobotRaconteurRemoteException` class represents an exception that has been passed from the other side of the connection.

`MessageErrorType` **ErrorCode**

The error code of the error.

`string` **Error**

The name of the exception that was thrown remotely. This is non-standard between languages.

`string` **Message**

The message associated with the exception.